UNIT 2 SECURITY IMPLEMENTATION

Structure **Page Nos.**

1.0 INTRODUCTION

In unit 1 we have discussed web services and its advantages, web security concepts, http authentication etc. This unit the following paragraphs and sections describes security implementation, security&servlet, form based custom authentication, and SSL authentication.

Intranet users are commonly required to use, a separate password to authenticate themselves to each and every server they need to access in the course of their work. Multiple passwords are an ongoing headache for both users and system administrators. Users have difficulty keeping track of different passwords, tend to choose poor ones, and then write them down in obvious places. Administrators must keep track of a separate password database on each server and deal with potential security problems related to the fact that passwords are sent over the network routinely and frequently.

2.1 OBJECTIVES

After going through this unit, you should be able to learn about:

- the threats to computer security;
- what causes these threats;
- various security techniques;
- implementing Security Using Servlets, and
- implementing Security Using EJB's.

2.2 SECURITY IMPLEMENTATION

In this section, we will look at security implementation issues.

2.2.1 Security Considerations

System architecture vulnerabilities can result in the violation of the system's security policy. Some of these are described below:

Covert Channel: It is a way for an entity to receive information in an unauthorised manner. It is an information flow that is not controlled by a security mechanism. It is an unintended communication, violating the system's security policy, between two of more users/subjects sharing a common resource.

This transfer can be of two types:

(1) Covert Storage Channel: When a process writes data to a storage location and another process directly or indirectly reads it. This situation occurs when the processes are at different security levels, and therefore are not supposed to be sharing sensitive data.

(2) Covert Timing Channel: One process relays information to another by modulating its use of system resources. There is not much a user can do to countermeasure these channels. But for Trojan that uses HTTP protocol, intrusion detection and auditing may detect a covert channel. Buffer overflow or Parameter checking or **"smashing the stack":** Failure to check the size of input streams specified by the parameters. For example, buffer overflow attack exploits this vulnerability in some operating systems and programs. This happens when programs do not check the length of data that is inputted into a program and then processed by the CPU. The various countermeasures are: (a) proper programming and good coding practices, (b) Host IDS, (c) File system permission and encryption, (d) Strict access control, and (e) Auditing.

Maintenance Hook or Trap Door or Back Doors: These are instructions within the software that only the developers know about and can invoke. This is a mechanism designed to bypass the system's security protections. The various countermeasures are:

(a) Code reviews and unit integration testing.

(b) Host intrusion detection system.

(c) Using file permissions to protect configuration files and sensitive information from being modified.

(d) Strict access control.

(e) File system encryption, and

(f) Auditing.

Timing Issues or Asynchronous attack or Time of Check to Time of Use attack:

This deals with the timing difference of the sequences of steps a system takes to complete a task. This attack exploits the difference in the time that security controls were applied and the time the authorised service was used. A time-of-check versus time-of-use attack, also called race conditions, could replace autoexec.bat.

The various countermeasures for such type of attacks are:

(a) Host intrusion detection system.

(b) File system permissions and encryption.

(c) Strict access control mechanism, and

(e) Auditing.

2.2.2 Recovery Procedures

When a trusted system fails, it is very important that the failure does not compromise the security policy requirements. The recovery procedures also should not give any opportunity for violation of the system's security policy. The system restart must be in a secure mode. Startup should be in the maintenance mode that permits access the only privileged users from privileged terminals.

Fault-tolerant System: In this system, the computer or network continues to function even when a component fails. In this the system has the capability of detecting the fault and correcting the fault as well.

Failsafe System: In this system, the program execution is terminated and the system is protected from being compromised when a system (hardware or software) failure occurs is detected.

Failsoft or resilient: When a system failure occurs and is detected, selected non-critical processing is terminated. The system continues to function in a degraded mode.

Failover : This refers to switching to a duplicate "hot" backup component in real time when a hardware or software failure occurs.

Cold Start: This is required when a system failure occurs and the recovery procedures cannot return the system to a known, reliable, secure state. The maintenance mode of the system is usually employed to bring data consistency through external intervention.

Check Your Progress 1

1) What are the different system architecture vulnerabilities?

……..……...
……
……

2) What are the counter measures for these system architecture vulnerabilities?

……..……...
……
……

3) What are the different procedures of recovery?

……..……...
……
……

2.3 SECURITY AND SERVLET

In this section, we will look at the security issues related to Java and its environment.

Java Security

In Java Security, there is a package, java.security.acl, that contains several classes that you can use to establish a security system in Java. These classes enable your development team to specify different access capabilities for users and user groups. The concept is fairly straightforward. A user or user group is granted permission to functionality by adding that user or group to an access control list.

For example, consider a java.security.Principal called testUser as shown below:

Principal testUser = new PrincipalImpl (“testUser”);

Now, you can create a Permission object to represent the capability of reading from a file.

Permission fileRead = new PermissionImpl (“readFile”);

Once, you have created the user and the user’s permission, you can create the access control list entry. Its important to note that the security APIs require that the owner of the access list be passed in order to ensure that this is truly the developer’s desired action. It is essential that this owner object be protected carefully.

Acl accessList = new AclImpl (owner, “exampleAcl”); In its final form, the access list will contain a bunch of access list entries.

You can create these as follows:

AclEntry aclEntry = new AclEntryImpl (testUser);

aclEntry.addPermission(fileRead);

accessList.addEntry(owner, aclEntry);

The preceding lines create a new AclEntry object for the testUser, add the fileRead permission to that entry, and then add the entry to the access control list. You can now check the user permissions quite easily, as follows:

boolean isReadFileAuthorised = accessList.checkPermission(testUser, readFile);

Check Your Progress 2

1) Explain security provided by java?

……..……...

……

……

2.4 FORM BASED CUSTOM AUTHENTICATION

In this section, we will examine how to use form to authenticate clients.

2.4.1 Use of Forms to Authenticate Clients

A common way for servlet-based systems to perform authentication is to use the session to store information indicating that a user has logged into the system. In this scheme, the authentication logic uses the HttpSession object maintained by the servlet engine in the Web server.

A base servlet with knowledge of authentication is helpful in this case. Using the service method of the BaseServlet, the extending servlets can reuse the security Check functionality.

The service method is shown in the following sample code snippet:

```
  Public void service(HttpServletRequest request, HttpServletResponse
response)   throws IOException, ServletException
  {

   // check if a session has already been created for this user don't create a new session
    HttpSession session = request.getSession( false);
    String requestedPage = request.getParameter(Constants.REQUEST);
    if ( session != null)
    {
      // retrieve authentication parameter from the session
      Boolean isAuthenticated = (Boolean)
      session.getValue(Constants.AUTHENTICATION);
      // Check if the user is not authenticated
      if ( !isAuthenticated.booleanValue() )
      {
        // process the unauthenticated request
        unauthenticatedUser(response, requestedPage);
      }

    }
    else // session does not exist
    {
     // therefore the user is not authenticated process the unauthenticated request
      unauthenticatedUser(response, requestedPage);
     }
  }
```

The BaseServlet attempts to retrieve the session from the servlet engine. On retrieval, the servlet verifies that the user has been granted access to the system. Should either of these checks fail, the servlet redirects the browser to the login screen. On the login screen, the user is prompted to give a username and password. Note that, the data passed from the browser to the Web server is unencrypted unless you use Secure Socket Layer (SSL).

The LoginServlet uses the username/password combination to query the database to ensure that this user does indeed have access to the system. If, the check fails to return a record for that user, the login screen is redisplayed. If, the check is successful, the following code stores the user authentication information inside a session variable.

```
// create a session
session = request.getSession(true);


// convert the boolean to a Boolean
Boolean booleanIsAuthenticated = new Boolean ( isAuthenticated);


// store the boolean value to the session
session.putValue(Constants.AUTHENTICATION,  booleanIsAuthenticated);
```

This example assumes that any user who successfully authenticates to the system has access to the pages displayed prior to login.

Providing Security through EJB's in Java

In the EJB's deployment descriptor, the following code identifies the access control entries associated with the bean:

```
(accessControlEntries
DEFAULT [administrators basicUsers]
theRestrictedMethod [administrators]
); end accessControlEntries
```

Administrators have access to the bean by default and constitute the only user group that has access to theRestrictedMethod. Once you've authorised that the administrators have access to the bean, you now need to create properties detailing which users are in the administrators group.

For this, the weblogic.properties file must include the following lines:

```
weblogic.password.SanjayAdministrator=Sanjay
weblogic.security.group.administrators=SanjayAdministrator
weblogic.password.User1Basic=User1
weblogic.security.group.basicUsers=User1Basic
```

The above method established the users who have access to the bean and have restricted certain specific methods. This limits the potential for malicious attacks on your Web server to reach the business logic stored in the beans.

The final step in this EJB authorisation is to establish the client connection to the bean. The client must specify the username/password combination properly in order to have access to the restricted bean or methods. For example client communication can be as follows:

```
try{
Properties myProperties = new Properties();
myProperties.put( Context.INITIAL_CONTEXT_FACTORY,
" weblogic.jndi.T3InitialContextFactory");
myProperties.put(Context.PROVIDER_URL, "t3://localhost:7001");
myProperties.put(Context.SECURITY_PRINCIPAL, "Sanjay");
myProperties .put(Context.SECURITY_CREDENTIALS, "san");
ic = new InitialContext(myProperties);
}
catch (Exception e) { ... }
```

Since, you've passed the SanjayAdministrator user to the InitialContext, you'll have access to any method to which the administrators group has been granted access. If, your application makes connections to external beans or your beans are used by external applications, you should implement this security authorisation.

2.4.2 Use Java's Multiple-Layer Security Implementation

The following examples are to demonstrate a complete system approach to the security problem.

- In the first example, the form-based authentication scheme was implemented. It checked only to ensure that the user was listed in the database of authenticated users. A user listed in the database was granted access to all functionality within the system without further definition.
- In the second example, the EJB authorised the user attempting to execute restricted methods on the bean and this protects the bean from unauthorised access, but does not protect the Web application.
- The third example, was that of the Java Security Access Control package is given to explains how to use a simple API to verify that a user has access to a certain functionality within the system. Using these three examples. It is possible to create a simple authentication scheme that limits the user's access to web-based components of a system, including back-office systems.

Delegate Security to the Java Access Control Model

The first step is to create delegate classes to wrap the security functionality contained in the Java Access Control Model classes. By wrapping the method calls and interfaces, the developer can ensure that the majority of the code in the system can function independently of the security implementation. In addition, through the delegation pattern, the remainder of the code can perform security functionality without obtaining specific knowledge of the inner workings of security model.

The first main component of this example is the User. The code that implements the interface can delegate calls to the java.security.Principal interface.

For example, to retrieve a user's telephone number, implement a method called getPhoneNumber(). Another approach to obtaining this user data involves the use of XML.Convert data stored in the database into an XMLDocument from which data could be accessed by walking the tree.

The second main component is interface. The classes that implement this interface use the implementation of the java.security.acl.Permission interface to execute their functionality. In a Web-based system, there is a need to identify both the name of the action and the URL related to that action.

The last major component is the WebSecurityManager object and this is responsible for performing the duties related to user management, features management, and the access control lists that establish the relationships between users and desired features.

WebSecurityManager can be implemented in many ways including, but not limited to, a Java bean used by JSP, a servlet, an EJB, or a CORBA/RMI service. The choice is with system's designer. In this simple example, the WebSecurityManager is assumed to run in the same JVM as the servlets/JSP.

In the following example it is assumed that: first, the information relating the users and their permissions is stored in a relational database; second, this database is already populated.

This example builds off of, the framework detailed in the prior example of servlet-based user authentication. The service method is as under:

```
Public void service (HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
  {

// check  if a session has already been created for this user don't create a new session.
    HttpSession session = request.getSession( false);

    String sRequestedFeature = request.getParameter(Constants.FEATURE);

    if ( session != null)
    {
      // retrieve User object
      User currentUser = (User) session.getValue(Constants.USER);

      Feature featureRequested = null;
      try {
        // get the page from Web Security Manager
        featureRequested = WebSecurityManager.getFeature(
sRequestedFeature);
      } catch ( WebSecurityManagerException smE)
      {
        smE.printStackTrace();
```

```
        }

        if ( WebSecurityManager.isUserAuthenticated( currentUser,
featureRequested) )
        {
          // get page from feature
          String sRequestedPage = featureRequested.getFeaturePath();

          // redirect to the requested page
          response.sendRedirect( Constants.LOGIN2 + sRequestedPage);
        } else {
          // redirect to the error page
          response.sendRedirect( Constants.ERROR + sRequestedFeature);
        }
      } else {
        // redirect to the login servlet (passing parameter)
        response.sendRedirect( Constants.LOGIN2 + sRequestedFeature);
      }
    }
```

In this code, the user is authenticated against the access control list using the requested feature name. The user object is retrieved from the session. The feature object corresponding to the request parameter is retrieved from the SecurityManager object. The SecurityManager then checks the feature against the access control list that was created on the user login through the implementation of the access control list interface.

Upon login, the username/password combination is compared to the data stored in the database. If successful, the User object will be created and stored to the session. The features related to the user in the database are created and added to an access control list entry for the user. This entry is then added to the master access control list for the application. From then on, the application can delegate the responsibility of securing the application to the Java Access Control Model classes.

Here's a code showing how the features are added to the access control list for a given user.

```
  private static void addAclEntry(User user, Hashtable hFeatures)
    throws WebSecurityManagerException
  {
    // create a new ACL entry for this user
    AclEntry newAclEntry = new AclEntryImpl(  user);
    // initialize some temporary variables
    String sFeatureCode = null;
    Feature feature = null;
```

```
    Enumeration enumKeys = hFeatures.keys();
    String keyName = null;

    while ( enumKeys.hasMoreElements() )
    {
      // Get the key name from the enumeration
      keyName = (String) enumKeys.nextElement();

      // retrieve the feature from the hashtable
      feature = (Feature) hFeatures.get(keyName);

      // add the permission to the aclEntry
      newAclEntry.addPermission( feature );
    }
    try {
      // add the aclEntry to the ACL for the _securityOwner
      _aclExample.addEntry(_securityOwner, newAclEntry);
    } catch (NotOwnerException noE)
    {
      throw new WebSecurityManagerException("In addAclEntry", noE);
    }
  }
```

The addAclEntry method is passed a User object and an array of Feature objects. Using these objects, it creates an AclEntry and then adds it to the Acl used by the application. It is precisely this Acl that is used by the BaseServlet2 to authenticate the user to the system.

Conclusion

Securing a Web system is a major requirement this section has put forth a security scheme that leverages the code developed by Sun Microsystems to secure objects in Java. Although this simple approach uses an access control list to regulate user access to protected features, you can expand it based on your requirements.

Check Your Progress 3

1) Explain form based custom authentication methods used by servlets and EJB.

……...

……

……

2) What are the advantages of using Java's multiple-layer security implementation?

……...

……

……

2.5 RETRIEVING SSL AUTHENTICATION

The following section discuss the issues relating to SSL authentication.

2.5.1 SSL Authentication

SSL uses certificates for authentication — these are digitally signed documents which bind the public key to the identity of the private key owner. Authentication happens at connection time, and is independent of the application or the application protocol.

Certificates are used to authenticate clients to servers, and servers to clients; the mechanism used is essentially the same in both cases. However, the server certificate is mandatory — that is, the server must send its certificate to the client — but the client certificate is optional: some clients may not support client certificates; other may not have certificates installed. Servers can decide whether to require client authentication for a connection.

A certificate contains

- Two distinguished names, which uniquely identify the issuer (the certificate authority that issued the certificate) and the subject (the individual or organisation to whom the certificate was issued). The distinguished names contain several optional components:
  - Common name
  - Organisational unit
  - Organisation
  - Locality
  - State or Province
  - Country
- A digital signature. The signature is created by the certificate authority using the public-key encryption technique:
  i) A secure hashing algorithm is used to create a digest of the certificate's contents.
  ii) The digest is encrypted with the certificate authority's private key.
  The digital signature assures the receiver that no changes have been made to the certificate since it was issued:

  a) The signature is decrypted with the certificate authority's public key.
  b) A new digest of the certificate's contents is made, and compared with the decrypted signature. Any discrepancy indicates that the certificate may have been altered.

- The subject's domain name. The receiver compares this with the actual sender of the certificate.
- The subject's public key.

SSL Encryption

The SSL protocol operates between the application layer and the TCP/IP layer. This allows it to encrypt the data stream itself, which can then be transmitted securely, using any of the application layer protocols.

Two encryption techniques are used:

- Public key encryption is used to encrypt and decrypt certificates during the SSL handshake.
- A mutually agreed symmetric encryption technique, such as DES (data encryption standard), or triple DES, is used in the data transfer following the handshake.

The SSL Handshake

The SSL handshake is an exchange of information that takes place between the client and the server when a connection is established. It is during the handshake that client and server negotiate the encryption algorithms that they will use, and authenticate one another. The main features of the SSL handshake are:

- The client and server exchange information about the SSL version number and the cipher suites that they both support.
- The server sends its certificate and other information to the client. Some of the information is encrypted with the server's private key. If , the client can successfully decrypt the information with the server's public key, it is assured of the server's identity.
- If, client authentication is required, the client sends its certificate and other information to the server. Some of the information is encrypted with the client's private key. If, the server can successfully decrypt the information with the client's public key, it is assured of the client's identity.
- The client and server exchange random information which each generates and which is used to establish session keys: these are symmetric keys which are used to encrypt and decrypt information during the SSL session. The keys are also used to verify the integrity of the data.

Check Your Progress 4

1) What do you understand by SSL Authentication?

……..……...

……

……

2.5.2 Using SSL Authentication in Java Clients

2.5.2.1 JSSE (Java Secure Socket Extesnsion) and Web Logic Server

JSSE is a set of packages that support and implement the SSL and TLS v1 protocols, making those capabilities available. BEA WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between Web Logic Server clients and servers, Java clients, Web browsers, and other servers. Web Logic Server's Java Secure Socket Extension (JSSE) implementation can be used by WebLogic clients. Other JSSE implementations can be used for their client-side code outside the server as well.

The following restrictions apply when using SSL in WebLogic server-side applications:

- The use of third-party JSSE implementations to develop WebLogic server applications is not supported. The SSL implementation that WebLogic Server

uses is static to the server configuration and is not replaceable by user applications. You cannot plug different JSSE implementations into WebLogic Server to have it use those implementations for SSL.

- The WebLogic implementation of JSSE does support JCE Cryptographic Service Providers (CSPs), however, due to the inconsistent provider support for JCE, BEA cannot guarantee that untested providers will work out of the box. BEA has tested WebLogic Server with the following providers:

  i) The default JCE provider (SunJCE provider) that is included with JDK

  ii) The nCipher JCE provider.

WebLogic Server uses the HTTPS port for SSL. Only SSL can be used on that port. SSL encrypts the data transmitted between the client and WebLogic Server so that the username and password do not flow in clear text.

2.5.2.2 Using JNDI Authentication

Java clients use the Java Naming and Directory Interface (JNDI) to pass on credentials to the WebLogic Server. A Java client establishes a connection with WebLogic Server by getting a JNDI InitialContext. The Java client then, uses the InitialContext to look up the resources it needs in the WebLogic Server JNDI tree.

To specify a user and the user's credentials, set the JNDI properties listed in the Table A.

Table A : JNDI Properties

JNDI Properties	Meaning
INITIAL_CONTEXT_FACTORY	Provides an entry point into the WebLogic Server environment. The class weblogic.jndi.WLInitialContextFactory is the JNDI SPI for WebLogic Server.
PROVIDER_URL	Specifies the host and port of the WebLogic Server that provides the name service.
SECURITY_PRINCIPAL	Specifies the identity of the user when that user authenticates the required information to the default (active) security realm.

These properties are stored in a hash table that is passed to the InitialContext constructor. Notice the use of t3s, which is a WebLogic Server proprietary version of SSL. t3s uses encryption to protect the connection and communication between WebLogic Server and the Java client.

The following Example demonstrates how to use one-way SSL certificate authentication in a Java client.

Example1.0: Example of One-Way SSL Authentication Using JNDI

```
...
Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
          "weblogic.jndi.WLInitialContextFactory");
    env.put(Context.PROVIDER_URL, "t3s://weblogic:7002");
```

```
env.put(Context.SECURITY_PRINCIPAL, "javaclient");
env.put(Context.SECURITY_CREDENTIALS, "javaclientpassword");
ctx = new InitialContext(env);
```

2.5.2.3 SSL Certificate Authentication Development Environment

SSL Authentication APIs

To implement Java clients that use SSL authentication on WebLogic Server, you use a combination of Java SDK application programming interfaces (APIs) and WebLogic APIs.

Table B lists and describes the Java SDK APIs packages used to implement certificate authentication. The information in Table B is taken from the Java SDK API documentation and annotated to add WebLogic Server specific information. For more information on the Java SDK APIs, Table C lists and describes the WebLogic APIs used to implement certificate authentication.

Table B: Java SDK Certificate APIs Java SDK Certificate APIs

Java SDK Certificate APIs Java SDK Certificate APIs	Description
javax.crypto	This package provides the classes and interfaces for cryptographic operations. The cryptographic operations defined in this package include encryption, key generation and key agreement, and Message Authentication Code (MAC) generation. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. This package also supports secure streams and sealed objects. Many classes provided in this package are provider-based (see the java.security.Provider class). The class itself defines a programming interface to which applications may be written. The implementations themselves may then be written by independent third-party vendors and plugged in seamlessly as needed. Therefore, application developers may take advantage of any number of provider-based implementations without having to add or rewrite the code.
javax.net	This package provides classes for networking applications. These classes include factories for creating sockets. Using socket factories you can encapsulate socket creation and configuration behaviour.
javax.net.SSL	While the classes and interfaces in this package are supported by WebLogic Server, BEA recommends that you use the weblogic.security.SSL package when you use SSL with WebLogic Server.
java.security.cert	This package provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. It contains support for X.509 v3 certificates and X.509 v2 CRLs.

java.security.KeyStore	This class represents an in-memory collection of keys and certificates. It is used to manage two types of keystore entries: • Key Entry : This type of keystore entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorised access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain for the corresponding public key. Private keys and certificate chains are used by a given entity for self-authentication. Applications for this authentication include software distribution organisations which sign JAR files as part of releasing and/or licensing software. • Trusted Certificate Entry : This type of entry contains a single public key certificate belonging to another party. It is called a trusted certificate because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the subject (owner) of the certificate. This type of entry can be used to authenticate other parties.
java.security.PrivateKey	A private key. This interface contains no methods or constants. It merely serves to group (and provide type safety for) all private key interfaces. **Note:** The specialised private key interfaces extend this interface. For example, see the DSAPrivateKey interface in java.security.interfaces.
java.security.Provider	This class represents a "Cryptographic Service Provider" for the Java Security API, where a provider implements some or all parts of Java Security, including: • Algorithms (such, as DSA, RSA, MD5 or SHA-1). • Key generation, conversion, and management facilities (such, as for algorithm-specific keys). Each provider has a name and a version number, and is configured in each runtime it is installed in. To supply implementations of cryptographic services, a team of developers or a third-party vendor writes the implementation code and creates a subclass of the Provider class.
javax.servlet.http. HttpServletRequest	This interface extends the ServletRequest interface to provide request information for HTTP servlets. The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, and so on).

javax.servlet.http. HttpServletResponse	This interface extends the ServletResponse interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies. The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, and so on).
javax.servlet. ServletOutputStream	This class provides an output stream for sending binary data to the client. A ServletOutputStream object is normally retrieved via the ServletResponse.getOutputStream() method. This is an abstract class that the servlet container implements. Subclasses of this class must implement the java.io.OutputStream.write(int) method.
javax.servlet. ServletResponse	This class defines an object to assist a servlet in sending a response to the client. The servlet container creates a ServletResponse object and passes it as an argument to the servlet's service methods (doGet, doPost, and so on).

Table C: Web Logic Certificate APIs WebLogic Certificate APIs

web Logic Certificate APIs WebLogic Certificate APIs	**Description**
weblogic.net.http. HttpsURLConnection	This class is used to represent a Hyper-Text Transfer Protocol with SSL (HTTPS) connection to a remote object. This class is used to make an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server.
weblogic.security.SSL. HostnameVerifierJSSE	This interface provides a callback mechanism, so that implementers of this interface can supply a policy for handling the case where the host that's being connected to and the server name from the certificate SubjectDN must match. To specify an instance of this interface to be used by the server, set the class for the custom host name verifier in the Client Attributes fields that are located on the Advanced Options panel under the Keystore and SSL tab for the server (for example, myserver).
weblogic.security.SSL. TrustManagerJSSE	This interface permits the user to override certain validation errors in the peer's certificate chain and allows the handshake to continue. This interface also permits the user to perform additional validation on the peer certificate chain and interrupt the handshake if need be.
weblogic.security.SSL. SSLContext	This class holds all the state information shared across all sockets created under that context.
weblogic.security.SSL. SSLSocketFactory	This class delegates requests to create SSL sockets to the SSLSocketFactory.

SSL Client Application Components

At a minimum, an SSL client application comprises the following components:

- *Java client*

  A Java client performs these functions:

  - Initialises an SSLContext with client identity, a HostnameVerifierJSSE, a TrustManagerJSSE, and a HandshakeCompletedListener.
  - Creates a keystore and retrieves the private key and certificate chain.
  - Uses an SSLSocketFactory, and
  - Uses HTTPS to connect to a JSP served by an instance of WebLogic Server.

- *HostnameVerifier*

  The HostnameVerifier implements the weblogic.security.SSL.HostnameVerifierJSSE interface. It provides a callback mechanism so that implementers of this interface can supply a policy for handling the case where the host that is being connected to and the server name from the certificate Subject Distinguished Name (SubjectDN) must match.

- *HandshakeCompletedListener*

  The HandshakeCompletedListener implements the javax.net.ssl.HandshakeCompletedListener interface. It defines how the SSL client receives notifications about the completion of an SSL handshake on a given SSL connection. It also defines the number of times an SSL handshake takes place on a given SSL connection.

- *TrustManager*

  The TrustManager implements the weblogic.security.SSL.TrustManagerJSSE interface. It builds a certificate path to a trusted root and returns true if it can be validated and is trusted for client SSL authentication.

- *build script (build.xml)*

  This script compiles all the files required for the application and deploys them to the WebLogic Server applications directories.

2.5.2.4 Writing Applications that Use SSL

This section covers the following topics:

- Communicating Securely From WebLogic Server to Other WebLogic Servers
- Writing SSL Clients
- Using Two-Way SSL Authentication
- Two-Way SSL Authentication with JNDI
- Using a Custom Host Name Verifier

- Using a Trust Manager
- Using an SSLContext
- Using an SSL Server Socket Factory
- Using URLs to Make Outbound SSL Connections

Communicating Securely From WebLogic Server to Other WebLogic Servers

You can use a URL object to make an outbound SSL connection from a WebLogic Server instance acting as a client to another WebLogic Server instance. The weblogic.net.http.HttpsURLConnection class provides a way to specify the security context information for a client, including the digital certificate and private key of the client.
The weblogic.net.http.HttpsURLConnection class provides methods for determining the negotiated cipher suite, getting/setting a host name verifier, getting the server's certificate chain, and getting/setting an SSLSocketFactory in order to create new SSL sockets.

Writing SSL Clients

This section describes, by way of example, how to write various types of SSL clients. Examples of the following types of SSL clients are provided:

- SSLClient
- SSLSocketClient
- SSLClientServlet

Below, Example 1 shows a sample SSLClient, the relevant explanation is embeddede inside the code for easy understanding of the same.

Example 1: SSL Client Sample Code

```
package examples.security.sslclient;

import java.io.File;
import java.net.URL;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.Hashtable;
import java.security.Provider;
import javax.naming.NamingException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletOutputStream;
import weblogic.net.http.*;
import weblogic.jndi.Environment;
/** SSLClient is a short example of how to use the SSL library of
 * WebLogic to make outgoing SSL connections. It shows both how to
 * do this from a stand-alone application as well as from within
 * WebLogic (in a Servlet).
 *
 */
```

```
public class SSLClient {
  public void SSLClient() {}
  public static void main (String [] argv)
    throws IOException {
    if ((!((argv.length == 4) || (argv.length == 5))) ||
       (!(argv[0].equals("wls")))
       ) {
      System.out.println("example: java SSLClient wls
                server2.weblogic.com 80 443 /examplesWebApp/SnoopServlet.jsp");
      System.exit(-1);
    }
    try {
      System.out.println("----");
      if (argv.length == 5) {
        if (argv[0].equals("wls"))
          wlsURLConnect(argv[1], argv[2], argv[3], argv[4], System.out);
      } else {  // for null query, default page returned...
        if (argv[0].equals("wls"))
          wlsURLConnect(argv[1], argv[2], argv[3], null, System.out);
      }
      System.out.println("----");
    } catch (Exception e) {
      e.printStackTrace();
      printSecurityProviders(System.out);
      System.out.println("----");
    }
  }
  private static void printOut(String outstr, OutputStream stream) {
    if (stream instanceof PrintStream) {
      ((PrintStream)stream).print(outstr);
      return;
    } else if (stream instanceof ServletOutputStream) {
      try {
        ((ServletOutputStream)stream).print(outstr);
        return;
      } catch (IOException ioe) {
        System.out.println(" IOException: "+ioe.getMessage());
      }
    }
    System.out.print(outstr);
  }
  private static void printSecurityProviders(OutputStream stream) {
    StringBuffer outstr = new StringBuffer();
    outstr.append(" JDK Protocol Handlers and Security Providers:\n");
    outstr.append("   java.protocol.handler.pkgs - ");
    outstr.append(System.getProperties().getProperty(
                                  "java.protocol.handler.pkgs"));
    outstr.append("\n");
    Provider[] provs = java.security.Security.getProviders();
    for (int i=0; i<provs.length; i++)
      outstr.append("   provider[" + i + "] - " + provs[i].getName() +
                  " - " + provs[i].getInfo() + "\n");
    outstr.append("\n");
    printOut(outstr.toString(), stream);
  }
  private static void tryConnection(java.net.HttpURLConnection connection,
                                OutputStream stream)
    throws IOException {
```

```
    connection.connect();
    String responseStr = "\t\t" +
                         connection.getResponseCode() + " -- " +
                         connection.getResponseMessage() + "\n\t\t" +
                         connection.getContent().getClass().getName() + "\n";
    connection.disconnect();
    printOut(responseStr, stream);
}
/*
 * This method contains an example of how to use the URL and
 *  URLConnection objects to create a new SSL connection, using
 *  WebLogic SSL client classes.
 */
public static void wlsURLConnect(String host, String port,
                                 String sport, String query,
                                 OutputStream out) {
  try {
    if (query == null)
      query = "/examplesWebApp/index.jsp";
    // The following protocol registration is taken care of in the
    //  normal startup sequence of WebLogic. It can be turned off
    //  using the console SSL panel.
    //
    // we duplicate it here as a proof of concept in a stand alone
    //  java application. Using the URL object for a new connection
    //  inside of WebLogic would work as expected.
    java.util.Properties p = System.getProperties();
    String s = p.getProperty("java.protocol.handler.pkgs");
    if (s == null) {
      s = "weblogic.net";
    } else if (s.indexOf("weblogic.net") == -1) {
      s += "|weblogic.net";
    }
    p.put("java.protocol.handler.pkgs", s);
    System.setProperties(p);
    printSecurityProviders(out);
    // end of protocol registration
    printOut(" Trying a new HTTP connection using WLS client classes -
            \n\thttp://" + host + ":" + port + query + "\n", out);
    URL wlsUrl = null;
    try {
      wlsUrl = new URL("http", host, Integer.valueOf(port).intValue(), query);
      weblogic.net.http.HttpURLConnection connection =
        new weblogic.net.http.HttpURLConnection(wlsUrl);
      tryConnection(connection, out);
    } catch (Exception e) {
      printOut(e.getMessage(), out);
      e.printStackTrace();
      printSecurityProviders(System.out);
      System.out.println("----");
    }
    printOut(" Trying a new HTTPS connection using WLS client classes -
             \n\thttps://" + host + ":" + sport + query + "\n", out);
    wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(), query);
    weblogic.net.http.HttpsURLConnection sconnection =
      new weblogic.net.http.HttpsURLConnection(wlsUrl);
   // only when you have configured a two-way SSL connection, i.e.
   // Client Certs Requested and Enforced is selected in Two Way Client Cert
   // Behavior field in the Server Attributes
```

```
    // that are located on the Advanced Options pane under Keystore & SSL
    // tab on the server, the following private key and the client cert chain
    // is used.
     File ClientKeyFile  = new File ("clientkey.pem");
     File ClientCertsFile  = new File ("client2certs.pem");
     if (!ClientKeyFile.exists() || !ClientCertsFile.exists())
{
  System.out.println("Error : clientkey.pem/client2certs.pem
                    is not present in this directory.");
  System.out.println("To create it run - ant createmycerts.");
  System.exit(0);
}
     InputStream [] ins = new InputStream[2];
     ins[0] = new FileInputStream("client2certs.pem");
     ins[1] = new FileInputStream("clientkey.pem");
     String pwd = "clientkey";
     sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());
     tryConnection(sconnection, out);
    } catch (Exception ioe) {
     printOut(ioe.getMessage(), out);
     ioe.printStackTrace();
    }
  }
}
```

Example 2: SSLSocketClient Sample Code

The SSLSocketClient sample demonstrates how to use SSL sockets to go directly to the secure port to connect to a JSP served by an instance of WebLogic Server and display the results of that connection. It shows how to implement the following functions:

- Initializing an SSLContext with client identity, a HostnameVerifierJSSE, and a TrustManagerJSSE
- Creating a keystore and retrieving the private key and certificate chain
- Using an SSLSocketFactory
- Using HTTPS to connect to a JSP served by WebLogic Server
- Implementing the javax.net.ssl.HandshakeCompletedListener interface
- Creating a dummy implementation of the weblogic.security.SSL.HostnameVerifierJSSE class to verify that the server the example connects to is running on the desired host

```
package examples.security.sslclient;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Hashtable;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.Certificate;
import weblogic.security.SSL.HostnameVerifierJSSE;
```

```
import weblogic.security.SSL.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSession;
import weblogic.security.SSL.SSLSocketFactory;
import weblogic.security.SSL.TrustManagerJSSE;
  /**
  * A Java client demonstrates connecting to a JSP served by WebLogic Server
  * using the secure port and displays the results of the connection.
  */
public class SSLSocketClient {
  public void SSLSocketClient() {}
  public static void main (String [] argv)
    throws IOException {
    if ((argv.length < 2) || (argv.length > 3)) {
      System.out.println("usage:   java SSLSocketClient host sslport
                              <HostnameVerifierJSSE>");
      System.out.println("example: java SSLSocketClient server2.weblogic.com 443
                              MyHVClassName");
      System.exit(-1);
    }
    try {
      System.out.println("\nhttps://" + argv[0] + ":" + argv[1]);
      System.out.println(" Creating the SSLContext");
      SSLContext sslCtx = SSLContext.getInstance("https");
      File KeyStoreFile  = new File ("mykeystore");
      if (!KeyStoreFile.exists())
{
   System.out.println("Keystore Error : mykeystore is not present in this
                        directory.");
   System.out.println("To create it run - ant createmykeystore.");
   System.exit(0);
}
      System.out.println(" Initializing the SSLContext with client\n" +
                        " identity (certificates and private key),\n" +
                        "  HostnameVerifierJSSE, AND NulledTrustManager");
     // Open the keystore, retrieve the private key, and certificate chain
      KeyStore ks = KeyStore.getInstance("jks");
      ks.load(new FileInputStream("mykeystore"), null);
      PrivateKey key = (PrivateKey)ks.getKey("mykey", "testkey".toCharArray());
      Certificate [] certChain = ks.getCertificateChain("mykey");
      sslCtx.loadLocalIdentity(certChain, key);
      HostnameVerifierJSSE hVerifier = null;
      if (argv.length < 3)
        hVerifier = new NulledHostnameVerifier();
      else
        hVerifier = (HostnameVerifierJSSE) Class.forName(argv[2]).newInstance();
      sslCtx.setHostnameVerifierJSSE(hVerifier);
      TrustManagerJSSE tManager = new NulledTrustManager();
      sslCtx.setTrustManagerJSSE(tManager);
      System.out.println(" Creating new SSLSocketFactory with SSLContext");
      SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();
      System.out.println(" Creating and opening new SSLSocket with
                           SSLSocketFactory");
      // using createSocket(String hostname, int port)
      SSLSocket sslSock = (SSLSocket) sslSF.createSocket(argv[0],
                                          new Integer(argv[1]).intValue());
      System.out.println(" SSLSocket created");
      sslSock.addHandshakeCompletedListener(new MyListener());
      OutputStream out = sslSock.getOutputStream();
```

```
    // Send a simple HTTP request
    String req = "GET /examplesWebApp/ShowDate.jsp HTTP/1.0\r\n\r\n";
    out.write(req.getBytes());
    // Retrieve the InputStream and read the HTTP result, displaying
    // it on the console
    InputStream    in = sslSock.getInputStream();
    byte buf[] = new byte[1024];
    try
    {
      while (true)
      {
        int amt = in.read(buf);
        if (amt == -1) break;
        System.out.write(buf, 0, amt);
      }
    }
    catch (IOException e)
    {
      return;
    }
    sslSock.close();
    System.out.println(" SSLSocket closed");
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
}
```

Example 3: SSLClientServlet Sample Code

The SSL ClientServlet Sample code is given below. For easy understanding details are provided inside the code.

```
package examples.security.sslclient;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;
/**
 * SSLClientServlet is a simple servlet wrapper of
 * examples.security.sslclient.SSLClient.
 *
 */
public class SSLClientServlet extends HttpServlet {
  public void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    response.setHeader("Pragma", "no-cache"); // HTTP 1.0
    response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
    ServletOutputStream out = response.getOutputStream();
    out.println("<br><h2>ssl client test</h2><br><hr>");
    String[] target = request.getParameterValues("url");
    try {
```

```
    out.println ("<h3>wls ssl client classes</h3><br>");
    out.println ("java SSLClient wls localhost 7001 7002
                /examplesWebApp/SnoopServlet.jsp<br>");
    out.println("<pre>");
    SSLClient.wlsURLConnect("localhost", "7001", "7002",
                        "/examplesWebApp/SnoopServlet.jsp", out);
    out.println("</pre><br><hr><br>");
   } catch (IOException ioe) {
    out.println ("<br><pre> "+ioe.getMessage () +"</pre>");
    ioe.printStackTrace ();
   }
  }
}
```

Using Two-Way SSL Authentication

When using certificate authentication, WebLogic Server sends a digital certificate to the requesting client. The client examines the digital certificate to ensure that it is authentic, has not expired, and matches the WebLogic Server instance that presented it.

With two-way SSL authentication (a form of mutual authentication), the requesting client also presents a digital certificate to WebLogic Server. When the instance of WebLogic Server is configured for two-way SSL authentication, requesting clients are required to present digital certificates from a specified set of certificate authorities. WebLogic Server accepts only digital certificates that are signed by root certificates from the specified trusted certificate authorities.

The following sections describe the different ways two-way SSL authentication can be implemented in WebLogic Server.

- Two-way SSL Authentication with JNDI
- Using Two-way SSL Authentication Between WebLogic Server Instances
- Using Two-way SSL Authentication with Servlets

Two-Way SSL Authentication with JNDI

While using JNDI for two-way SSL authentication in a Java client, use the setSSLClientCertificate() method of the WebLogic JNDI Environment class. This method sets a private key and chain of X.509 digital certificates for client authentication.

For passing digital certificates to JNDI, create an array of InputStreams opened on files containing DER-encoded digital certificates and set the array in the JNDI hash table. The first element in the array, must contain an InputStream opened on the Java client’s private key file. The second element, must contain an InputStream opened on the Java client's digital certificate file. (This file contains the public key for the Java client.) Additional elements, may contain the digital certificates of the root certificate authority and the signer of any digital certificates in a certificate chain. A certificate chain allows WebLogic Server to authenticate the digital certificate of the Java client if that digital certificate was not directly issued by a certificate authority registered for the Java client in the WebLogic Server keystore file.

You can use the weblogic.security.PEMInputStream class to read digital certificates stored in Privacy Enhanced Mail (PEM) files. This class provides a filter that decodes the base 64-encoded DER certificate into a PEM file.

Example 4: Example of a Two-Way SSL Authentication Client That Uses JNDI

This example demonstrates how to use two-way SSL authentication in a Java client.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import weblogic.jndi.Environment;
import weblogic.security.PEMInputStream;
import java.io.InputStream;
import java.io.FileInputStream;
Public class SSLJNDIClient
{
  public static void main (String [] args) throws Exception
  {
    Context context = null;
    try {
      Environment env = new Environment ();
      // set connection parameters
      env.setProviderUrl("t3s://localhost:7002");
      // The next two set methodes are optional if you are using
      // a UserNameMapper interface.
      env.setSecurityPrincipal("system");
      env.setSecurityCredentials("weblogic");
      InputStream key = new FileInputStream("certs/demokey.pem");
      InputStream cert = new FileInputStream("certs/democert.pem");
      // wrap input streams if key/cert are in pem files
      key = new PEMInputStream(key);
      cert = new PEMInputStream(cert);
      env.setSSLClientCertificate (new InputStream [] {key, cert});
      env.setInitialContextFactory(Environment.
              DEFAULT_INITIAL_CONTEXT_FACTORY);
      context = env.getInitialContext ();
      Object myEJB = (Object) context. lookup ("myEJB");
    }
    finally {
      if (context != null) context.close ();
    }
  }
}
```

The code in Example 4 generates a call to the WebLogic Identity Assertion provider that implements the weblogic.security.providers.authentication.UserNameMapper interface. The class that implements the UserNameMapper interface returns a user object if the digital certificate is valid. WebLogic Server stores this authenticated user object on the Java client's thread in WebLogic Server and uses it for subsequent authorisation requests when the thread attempts to use WebLogic resources protected by the default (active) security realm.

Example 5: Establishing a Secure Connection to Another WebLogic Server Instance

You can use two-way SSL authentication in server-to-server communication in which one WebLogic Server instance is acting as the client of another WebLogic Server instance. Using two-way SSL authentication in server-to-server communication enables you to have dependable, highly-secure connections, even without the more common client/server environment.

This example shows an example of how to establish a secure connection from a servlet running in one instance of WebLogic Server to a second WebLogic Server instance called server2.weblogic.com.

```
FileInputStream [] f = new FileInputStream[3];
   f[0]= new FileInputStream("demokey.pem");
   f[1]= new FileInputStream("democert.pem");
   f[2]= new FileInputStream("ca.pem");
Environment e = new Environment ();
e.setProviderURL("t3s://server2.weblogic.com:443");
e.setSSLClientCertificate(f);
e.setSSLServerName("server2.weblogic.com");
e.setSSLRootCAFingerprints("ac45e2d1ce492252acc27ee5c345ef26");

e.setInitialContextFactory
            ("weblogic.jndi.WLInitialContextFactory");
Context ctx = new InitialContext(e.getProperties())
```

In Example 5, the WebLogic JNDI Environment class creates a hash table to store the following parameters:

- setProviderURL—specifies the URL of the WebLogic Server instance acting as the SSL server. The WebLogic Server instance acting as SSL client calls this method. The URL specifies the t3s protocol which is a WebLogic Server proprietary protocol built on the SSL protocol. The SSL protocol protects the connection and communication between the two WebLogic Servers instances.

- setSSLClientCertificate—specifies a certificate chain to be used for the SSL connection. You use this method to specify an input stream array that consists of a private key (which is the first input stream in the array) and a chain of X.509 certificates (which make up the remaining input streams in the array). Each certificate in the chain of certificates is the issuer of the certificate preceding it in the chain.

- setSSLServerName—specifies the name of the WebLogic Server instance acting as the SSL server. When the SSL server presents its digital certificate to the server acting as the SSL client, the name specified using the setSSLServerName method is compared to the common name field in the digital certificate. In order for hostname verification to succeed, the names must match. This parameter is used to prevent man-in-the-middle attacks.

- setSSLRootCAFingerprint—specifies digital codes that represent a set of trusted certificate authorities. The root certificate in the certificate chain received from the WebLogic Server instance acting as the SSL server has to match one of the fingerprints specified with this method to be trusted. This parameter is used to prevent man-in-the-middle attacks.

Using Two-Way SSL Authentication with Servlets

To authenticate Java clients in a servlet (or any other server-side Java class), you must check whether the client presented a digital certificate and if so, whether the certificate was issued by a trusted certificate authority. The servlet developer is responsible for asking whether the Java client has a valid digital certificate. When developing servlets with the WebLogic Servlet API, you must access information about the SSL connection through the getAttribute () method of the HTTPServletRequest object.

The following attributes are supported in WebLogic Server servlets:

- javax.servlet.request.X509Certificate
  java.security.cert.X509Certificate []—returns an array of the X.509 certificate.
- javax.servlet.request.cipher_suite—returns a string representing the cipher suite used by HTTPS.
- javax.servlet.request.key_size— returns an integer (0, 40, 56, 128, 168) representing the bit size of the symmetric (bulk encryption) key algorithm.
- weblogic.servlet.request.SSLSession
  javax.net.ssl.SSLSession—returns the SSL session object that contains the cipher suite and the dates on which the object was created and last used.

You have access to the user information defined in the digital certificates. When you get the javax.servlet.request.X509Certificate attribute, it is an array of the java.security.cert X.509 certificate. You simply cast the array to that and examine the certificates.

A digital certificate includes information, such as the following:

- The name of the subject (holder, owner) and other identification information required to verify the unique identity of the subject, such as the uniform resource locator (URL) of the Web server using the digital certificate, or an individual user's e-mail address.
- The subject's public key.
- The name of the certificate authority that issued the digital certificate.
- A serial number.
- The validity period (or lifetime) of the digital certificate (as defined by a start date and an end date).

Example 6: Host Name Verifier Sample Code

A host name verifier validates that the host to which an SSL connection is made is the intended or authorised party. A host name verifier is useful when a WebLogic client or a WebLogic Server instance is acting as an SSL client to another application server. It helps prevent man-in-the-middle attacks.

The following program verify the host name.

```
Package examples.security.sslclient;
/**
 * HostnameVerifierJSSE provides a callback mechanism so that
 * implementers of this interface can supply a policy for handling
 * the case where the host that's being connected to and the server
 * name from the certificate SubjectDN must match.
 *
 * This is a null version of that class to show the WebLogic SSL
 * client classes without major problems. For example, in this case,
 * the client code connects to a server at 'localhost' but the
 * democertificate's SubjectDN CommonName is 'bea.com' and the
 * default WebLogic HostnameVerifierJSSE does a String.equals () on
```

```
 * those two hostnames.
 *
 */
Public class NulledHostnameVerifier implements
                weblogic.security.SSL.HostnameVerifierJSSE {
  public boolean verify(String urlHostname, String certHostname)
 {
    return true;
  }
}
```

Example 7: TrustManager Code Example

The weblogic.security.SSL.TrustManagerJSSE interface allows you to override validation errors in a peer's digital certificate and continue the SSL handshake. You can also use the interface to discontinue an SSL handshake by performing additional validation on a server's digital certificate chain.

The following is a example TrustManager

```
Package examples.security.sslclient;

import weblogic.security.SSL.TrustManagerJSSE;
import javax.security.cert.X509Certificate;
Public class NulledTrustManagerJSSE implements TrustManagerJSSE {
  public boolean certificateCallback(X509Certificate[] o, int validateErr) {
    System.out.println(" --- Do Not Use In Production ---\n" + "  By using this " +
                "NulledTrustManager, the trust in the server's identity "+
                "is completely lost.\n --------------------------------");
    for (int i=0; i<o.length; i++)
      System.out.println(" certificate " + i + " -- " + o[i].toString());
    return true;
  }
}
```

The SSLSocketClient example uses the custom trust manager shown above. The SSLSocketClient shows how to set up a new SSL connection by using an SSL context with the trust manager.

Using a Handshake Completed Listener

Example 8: HandshakeCompletedListener Code Example

The *javax.net.ssl.HandshakeCompletedListener* interface defines how the SSL client receives notifications about the completion of an SSL protocol handshake on a given SSL connection. It also defines the number of times an SSL handshake takes place on a given SSL connection. Example 8 shows a HandshakeCompletedListener interface code example. A sample coding is given below:

```
Package examples.security.sslclient;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Hashtable;
import javax.net.ssl.HandshakeCompletedListener;
import javax.net.ssl.SSLSession;
```

```
public class MyListener implements HandshakeCompletedListener
{
public void handshakeCompleted(javax.net.ssl.
                        HandshakeCompletedEvent event)
  {
   SSLSession session = event.getSession();
   System.out.println("Handshake Completed with peer " +
                session.getPeerHost());
   System.out.println("   cipher: " + session.getCipherSuite());
   javax.security.cert.X509Certificate[] certs = null;
   try
   {
    certs = session.getPeerCertificateChain();
   }
   catch (javax.net.ssl.SSLPeerUnverifiedException puv)
   {
    certs = null;
   }
   if  (certs != null)
   {
    System.out.println("   peer certificates:");
    for (int z=0; z<certs.length; z++) System.out.
          println("certs["+z+"]: " + certs[z]);
   }
   else
   {
    System.out.println("No peer certificates presented");
   }
  }
}
```

Using an SSLContext

Example 9: SSL Context Code Example

```
import weblogic.security.SSL.SSLContext;
SSLcontext sslctx = SSLContext.getInstance ("https")
```

The SSLContext class is used to programmatically configure SSL and retain SSL session information. For example, all sockets that are created by socket factories provided by the SSLContext class can agree on session state by using the handshake protocol associated with the SSL context. Each instance can be configured with the keys, certificate chains, and trusted root certificate authorities that it needs to perform authentication. These sessions are cached so that other sockets created under the same SSL context can potentially reuse them later. For more information on session caching see SSL Session Behavior in *Managing WebLogic Security*. To associate an instance of a trust manager class with its SSL context, use the weblogic.security.SSL.SSLContext.setTrustManagerJSSE () method

Using an SSL Server Socket Factory

Example 10: SSLServerSocketFactory Code Example

Instances of the SSLServerSocketFactory class create and return SSL sockets. This class extends javax.net.SocketFactory. A sample code is given below:

```
import weblogic.security.SSL.SSLSocketFactory;
 SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();
```

Example 11: One-Way SSL Authentication URL Outbound SSL Connection Class

You can use a URL object to make an outbound SSL connection from a WebLogic Server instance acting as a client to another WebLogic Server instance. WebLogic Server supports both one-way and two-way SSL authentication for outbound SSL connections.

That Uses Java Classes Only

```
import java.net.URL;
import java.net.URLConnection;
import java.net.HttpURLConnection;
import java.io.IOException;
Public class simpleURL
{
  public static void main (String [] argv)
  {
   if (argv.length != 1)
   {
    System.out.println("Please provide a URL to connect to");
    System.exit(-1);
   }
   setupHandler();
   connectToURL(argv[0]);
  }
  private static void setupHandler()
  {
   java.util.Properties p = System.getProperties();
   String s = p.getProperty("java.protocol.handler.pkgs");
   if (s == null)
    s = "weblogic.net";
   else if (s.indexOf("weblogic.net") == -1)
    s += "|weblogic.net";
   p.put("java.protocol.handler.pkgs", s);
   System.setProperties(p);
  }
  private static void connectToURL(String theURLSpec)
  {
   try
   {
    URL theURL = new URL(theURLSpec);
    URLConnection urlConnection = theURL.openConnection();
    HttpURLConnection connection = null;
    if (!(urlConnection instanceof HttpURLConnection))
    {
     System.out.println("The URL is not using HTTP/HTTPS: " +
                        theURLSpec);
     return;
    }
    connection = (HttpURLConnection) urlConnection;
    connection.connect();
    String responseStr = "\t\t" +
          connection.getResponseCode() + " -- " +
          connection.getResponseMessage() + "\n\t\t" +
              connection.getContent().getClass().getName() + "\n";
    connection.disconnect();
    System.out.println(responseStr);
   }
   catch (IOException ioe)
```

```
    {
     System.out.println("Failure processing URL: " + theURLSpec);
     ioe.printStackTrace();
    }
   }
}
```

WebLogic Two-Way SSL Authentication URL Outbound SSL Connection

Example 12: WebLogic Two-Way SSL Authentication URL Outbound SSL Connection Code Example

For two-way SSL authentication, the weblogic.net.http.HttpsURLConnection class provides a way to specify the security context information for a client, including the digital certificate and private key of the client. Instances of this class represent an HTTPS connection to a remote object.

```
wlsUrl = new URL ("https", host, Integer.valueOf(sport).intValue(),
            query);
weblogic.net.http.HttpsURLConnection sconnection =
       new weblogic.net.http.HttpsURLConnection(wlsUrl);
InputStream [] ins = new InputStream[2];
     ins[0] = new FileInputStream("client2certs.pem");
     ins[1] = new FileInputStream("clientkey.pem");
     String pwd = "clientkey";
     sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());
```

Check Your Progress 5

1) Compare and Contrast SSL Authentication in Java Clients.

……..……...

……

……

2) JSSE and Web Logic Server.

……..……...

……

……

3) What is JNDI Authentication?Explian with suitable example.

……..……...

……

……

2.6 SUMMARY

Software authentication enables a user to authenticate once and gain access to the resources of multiple software systems. Securing a Web system is a major requirement for the development team.

This unit has put forth a security scheme that leverages the code developed by Sun Microsystems to secure objects in Java. Although this simple approach uses an access control list to regulate user access to protected features, you can expand it based on

the requirements of your user community to support additional feature-level variations or user information.

2.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) System Architecture Vulnerabilities can result in violations of security policy. This may please flaw in system design, poor security parameters, open ports, poor management, access control vulnerabilities etc. This include covert channel analysis, maintenance hook or back door or trap doors, buffer overflow etc.

2) Code reviews and unit integration testing, Host intrusion detection system, Use file permissions to protect configuration files and sensitive information from being modified, Strict access control, File system encryption, Auditing.

3) Fault-tolerant System, Failsafe system, failsoft or resilient, Failover, and Cold Start.

Check Your Progress 2

1) Discuss java.security.acl class and its features with suitable example. There is a package, java.security.acl, that contains several classes that you can use to establish a security system in Java. These classes enable your development team to specify different access capabilities for users and user groups. The concept is fairly straightforward. A user or user group is granted permission to functionality by adding that user or group to an access control list.

Check Your Progress 3

1) Discuss the example or similar example as shown in 2.4.2.

2) Implementation without knowledge of inner working of the Operating System, platform independence, enhancing security with multiple security features, In addition, a J2EE server without much customisation may support the EJB mapping that was described earlier in the article. Java also provides some other additional methods of security ranging from digital signatures to the JAAS specification that can be used to protect the class files against unauthorized access.

Check Your Progress 4

1) Discuss SSL Authentication that takes place between the Application layer and the TCP/IP layer. The SSL protocol operates between the application layer and the TCP/IP layer. This allows it to encrypt the data stream itself, which can then be transmitted securely, using any of the application layer protocols. In this both symmetric and asymmetric encryption is used.

Check Your Progress 5

1) Hint: Discuss JSSE set of packages that support and implement the SSL. Discuss

2) Web Logic Server’s Java Secure Socket Extension (JSSE) implementation can be used by WebLogic clients. Other JSSE implementations can be used for their client-side code outside the server as well.

3) Java clients use the Java Naming and Directory Interface (JNDI) to pass credentials to WebLogic Server. A Java client establishes a connection with WebLogic Server by getting a JNDI `InitialContext`. The Java client then uses the `InitialContext` to look up the resources it needs in the WebLogic Server JNDI tree. Please discuss with suitable example.

2.8 FURTHER READINGS/REFERENCES

- Stalling William, *Cryptography and Network Security, Principles and Practice,* 2000, SE, PE.
- Daview D. and Price W., *Security for Computer Networks,* New York:Wiley, 1989.
- Chalie Kaufman, Radia Perlman, Mike Speciner, *Network Security,* Pearson Education.
- B. Schnier, *Applied Cryptography*, John Wiley and Sons
- Steve Burnett & Stephen Paine, *RSA Security's Official Guide to Practice,* SE, PE.
- Dieter Gollmann, *Computer Security*, John Wiley & Sons.

Reference websites:

- *World Wide Web Security FAQ:* http://www.w3.org/Security/Faq/www-security-faq.html
- *Web Security*: http://www.w3schools.com/site/site_security.asp
- *Authentication Authorisation and Access Control:* http://httpd.apache.org/docs/1.3/howto/auth.html
- *Basic Authentication Scheme:* http://en.wikipedia.org/wiki/Basic_authentication_scheme
- *OpenSSL Project:* http:/www.openssl.org
- *Request for Comments 2617 :* http://www.ietf.org/rfc/rfc2617.txt
- *Sun Microsystems Enterprise JavaBeans Specification:* http://java.sun.com/products/ejb/docs.html.
- *Javabeans Program Listings*: http:/e-docs.bea.com