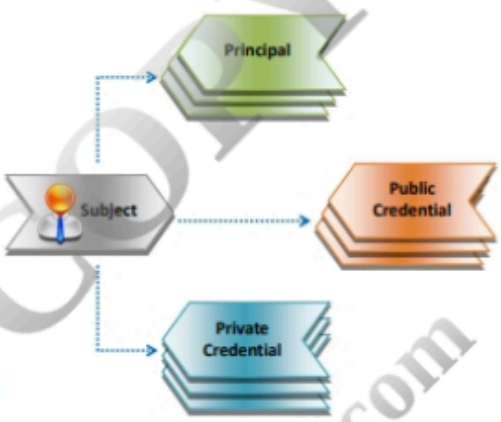
JAAS Subject and Principal

- The Subject class is considered as the **central class** of JAAS
- The Subject represents information for a single **user, entity** or **system**
- The Subject includes the entity **principals, public** (public keys) and **private** (passwords, private keys) credentials
- A Principal **encapsulates** features or properties of a subject
- The JAAS API interface used to represent Principal is `java.security.Principal`
- A Subject is associated with **identities** or principals throughout the authentication process
- A Subject can contain **multiple principals**

For Example, a user with a name principal (George), a social security number principal (123-12-1234) and a username principal (GeorgeD), all of which differentiates the subject from other subjects



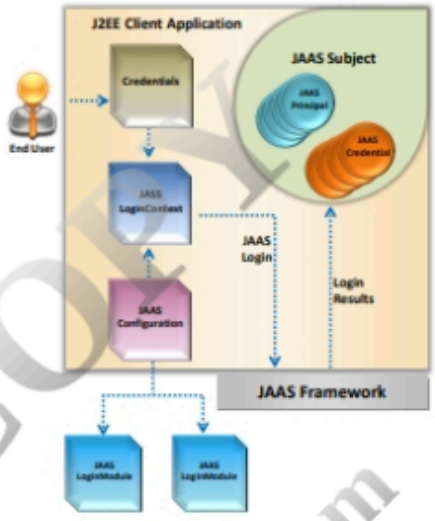
JAAS Subject and Principal (Cont'd)

- **To retrieve principals from the associated subject, the following methods can be used**

Methods	Description
`public Set getPrincipals() {...}`	Retrieves all the Principals from the specified subject
`public Set getPrincipals(Class c) {...}`	Retrieves only the Principals that are instances of class 'c' or its subclasses
`Pubic Set getPrivateCredentials() {...}`	Retrieves the private credentials from the specified subject
`Public Set getPrivateCredentials (Class c) {...}`	Retrieves only the private credentials that are instances of class 'c' or its subclasses
`Public Set getPublicCredentials () {...}`	Retrieves the public credentials from the specified subject
`Public Set getPublicCredentials (Class c) {...}`	Retrieves only the public credentials that are instances of class 'c' or its subclasses
`Public boolean isReadOnly() {...}`	Tests a Subject's read only state
`Public void setReadOnly() {...}`	To set Subject to read only state

Authentication in JAAS

- Authentication is used to verify whether the user is **eligible** to **access** the application
- JAAS enables **plugging** of various authentication modules at **runtime**
- The client application interacts with JAAS through the **LoginContext object**
- Authentication in JAAS can be implemented using the following set of modules as represented in the diagram
  - LoginContext
  - LoginModule
  - CallbackHandler



Authentication Steps in JAAS

1. The initial step is to create a **LoginContext object**. The LoginContext object checks with the configuration file to identify which **LoginModule** is to be loaded. Optionally, a CallbackHandler can be passed to the LoginContext
2. Login method from LoginContext is called to implement authentication. This process loads the predefined LoginModule to verify if the user can be authenticated
3. Integrate credentials and principals with the subject for successful logins
4. In case of login failure, throw a **LoginException**
5. To log out from application, make use of **logout** function from LoginContext

Authorization in JAAS

- Authorization of an user is used to identify whether the (authenticated) user is **eligible** to **perform** certain **functionalities** in the application like accessing a resource
- Since JAAS is developed on the existing Java security model, the process of authorization is **policy-based**
- The policy configuration file consists of a list of entries like "**keystore**" and "**grant**"
- The grant entry provides with the permissions granted for the authenticated codes or principals to perform the security-sensitive operations
- JAAS also supports **principal-based** policy entry
- Permissions are granted in policy with regard to **specific principals**

The basic format of a grant entry looks as:

```
1: grant Codebase "codebase_URL" Signedby
   "signer_name,"
2: Principal principal_class_name
   "principal_name",
3: Principal principal_class_name
   "principal_name",
4: $ {
5: permission permission_class_name
   "target_name", "action",
6: permission permission_class_name
   "target_name", "action",
7: $
8: }
```

Source: http://www2.sys-con.com

Authorization in JAAS (Cont'd)

- The **Policy object** in JAAS indicates the system security policy for a Java application environment
- Once the authentication of user is done, the authorization takes place through the `Subject.doAs` function or `doAsPrivileged` function from Subject class
- The **doAs** function dynamically combines the Subject with the current **AccessControlContext** and then triggers the run method to execute the security checks

Authorization Steps in JAAS

1. The initial step is to invoke `Subject.doAs` or `doAsPrivileged`
2. Check the permission by calling `SecurityManager.checkPermission` or other check methods
3. The SecurityManager redirects the check to the **AccessController**
4. The AccessController checks whether the relevant AccesControlContext includes all the **permissions** required for the action to be performed
5. AccessControlContext is updated by the **SecurityManager** with the permission policies given to the Subject

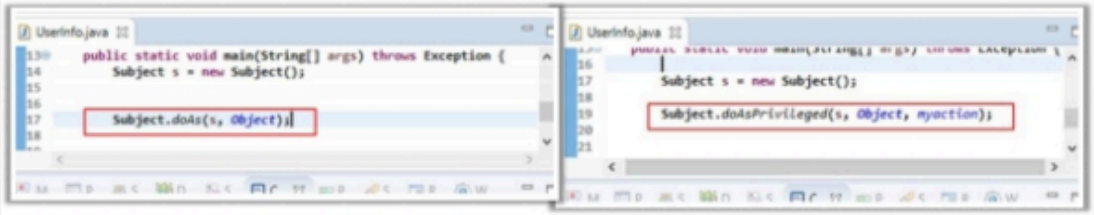
Subject Methods doAs() and doAsPrivileged()

Object doAs (Subject, PrivilegedAction)

- The specified privileged action is carried out as the **specified Subject**
- This method combines the Subject with the AccessControlContext instance of the current thread adding the Subject's permissions to the permissions of that access control context
- A new access control context is created with the combined permissions

Object doAsPriviledgedAction (Subject, PrivilededAction, AccessControlContext)

- This method also possesses the same functionality as the **doAs ()** method but within the specified access control context
- It is mainly used for specifying a null access control context



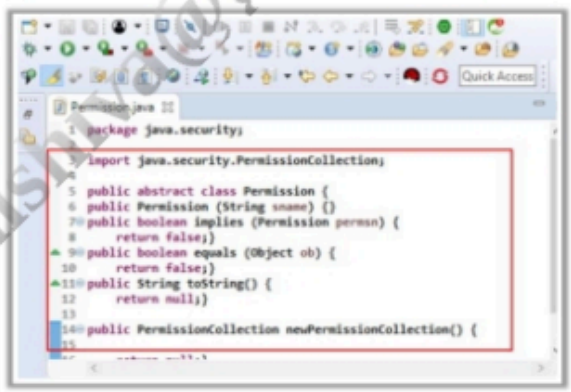
Impersonation in JAAS

- JAAS can be used to implement impersonation at the **application level**
- The Subject class in JAAS enables the execution of a specific **sensitive operation** as another user's entity

- When the above code is executed, the **doAs()** method links the impersonated Subject with the current AccessControlContext and performs the action
- The Subject is removed from the AccessControlContext at the end of **doAs method call**

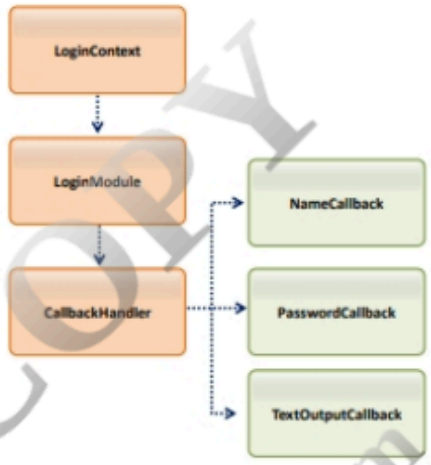
JAAS Permissions

- Permissions play an important role in authorization
- Permissions control the access to **resources**
- JAAS permissions are developed from the existing Java security model
- This model provides an efficient way of controlling access to resources like **sockets** and **files** which excludes URLs
- To integrate JAAS with a web application, a new **permission class** should be created
- The permission class can be created using the following two ways:
  - To extend `java.security.BasicPermission`, which links the permissions to URLs
  - To create URLPermission class that extends `java.security.Permission` class



LoginContext in JAAS

- The **LoginContext** is mainly used for the log in purpose
- The code used for authentication creates a new instance of a LoginContext, and it in turn uses a **Configuration** to identify which login module is used to authenticate a subject
- The login code then executes the login() method from the **LoginContext**



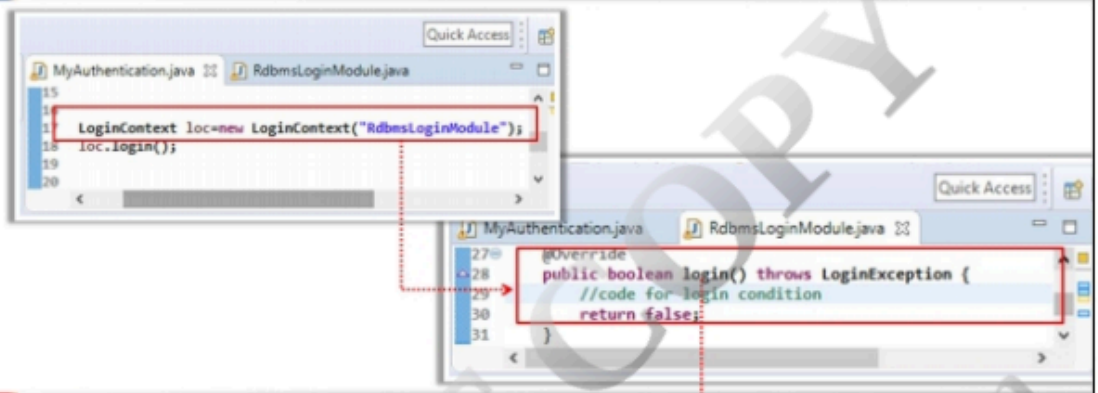
Creating LoginContext

1. The **name parameter** is used to get the login Configuration
2. The constructor without a **CallbackHandler** class depends on the default `CallbackHandler` or on the `LoginModules` to get information
3. Callers of a `LoginContext` constructor should require an AuthPermission with target **"createLoginContext.<name>"**

```
MyAuthentication.java
1  import javax.security.auth.login.LoginContext;
2  import javax.security.auth.login.LoginException;
3
4  public class MyAuthentication {
5  public static void main(String[] args) {
6      System.setProperty("java.security.auth.login.config", "jaas.config");
7      String name = "username";
8      String password = "userPassword";
9
10     try {
11         LoginContext lc = new LoginContext("Test", new TestCallBackHandler(name, password));
12         lc.login();
13     } catch (LoginException e) {
14         e.printStackTrace();
15 }
16 }
17 }
```

LoginContext Instantiation

- `LoginContext` uses **application identifier** to load the appropriate configuration block



- LoginContext locates the JAAS configuration file by the **javax.security.auth.login.config** system property as shown here:

  "java -Djavax.security.auth.login.config=jaas.config MyApp"

JAAS Configuration

The configuration file format consists of the following entries:

- **LoginEntry**: Labels a single entry in the login configuration. The application code implicitly defines the LoginEntry label where a particular **LoginEntry** entry is searched by its login configuration
- **ModuleClass**: Specifies the class name of a JAAS login module
  Example: **com.sun.security.auth.module.Krb5LoginModule** specifies the class name of the Kerberos login module
- **Flag** : Specifies the way to respond when the current login module gives an authentication failure. The flag can be set with anyone of the following values
  - **Required** - Authentication should succeed irrespective of success or failure, proceed to the next login module in this entry
  - **Requisite** - Authentication should succeed. For success condition, proceed to the next login module, and in case of failure, return immediately without executing the remaining login modules
  - **Sufficient** - It is not essential for this login module to succeed in authentication. For success condition, return immediately without executing the remaining login modules, and in case of failure, proceed to the next login module
  - **Optional** - It is not essential for this login module to succeed in authentication. Irrespective of success or failure, always proceed to the next login module in this entry
- **Option="Value"** - Zero or more option settings can be passed to the login module after the flag. The options are given in the form of space-separated lists. The login module lines should be terminated using a semicolon (;)

The JAAS Login Configuration File's General Format:

Locating JAAS Configuration File

The location of JAAS login configuration files follows two approaches:

1. Set a System Property

The system property value `java.security.auth.login.config` is set to the location of the login configuration file

E.g. the system property can be set on the command line as:

```
1: java -
2: Djava.security.auth.login.config=/var/activemq/config/login.config
3: ...
```

2. Configure the JDK

JAAS checks the `$JAVA_HOME/jre/lib/security/java.security` security properties file for entries of the form if the system property is not set

```
1: login.config.url.1=file:C:/activemq/config/login.config
```

- When more than one entry is found, `login.config.url.n`, the entries should be consecutively numbered
- A single configuration file is used for storing login files in `java.security` package

JAAS CallbackHandler and Callbacks

- When `CallbackHandlers` are specified in application, the **LoginModules** remains independent regardless of the application interaction with users
- Login modules make use of **`javax.security.auth.callback.CallbackHandler`** to retrieve authentication information from the user
- The CallbackHandler **interface** is implemented in applications and then passes it to the `LoginContext` that sends it directly to the `LoginModules`
- The LoginModule makes use of the `CallbackHandler`:
  - To get input from the users

    E.g. to get password, pin number, etc.
  - To give information to users

    E.g. status information
- For **example**:
  - `CallbackHandler` implementation for a GUI application might display a window and request user for input
  - `CallbackHandler` implementation for a non-GUI might simply requests user to give input on the command line

JAAS CallbackHandler and Callbacks (Cont'd)

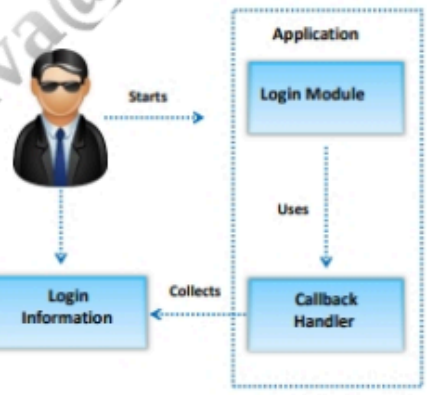
- CallbackHandler interface should implement the following method:

```
@Override
public void handle(Callback[] arg0) throws IOException, UnsupportedCallbackException
{
}
```

- To process **NameCallback**, the `CallbackHandler` initially retrieves the name from user by requesting the user and then calls the **setName** function of the `NameCallback` to store the name
- The login module passes an **array** of **Callbacks** to the `CallbackHandler` handle method to execute the requested user interaction and sets values to the Callbacks
- The package **javax.security.auth.callback** consists of the Callback interfaces and many other implementations
- Example of Callbacks include `NameCallback` (for user name) and `PasswordCallback` (for password)
- LoginModule sends an array of `Callbacks` directly to the `CallbackHandler's` handle method

Login to Standalone Application

- **Step 1:** The user **initiates** the application
- **Step 2:** **Login** module is invoked
- **Step 3 & 4:** The Login module makes use of the **callback handler** to collect login information from the user using callbacks
- **Step 5:** Login **information** entered by the user
- **Step 6:** Login module gets the information and decides whether to allow the user into the application or not



Login to Standalone Application

JAAS Client

- The code below given is an example of JAAS client where :
  - The **username**, **password**, and **URL** are read as input arguments from the command line
  - Connection to the URL is made, and the client is **authenticated** using the given username and password
  - A **privileged action** is executed under the newly implemented authenticated subject

MyClient.java

```
1  import javax.security.auth.Subject;
2  import javax.security.auth.login.LoginContext;
3
4  public class MyClient {
5  public static void main(String[] args) {
6      String user = args[0];
7      String pass = args[1];
8      String url = args[2];
9      LoginContext lc = null;
10
11     try {
12
13         lc=new LoginContext("Test", new TestCallBackHandler(user, pass,url));
14
15     } catch (Exception e) {
16         e.printStackTrace();
17     }
18     try {
19         lc.login();
20
21     } catch (Exception e) {
22         e.printStackTrace();
23         System.exit(-1);
24     }
25
26     Subject sub=lc.getSubject();
27     SimpleAction simpleAction = new SimpleAction(url);
28
29 }
30 }
```

LoginModule Implementation in JAAS

- The LoginModule interface should be implemented to perform authentication
- More than one login module can be used at a time and JASS logs in through each of them
- Depending upon various successful attempts, JAAS can be made to allow or deny logins

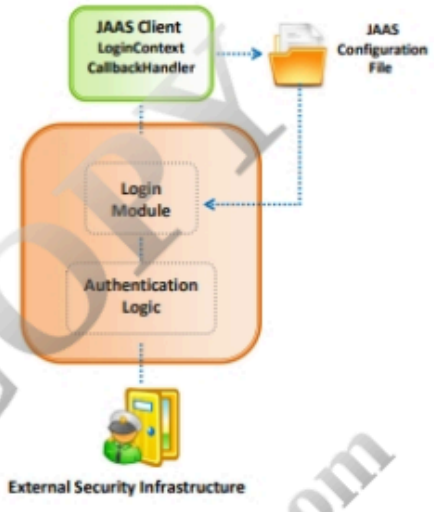
LoginModule implements 5 methods in its interface namely:

- `Initialize()`
- `Login()`
- `Commit()`
- `Abort()`
- `Logout()`

LoginModule Implementation in JAAS (Cont'd)

Implementation Methods

```
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

public class RdbmsLoginModule implements javax.security.auth.spi.LoginModule {

    @Override
    public boolean abort() throws LoginException {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean commit() throws LoginException {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public void initialize(Subject arg0, CallbackHandler arg1, Map arg2, Map arg3) {

    }
    @Override
    public boolean login() throws LoginException {

        return false;
    }

    @Override
    public boolean logout() throws LoginException {
        // TODO Auto-generated method stub
        return false;
    }
```



Methods Associated with LoginModule

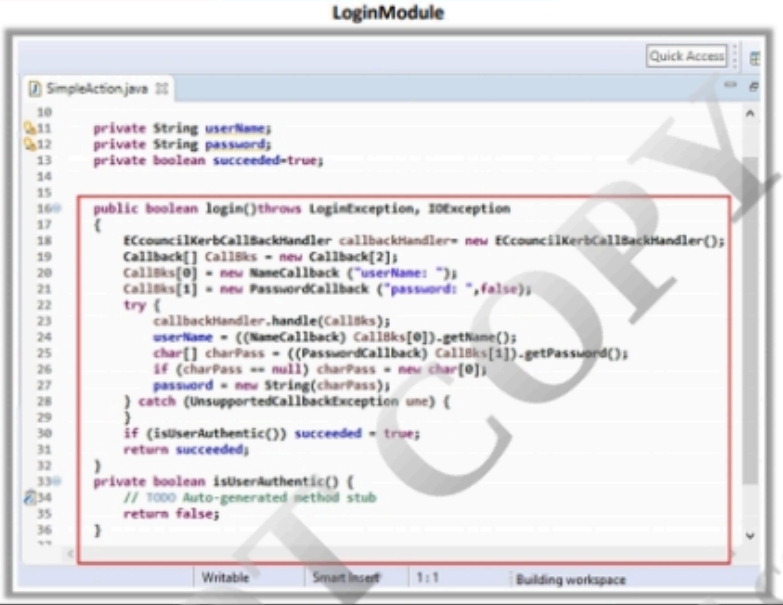
Method	Description
`initialize()`	Called once the LoginModule is created
`login()`	Implements authentication
`commit()`	Invoked by LoginContext once it has accepted the output from all LoginModules defined in the application Principals and credentials to the Subject are assigned here
`abort()`	Executed when any LoginModule for this application fails Principals or credentials are not assigned to the Subject
`logout()`	Removes all the Principals and credentials associated with the Subject

LoginModules in J2SE

Method	Description
`JndiLoginModule`	Confirms against a directory service configured under JNDI (Java Naming and Directory Interface)
`Krb5LoginModule`	Authenticates using Kerberos protocols
`NTLoginModule`	Authenticates using the current user's NT security information
`UnixLoginModule`	Authenticates using the current user's Unix security information

LoginModule Example

LoginModule



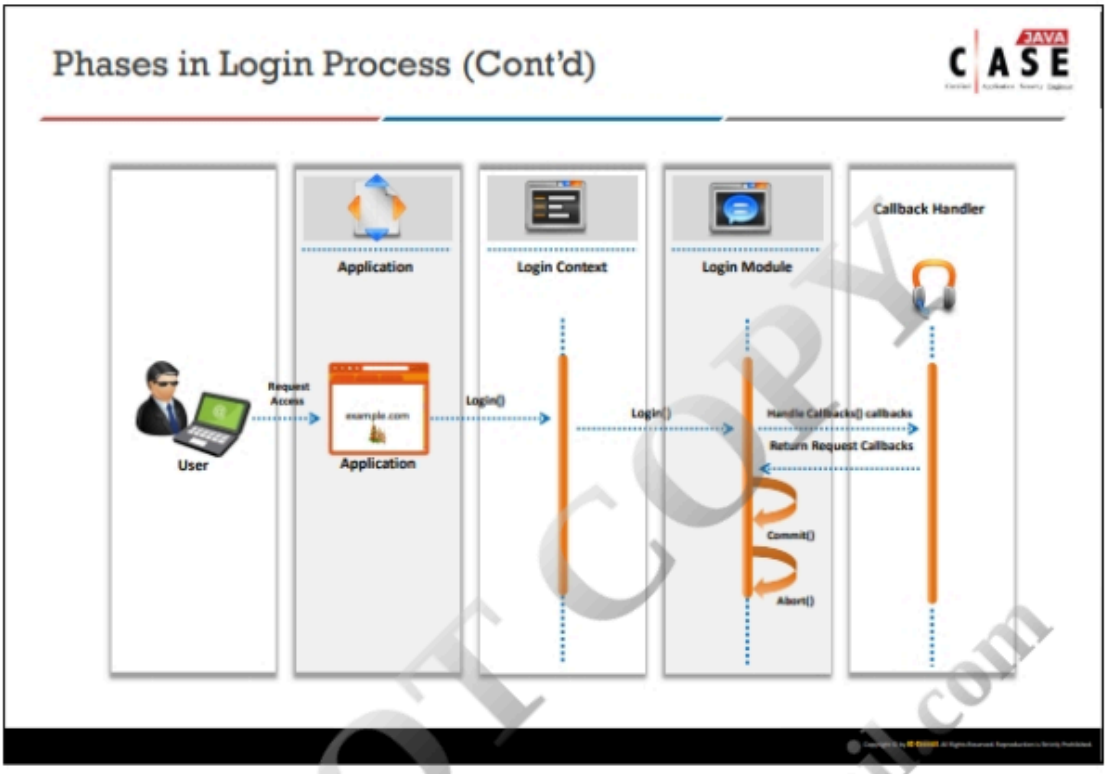
Phases in Login Process

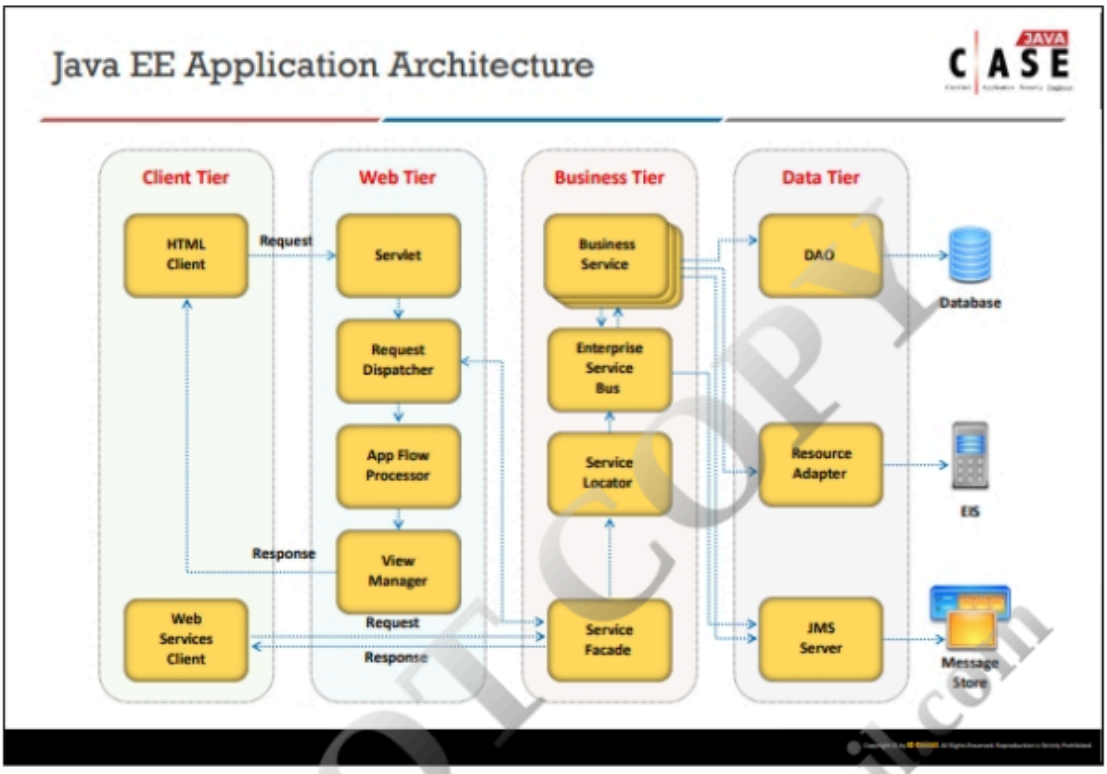
- The application calls the `login()` function of the LoginContext class that **authenticates** the Subject
- The **LoginContext** in turn calls the `login()` function of each login module in the authentication stack for application's policy configuration

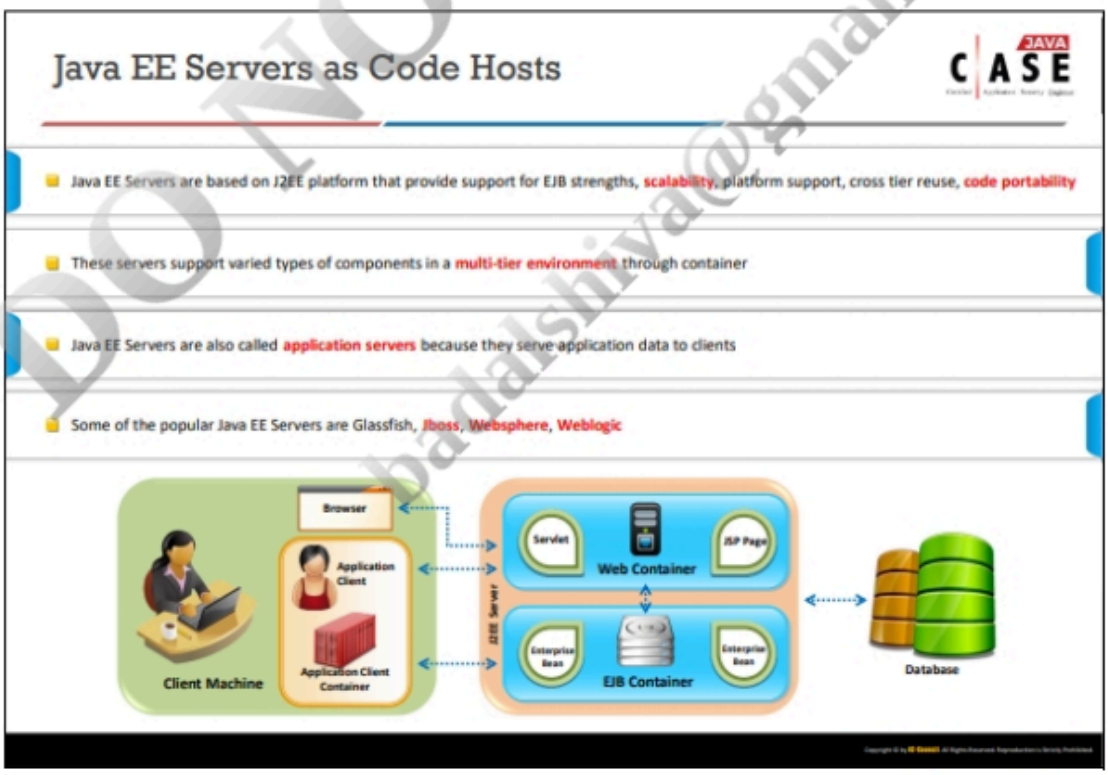
The **user authentication** is done by the login module in two phases

1. The `login()` method uses the CallbackHandler to get the authentication information from the user
2. The `commit()` method is executed if the user is successfully authenticated, and the abort() method is called if the authentication fails

Phases in Login Process (Cont'd)







Declaring Roles

Method permissions are defined using the following annotations:

@DeclareRoles

- This annotation is used to specify all the roles that are used by the application
- It also includes the roles that are not defined in the @RolesAllowed annotation
- The following syntax is used when declaring more than one role:
  - `@DeclareRoles({"Administrator", "Manager", "Employee"})`

@RolesAllowed("list-of-roles")

- This annotation specifies the security roles that enable access to methods in an application
- It can be defined on a class or on one or more methods

Calculate.java | DeclreRoles...

```
import org.springframework.stereotype.Controller;

@DeclreRoles("Office Admin")
@Controller
public class Calculate {
/*..
    ...
    ..
    ..*/
```

Calculate.java

```
import org.springframework.stereotype.Controller;

@DeclreRoles({"administrator","clerk"})
@Controller
public class Calculate {

@RolesAllowed("Administrator")
public void setNewRate(int rate)
{
```

Declaring Roles (Cont'd)

@DenyAll

- This annotation specifies that no security roles are allowed to run the specified method or methods
- These methods are eliminated from execution in the Java EE container

@PermitAll

- This annotation specifies that all security roles are allowed to run on the specified method or methods

Example Code for `@DenyAll` Annotation

Calculate.java

```
import org.springframework.stereotype.Controller;

@DeclreRoles({"administrator","clerk"})
@Controller
public class Calculate {

@RolesAllowed("Administrator")
public void setNewRate(int rate)
{

}
@DenyAll
public double convertCurrency()
{
```

Example: Code of `@PermitAll` Annotation

Calculate.java

```
import javax.annotation.security.DenyAll;
import javax.annotation.security.PermitAll;

import org.springframework.stereotype.Controller;

@DeclreRoles({"administrator","clerk"})
@Controller
public class Calculate {

@RolesAllowed("Administrator")
public void setNewRate(int rate)
{
}
@PermitAll
public double convertCurrency()
{
```

HTTP Authentication Schemes

The authentication schemes that are supported by HTTPClient

Basic

- The basic authentication is defined in **RFC 2617**
- It is not secure as it passes the credentials in clear **text form**
- The security of basic authentication scheme can be enhanced by combining it with **TLS/SSL encryption**
- The `getRequestingPrompt()` method returns the Basic authentication realm as given by the server

Digest

- Digest authentication scheme is defined in RFC 2617
- It is more **secure** than the Basic authentication and suited for applications that do not require security through TLS/SSL
- The `getRequestingPrompt()` method returns the Digest authentication realm as given by the server

NTLM

- NTLM authentication scheme is defined by **Microsoft** and is more secure than Basic and less secure than Digest authentication
- NTLM is used with **proxies** or **servers** and not both at the same time

HTTP Authentication Schemes (Cont'd)

HTTP Negotiate (SPNEGO)

- HTTP Negotiate enables any GSS authentication to be used as a HTTP authentication protocol
- At present, it supports only **NTLM** and **Kerberos**
- Kerberos 5 Configuration: When SPNEGO calls JGSS, it in turn calls the Kerberos V5 loginmodule to perform real work

The code used to configure Kerberos using Java system property **`java.security.krb5.conf.`** is

```
1: java -
   Djava.security.krb5.conf=krb5.conf
   \
2: -
   Djavax.security.auth.useSubjectCre
   dsOnly=false \
3: ClassName
```

Common Mistakes (Cont'd)

Methods to safeguard web applications against common mistakes that occur in authorization process are:

- **Principle of least privilege**
  - Ensure **development**, **testing** and demonstration source code and environments be run at least privileges
  - Avoid web application servers running at **privileged accounts** such as administrator, root, sysman, sa, etc.
  - User accounts should have enough privileges according to their **tasks**
  - Enable web applications' access to **database** through limited accounts only
  - Ensure database access through **parameterized stored procedures**
  - Evaluate and implement **code access permissions**
- **Protecting access to static resources on web server**
  - Do not let content save on web server in cases of **static content**
  - **Unauthorized access** to static content on the web should be prevented
  - Save sensitive files with **random names** and clean temporary files
- **Reauthorize users or code for accessing high value activities or after idle out**

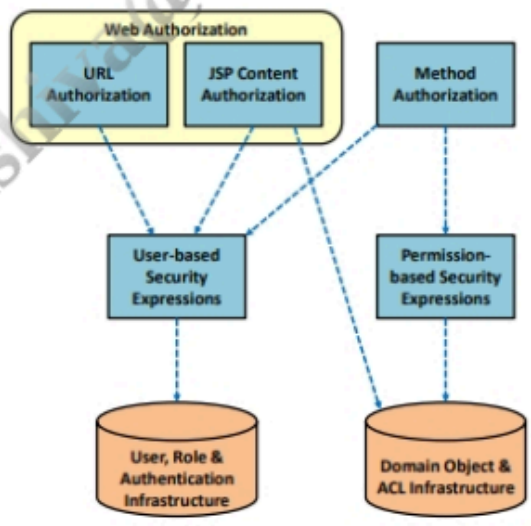
Common Mistakes (Cont'd)

- **Authorization matrix**
  - Ensure code uses built-in authorization framework or access check through **centralized authorization facilities**
- **Controlling access to protected resources**
  - Avoid using **custom authorized code**
  - Use only framework authorization and built-in platform facilities
- **Never implement client-side authorization tokens**
  - In case of a satisfactory authentication, associate session IDs with authorization **tokens**, **state** or **flags**
- **Time-based authorization should be implemented in sensitive web applications**
- **Be cautious of custom authorization controls**
  - Write **precise code**
  - In case of custom code that performs authorization, ensure fail safe authentication mechanisms and **exception handling** capability
  - Ensure authorization framework controls cover in totality to the application

Authentication and Authorization in Spring Security Framework

Spring Security Framework

- Spring Security is a lightweight, customizable framework for Authentication and Authorization
- Spring Security uses **javax.servlet.Filter** to implement Authentication and Authorization
- Servlet API and Spring Web MVC can implement Spring Security Framework



Printed by: badalshiva@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Spring Security Framework (Cont'd)

Spring Security Features

- Implement user authentication and authorization
- Provide login logout feature
- Provides role-based authorization control
- Provides link for database-based authentication and authorization
- Encrypted password support
- Supports form authentication
- Provides page-based user authentication and authorization

Authentication Types

- Http Basic Authentication
- Form-based Authentication

Authorization Types

- Web Authorization
- Method Authorization

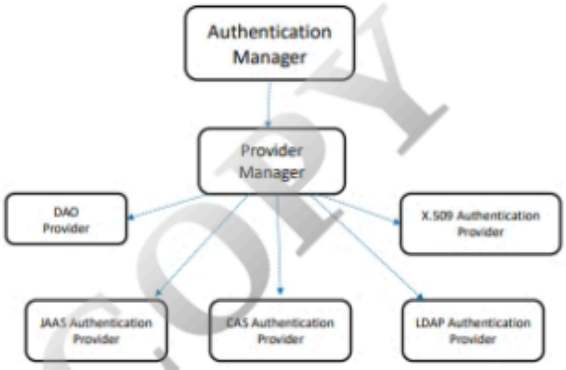
Spring Security Modules

Module	Description
Core (spring-security-core.jar)	Includes authentication and access-control API's
Web (spring-security-web.jar)	Includes servlet filters and web based authentication API's . Required for web application with Web authentication and URL Authorization
Remoting (spring-security-remoting.jar)	Includes Spring Remoting API's. Required for Remote Client Applications.
Config (spring-security-config.jar)	Includes API to enable using Spring Security XML namespace as XML configuration
LDAP (spring-security-ldap.jar)	Includes API's supporting LDAP (Light weight Directory Authentication)
ACL (spring-security-acl.jar)	Includes API's for implementing domain object security using access control lists(ACL's)
CAS (spring-security-cas.jar)	Includes API's for implementing Spring Security web authentication with a CAS single sign-on server.
OpenID (spring-security-openid.jar)	Includes API's for implementing authentication against an external OpenID server

Spring Authentication

Authentication Process

- Spring Security Supports a variety of pluggable Authentication Providers, e.g. , LDAP, X.509 (Certificates), Database (JDBC), JAAS, OAuth
- **AuthenticationManager** (Interface for processes all authentication requests) is the core for processing all authentication requests
- **AuthenticationProvider** (Interface for Authentication process) is used to define authentication provider
- AuthenticationProvider implements **UserDetailsService** (Service interface for fetching UserDetails object), which is responsible to get the User details
- The retrieved user details are **matched** with the provided username/ password while login



Authentication Manager and Provider Manager

Storing Username and Password

- **<authentication-provider> tags** links to **DaoAuthenticationProvider** and calls the **UserDetailsService** implementation

Storing User Details in Application Context

- When the number of users for the application are less, we can directly add the login and passwords in XML
- **org.springframework.security.core.userdetails.memory.InMemoryDaoImpl** bean is initialized for <user-service>

Storing User Details in Database

- When the number of users for the application are more, the login and passwords are stored in the database
- **org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl** class is used

Configuration to Store Login Details in Application Context

```
*ApplicationContext.xml
<sec:authentication-manager id="authManager">
<sec:authentication-provider><sec:user-service>
<sec:user name="admin" password="password" authorities="ROLE_ADMIN,ROLE_USER"/>
<sec:user name="user" password="password" authorities="ROLE_USER"/>
</sec:user-service></sec:authentication-provider>
</sec:authentication-manager>
```

Configuration to Store Login Details in Database

```
<sec:authentication-manager id="authManager">
  <sec:authentication-provider>
    <sec:jdbc-user-service data-source-ref="dataSource" />
  </sec:authentication-provider></sec:authentication-manager>
```

Storing Username and Password (Cont'd)

Storing User Details in Custom User Table

- To store and retrieve Login details from a custom table, configure the fetch queries, to be executed for getting the user credentials

Configuration to alter fetch query to get credentials from USER table with id, username, password columns and Role table which has username, role columns

ApplicationContext.xml

```
<sec:authentication-manager id="authenticationManager">
<sec:authentication-provider>
<!-- TBD <password-encoder hash="md5"/> --><sec:jdbc-user-service id="userDetailsService"
data-source-ref="dataSource"
users-by-username-query="SELECT username, password, true as enabled FROM USER WHERE username=?"
authorities-by-username-query= "SELECT user.username, role.role as authorities FROM ROLE role, USER
WHERE role.user_id=user.id and user.username=?"/>
</sec:authentication-provider>
</sec:authentication-manager>
```

Securing Authentication Provider

Encode Password using -SHA1

- While storing user details in **Application Context**, the password is configured as plain text
- Encode the password using **SHA1** using **<password-encoder>** element

Vulnerable Authentication Provider Configuration

ApplicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<sec:authentication-manager id="authManager">
<sec:authentication-provider>
<sec:user-service>
<sec:user name="admin" password="password" authorities="ROLE_ADMIN,ROLE_USER"/>
<sec:user name="user" password="password" authorities="ROLE_USER"/>
</sec:user-service></sec:authentication-provider>
</sec:authentication-manager>
```

Secure Authentication Provider Configuration

ApplicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<sec:authentication-manager id="authManager">
<sec:authentication-provider>
<password-encoder hash="sha" />
<sec:user-service>
<sec:user name="admin" password="password" authorities="ROLE_ADMIN,ROLE_USER"/>
<sec:user name="user" password="password" authorities="ROLE_USER"/>
</sec:user-service></sec:authentication-provider>
</sec:authentication-manager>
```

Implementing HTTP Basic Authentication

- Defining ‹**http-basic**› defines a **BasicAuthenticationFilter** filter
- On successfully authentication of the user, the Authentication object is added to Spring **SecurityContext**
- **SecurityContextHolder** class is used to access the security context

Configuring HTTP Basic Authentication

spring-bean.xml | root-contex... | spring-secu...

```
<http auto-config="true">
    <intercept-url pattern="/admin**" access="ROLE_ADMIN" />
    <logout logout-success-url="/admin" />
</http>
```

BasicAuthenticationFilter Bean Declaration

spring-bean.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

 <bean id="basicAuthenticationFilter"
class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
</bean>
<bean id="authenticationEntryPoint"
 class="org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint">
<property name="realmName" value="Name Of Your Realm"/>
</beans>
```

Form-based Authentication

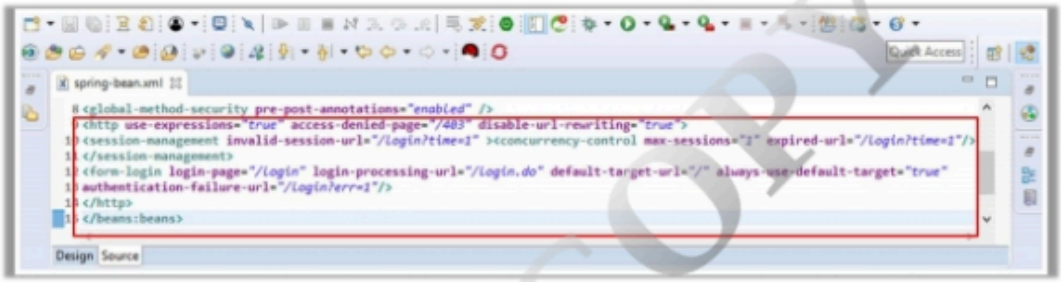
- Spring Security includes a **login form** for authentication by default

Configuring Form-based Authentication

- **global-method-security:** Will enable the **@PreAuthorize**, **@PostAuthorize** annotations tag
- **session-management:** Includes **SessionManagementFilter** to Spring Security's chain of filters
- **form-login:** Configure **Form-based Authentication**
  - **default-target-url:** Redirects to specified page on successful authentication
  - **authentication-failure-url:** Redirects to specified page when authentication fails
  - **always-use-default-target:** When set to true will redirect user to the specified page after login
  - **authentication-success-handler-ref:** Gets executed on successful authentication
  - **authentication-failure-handler-ref:** Gets executed on authentication fails
- **username-parameter:** Link the defined username property of **login.jsp** page
- **password-parameter:** Link the defined password property of **login.jsp** page

Form-based Authentication (Cont'd)

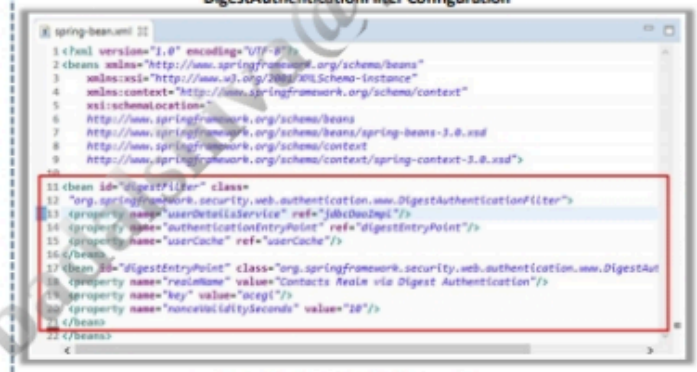
Sample Login Form Configuration



Implementing Digest Authentication

- Implementing Digest Authentication will encrypt the user password using hashing algorithms before sending it to the server
- MD5, SHA, BCrypt, SCrypt and PBKDF2WithHmacSHA1 are the hashing algorithms used for hashing password
- Digest contains **"nonce"**
  - Prevents plaintext attacks for retrieving plain text from cryptographic hash functions
  - It contains timestamps and prevents replay attacks
- **Example :** "nonce" value format generated by server
  - **expirationTime:** Expiration time in millisecond
  - **key:** Its private key to protect "nonce" changes

DigestAuthenticationFilter Configuration



Example : Unauthorized Response

```
HTTP/1.1 401 Unauthorized Server: Apache-Coyote/1.1 Set-Cookie:
JSESSIONID=CF0233C...; Path=/spring-security-mvc-digest-auth/;
HttpOnly WWW-Authenticate: Digest realm="Contacts Realm via
Digest Authentication", qop="auth",
nonce="MTM3MzYzODE2NTg3OTo3MmYxN2JkOWYxZTc4MzdmMzBiN2Q0YmY0ZTU0
N2RkZg==" Content-Type: text/html;charset=utf-8 Content-Length:
1061 Date: Fri, 12 Jul 2013 14:04:25 GMT
```

Security Expressions

- Spring security includes **Spring Expression Language** (SpEL) security expressions for authorization
- Security expressions are evaluated against a context-dependent "root object" called **SecurityExpressionRoot**
- In web context, the root object is called **WebSecurityExpressionRoot**, which is derived from SecurityExpressionRoot

Security Expressions (Cont'd)

User Security Expression Terms and Predicates

Term	Description
Authentication	Current user's Authentication object, derived from SecurityContext
Principal	Current user's principal object, derived from Authentication object

Expression	Description
hasRole([role])	User has the specified role then return "true"
permitAll	Always true
denyAll	Always false
isAnonymous()	Return true if user is anonymous
isRememberMe()	Return true if user is authenticated using Remember-Me option
isAuthenticated()	Return true if the user is not anonymous user
isFullyAuthenticated()	Return true if the user is not anonymous user and is not authenticated using Remember-Me option

WebSecurityExpressionRoot Terms

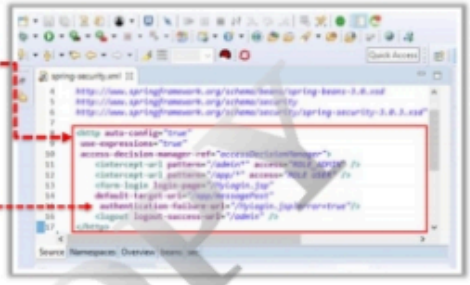
Term	Description
request	Denotes the HttpServletRequest
hasIpAddress(ipAddr)	Returns true if the client IP address same as the specified IP address

URL-based Authorization

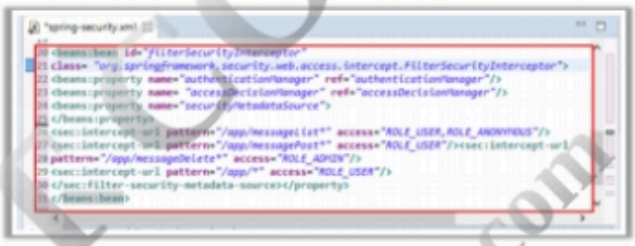
- To implement URL-based Authorization, use **<intercept-url>**
- <intercept-url> contains the following attributes
  - **Pattern**: Specifies the url patterns
  - **Access**: Contains the list of user roles that can access the url
  - **Method**: Optional parameter specifying a HTTP method for authorization
- The specified urls to be intercepted are sent as **metadata** to **FilterSecurityInterceptor**
- Ensure that the specified url-patterns in the <intercept-url> **ends with "*"**, otherwise an attacker can **pass parameters** to the Url to bypass the Authorization rule

Enable expression-based web URL authorization

Sample intercept-url configuration in XML



Sample intercept-url Configuration without XML



JSP Page Content Authorization

- Authorizing **JSP page** content based on the logged in users status, role, etc.
- Include the following to implement web URL authorization

Include Tag Library to Implement Web URL Authorization

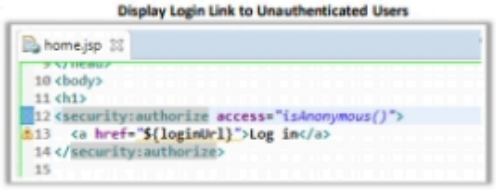
Tag Library Attributes List

Tag attributes	Description
access	Show tag content when specified Web security access expression is true
url	Show tag content when the user has access permissions for the specified url
Method	Limit authorization to the specified HTTP method (e.g., GET, POST, PUT, DELETE)

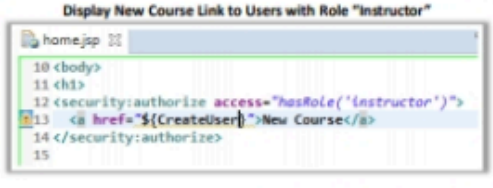
Tag Library Tags List

Term	Description
<security:authentication>	Fetch the current Authentication object to the JSP
<security:authorize>	Display or hide the tag content based on the current user matches the specified condition
<security:accesscontrollist>	Display or hides the tag content based on current user permission matches the specified domain object

Display Login Link to Unauthenticated Users



Display New Course Link to Users with Role "Instructor"



JSP Page Content Authorization with Domain Object's ACL

- Authorizing JSP page content based on the logged in users' permission on domain objects
- <security:accesscontrollist> tag displays or hides content of JSP page based on permissions of logged in user for a domain object's ACL
- Expressions are not used by <security:accesscontrollist> tag

Tag attributes	Description
hasPermission	Show or hide content when at least one of the specified permissions exist (list of numeric permissions)
domainObject	Domain object for checking the specified permission

Display Edit link on blog when the logged in user is Admin or had edit permission on the blog object

"home.jsp

```
10 <body>
11 <h1>
12 <security:accesscontrollist
13    hasPermission="2,16" domainObject="${blog}">
14    <a href="${editUrl}">Edit message</a>
15 </security:accesscontrollist>
```

Design | Preview

Method Authorization

- Spring security use security expression annotations to protect methods and classes
- Include the global-method-security namespace to implement @Pre/@Post annotations

XML Configuration for @Pre/@Post Annotations

spring-security.xml

```
<global-method-security pre-post-annotations="enabled">
</global-method-security>
```

JAVA Configuration for @Pre/@Post Annotations

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class JSecurityConfig extends
WebSecurityConfigurerAdapter {   ...}
```

@PreAuthorize

- @PreAuthorize ensures that the method is invoked only when the specified expression is true

@PostAuthorize

- @PostAuthorize ensures that the method returns value only when the specified expression is true

Only Guest User can Create guestlist

```
return "home";
}
@PreAuthorize("hasRole('ROLE_guest')") public void create(guestlist guest) {
}
```

Only Admin can read the Message

```
@PostAuthorize("hasRole('admin') or  hasPermission(FilterObject, admin)")
public Message getMessage(long id) {
    return msg;  }
```

- **Implementing Filter using @PreFilter and @PostFilter**
  - Filter collections and arrays based on the specified expression

Configuring Anonymous Login

- Spring Security creates by default an anonymous role, which can be used for providing anonymous login
- Using 'ROLE_ANONYMOUS' or 'IS_AUTHENTICATED_ANONYMOUSLY' gives anonymous users access to any page
- The default roles for anonymous users can be altered by overriding the default configuration

Configuring Custom Anonymous Role

spring-security.xml

```
<sec:intercept-url pattern="/login.jsp*" filters="none" />
<sec:intercept-url pattern="/*" access="ROLE_USER"/>
<!-- Defines a custom role in place of ROLE_ANONYMOUS.
ROLE_ANONYMOUS will no more work, use ROLE_GUEST instead of it-->
<sec:anonymous username="guest" granted-authority="ROLE_GUEST" />
</sec:http>
<p style="text-align: justify;">Here is the how the underlying filter can be defined
<sec:custom-filter position="ANONYMOUS_FILTER" ref="anonymousFilter" />
<bean id="anonymousFilter" class="org.springframework.security.web.authentication.Ano
    <property name="userAttribute" value="ROLE_GUEST" />
</bean>
```

Logout Feature Configuration

- Spring security provides **handler** to support logout feature
- **/j_spring_security_logout** is the default log-out url
- Define **logout-url** *attribute to* customize default logout url
- Define **logout-success-url** *to redirect to a page after logout*
- For conditional redirection to different pages after logout, implement **LogoutSuccessHandler** and link it to **‹logout›** element

Configuring Logout Feature

spring-security.xml

```
<http auto-config="true">
    <logout logout-success-url="/app/messageList" />
</http>
```

Source | Namespaces | Overview | beans | sec

Implementing LogoutSuccessHandler

spring-security.xml

```
<http auto-config="true">
    <logout logout-success-url="/app/messageList"
    success-handler-ref="customLogoutSuccessHandler"/>
</http>
```

Source | Namespaces | Overview | beans | sec

Define Logout Filter

spring-security.xml

```
<bean id="logoutFilter" class="org.springframework.security.web.authentication.logout.LogoutFilter">
<constructor-arg value="/pages/Security/logout.html" /> <constructor-arg> <list>
<bean class="org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"/>
</list> </constructor-arg> <property name="filterProcessesUrl" value="/j_logMeOut"/> </bean>
```

Remember-Me Authentication

- Spring security supports **Remember-Me feature**
- Once the Remember-Me feature is enabled, a **cookie base64(username + ":" + expirationTime + ":" + md5Hex(username + ":" + expirationTime + ":" password + ":" + key))** is sent to the browser after successful authentication at login
- Browser sends the received authentication **cookie** with every request to the server
- Spring security retrieves the **password** from the database for the logged in username
- Evaluates **md5Hex()** for the username, password along with expiration time and key and matches it with the supplied cookie
- If the evaluated values match with the supplied cookie, the user automatically gets logged in

Enabling Remember-Me Authentication

spring-security.xml

```
13  <http auto-config="true">
14      <logout logout-success-url="/app/messagelist"
15      success-handler-ref="customLogoutSuccessHandler"/>
16
17      <remember-me key="APPLICATIONKEY"/>
18  </http>
```

Integrating Spring Security with JAAS

- Spring Security contains a **package** to delegate authentication requests to JAAS
- Spring Security's authentication process takes the **username** and **password** provided by the user and stores it in the authentication object
- Each authentication object contains one **principle**

JAAS package components

- Authentication
- AuthenticationProvider
- LoginContext
- AuthorityGranter
- JaasAuthenticationToken

JAAS based Classes and Interfaces

- org.springframework.security.authentication.jaas
- AbstractJaasAuthenticationProvider
- AuthorityGranter
- DefaultJaasAuthenticationProvider
- DefaultLoginExceptionResolver
- JaasAuthenticationCallbackHandler
- JaasAuthenticationToken
- JaasGrantedAuthority
- JaasNameCallbackHandler
- LoginExceptionResolver
- SecurityContextLoginModule

Spring JAAS Implementation

Implement AuthorityGranter Interface

RoleGranter.java

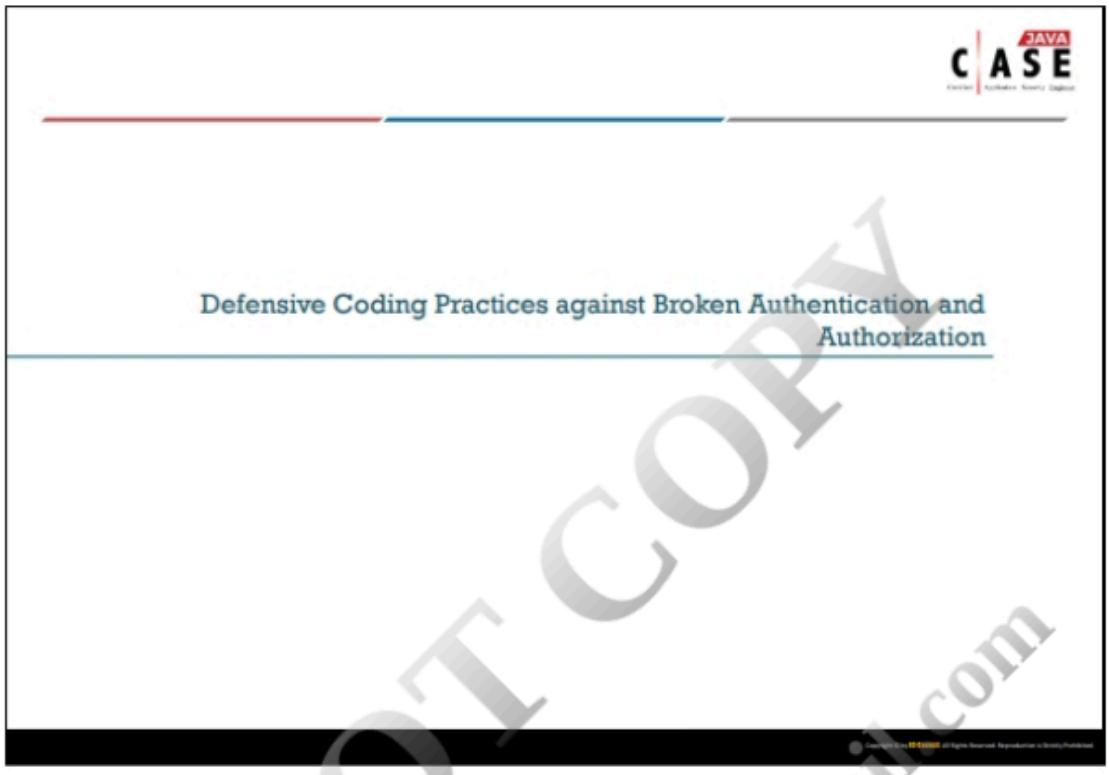
```
1  package com.myproject.webapp;
2
3  import java.security.Principal;
4  import java.util.Collections;
5  import java.util.Set;
6
7  import org.springframework.security.authentication.jaas.Auth
8
9  public class RoleGranter implements AuthorityGranter{
10
11     @Override
12     public Set<String> grant(Principal principal){
13         if (principal.getName().equals("admin"))
14         return Collections.singleton("ADMIN");
15         else
16         return Collections.singleton("ENDUSER");
17     }
18 }
```

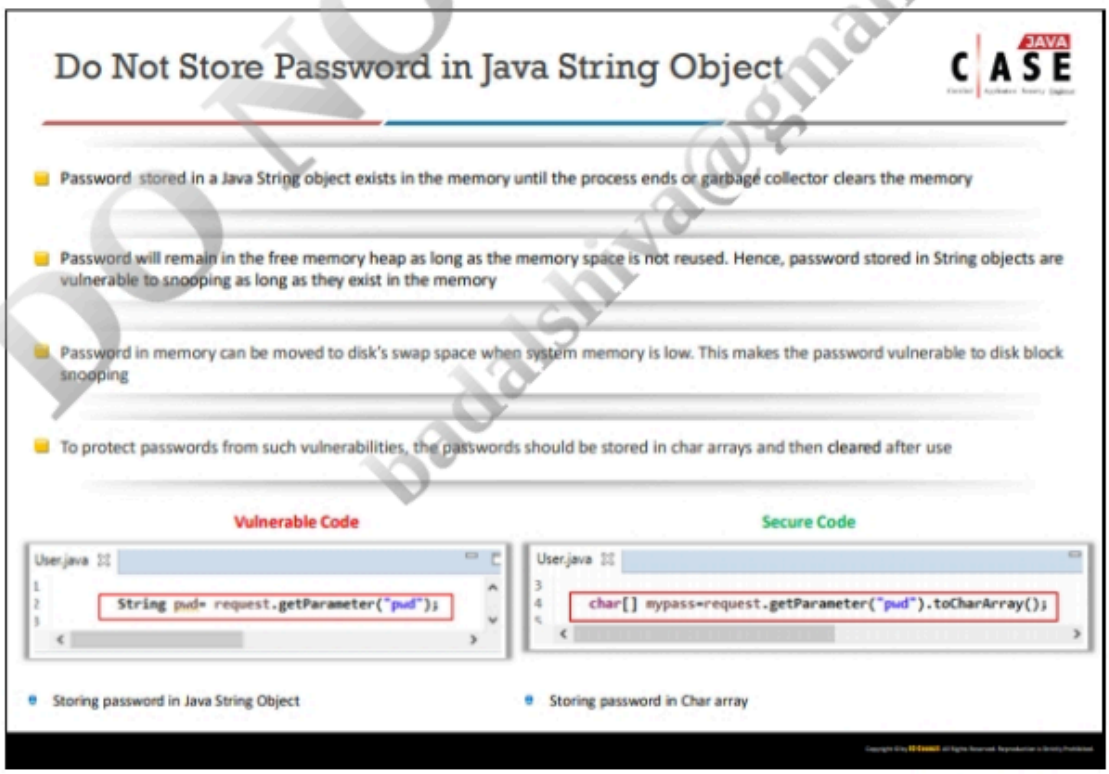
Spring JAAS Implementation (Cont'd)

Implementation of JaasGrantedAuthority and UserPrinciple

ItsSecureController.java

```
1  package jaaSimpl;
2
3  import java.nio.file.attribute.UserPrincipal;
4
5  import org.springframework.security.authentication.jaas.JaasGrantedAuthority;
6  import org.springframework.security.core.Authentication;
7  import org.springframework.security.core.context.SecurityContextHolder;
8  import org.springframework.ui.ModelMap;
9  import org.springframework.web.bind.annotation.RequestMapping;
10
11 public class ItsSecureController {
12
13    @RequestMapping(value="HomeUser/Home")
14    public String getUser(ModelMap model)
15    {
16    Authentication auth= SecurityContextHolder.getContext().getAuthentication();
17
18    JaasGrantedAuthority jaasGrantedAuthority=(JaasGrantedAuthority)(auth.getAuthorities());
19    UserPrincipal usrPrincipal =(UserPrincipal)jaasGrantedAuthority.getPrincipal();
20        model.addAttribute("userPrincipal",usrPrincipal);
21    return "HomeUser/Home";
22    }
```





Avoid Cookie Based Remember-Me Use Persistent Remember-Me

- Implementing **TokenBasedRememberMeServices** stores the MD5 hash of user password in the cookie
- Cookie-based Remember-Me functionality is vulnerable to cookie attacks
- Implementing **PersistentTokenBasedRememberMeServices** stores a unique persistent token in the database
- This token is regenerated and stored every time when a user logs in using persisted Remember-Me functions
- Hence, **PersistentTokenBasedRememberMeServices** approach prevents brute force attack

Vulnerable Code

spring-security.xml

```
<http auto-config="true">
    <logout logout-success-url="/app/messagelist"
    success-handler-ref="customLogoutSuccessHandler"/>
    <remember-me key="APPLICATIONKEY"/>
</http>
```

- Implementing **TokenBasedRememberMeServices** for Remember-Me functionality

Secure Code

spring-security.xml

```
<http auto-config="true">
    <logout logout-success-url="/app/messagelist"
    success-handler-ref="customLogoutSuccessHandler"/>
    <remember-me key="APPLICATIONKEY" data-source-ref="dataSource"
     token-validity-seconds="86400"/>
</http>
```

- Implementing **PersistentTokenBasedRememberMeServices** for Remember-Me functionality

Implement Appropriate Session Timeout

- Ensure that the **timeout** of session is not too long
- Session is properly **invalidate** or **terminated**
- Set appropriate session time out
- Use **Tomcat/config/web.xml** to set session time out at server level
- Use **$Tomcat/webapps/myproje/WEB-INF/web.xml** to set session time out at application level

Vulnerable Code

web.xml

```
</web-resource-collection>
</security-constraint>
<session-config>
<session-timeout>-1</session-timeout>
</session-config>
```

- Session time out is set to unlimited duration

Secure Code

web.xml

```
</web-resource-collection>
</security-constraint>
<session-config>
<session-timeout>15</session-timeout>
</session-config>
```

- Session time out is set to 15 mins

Implement Appropriate Session Timeout (Cont'd)

Secure Code : Setting Different Session Timeout for Different Users

web.xml | LoginServlet.java

```
{
    HttpSession session=req.getSession(true);

    if(req.isUserInRole("ADMIN"))
    {
        session.setMaxInactiveInterval(30*60);
    }else
    {
        session.setMaxInactiveInterval(15*60);
    }
```

Secure Code : Setting Session Timeout Programmatically

AppHttpSessionListener.java

```
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class AppHttpSessionListener implements HttpSessionListe

    @Override
    public void sessionCreated(HttpSessionEvent event) {

        event.getSession().setMaxInactiveInterval(15 * 60);

    }

    @Override
    public void sessionDestroyed(HttpSessionEvent event) {
            //destroy session
```

Note: `setMaxInactiveInterval` will set session time out separately for Admin and user roles

Prevent Session Stealing by Securing SessionID Cookie

- Session stealing happens when data is transferred over network as clear-text. It can be prevent by setting authentication on server
- Set AuthCookieEnabled="true" in the WebServer element in config.xml of WebLogic Server
- When AuthCookieEnabled is set to true, while authenticating using HTTPS, the server sends a new secure cookie, _WL_AUTHCOOKIE_JSESSIONID
- When session id uses default JSESSIONID, it can be stolen because it is not in encrypted format, but _WL_AUTHCOOKIE_JSESSIONID is in encrypted format over HTTPS

Setting AuthCookieEnabled in config.xml

```
<WebServer Name="myserver" AuthCookieEnabled="true"/>
```

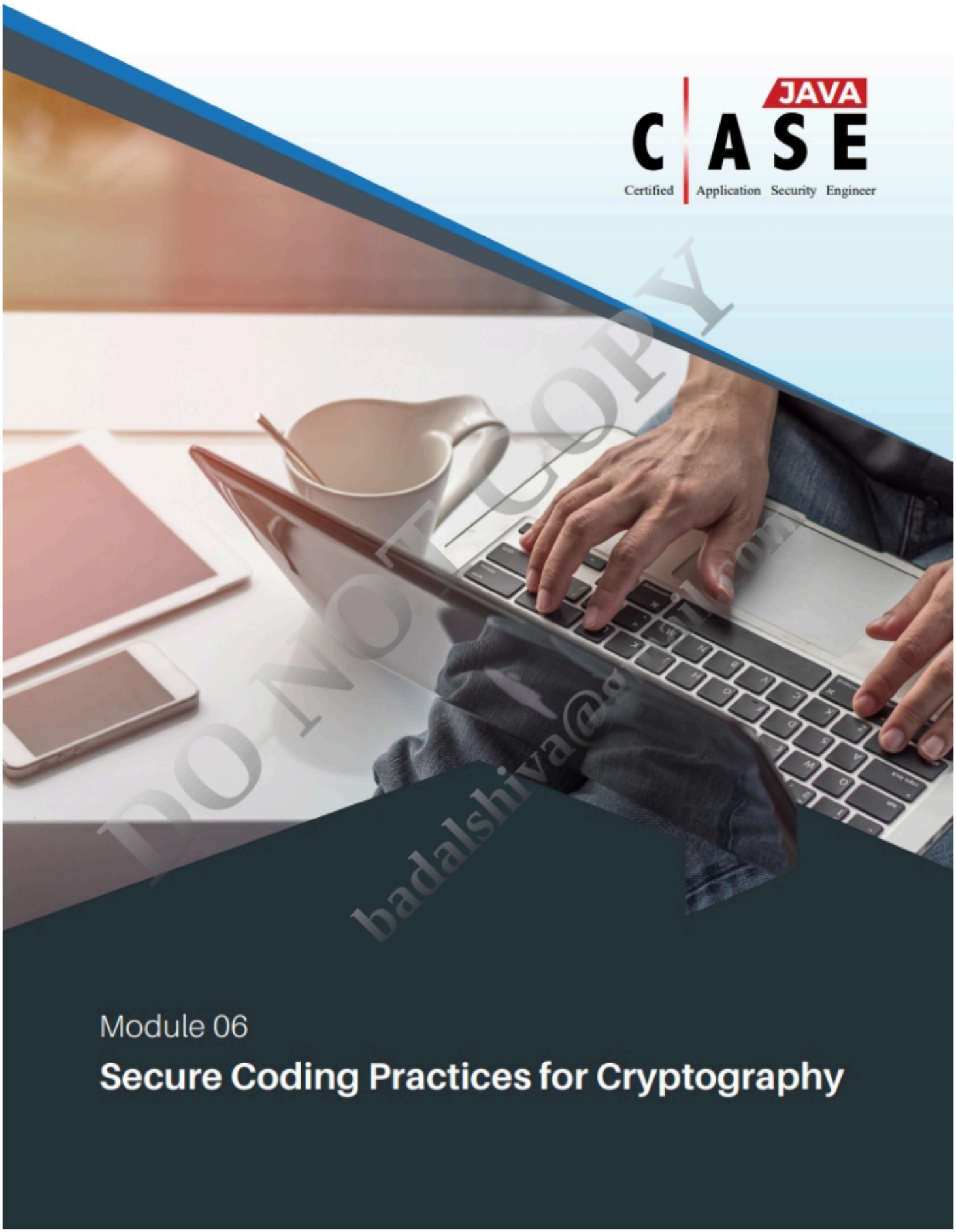
Note: By default, AuthCookieEnabled is set to true

Secure Development Checklists: Broken Authentication and Session Management

Check whether your **server** stores or validates **user passwords**
Do not transmit passwords over the **network** in **clear text** format
Check whether the security mechanism has provisions to **create, modify, delete**, and **validate** user passwords
Ensure that the client provides **authentication** information only after the **server** has provided its own authentication **credential** for **verification**
Check if the application ever reissues a password given to it by a **client** to a **third-party** application
Check if the application supports **Kerberos** and other **authentication** methods
Avoid allowing the application to support **unauthenticated** (guest) access
Verify if the application uses **Open Directory** for all **authentications**
Implement **built-In session management**
Ensure **login** page is **SSL-protected**
To prevent session fixation, **invalidate** the **HttpSession** before login. Take proper care to invalidate session, and clear all authentication related data on **Logout**
Set appropriate **session timeout** to ensure that sessions are not reused by attackers
Ensure proper implementation for **forgot password, reset password** and **Remember-Me** functionality as they can compromise security
Implement **multiple authentication** systems (like token authentication, OTP) along with Username password checking
Check the user **access privileges** before accessing resources
Validate the URL before accepting or submitting

Module Summary

- The main purpose of authentication is to verify a user's access to a protected segment of the web application
- Authentication is of two types: basic authentication and form-based authentication
- J2EE container services provide application tiers, and components with authentication and authorization facilities identifying service providers and callers
- Authentication in a web application is role-based, i.e., a user needs to be assigned a role in order to access the web application, e.g., customer, developer or a manager
- Authorization is the process that controls access rights of principals to system resources that include: Access to users, Access to processes, Access to machines
- Authorization shows "who is executing" "where is the code is residing" "who is the owner of the code"
- Access control comprises three models: discretionary access control (DAC), mandatory access control (MAC) and role-based access control (RBAC)



This page is intentionally left blank.

Module Objectives

1	Understand Fundamental Concepts and Need of Cryptography in Java
2	Discuss Encryption and Secret Keys
3	Discuss Implementation of Cipher Class
4	Discuss Implementation of Digital Signatures
5	Discuss Implementation of Secure Socket Layer (SSL)
6	Discuss Secure Key Management
7	Discuss Implementation of Digital Certificates
8	Discuss Implementation of Hashing
9	Explain Java Card Cryptography
10	Explain Crypto Module in Spring Security
11	Learn Dos and Don'ts in Java Cryptography

Java Cryptography

Need for Java Cryptography

- When a secret needs to be kept, you have to find a way to protect it from:
  - Misuse of data
  - Unauthorized access
  - Data theft
  - Deleting of data
- Protecting applications against attacks and maintaining privacy is a perplexing task for organizations due to **decentralized operations**, **diverse Internet users** and **distributed systems**
- Data collected through attacks against applications can be used for creating unauthorized profiles, deleting authorized profiles, misuse and manipulation of sensitive data, jeopardizing trade secrets and business transactions
- Due to threats to identity and privacy, IT developers are now focusing on providing security solutions that **enhance**, **support**, and **manage** associated risks with **data storage**, **access**, and **usage**
- Cryptography uses **ciphers** to encrypt data, **digital certificates** to authenticate trusted sites, and **digital signatures** to ensure that data is not compromised or tampered

Java Security with Cryptography

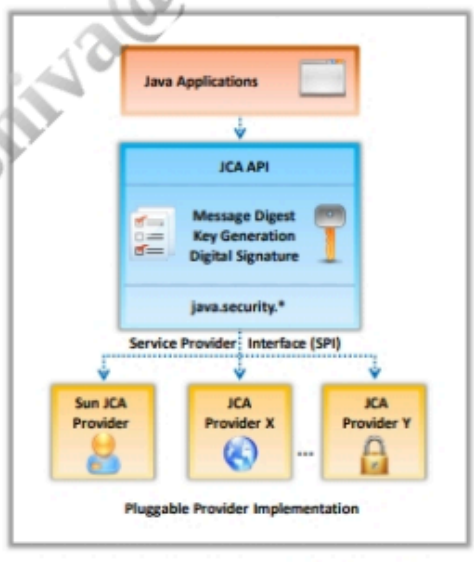
- Cryptography known as **secret writing**, can convert data into a scrambled code that is **encrypted**, **decrypted** and sent across a private or public network
- Cryptography deals with **security** issues in accordance with:
  - Privacy
  - Integrity
  - Authentication
  - Non-repudiation
- The Java platform, through APIs, addresses major security areas such as **cryptography**, authentication, **public key infrastructure**, secure communication and access control
- With these APIs, developers can easily design and write application code by integrating both **low-level** and **high-level security functions**
- Java platform offers cryptographic operations using **APIs** such as
  - JCA (Java Cryptography Architecture)
  - JCE (Java Cryptography Extension)

Java Security with Cryptography (Cont'd)

- **Design principles for JCA and JCE**
  1. Algorithm Independence
  2. Algorithm Extensibility
  3. Implementation Independence
  4. Implementation Interoperability

Java Cryptography Architecture (JCA)

- JCA provides an **architectural framework** for executing the main cryptographic services in the Java platform
- Java cryptographic operations are realized through **engines classes** that abstract with a particular cryptographic concept
- JCA classes are located in **java.security package**
- JCA supports many of the standard **algorithms** such as RC4, RC2, PKCS#5, SHA, Triple DES, AES, DSA, RSA, etc.



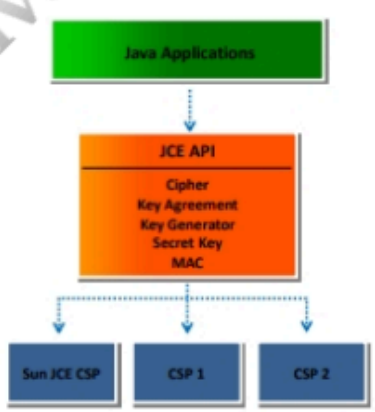
Java Cryptography Architecture (JCA) (Cont'd)

Java Cryptography Architecture Engine Includes

- Message Digests (create hash value)
- Digital Signatures (create signatures)
- Key Pair Generator (creates a pair of keys)
- Key Factories (break down a key)
- Key Store (manages and stores a key)
- AlgorithmParameters (encoding and decoding)
- AlgorithmParametersGenerator (generating parameters)
- CertificateFactory (generates public key certificates)
- CertPathBuilder (creates relationship chains between certs)
- CertStore (manages and stores certs)

Java Cryptography Extension (JCE)

- JCE includes classes for encryption and decryption, used for ciphering and deciphering functionality
- JCE engines include
  - Ciphers (manage encryption/decryption)
  - KeyGenerator (creates secret keys used by ciphers)
  - SecretKeyfactory (operates on SecretKey objects)
  - Key agreement (key agreement protocol)
  - MAC (message authentication code functionality)
- JCE engines are located in javax.crypto package
- SunJCE provider supports algorithms such as DES, DESede, Triple-DES, Blowfish, RSA, PBEWithMD5AndDES, HmacMD5, HmacSHA1, etc.



Encryption and Secret Keys

Attack Scenario: Inadequate/Weak Encryption

1. The data transmitted across the network can be intentionally or unintentionally **modified** or **manipulated** hence it cannot be considered as secure
2. Inadequate or weak encryption can lead to data breaches where an **intruder** can exploit prevailing **vulnerabilities** between server and client communications
3. In weak encryption based attacks, an attacker can **decrypt sensitive data** using brute force attacks
4. Weak encryption keys can allow attackers to perform reverse engineering, man-in-the-middle attacks, spoof content, phishing attacks

Encryption: Symmetric and Asymmetric Key

- Encryption is a method of transforming plain text into something **unintelligible (cipher) text** in order to protect it from data compromise
- Symmetric encryption (secret-key, shared-key, and private-key) uses the same key for **encryption** as it does for **decryption**
- Asymmetric encryption (public-key) uses different encryption keys for encryption and decryption. These keys are known as **public** and **private keys**
- In Java, a cipher object is created in the process of encryption and decryption with a specific algorithm such as **DES** for symmetric and **RSA** for asymmetric encryption
- The **javax.crypto** package provides a framework for symmetric and asymmetric encryption with cipher implementations

Encryption/Decryption Implementation Methods

- **Implementation methods for encryption and decryption**

1. Symmetric Key Generation

```
import java.security.NoSuchAlgorithmException;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
public class Basic {
private SecretKey generateKey ()
    throws NoSuchAlgorithmException {
KeyGenerator key=KeyGenerator.getInstance("DESede");
```

2. Encryption

```
private byte[] encrypt(byte[] message) throws
InvalidKeyException,
IllegalBlockSizeException,
BadPaddingException,
NoSuchAlgorithmException,
NoSuchPaddingException {
    Key aesKey = new SecretKeySpec(message, "AES");
Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, aesKey);
    return cipher.doFinal(message);
```

3. Decryption

```
byte[] encrypt(byte[] message) throws
InvalidKeyException,
IllegalBlockSizeException,
BadPaddingException,
NoSuchAlgorithmException,
NoSuchPaddingException {
    Key aesKey = new SecretKeySpec(message, "AES");
Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, aesKey);
    return cipher.doFinal(message);
```

SecretKeys and KeyGenerator

- SecretKeys are used in **symmetric encryption algorithms** and use the same keys for the encryption and decryption process
- KeyGenerator class creates the SecretKeys objects used by **cipher** for encryption and decryption
- KeyGenerator class (`javax.crypto.KeyGenerator`); is used to create **SecretKeys**
  - `SecretKey myKey = keyGen.generateKey();`

Implementation Methods of KeyGenerator Class

```
public abstract class KeyGeneratorSpi
```

- KeyGenerator class abstracts with service provider interface class

```
protected abstract SecretKey engineGenerateKey()
```

- Generates secret keys using installed random number generator and installed algorithm parameter specification

```
protected abstract void engineInit(SecureRandom secRan)
```

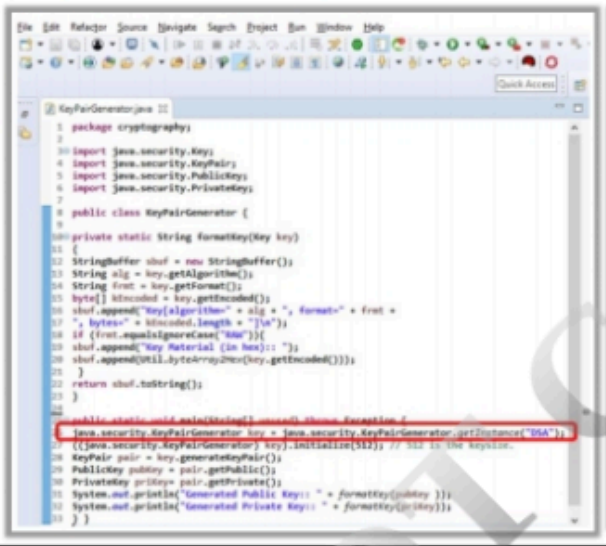
- Initializes key generation engine with its generated random number

```
protected abstract void engineInit(AlgorithmParameterSpecaps,
  SecureRandomsecRan)
```

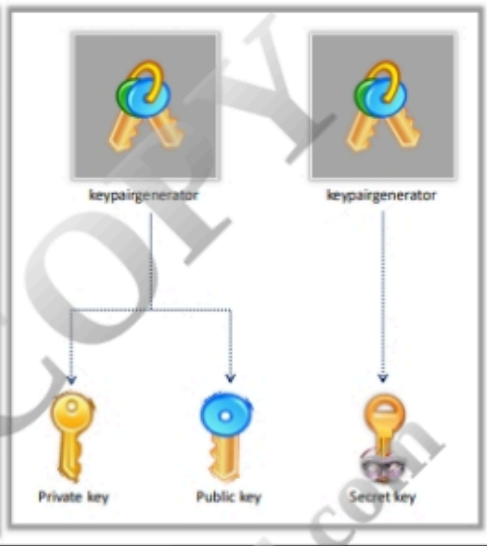
```
public final void engineInit(int strength, SecureRandom secRan)
```

Creating SecretKey with KeyGenerator Class

- In the code, **KeyPairGenerator** class is used for creating public and private keys

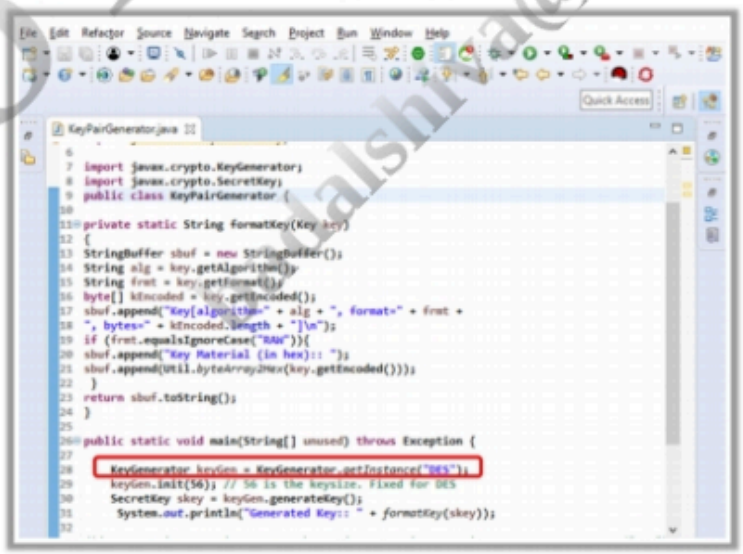


```
package cryptography;

import java.security.Key;
import java.security.KeyPair;
import java.security.PublicKey;
import java.security.PrivateKey;

public class KeyPairGenerator {

private static String formatKey(Key key)
{
StringBuffer sbuf = new StringBuffer();
String alg = key.getAlgorithm();
String frmt = key.getFormat();
byte[] kEncoded = key.getEncoded();
sbuf.append("Key[algorithm=" + alg + ", format=" + frmt +
", bytes=" + kEncoded.length + "]\n");
if (frmt.equalsIgnoreCase("RAW")){
sbuf.append("Key Material (in hex):: ");
sbuf.append(Util.byteArray2Hex(key.getEncoded()));
 }
return sbuf.toString();
}

public static void main(String[] unused) throws Exception {
  java.security.KeyPairGenerator key = java.security.KeyPairGenerator.getInstance("DSA");
((java.security.KeyPairGenerator) key).initialize(512); // 512 is the keysize.
KeyPair pair = key.generateKeyPair();
PublicKey pubKey = pair.getPublic();
PrivateKey priKey= pair.getPrivate();
System.out.println("Generated Public Key:: " + formatKey(pubKey ));
System.out.println("Generated Private Key:: " + formatKey(priKey));
} }
```



Creating SecretKey with KeyGenerator Class (Cont'd)

- In the code, **KeyGenerator** class is used for creating secret keys



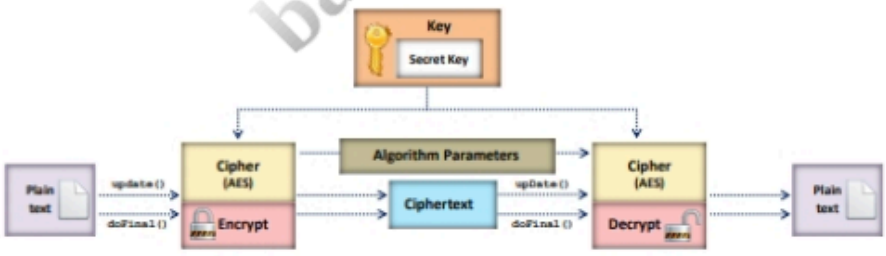
```
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
public class KeyPairGenerator {

private static String formatKey(Key key)
{
StringBuffer sbuf = new StringBuffer();
String alg = key.getAlgorithm();
String frmt = key.getFormat();
byte[] kEncoded = key.getEncoded();
sbuf.append("Key[algorithm=" + alg + ", format=" + frmt +
", bytes=" + kEncoded.length + "]\n");
if (frmt.equalsIgnoreCase("RAW")){
sbuf.append("Key Material (in hex):: ");
sbuf.append(Util.byteArray2Hex(key.getEncoded()));
 }
return sbuf.toString();
}

public static void main(String[] unused) throws Exception {

  KeyGenerator keyGen = KeyGenerator.getInstance("DES");
    keyGen.init(56); // 56 is the keysize. Fixed for DES
    SecretKey skey = keyGen.generateKey();
     System.out.println("Generated Key:: " + formatKey(skey));
```

Cipher Class

The Cipher Class

- The Cipher class in `javax.crypto` package performs the main functionality of cryptographic cipher using encryption and decryption
- It is mostly implemented by programmers for secure data transfers by encrypting data from the sending end and decrypting data from the receiving end in order to safeguard data from intrusions while in transmission

Methods	Description
`getAlgorithm()`	Returns the name of the algorithm used by the cipher object
`getIV()`	Retrieves the initialization vector (IV) in a new buffer class
`getParameters()`	Retrieves the parameters used by the cipher
`doFinal()`	Performs encryption and decryption process as initialized by the cipher
`getBlockSize()`	Retrieves block size (bytes)



Implementation Methods of Cipher Class

Description	Methods
Creating a cipher object	`public static Cipher getInstance(String transformation)` `public static Cipher getInstance(String transformation, String provider)`

Initializing a cipher object	`public void init(int opmode, Key key);`
	`public void init(int opmode, Certificate certificate)`
	`public void init(int opmode, Key key, SecureRandom random);`
	`public void init(int opmode, Certificate certificate, SecureRandom random`

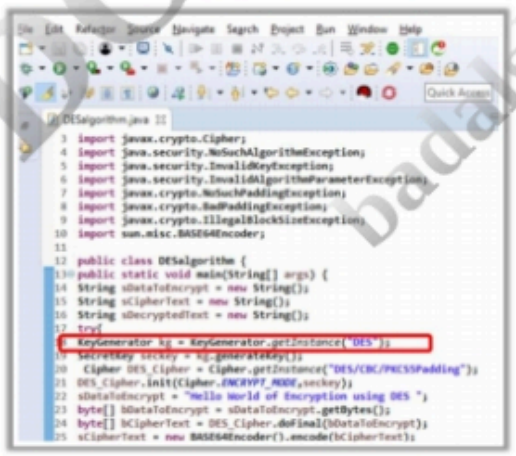
Encrypting and Decrypting data	`public byte[] doFinal(byte[] input);` `public byte[] update(byte[] input);`
Wrapping and Unwrapping Keys	`public final byte[] wrap(Key key);` `public final Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType));`

Managing Algorithm Parameters	`Init method`
	`Getparameters method`
	`getIV method`

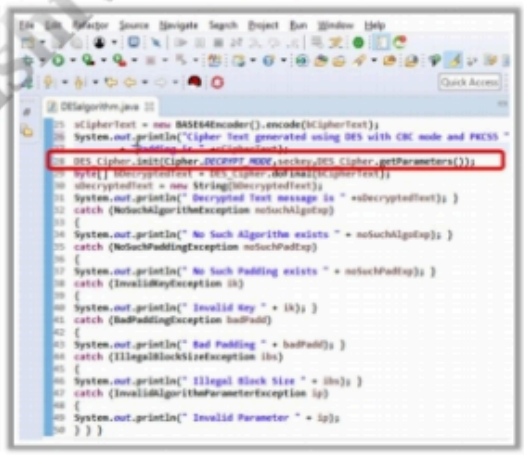
Cipher Output Considerations	`public int getOutputSize(int inputLen)`

Insecure Code for Cipher Class using DES Algorithm

- The code uses the DES algorithm for creating Cipher class, although DES is a standard algorithm often used for constructing Cipher class, it is considered weak and prone to attacks



```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
import java.security.InvalidAlgorithmParameterException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import sun.misc.BASE64Encoder;

public class DESalgorithm {
public static void main(String[] args) {
String sDataToEncrypt = new String();
String sCipherText = new String();
String sDecryptedText = new String();
try{
KeyGenerator kg = KeyGenerator.getInstance("DES");
SecretKey seckey = kg.generateKey();
Cipher DES_Cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
DES_Cipher.init(Cipher.ENCRYPT_MODE,seckey);
sDataToEncrypt = "Hello World of Encryption using DES ";
byte[] bDataToEncrypt = sDataToEncrypt.getBytes();
byte[] bCipherText = DES_Cipher.doFinal(bDataToEncrypt);
sCipherText = new BASE64Encoder().encode(bCipherText);
```

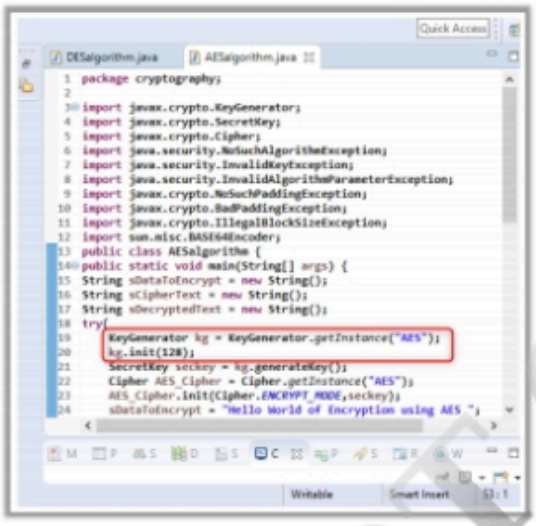


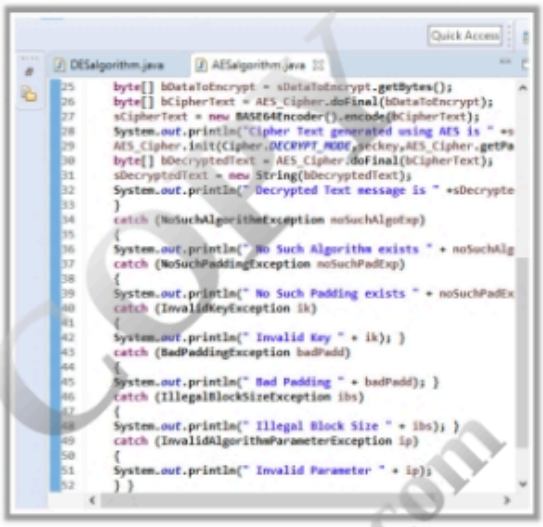
```
sCipherText = new BASE64Encoder().encode(bCipherText));
System.out.println("Cipher Text generated using DES with CBC mode and PKCS5 "
DES_Cipher.init(Cipher.DECRYPT_MODE,seckey,DES_Cipher.getParameters()));
byte[] bDecryptedText = DES_Cipher.doFinal(bCipherText);
sDecryptedText = new String(bDecryptedText);
System.out.println(" Decrypted Text message is " +sDecryptedText); }
catch (NoSuchAlgorithmException noSuchAlgoExp)
{
System.out.println(" No Such Algorithm exists " + noSuchAlgoExp); }
catch (NoSuchPaddingException noSuchPadExp)
{
System.out.println(" No Such Padding exists " + noSuchPadExp); }
catch (InvalidKeyException ik)
{
System.out.println(" Invalid Key " + ik); }
catch (BadPaddingException badPadd)
{
System.out.println(" Bad Padding " + badPadd); }
catch (IllegalBlockSizeException ibs)
{
System.out.println(" Illegal Block Size " + ibs); }
catch (InvalidAlgorithmParameterException ip)
{
System.out.println(" Invalid Parameter " + ip);
} } }
```

Secure Code for Cipher Class using AES Algorithm

JAVA CASE

- The code uses the **AES** algorithm (considered as a strong algorithm according to OWASP) for creating **Cipher class** with **strong key size (128)**



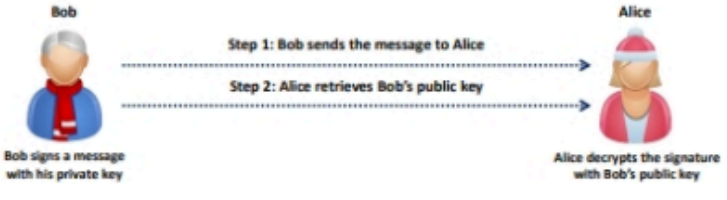


Attack Scenario: Man-in-the-Middle Attack

- Man-in-the-Middle (MITM) attacks allow an attacker to access sensitive information by **intercepting** and **altering communications** between an end-user and webservers
- For any data that travels across the network, there is a possibility of manipulation or modification of data by intruders
- If the data is not adequately encrypted using digital signatures or digital certificates then there is a possibility of **data leakage**
- The information that travels may contain **personal data**, credit card numbers and passwords

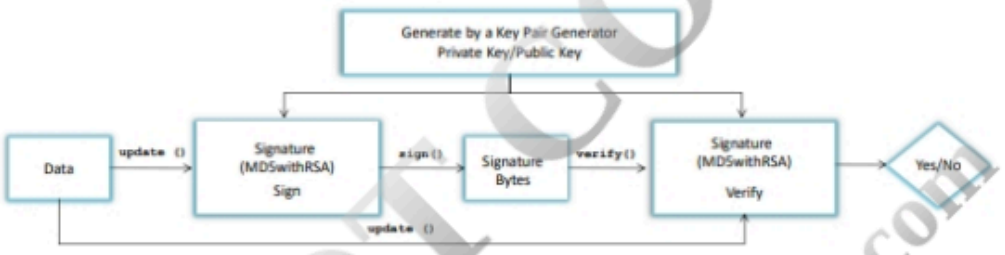
Digital Signatures

1. Digital signatures are used for protecting and maintaining the **integrity** of **digital data**, downloads, the source of the message and a software license file
2. JCA provides a specific framework for digital signatures with **java.security.Signature class** that provides the functionality of signing and verifying digital signatures
3. The Signature class and SignedObject class provide the basic functionality of **digitally signing** and **verifying** digital signatures in JCA



The Signature Class

- The Signature class is designed to provide applications for the functionality of a cryptographic digital signature algorithm such as DSA or MD5withRSA
- The Signature class can be used for checking whether a signature is in fact authentic
- Signature class methods include `getInstance( )`, `initSign( )`,`initVerify( )`,`update( )`, `sign( )`,and `verify( )`
- Only a private or public key owner can create a signature
- Public key for the desired private key is used to generate signature
- Public key and signature cannot reveal any details about private key

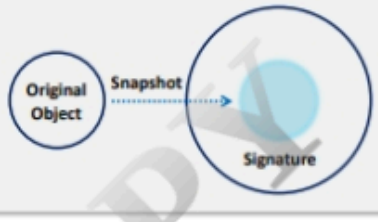


Implementation Methods of Signature Class

Methods	Description
`Signature(String algorithm)`	Generates a signature object with a specific algorithm
`getAlgorithm()`	Returns the Signature object algorithm
`getInstance(String algorithm)`	Generates a signature object with a specific digest algorithm
`Getparameters()`	Returns the signature object's parameters
`sign()`	Retrieves signature bytes of data
`initSign(PrivateKey privateKey)`	Initializes the object for signing
`initVerify(Certificate certificate)`	Initializes the object for verification using a certificate
`initVerify(PublicKey publicKey)`	Initializes the object for verification
`update(byte b)`	Updates the data to be signed or verified by a byte

The SignedObjects

- Java provides developers with object level protection mechanisms, protecting the **integrity** and **confidentiality** of objects inside a runtime environment (memory) or that are in **transit** (stored in IP packets or saved on disk)
- **SignedObject** class is used for creating **authentic runtime objects** whose integrity cannot be compromised without being detected



- **SignedObject**

  (`java.security.SignedObject`):
  - Creates another serializable object (a copy of **serializable object**)
  - Creates in **serialized form**
  - Creates a signature using algorithms (**DSA**, **SHA-1**, **MD5**)
  - If the signature is not null then the signed object may contain a valid **digital signature**

- **Signedobjects are used as:**
  - Unforgetable **authorization tokens**
  - Sign and serialize **objects/data**
  - Verify authenticity of **digital signatures**
  - Construct a logical sequence of **signatures**

Implementing Methods of SignedObjects

Signing an object

```
public void test() throws NoSuchAlgorithmException,
NoSuchProviderException, InvalidKeyException, SignatureException, IOException
{
Signature signingEngine = Signature.getInstance(algorithm, provider);
SignedObject sobj = new SignedObject(myobject, signingKey, signingEngine);
}
```

Verifying a SignedObject

```
Signature verificationEngine = Signature.getInstance(algorithm, provider);
try {
if (sobj.verify(publickey, verificationEngine))
{
  Object myob = sobj.getObject();
}
catch (java.lang.ClassNotFoundException e) {
```

- **myobject:** The serialized object that needs to be signed
- **signingKey:** A private key such as DSAPrivateKey, RSAPrivateKey, RSAPrivateCrtKey
- **signingEngine:** Signature Algorithms can be SHA1withDSA, AES, etc.
- **publickey:** Key used for verification
- **verificationEngine:** Signature verification engine

Methods	Description
`getAlgorithm()`	Gets the name of the algorithm used to seal an object
`getObject()`	Gets the encapsulate object
`getSignature()`	Gets the signature of the signed object
`verify(PublicKey verificationKey, Signature verificationEngine)`	Verifies if a signedobject contains a valid signature

The SealedObjects

- Sealedobject class helps developers in creating an object and protecting its confidentiality using **cryptographic algorithms**
- Sealedobject (`javax.crypto.SealedObject`)
  - Aims at protecting object confidentiality
  - Creates serializable object
  - Creates cryptographic algorithm (DES, IDEA, RC4)
  - Encryption and decryption
  - Deserialization of original object
- The proper way of maintaining the integrity and confidentiality of **Java objects** that travel across network is to:
  - First, create **signedobject**
  - Second, create **sealedobject**

Implementation Methods of SealedObject

Encryption

Signed.java

```
KeyGenerator k = KeyGenerator.getInstance ("DES");
SecretKey desKey = k.generateKey();
Cipher c = Cipher.getInstance ("DES");
c.init(Cipher.ENCRYPT_MODE, deSKEY);
String str = new String ("Good Morning");
SealedObject Sobj = new SealedObject (str, c);
```

Decryption

Signed.java

```
KeyGenerator k = KeyGenerator.getInstance ("DES");
SecretKey desKey = k.generateKey();
Cipher c = Cipher.getInstance ("DES");
c.init(Cipher.DECRYPT_MODE, deSKEY);
String str = new String ("Good Morning");
SealedObject Sobj = new SealedObject (str, c);
```

Methods	Description
`getAlgorithm()`	Returns the name of the algorithm used to seal an object
`getObject(Cipher c)`	Retrieves the original (encapsulate) object
`getObject(Key key)`	Retrieves the original (encapsulate) object

Insecure and Secure Code for Signed/Sealed Objects

Vulnerable Code

- In the given code, sealing is performed before signing and that questions the integrity of the signer's true identity


```
import java.security.Signature;
import java.security.SignedObject;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SealedObject;

public class SecureSigned {
    public static void main(String[] args) throws IOException,
    GeneralSecurityException, ClassNotFoundException {
    SerializableMap<String, Integer> map = buildMap();  // Build map
    KeyGenerator KeyGen;    // Generate sealing key & seal map
    KeyGen = KeyGenerator.getInstance("AES");
    KeyGen.init(new SecureRandom());
    Key k = KeyGen.generateKey();
    Cipher cipher1 = Cipher.getInstance("AES");
    cipher1.init(Cipher.ENCRYPT_MODE, k);
    SealedObject sealedMap = new SealedObject(map, cipher1);        // Generate signing p
    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DES");
    KeyPair kypair = keyPairGen.generateKeyPair();
    Signature sign = Signature.getInstance("SHA1withDSA");
    SignedObject signedMap = new SignedObject(sealedMap, kypair.getPrivate(), sign);
    ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data")); // Seri
    out.writeObject(signedMap); out.close();
    ObjectInputStream ins = new ObjectInputStream(new FileInputStream("data"));  // Deseri
    signedMap = (SignedObject) ins.readObject(); ins.close();
    if (!signedMap.verify(kypair.getPublic(), sign)) {  // Verify signature and retrieve
        throw new GeneralSecurityException("Map failed verification"); }
        sealedMap = (SealedObject) signedMap.getObject();
        cipher1 = Cipher.getInstance("AES"); // Unseal map
        cipher1.init(Cipher.DECRYPT_MODE, k);
        map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher1);
        InspectMap(map); // Inspect map
        }
```


Insecure and Secure Code for Signed/Sealed Objects (Cont'd)

Secure Code

- In the given code, signing is performed before sealing the object and that guarantees the authenticity of the object


```
import java.security.SecureRandom;
import java.security.Signature;
import java.security.SignedObject;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SealedObject;

public class SecureSigned {
    public static void main(String[] args) throws IOException,
    GeneralSecurityException, ClassNotFoundException {
    SerializableMap<String, Integer> map = buildMap();  // Build map
    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("DSA");
    KeyPair kypair = keyPairGen.generateKeyPair();
    Signature sign = Signature.getInstance("SHA1withDSA");
    SignedObject signedMap = new SignedObject(map, kypair.getPrivate(), sign);
    KeyGenerator KeyGen;    // Generate sealing key & seal map
    KeyGen = KeyGenerator.getInstance("AES");
    KeyGen.init(new SecureRandom());
    Key k = KeyGen.generateKey();
    Cipher cipher1 = Cipher.getInstance("AES");
    cipher1.init(Cipher.ENCRYPT_MODE, k);
    SealedObject sealedMap = new SealedObject(map, cipher1);
    // Generate signing public/private key pair & sign map
    ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data"));
    out.writeObject(signedMap); out.close();
    ObjectInputStream ins = new ObjectInputStream(new FileInputStream("data"));
    signedMap = (SignedObject) ins.readObject(); ins.close();
    if (!signedMap.verify(kypair.getPublic(), sign)) {  // Verify signature and
        throw new GeneralSecurityException("Map failed verification"); }
        sealedMap = (SealedObject) signedMap.getObject();
        cipher1 = Cipher.getInstance("AES"); // Unseal map
        cipher1.init(Cipher.DECRYPT_MODE, k);
        map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher1);
        InspectMap(map); // Inspect map
```

Java XML Digital Signature

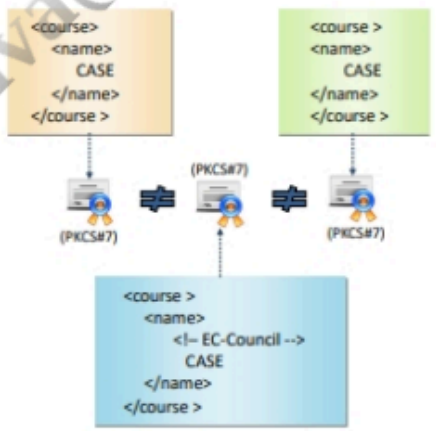
- XML Signature syntax, processing rules and representing are defined by W3C XML Signature Working Group and W3C XML Security Specifications Maintenance Working Group
- Java XML Digital Signature API Specification (JSR 105) is to define a standard Java API for generating and validating XML signatures. The API are located in javax.xml.crypto package
- The purposes are to provide:
  - XML Canonicalization
  - Signature Generation
    - Detached XML Digital Signature
    - Enveloped XML Digital Signature
    - Enveloping XML Digital Signature
  - Signature Validation

Example: XML Digital Signature

Java XML Digital Signature (Cont'd)

XML Canonicalization

- **Digital signature (PKCS#7)** only works if the verification calculations are performed on exactly the same bits as the signing calculations, which is inappropriate for XML data
- XML data is subject to surface representation changes which needs to be canonicalized to DOM-Level, character encoding and XML namespace while calculating the signature



Java XML Digital Signature (Cont'd)

Detached XML Digital Signature

- The signature is detached from the content it signs. It applies to separate data objects and also the signature and data object reside within the same XML document as sibling elements


```
//create a DOM XMLSignatureFactory
//create a DOM XMLSignatureFactory
XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
//using the SHA1 digest algorithm
Reference ref = (Reference) fac.newReference("http://www.w3.org/TR/xmlstylesheet",
            fac.newDigestMethod(DigestMethod.SHA1, null));
// Create the SignedInfo
SignedInfo si = fac.newSignedInfo(fac.newCanonicalizationMethod
            (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
            (C14NMethodParameterSpec) null),
            fac.newSignatureMethod(SignatureMethod.DSA_SHA1, null),
            Collections.singletonList(ref));
// Create a DSA KeyPair
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
kpg.initialize(512);
KeyPair kp = kpg.generateKeyPair();
// Create DSA public key KeyValue object
KeyInfoFactory kif = fac.getKeyInfoFactory();
KeyValue kv = kif.newKeyValue(kp.getPublic());
// Create a KeyInfo and add the KeyValue to it
KeyInfo ki = kif.newKeyInfo(Collections.singletonList(kv));
// Create the XMLSignature object
XMLSignature signature = fac.newXMLSignature(si, ki);
//Create the Document that will hold the resulting XMLSignature
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
Document doc = dbf.newDocumentBuilder().newDocument();
// Create a DOMSignContext and set the signing key
DOMSignContext signContext = new DOMSignContext(kp.getPrivate(), doc);
// Marshal, generate (and sign) the detached XMLSignature
signature.sign(signContext);
//output the resulting document
OutputStream os;
if (args.length > 0) {
  os = new FileOutputStream(args[0]);
} else {
  os = System.out;
}
TransformerFactory tf = TransformerFactory.newInstance();
Transformer trans = tf.newTransformer();
trans.transform(new DOMSource(doc), new StreamResult(os));
}
```


Java XML Digital Signature (Cont'd)

Enveloping XML Digital Signature

- The signature is over the content found within an object element of the signature itself


```
private void main(String args[]) throws NoSuchAlgorithmException, InvalidAlgorithmParame
    //create a DOM XMLSignatureFactory
    XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
    Reference ref = (Reference) fac.newReference("#object",
       fac.newDigestMethod(DigestMethod.SHA1, null));
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    dbf.setNamespaceAware(true);
    Document doc = dbf.newDocumentBuilder().newDocument();
    Node text = doc.createTextNode("some text");
    XMLStructure content = new DOMStructure(text);
    XMLObject obj = fac.newXMLObject
        (Collections.singletonList(content), "object", null, null);
    // Create the SignedInfo
    SignedInfo si = fac.newSignedInfo(fac.newCanonicalizationMethod
            (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
            (C14NMethodParameterSpec) null),
            fac.newSignatureMethod(SignatureMethod.DSA_SHA1, null),
            Collections.singletonList(ref));
    // Create a DSA KeyPair
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
    kpg.initialize(512);
    KeyPair kp = kpg.generateKeyPair();
    // Create DSA public key KeyValue object
    KeyInfoFactory kif = fac.getKeyInfoFactory();
    KeyValue kv = kif.newKeyValue(kp.getPublic());
    // Create a KeyInfo and add the KeyValue to it
    KeyInfo ki = kif.newKeyInfo(Collections.singletonList(kv));
    // Create the XMLSignature object
    XMLSignature signature = fac.newXMLSignature(si, ki,
                    Collections.singletonList(obj), null, null);
    DOMSignContext dsc = new DOMSignContext(kp.getPrivate(), doc);
```

Java XML Digital Signature (Cont'd)

Enveloped XML Digital Signature

- The signature is over the XML content that contains the signature as an element. The content provides the root XML document element


```
<Envelope>
  <data>...</data>
  <Signature>...</Signature>
</ Envelope >
```

Myclass.java

```
47
48
49  private void main(String args[]) throws NoSuchAlgorithmException, InvalidAlgorithmParam
50      //create a DOM XMLSignatureFactory
51
52      XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
53      Reference ref = (Reference) fac.newReference("",
54        fac.newDigestMethod(DigestMethod.SHA1, null),
55        Collections.singletonList
56          (fac.newTransform
57      (Transform.ENVELOPED, (TransformParameterSpec) null)), null, null);
58      // Create the SignedInfo
59      SignedInfo si = fac.newSignedInfo(fac.newCanonicalizationMethod
60                (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
61                (C14NMethodParameterSpec) null),
62                fac.newSignatureMethod(SignatureMethod.DSA_SHA1, null),
63                Collections.singletonList(ref));
64      // Create a DSA KeyPair
65      KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
66      kpg.initialize(512);
67      KeyPair kp = kpg.generateKeyPair();
68      // Create DSA public key value object
69      KeyInfoFactory kif = fac.getKeyInfoFactory();
70      KeyValue kv = kif.newKeyValue(kp.getPublic());
71      // Create a KeyInfo and add the KeyValue to it
72      KeyInfo ki = kif.newKeyInfo(Collections.singletonList(kv));
73      // Instantiate the document to be signed
74      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
75      dbf.setNamespaceAware(true);
76      Document doc =     dbf.newDocumentBuilder().parse(new FileInputStream(""));
77      // Create a DOMSignContext and specify the DSA PrivateKey
78      DOMSignContext dsc = new DOMSignContext
79              (kp.getPrivate(), doc.getDocumentElement());
80      // Create the XMLSignature object
81      XMLSignature signature = fac.newXMLSignature(si, ki);
```

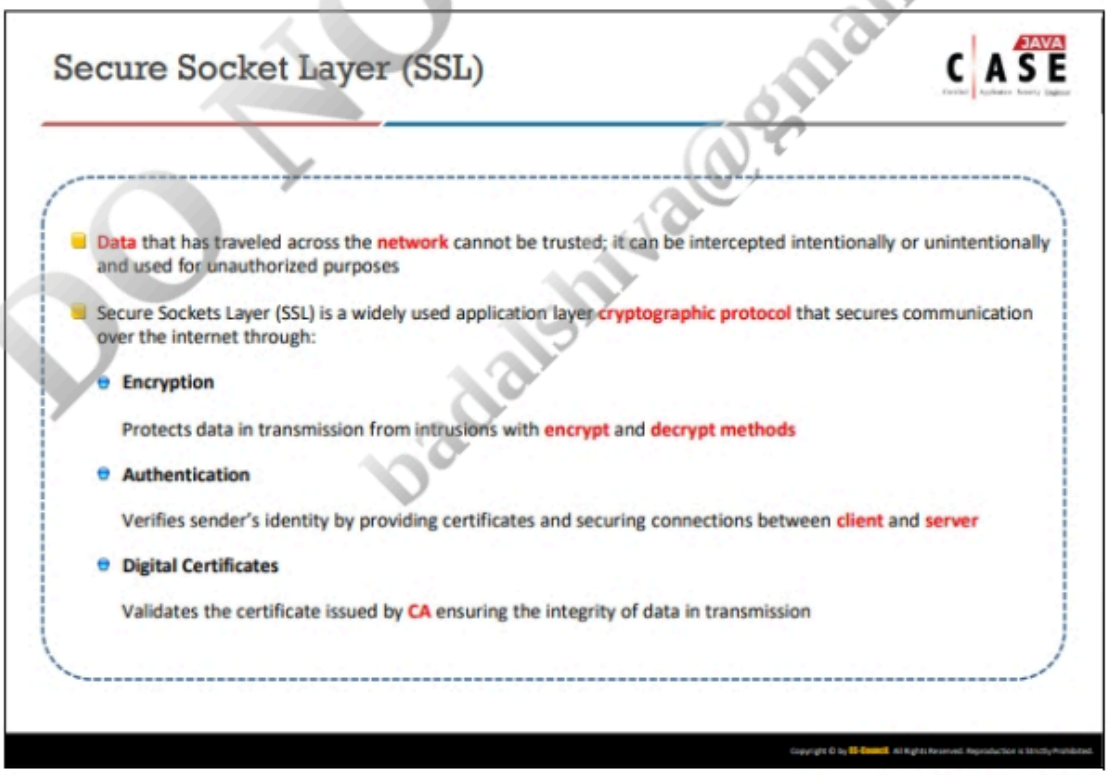

Java XML Digital Signature (Cont'd)

Validate XML Digital Signature

- The validation process includes signature validation and SignedInfo reference validation

Myclass.java | KeyValueKeySelector.java

```
55
56      // Instantiate the document to be validated
57      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
58      dbf.setNamespaceAware(true);
59      Document doc =
60              dbf.newDocumentBuilder().parse(new FileInputStream(""));
61      // Find Signature element
62      NodeList nl =
63          doc.getElementsByTagNameNS(XMLSignature.XMLNS, "Signature");
64      if (nl.getLength() == 0) {
65          throw new Exception("Cannot find Signature element");}
66      // Create a DOM XMLSignatureFactory
67      XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
68      // Create a DOMValidateContext and specify a KeyValue KeySelector
69      DOMValidateContext valContext = new DOMValidateContext
70      (new KeyValueKeySelector(), nl.item(0));
71      // unmarshal the XMLSignature
72      XMLSignature signature = fac.unmarshalXMLSignature(valContext);
73      // validate the XMLSignature (generated above)
74      boolean coreValidity = signature.validate(valContext);
75      // Check core validation status
76      if (coreValidity == false) {
77          System.err.println("Signature failed core validation");
78          boolean sv = signature.getSignatureValue().validate(valContext);
79          System.out.println("signature validation status: " + sv);
80          // check the validation status of each Reference
81          Iterator i =  signature.getSignedInfo().getReferences().iterator();
82          for (int j=0; i.hasNext(); j++) {
83              boolean refValid =
84                  ((XMLSignature) i.next()).validate(valContext);
85              System.out.println("ref["+j+"] validity status: " + refValid);
86          }
87      } else {
88          System.out.println("Signature passed core validation");
89      }
```



Java Secure Socket Extension (JSSE)

- JSSE provides API framework and its **implementation** of the **SSL** and **TLS protocols** imbibed with functionalities such as encryption, server authentication, client authentication and message integrity
- It also provides support for **HTTPS** and several **cryptographic algorithms** that minimize subtle and dangerous security vulnerabilities
- `javax.net.ssl.SSLSocket` class supports standard socket methods to secure socket communication
- `javax.net` and `javax.net.ssl` packages are the standard JSSE APIs that include important classes such as:
  - SSLServerSocket
  - SSLServerSocketFactory
  - SSLSocket
  - SocketFactory
  - ServerSocketFactory
  - SSLSocketFactory classes

SSL and Security: Example 1

- Sockets are used as a communication channel between the server and client in **Java applications**

Vulnerable Code

```
int port = availablePortNumber;
ServerSocket sSkt;
try {
sSkt = new ServerSocket(port);
Socket s = sSkt.accept();
OutputStream outStr = s.getOutputStream();
InputStream inStr = s.getInputStream();
// Send messages to the client through
// the OutputStream
// Receive messages from the client
// through the InputStream
}
catch (IOException e) {
}
```

- The server code uses **regular sockets** that fail to protect data in transmission

Secure Code

```
int port = availablePortNumber;
SSLServerSocket sSkt;
try {
SSLServerSocketFactory sslSrvSkt =
(SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();
sSkt=(SSLServerSocket)sslSrvSkt.createServer
OutputStream outStr = s.getOutputStream();
InputStream inStr = s.getInputStream();
// Send messages to client through OutputStr
// Receive messages from client via Inputstr
}
catch (IOException e) {
}
```

- The server code uses **secure sockets** to protect data while in transmission

SSL and Security: Example 2

Vulnerable Code

- The code uses **regular sockets** for a server application that fails to protect data in transmission

```
import java.io.*;
import java.net.*;
public class SSLSecurity {

    public void EcoServer(String[] args) throws IOException {
        ServerSocket SrvSkt = null;
        try {
        SrvSkt = new ServerSocket(9999);
        Socket s = SrvSkt.accept();
        PrintWriter out = new PrintWriter(s.getOutputStream(), true);
        BufferedReader br = new BufferedReader(
        new InputStreamReader(s.getInputStream()));
        String inLine;
        while ((inLine = br.readLine()) != null) {
        System.out.println(inLine);
        out.println(inLine);
        }
        } finally {
        if (SrvSkt != null) {
        try {
        SrvSkt.close();
        } catch (IOException x) {
        // handle error
        } } }
        public void EcoClient(String[] args) throws IOException {
            Socket s = null;
            try {
            s = new Socket("localhost", 9999);
            PrintWriter out = new PrintWriter(s.getOutputStream(), tru
            BufferedReader br = new BufferedReader(
            new InputStreamReader(s.getInputStream()));
            BufferedReader In = new BufferedReader(
            new InputStreamReader(System.in));
            String user_input;
            while ((user_input = In.readLine()) != null) {
            out.println(user_input);
            System.out.println(br.readLine());
            }
```


SSL and Security: Example 2 (Cont'd)

Secure Code

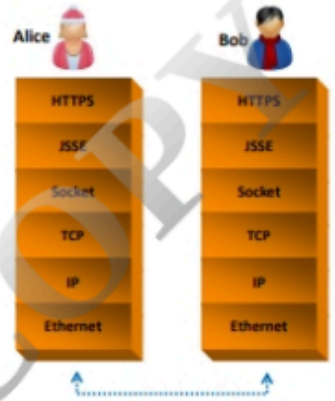
- The code uses **SSLSocket** that blocks any attempts to connect any ports that are not using SSL

```
Demo.java
package com.demo;

import java.io.IOException;
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSocket;
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class Demo{

    public static void main(String[] args) throws IOException {
        SSLServerSocket sslSrvSkt = null;
        try {
        SSLServerSocketFactory sslSrvSktFactory =
        (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
        sslSrvSkt = (SSLServerSocket) sslSrvSktFactory.
        createServerSocket(9999);
        SSLSocket sslSkt = (SSLSocket) sslSrvSkt.accept();
        PrintWriter out = new PrintWriter(sslSkt.getOutputStream(),true);
        BufferedReader br = new BufferedReader(
        new InputStreamReader(sslSkt.getInputStream()));
        String inLine;
        while ((inLine = br.readLine()) != null) {
        System.out.println(inLine);
        out.println(inLine); }
        } finally {
if (sslSrvSkt != null) {
try {
sslSrvSkt.close();
} catch (IOException x) {
// handle error
}
```

```
Demo.java
    public void EchoClient() throws UnknownHostException, IOException
    {

        SSLSocket sslSkt = null;
        try {
        SSLSocketFactory sslSktFactory =
        (SSLSocketFactory) SSLSocketFactory.getDefault();
        sslSkt =
        (SSLSocket) sslSktFactory.createSocket("localhost", 9999);
        BufferedReader br = new BufferedReader(
        new InputStreamReader(sslSkt.getInputStream()));
        BufferedReader In = new BufferedReader(
        new InputStreamReader(System.in));
        String user_input;
        while ((user_input = In.readLine()) != null) {
        out.println(user_input);
        System.out.println(br.readLine()); }
        } finally {
        if (sslSkt != null) {
        try {
        sslSkt.close();
```

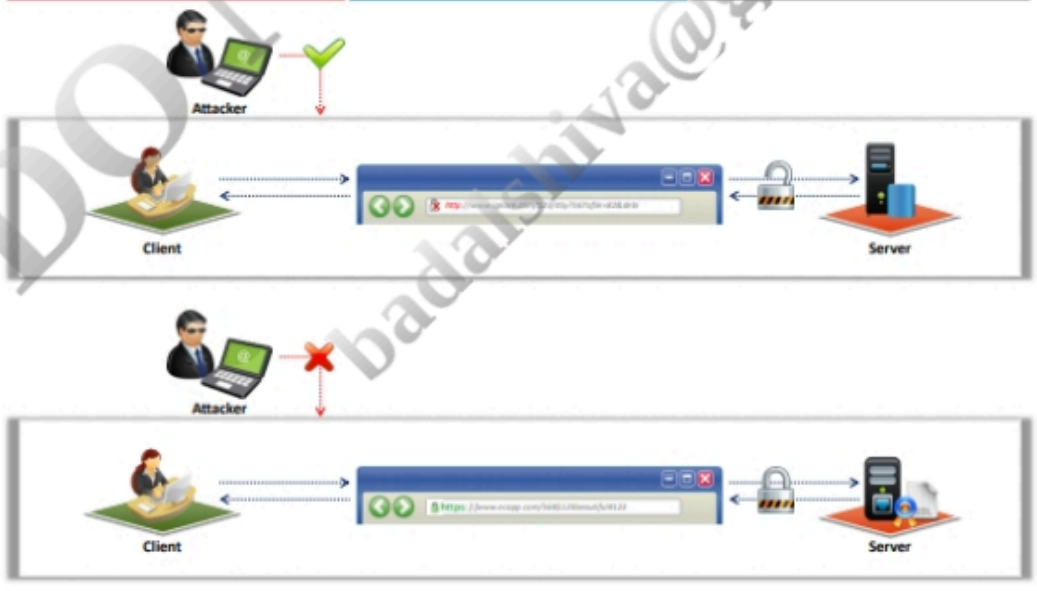
JSSE and HTTPS

- HTTP and HTTPS are **request** and **response** protocols where a client sends a request and a server sends a response to the client
- Some methods supported by these protocols include:
  - `GET`
  - `POST`
  - `DELETE`
  - `PUT`
- HTTPS protocol:
  - Creates a secure connection via SSL/TLS sockets
  - Verifies sender's identity by checking certificates
  - Performs requesting/receiving of data
- `javax.net.ssl.HttpsURLConnection` is an extension of `java.net.HttpsURLConnection` class supporting https-specific features
- Classes used for **HTTPS URL connections** include
  - `java.net.URL`, `java.net.URLConnection`,
  - `java.net.HttpURLConnection`, and
  - `javax.net.ssl.HttpURLConnection`



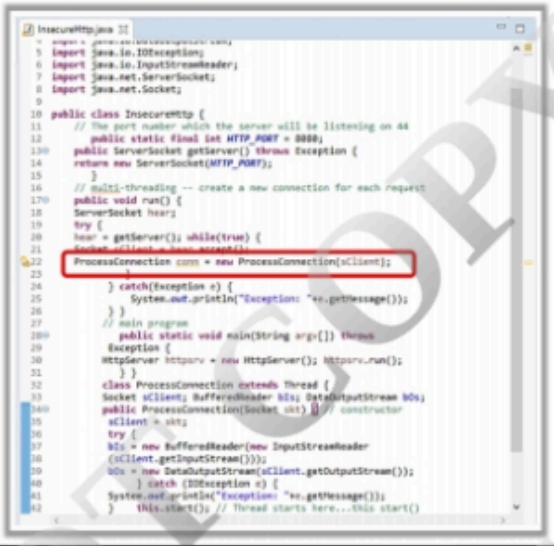
JSSE encapsulates sockets and TCP/IP layers Cross-Reference

JSSE and HTTPS (Cont'd)



Insecure HTTP Server Code

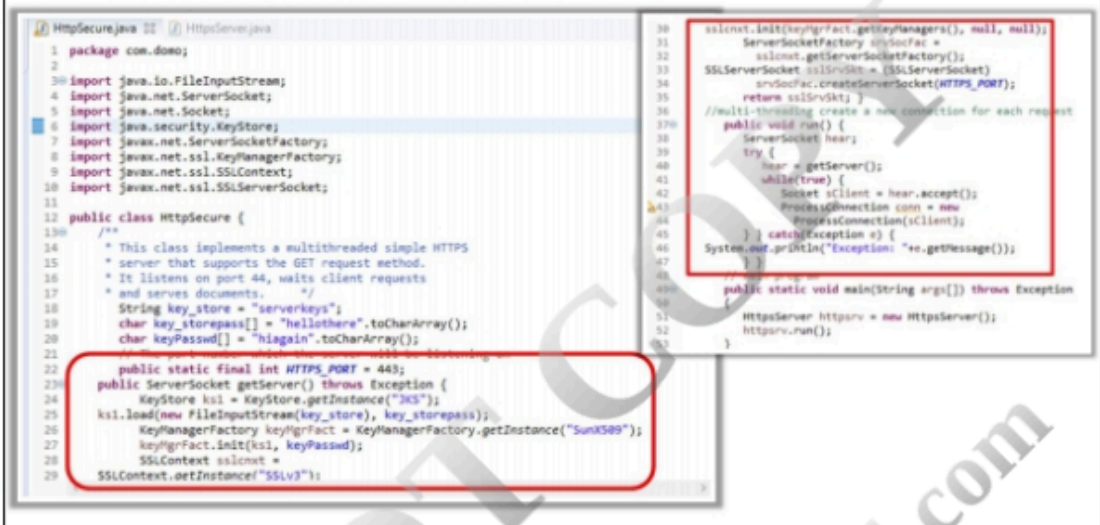
- In the code, **ProcessConnection** class is used for creating a new request in **multi-threaded HTTP server** and **shipDocument** method to send a document to the sever; however, the authenticity of the communication is questioned



Insecure HTTP Server Code (Cont'd)

Secure HTTP Server Code

- In the code, before creating a connection with the HTTP server, a certificate is generated to make it more secure and it uses the **getServer()** method to make a secure server socket connection on HTTPS default port number 443



```
HttpSecure.java   HttpsServer.java
1  package com.domo;
2
3  import java.io.FileInputStream;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6  import java.security.KeyStore;
7  import javax.net.ServerSocketFactory;
8  import javax.net.ssl.KeyManagerFactory;
9  import javax.net.ssl.SSLContext;
10 import javax.net.ssl.SSLServerSocket;
11
12 public class HttpSecure {
13     /**
14      * This class implements a multithreaded simple HTTPS
15      * server that supports the GET request method.
16      * It listens on port 44, waits client requests
17      * and serves documents.      */
18       String key_store = "serverkeys";
19       char key_storepass[] = "hellothere".toCharArray();
20       char keyPasswd[] = "hiagain".toCharArray();
21
22       public static final int HTTPS_PORT = 443;
23   public ServerSocket getServer() throws Exception {
24         KeyStore ks1 = KeyStore.getInstance("JKS");
25   ks1.load(new FileInputStream(key_store), key_storepass);
26         KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance("SunX509");
27         keyMgrFact.init(ks1, keyPasswd);
28         SSLContext sslcnxt =
29   SSLContext.getInstance("SSLv3");
```

```
30       sslcnxt.init(keyMgrFact.getKeyManagers(), null, null);
31       ServerSocketFactory srvSocFac =
32           sslcnxt.getServerSocketFactory();
33   SSLServerSocket sslSrvSkt = (SSLServerSocket)
34           srvSocFac.createServerSocket(HTTPS_PORT);
35       return sslSrvSkt; }
36   //multi-threading create a new connection for each request
37⊖    public void run() {
38       ServerSocket hear;
39       try {
40           hear = getServer();
41           while(true) {
42               Socket sClient = hear.accept();
43               ProcessConnection conn = new
44                   ProcessConnection(sClient);
45           } } catch(Exception e) {
46   System.out.println("Exception: "+e.getMessage());
47       } }
48
49⊖    public static void main(String args[]) throws Exception
50       {
51           HttpsServer httpsrv = new HttpsServer();
52           httpsrv.run();
53       }
```

Attack Scenario: Poor Key Management

- In the present scenario, most of the attacks are aimed at **key management** rather than **cryptographic algorithms**

- **Poor Key Management Threats include:**
  - Mishandling of **Keys**
  - Incorrect Implementation of **Key Generation**
  - Confidentiality **Compromise**
  - Unauthorized use of **Public** or **Secret Keys**
  - Compromise of **Authenticity**

Keys and Certificates

- In Java, there is a separate `java.security.cert.Certificate` class that manages all interactions with certificates using the **Certificate** Class, the **CertificateFactory** Class and the **X509Certificate** Class
- Keys and certificates are often associated with some person or an organization and they deal in how keys are **stored**, **transmitted** and **shared** in the Java Security package i.e., JCE
- Keys are an important component of many Java cryptographic algorithms, generally used for **generating** and **verifying** digital signatures or performing encryption
  - Asymmetric keys (public and private keys) are used for **digital signatures**
  - Symmetric keys (secret keys) are associated with implementation of **certificates**
- The main classes that operate on keys include:
  - The `KeyPairGenerator` class (generates keys)
  - The `KeyFactory` class (interprets key objects and their external representations)
- Certificates are used for **authenticating keys** that are used for digital signing (using secret keys or private keys)

Key Management System

- The key management system provides **private keys** to create digital signatures and **public keys** for verifying the digital signatures
- Key management system manages keys (**public**, **private**, **secret keys**) contained within a certificate, stores those keys and retrieves them programmatically using tools

- **Elements of key management system include:**

Keys	Used for **signing data** e.g., JAR files that are usually provided by an entity (an individual or an organization) and include **public key**, **private key** or both public and private keys
Certificates	Used for verifying **digital signatures** (public key) provided by an entity
Identities	Used for **managing identities** with their keys that are stored in the key database (**keystore**)

KeyStore

- KeyStore provides standard mechanism for managing and storing **cryptographic keys** and **certificates** (password-protected database)
- KeyStore is an important class in **JCA** that manages Java's key management system (`java.security.KeyStore`)
- KeyStore implements entries using `KeyStore.Entry` interface for managing entries such as `KeyStore.PrivateKeyEntry` `KeyStore.SecretKeyEntry` and `KeyStore.TrustedCertificateEntry`

Implementation Method of KeyStore Class

Methods	Description
`aliases()`	Lists all the alias names of this keystore
`getCertificate(String alias)`	Returns the certificate associated with the given alias
`getCertificateAlias(Certificate cert)`	Returns the (alias) name of the first keystore entry whose certificate matches the given certificate
`getInstance(String type)`	Generates a keystore object of the given type
`getKey(String alias, char[] password)`	Returns the key associated with the given alias, using the given password to recover it
`getType()`	Returns the type of this keystore
`load(InputStream stream, char[] password)`	Loads this KeyStore from the given input stream
`size()`	Retrieves the number of entries in this keystore
`store(KeyStore.LoadStoreParameter param)`	Stores this keystore using the given LoadStoreParameter
`store(OutputStream stream, char[] password)`	Stores this keystore to the given output stream, and protects its integrity with the given password

KeyStore: Persistent Data Stores

- Applications that store cryptographic keys in **persistent** and **temporary data stores** or memory are prone to attacks from unauthorized users
- If an application does not securely guard the data store location, an attacker can use insecure **keystore bugs** to read data related to cryptographic keys and certificates
- To protect **keystore** from this type of attack, developers should securely **manage data** in both temporary and persistent data stores
- In the below insecure code, **keystore files** and **registry keys** are accessed due to insecurely managed permissions

MyKeystore.java

```
1  package com.domo;
2
3  import java.io.File;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.security.KeyStore;
7  import java.security.KeyStoreException;
8  import java.security.NoSuchAlgorithmException;
9  import java.security.cert.CertificateException;
10 public class MyKeystore {
11     public static void main(String[] args) throws KeyStoreException, NoSuchAlgorithmException, CertificateException,
12         KeyStore key=KeyStore.getInstance("JKS");
13         String file = System.getProperty("java.home") +"/lib/security/myKeyStore.jks";
14         FileInputStream fstream = new FileInputStream(new File(file));
15                 key.load(fstream, "storeit".toCharArray());
16     }
17 }
```

- Secure code solution for the above code is to give restricted or no permission to the folder `java.home/lib/security` that holds the storage facility for all keystore files and registry files

Key Management Tool: KeyTool

- The `java.security.KeyStore` provides a repository for cryptographic keys and certificates
- Keytool is an inbuilt tool that is used to generate key pairs, import digital certificates, export existing keys, create self-signed certificates, etc.

Commands for Generating

- Generate a Java keystore and key pair
  - `keytool -genkey -alias mydomain -keyalg RSA -keystore keystore.jks`
- Generate a certificate signing request (CSR) for an existing Java keystore
  - `keytool -certreq -alias mydomain -keystore keystore.jks -file mydomain.csr`
- Generate a keystore and self-signed certificate
  - `keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -validity 360`

Commands for Importing

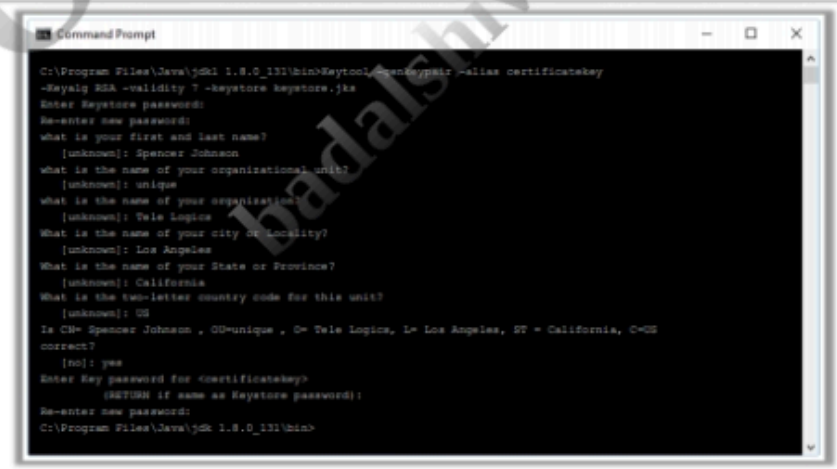
- Import a root or intermediate CA certificate to an existing Java keystore
  - `keytool -import -trustcacerts -alias root -file Thawte.crt -keystore keystore.jks`
- Import a signed primary certificate to an existing Java keystore
  - `keytool -import -trustcacerts -alias mydomain -file mydomain.crt -keystore keystore.jks`

Commands for Checking

- Check a stand-alone certificate
  - `keytool -printcert -v -file mydomain.crt`
- Check which certificates are in a Java keystore
  - `keytool -list -v -keystore keystore.jks`
- Check a particular keystore entry using an alias
  - `keytool -list -v -keystore keystore.jks -alias mydomain`

Key Management Tool: KeyTool (Cont'd)

- Generates a private key in the keystore:
  - `keytool -genkeypair -alias certificatekey -keyalg RSA -validity 7 -keystore keystore.jks`



Key Management Tool: KeyTool (Cont'd)

- Displays **contents** of the keystore:
  - **keytool -list -v -keystore keystore.jks**

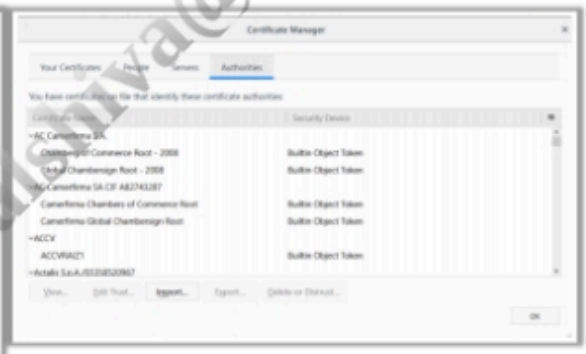
Digital Certificates

Digital Certificates

- Digital certificates are used for **identifying** the **author** (who created the keys) and generally issued by a certification authority (CA)
- A Digital certificate includes:
  - User (entity) Information
  - User's public key
  - Digital signature of the CA
  - Issue and expiry date
- Types of Digital Certificates
  - Secure Socket Layer (SSL) Server certificates
  - Code Signing certificates
  - Client Certificates

Certification Authorities

- In Java, there are defined sets of trusted certificates provided by some trusted certification authorities such as **VeriSign**, **Entrust** or **Thawte**
- The Java platform has a special inbuilt key store, cacert that contains certificates from trusted CAs (`$JREHOME/lib/security/cacerts`)
- In Java, the class that represents digital certificates is `java.security.cert.Certificate`
- Example: ANSI standards and X509 certificates use **distinguished name** (DN) convention to identify the entities that include
  - commonName (CN)
  - organizationUnit (OU)
  - organizationName (O)
  - localityName (L)
  - stateName (S)
  - country (C)



Certification Authorities (Cont'd)

Source: http://www.comodo.com

Source: http://www.thawte.com

Source: http://www.verisign.com

Source: http://www.entrust.net

Signing Jars

- Java Archive (JAR) is a file format that contains multiple files i.e., **class files** and **subsidiary resources** associated with applets and applications
- It includes two important concepts:
  - **Signing JAR files**
    - Signing Jars is an effective way of conveying **trustworthiness** of a **JAR** through digital signature ensuring that the public key is using a certificate from a trusted CA
    - To sign a JAR file, a **private key** and its associated **public key** are required
  - **Verifying signed JAR files**
    - Verifying signed JAR implies that the signed JAR does not contain **malicious data** and also ensures that it is not modified or manipulated by untrusted sources

Signing JAR Tool: Jarsigner

- Jarsigner is an inbuilt **command line Java tool** used to sign a JAR file, and it generates a key from **keystore** and verifies the integrity of a **signed JAR file**

To Sign a JAR File

```
jarsigner -keystore keystore-name -
storepass keystore-password -keypass
key-password jar-file alias-name
```

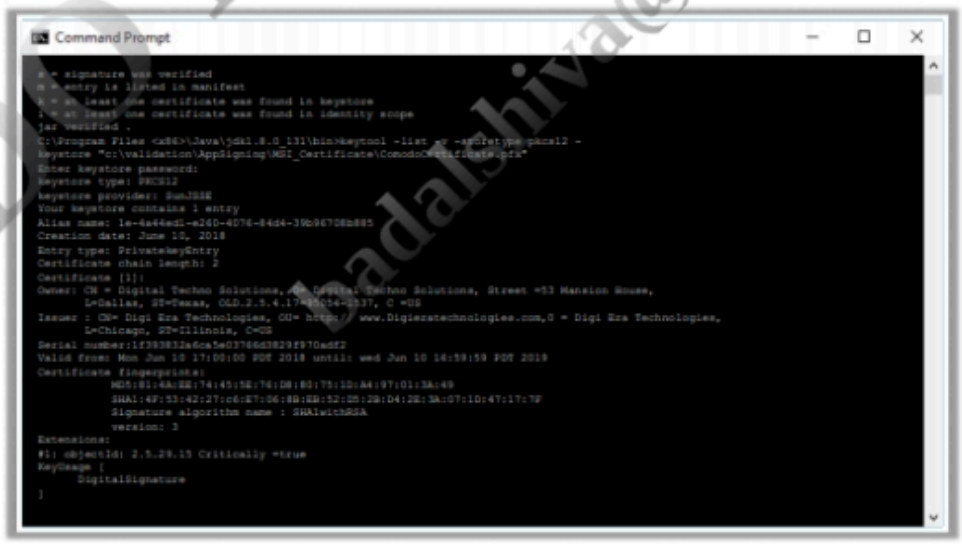
To Verify a Signed JAR File

```
jarsigner -verify jar-file
```

Options	Description
-keystore url	Specifies a desired keystore
- storepass password	Specifies a keystore password
-keypass password	Specifies an alias password
-sigfile file	Specifies a base name for .SF and .DSA files (.keystore created files)
-signedjar file	Specifies the name of the signed JAR file

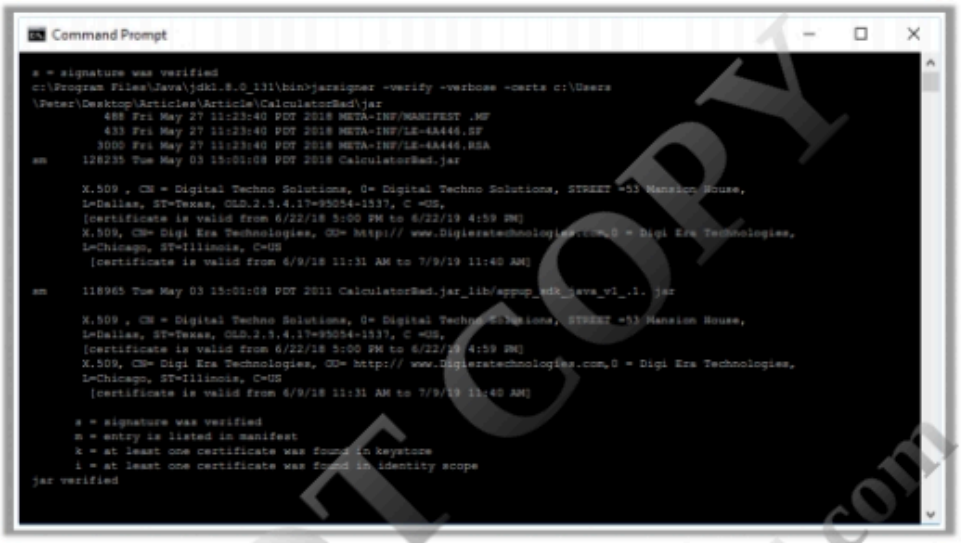
Signing JAR Tool: Jarsigner (Cont'd)

- Signing the application with the **code signing certificate**



Signing JAR Tool: Jarsigner (Cont'd)

- Verifying the **signed application**



Signed Code Sources