UNIT 12 SPRING SECURITY CONFIGURATION

Structure

12.0 INTRODUCTION

In unit 9, Spring Boot Application and a simple application without any security considerations are described. This unit will describe security features like authentication and authorization that create a secured Java Enterprise Application. The following sections will explain JCA and JSSE to secure our data at various levels. This unit will also describe how spring security simplifies the security implementation as compared to non-spring projects.

12.1 OBJECTIVE

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

- secure the data using JCA,
- establish Secure communication between client and server using JSSE,
- explain common web application security vulnerabilities,
- perform Authentication (AuthN) and Authorization (AuthZ),
- describe building blocks of Spring Security, and
- develop the secured applications using spring security.

12.2 INTRODUCTION TO WEB SECURITIES

Security is one of the topmost concern for any application. It includes many aspects such as sensitive data safety, robust website against hacking, sql injection, csrf etc. A different set of tools are used for various aspects of application security. This section describes how to secure sensitive information on backend applications, i.e. passwords

in the database, critical records etc. With the emergence of cloud services, security has become an indispensable requirement for sensitive information on every platform.

Encryption of data in-transit and at-rest

Data **in-transit** is data moving from/ through the internet or private network. Term Data protection in transit is the protection of data while it is moving from network to network or istransferred from a local storage device to a cloud storage device.

The term Data **at-rest** is data that is not moving from device to device or network to networks such as data stored on a hard drive, laptop or archived/stored in some other way. Data protection at rest aims to secure inactive data stored on any device or network.

Safeguarding sensitive data both **in-transit** and **at-rest** is vital for modern enterprises as attackers are persistently trying innovative ways to compromise systems and steal data. Data can be exposed to many security threats, both in-transit and at-rest, which requires safeguard against such threats. There are multiple approaches to protect data in-transit and at-rest. Encryption plays a vital role in data protection and is a very popular tool for securing data both in-transit and at-rest. Enterprises often encrypt sensitive data prior to move and/or use encrypted connections such as HTTPS, SSL, TLS, FTPS etc., to protect the data in transit. Enterprises can simply encrypt sensitive data prior to storing them and/or choose to encrypt the storage drive itself.

12.2.1 INTRODUCTION OF JAVA CRYPTOGRAPHY ARCHITECTURE (JCA)

Cryptography

Cryptography is the art and science of making a cryptosystem that is capable of providing information security. Cryptography aims to provide the following:

- Secret Communication
- Authentic Information
- Store Sensitive Information

Cryptography Primitives

Cryptography primitives are simply the tools and techniques in Cryptography that can be particularly used to provide a set of desired security services −

- Encryption
- Hash functions
- Message Authentication codes (MAC)
- Digital Signatures

Java Cryptography Architecture (JCA)

The ***Java Cryptography Architecture (JCA)*** is based on “provider” architecture. It is a set of APIs’ to cryptography such as encryption, message digests, key generation and management, digital signatures, certificates etc. It enables you to encrypt and decrypt data in java, manage keys, sign and authenticate messages etc.
JCA provides some general-purpose classes and interfaces. JCA has “provider” based architecture; thus actual functionality and implementation are provided by providers. The encryption algorithm used for encryption and decryption in Cipher class depends on the concrete provider used.

Core Classes and Interfaces

Following java packages has been defined into Java cryptography API.

- javax.crypto
- javax.crypto.spec
- javax.crypto.interfaces

- java.security
- java.security.cert
- java.security.spec
- java.security.interfaces

Above defines, packages contain many core classes and interfaces. Some of thefrequently used ones are as follows:

- Provider
- Cipher
- MessageDigest
- Signature
- Mac
- KeyPairGenerator
- KeyGenerator
- KeyStore

The following sections will give you insights of some frequently used classes and its feature.

Provider

JCA has a "provider" based architecture. Java SDK provides a default provider. Default providers may not support the encryption algorithms you want to use. In order to use a different provider instead of the default, Java Cryptography API provides a method to set the desired provider. Cryptography provider can be set as below-

Security.addProvider(…)

Cipher

The ***Cipher (javax.crypto.Cipher)*** class is the core of JCA framework. This class provides cryptographic cipher for encryption and decryption. Cipher's getInstance method provides cipher object for encryption and decryption. Method getInstance expects the name of the requested transformation. A transformation can be any form either "algorithm/mode/padding" or "algorithm".

Cipher instance can be created using getInstnace method as –

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

Encryption/Decryption example using Cipher Class

```
package com.ignou;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.security.GeneralSecurityException;
import java.util.Base64;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
publicclassEncryptDecrypt
{
 publicstaticbyte[] Encrypt(String secretKey, String plainText)
 throwsGeneralSecurityException
 {
  byte[] keyBytes = secretKey.getBytes(Charset.forName("UTF-8"));
  if (keyBytes.length != 16)
  {
   thrownewIllegalArgumentException("Allowed key size is 16");
  }
  IvParameterSpecparamSpec = newIvParameterSpec(newbyte[16]);
```

```
  SecretKeySpeckeySpec = newSecretKeySpec(keyBytes, "AES");
  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
  cipher.init(Cipher.ENCRYPT_MODE, keySpec, paramSpec);
  byte[] cipherTextBytes = cipher.doFinal(plainText.getBytes(Charset.forName("UTF-8")));
  returncipherTextBytes;
 }
 publicstaticbyte[] Decrypt(String key, byte[] cipherText) throwsGeneralSecurityException
 {
  byte[] keyBytes = key.getBytes(Charset.forName("UTF-8"));
  if (keyBytes.length != 16)
  {
    thrownewIllegalArgumentException("Allowed key size is 16");
  }
  IvParameterSpecparamSpec = newIvParameterSpec(newbyte[16]);
  SecretKeySpeckeySpec = newSecretKeySpec(keyBytes, "AES");
  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
  cipher.init(Cipher.DECRYPT_MODE, keySpec, paramSpec);
  byte[] plainTextBytes = cipher.doFinal(cipherText);
  returnplainTextBytes;
 }
 publicstaticvoidmain(String[] args)
 {
  try
  {
   String key = "ThisIsASecretKey";
   String plainText = "Hello, Good Morning";
   byte[] ciphertextbytes = Encrypt(key, plainText);
   System.out.println("Cipher Text: " + Base64.getEncoder().encodeToString(ciphertextbytes));
   byte[] plaintextbytes = Decrypt(key, ciphertextbytes);
   String decryptedPlainText = newString(plaintextbytes, StandardCharsets.UTF_8);
   System.out.println("Decrypted Plain Text: "+decryptedPlainText);
  }
  catch (GeneralSecurityException e)
  {
   e.printStackTrace();
  }
 }
}
```

Output: Program execution will provide the below output. As you can see by encryption, you are getting encrypted text corresponding to plain text. This Cipher text is used for confidentiality. This encrypted text has been decrypted to the plain text using Cipher class in Decrypt mode.

```
Cipher Text: ww5BTFTQIgKfRVb4u/yi8Xt62XCGE+JOQ9F1KjFltvw=
Decrypted Plain Text: Hello, Good Morning
```

Message Digest

Message digests are a secured one-way hash functions that take arbitrary-sized data and output fixed-length hash value. A common solution to validate the encrypted data on the way to you has not been modified is Message Digest. A message digest is a hash value calculated from message data. Steps to send encrypted data with message digest and validation of encrypted data is as-

- Calculate the message digest from data before encryption
- Encrypt both data and message digest and send across the wire
- Once encrypted data is received, decrypt it to get the message and message digest

- Calculate message digest for the received message and compare it with the received message digest
- If the comparison evaluates to true, there is a high probability (but not a 100% guarantee) that the data was not modified.

Java provides a class, named ***MessageDigest*** which belongs to the package java.security, to create a message digest for message data. This class supports algorithms such as SHA-1, SHA-256, SHA-384, SHA-512, MD5 algorithms to convert an arbitrary-length message to a fixed-length message digest. Sample code to generate message digest for multiple blocks of data is as –

```
MessageDigest md = MessageDigest.getInstance("SHA-256");

byte[] msg1 = "This is msg1".getBytes("UTF-8");
md.update(msg1);
byte[] msg2 = "This is msg2".getBytes("UTF-8");
md.update(msg2);

// Call digest method to get the MD
byte[] digest = md.digest();
```

Mac

The term ***MAC*** stands for Message Authentication Code. The integrity of information transmitted over an unsecured channel can be validated using MAC. A MAC is like a Message Digest but uses an additional key to encrypt the message digest. Thus, A MAC is a message digest encrypted with a secret key. MAC from a message can be created using Java Mac Class. To verify the MAC of a message, it requires the key along with original data. Hence, a MAC is a more protected way to guard a block of data from modification than a message digest. Java code to calculate the MAC of a message data is as –

```
packagecom.ignou;
importjava.util.Base64;
importjavax.crypto.Mac;
importjavax.crypto.spec.SecretKeySpec;
publicclass MAC
{
   publicstaticvoidmain(String[] args) throws Exception
   {
      Mac mac = Mac.getInstance("HmacSHA256");
      byte[] keyBytes   = newbyte[]{'T','h','i','s','I','s','A','S','e' ,'c','r','e','t','K','e','y'};
      String algo  = "RawBytes";
      SecretKeySpec key = newSecretKeySpec(keyBytes,algo);
      mac.init(key); // MAC instance initialized with key

      byte[] msg1 = "This is msg1".getBytes("UTF-8");
      mac.update(msg1); // Update that data for which mac should be calculated
      byte[] msg2 = "This is msg2".getBytes("UTF-8");
      mac.update(msg2); // Update that data for which mac should be calculated
      byte[] macBytes = mac.doFinal(); // Calculate MAC for given blocks of data
      System.out.println("Message Authentication Code(MAC): " +
      Base64.getEncoder().encodeToString(macBytes));
   }
}
```

Execution of the above program will give the MAC. MAC helps to validate the integrity of the received msg. The person who receives and knows the key, he/she can validate the integrity of msg.

Output:

```
Message Authentication Code(MAC): IweNfSGl2Dqw05u57fAL29Sy0aYyWSvs22dZUzg1uXo=
```

Signature

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message or digital document. Digital signatures provide the added assurances of evidence of origin, identity and status of an electronic document or message. It also ensures the non-repudiation, i.e. the author of the message can't later deny that they were the source. Digital Signatures are based on public-key cryptography. The signer uses his private key to create the signature and signature can be authenticated only by the signer public key.

The ***Signature*** (java.security.Signature) class is used to create Signature instance in java. A digital signature is an encrypted message digest with the private key of the signing authority. The signing authority may be the device, person or organization that is to sign the data.

To create a Signature instance, you call the Signature.getInstance(...) method. Signature instance creation is shown below:

```
Signature signature = Signature.getInstance("SHA256WithDSA");
```

Signature creation and verification example

```
package com.ignou;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
public class CreateAndVerifySignature
{
   public static void main(String[] args) throws Exception
   {
      KeyPairGeneratorkeyPairGen = KeyPairGenerator.getInstance("DSA");
      // Initializing the key pair generator
      keyPairGen.initialize(2048);
      // Generate the pair of keys
      KeyPair pair = keyPairGen.generateKeyPair();
      // Get the private key from the key pair
      PrivateKeyprivateKey = pair.getPrivate();
      // Get the public key from the key pair
      PublicKeypublicKey = pair.getPublic();
      // Create signature instance
      Signature sign = Signature.getInstance("SHA256withDSA");
      // Initialize with private key
      sign.initSign(privateKey);
      // Data that needs to be signed
      byte[] bytes = "This is my digital signed data".getBytes();
      // Update signature instance with data to be signed
      sign.update(bytes);
      // Create the signature
      byte[] signature = sign.sign();
      // Initiate the signature with public key to verify signature
      sign.initVerify(pair.getPublic());
      // Update signature instance with data for which signature needs to verify
      sign.update(bytes);
      // Verify the signature
      booleanisValidSignature = sign.verify(signature);

      if (isValidSignature)
      {
         System.out.println("Signature verified");
      }
      else
      {
```

```
            System.out.println("Signature Failed");
        }
    }
}
```

☞Check Your Progress 1

1) Write about JCA with some of the core classes and interface with example

.....
.....
.....
.....

2) Write the characteristics of Message Digest (MD) and Message Authentication Code (MAC) and compare MD and MAC.

.....
.....
.....
.....

3) Alice wants to send some confidential data securely to Bob over the network. Alice uses digital signature to sign the data and did the following where S(m) = digital signature of m.
   - Compute S(m) using her private key
   - Send m, S(m) and her public key to Bob

   Bob did as below upon receiving the confidential data:
   - Bob receive S(m), m and Alice's public key
   - Bob verify the S(m) using m and Alice's key.

   Alice used private key to digitally sign the confidential data. Yet the above approach is attack prone. Figure out the possible attack and flaw in the above approach.

.....
.....
.....
.....

12.2.2 INTRODUCTION OF JAVA SECURE SOCKET EXTENSION (JSSE)

Data *in-transit* is vulnerable and can easily be modified or hacked by an unintended recipient. Private or critical data that travel across the network containing information such as password, credit card numbers etc., must be made unintelligible to unauthorized parties. Data integrity must be ensured while data transports. Many protocols have been designed to protect the privacy and integrity of in-transit data. The Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols are among them.

The Java Secure Socket Extension (JSSE) is a framework to enable secure internet communication. This framework provides an implementation for a Java version of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols. It also

provides API's for data encryption, server authentication, message integrity and optional client Authentication.
JSSE provides API framework along with its implementation.It acts as a building block to build secure web applications simply. It reduces the security vulnerabilities by abstracting the complex underlying security algorithms and handshaking mechanisms. The JSSE API boosts up the core network and cryptographic services defined by the java.security and java.net packages with add-on security.
Supported security protocols by JSSE API are as follows:

- TLS: version 1.0, 1.1, and 1.2
- SSL: version 3.0
- DTLS: versions 1.0 and 1.2

JSSE Features and Benefits

As per the Oracle JSSE reference guide, important benefits and features of JSSE are as follows:

- A standard component of the JDK
- Provider-based architecture makes it flexible and extensible
- Provides classes to create secure channels such as SSLSocket, SSLServerSocket, and SSLEngine
- Initiation or verification of secure communication using cipher suite negotiation
- Support for client and server authentication, which is part of the normal SSL/TLS/DTLS handshaking
- Encapsulated HTTP in the SSL/TLS protocol, which allows access to web pages using HTTPS
- Server session management APIs
- Server name indication extension to facilitate secure connections to virtual servers.
- Provides support for Server Name Indication (SNI) Extension, which extends the SSL/TLS/DTLS protocols to indicate what server name the client is attempting to connect to during handshaking.
- Prevents man-in-the-middle attacks by providing support for endpoint identification during handshaking.

The following section explains the SSL and describes how to implement SSL in Java using JSSE (Java Secure Socket Extension) API.

JSSE provides an SSL toolkit for Java applications. In addition to the necessary classes and interfaces, JSSE provides a handy command-line debugging switch that you can use to watch the SSL protocol in action. A simple example with the below code can be executed in debug mode to watch the handshaking details.

```
packagecom.ignou;
publicclassSecureConnection
{
    publicstaticvoidmain(String[] arstring)
    {
        try
        {
            new java.net.URL("https://www.google.com").getContent();
        }

        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

To run the above application in debug mode, we need to turn on SSL debugging. The application connects to the secure website https://www.google.com using the SSL protocol via HTTPS. In the command given below, the first option loads the HTTPS protocol handler, while the second option, the debug option, causes the program to print out its behavior.

java -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol -Djavax.net.debug=sslcom.ignou.SecureConnection

The above SecureConnection program using the java.net.URL class demonstrates the easiest way to add SSL to your applications. This approach is useful, but it is not flexible enough to let you create a secure application that uses generic sockets. The next section describes SSL and its implementation using JSSE.

In simple terms, we can understand that SSL provides a secured connection between server and client. SSL encrypts the communication between two devices operating over a network connection to provide a secure channel. Secure communication between web browsers and web servers is a very common example of SSL. The following three main information security principles is required for secured communication over a network, and SSL fulfils these:

- Encryption: Protects the data transmissions between involved entities
- Authentication: Ensures that the server we connect to is actually the intended server
- Data integrity: guarantee that data exchange between two entities is not modified by the third party

JSSE API

Java Security API is based on Factory Design Pattern extensively. JSSE provides many factory methods to instantiate. The following section will outline some important classes in JSSE API and the use of these classes.

SSLSocketFactory

The *javax.net.ssl.SSLSocketFactory* is used to create *SSLSocket* objects. Methods in the the *javax.net.ssl.SSLSocketFactory* class can be categorized into three categories. The first category consist of a single method ***static getDefault()***, that can retrieve the default SSL socket factory: **static SocketFactorygetDefault()**

The second category consist of five methods that can be used to create SSLSocket instances:

- Socket createSocket(String host, int port)
- Socket createSocket(InetAddress host, int port)
- Socket createSocket(String host, int port, InetAddressclientHost, int clientPort)
- Socket createSocket(InetAddress host, int port, InetAddressclientHost, int clientPort)
- Socket createSocket(Socket socket, String host, int port, booleanautoClose)

The third category consists of two methods which return the list of SSL cipher suites that are enabled by default and the complete list of supported SSL cipher suites:

- String [] getDefaultCipherSuites()
- String [] getSupportedCipherSuites()

SSLSocket

This *javax.net.ssl.SSLSocket* class extends *java.net.Socket* class in order to provide secure socket. SSLSockets are normal stream sockets but with an added layer of security over a network transport protocol, such as TCP. Since it extends java.net.Socket class, thus it supports all the standard socket methods and adds methods specific to secure sockets. *SSLSocket* instances construct an SSL connection to a named host at a specified port. This allows binding the client side of the connection to a given address and port. Few important methods in javax.net.ssl.SSLSocket are listed below:

- String [] getEnabledCipherSuites()
- String [] getSupportedCipherSuites()
- void setEnabledCipherSuites(String [] suites)
- booleangetEnableSessionCreation()
- void setEnableSessionCreation(boolean flag)
- booleangetNeedClientAuth()
- void setNeedClientAuth(boolean need)

The methods below change the socket from client mode to server mode. This affects who initiates the SSL handshake and who authenticates first:

- booleangetUseClientMode()
- void setUseClientMode(boolean mode)

Method void startHandshake() forces an SSL handshake. It's possible, but not common, to force a new handshake operation in an existing connection.

SSLServerSocketFactory

The *SSLServerSocketFactory* class creates *SSLServerSocket* instance. It is quite similar to *SSLSocketFactory*. Like *createSocket* method in *SSLSocketFactory*, *createServerSocket* method is there in *SSLServerSocketFactory* class to get *SSLServerSocket* instance.

SSLServerSocket

This class is subclass of *ServerSockets* and provides secure server sockets using protocols such as the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols. *SSLServerSocket* creates SSLSockets by accepting connections.

Example

The class *SecureEchoServer* uses JSSE API to create a secure server socket. It listens on the server socket for connections from secure clients. When executing the *SecureEchoServer*, you must specify the keystore to use. The keystore contains the server's certificate.

```
package com.ignou;
public class SecureEchoServer
{
  public static void main(String[] arstring)
  {
    try
    {
      SSLServerSocketFactorysslserversocketfactory = (SSLServerSocketFactory)
      SSLServerSocketFactory.getDefault();
      SSLServerSocketsslserversocket = (SSLServerSocket)
      sslserversocketfactory.createServerSocket(8888);
      SSLSocketsslsocket = (SSLSocket) sslserversocket.accept();
      InputStreaminputstream = sslsocket.getInputStream();
      InputStreamReaderinputstreamreader = new InputStreamReader(inputstream);
      BufferedReaderbufferedreader = new BufferedReader(inputstreamreader);
      String text = null;
```

```
        while ((text = bufferedreader.readLine()) != null)
        {
          System.out.println("Received From Client: "+text);
          System.out.flush();
        }
      }
      catch (Exception exception)
      {
        exception.printStackTrace();
      }
    }
}
```

The *SecureEchoClient* code used JSSE to securely connect to the server. When running the *SecureEchoclient*, you must specify the truststore to use, which contains the list of trusted certificates.

```
package com.ignou;
public class SecureEchoClient
{
  public static void main(String[] arstring)
  {
    try
    {
      SSLSocketFactorysslsocketfactory = (SSLSocketFactory)
      SSLSocketFactory.getDefault();
      SSLSocketsslsocket = (SSLSocket) sslsocketfactory.createSocket("localhost", 8888);
      InputStreaminputstream = System.in;
      InputStreamReaderinputstreamreader = new InputStreamReader(inputstream);
      BufferedReaderbufferedreader = new BufferedReader(inputstreamreader);
      OutputStreamoutputstream = sslsocket.getOutputStream();
      OutputStreamWriteroutputstreamwriter = new OutputStreamWriter(outputstream);
      BufferedWriterbufferedwriter = new BufferedWriter(outputstreamwriter);
      String text = null;
      System.out.println("Send Text to server");
      while ((text = bufferedreader.readLine()) != null)
      {
        bufferedwriter.write(text + '\n');
        bufferedwriter.flush();
      }
    }
    catch (Exception exception)
    {
      exception.printStackTrace();
    }
  }
}
```

The execution of the SecureEchoServer without keystore will give exception as **javax.net.ssl.SSLHandshakeException: no cipher suites in common** while execution of SecureEchoClient without trustStore will give exception as **javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure.**

Generate the self-signed SSL certificate using the Java keytool command and use the generated certificate to execute SecureEchoServer and SecureEchoClient code. Follow below steps to generate SSL certificate.

1. Open a command prompt or terminal
2. Run this command
   keytool -genkey -keyalg RSA -alias localhost -keystore selfsigned.jks -
   validity <days> -keysize 2048
   Where <days> indicate the number of days for which the certificate will be valid.

3. Enter a password for the keystore. Note this password as you require this for configuring the server
4. Enter the other required details
5. When prompted with "Enter key" password for <localhost>, press Enter to use the same password as the keystore password
6. Run this command to verify the contents of the keystore
   keytool -list -v -keystore selfsigned.jks

Put this generated selfsigned.jks file into the root directory of your source code and execute the below command to run SecureEchoServer and SecureEchoClient.

java -Djavax.net.ssl.keyStore=selfsigned.jks -Djavax.net.ssl.keyStorePassword=localhost com.ignou.SecureEchoServer

java -Djavax.net.ssl.trustStore=selfsigned.jks -Djavax.net.ssl.trustStorePassword=localhost com.ignou.SecureEchoClient

Note: Use the same password you used to create a self-signed certificate in step 3 for keyStorePassword and trustStorePassword. I used localhost as a password in self-signed certificate generation;therefore so in the above command, I used the same.

On the client terminal, send some text to the server, and the server will echo the same text on its terminal.

☛Check Your Progress 2

1) Explain JSSE with its features and benefit

.....
.....
.....
.....

2) List down the benefits of implementing SSL using JSSE over secure connection established using java.net.URL

.....
.....
.....
.....

3) List down the reasons for javax.net.ssl.SSLHandshakeException during the establishment of a secure connection

.....
.....
.....
.....

12.3 ISSUES AND CHALLENGES OF WEB SECURITY

With the advancement of the Web and other technology, the frequent usage of networks makes web applications vulnerable to a variety of threats. Advancements in web applications have also attracted malicious hackers and scammers, who are always coming up with new attack vectors. Hackers and attackers can potentially take various

path through the web application to cause risks to your business. Web application security deals specifically with the security surrounding websites, web applications and web services such as APIs.

Common Web App Security Vulnerabilities

Web application security vulnerabilities can range from targeted database manipulation to large-scale network disruption. Few methods of attack which commonly exploit are:

- **Cross site scripting (XSS)** – Cross site scripting (XSS) is a kind of client-side code injection attack. Attacker executes malicious script into web browser of the victim by including malicious code into benign and trusted websites. Attacker mostly uses the forums, message boards and web pages that allow comments to perform XSS. The use of user's input without validation in the output generated by web page make the web page vulnerable to XSS. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. XSS is most common in JavaScript, JavaScript has very restricted access to user's operation system and file system since most web browsers run JavaScript with restrictions. Malicious JavaScript can access all the objects to which a web page has access to. Web page access also includes user's cookies. Most often, session tokens are stored into cookies. Thus, if attackers obtain a user's session cookie, they can impersonate that user, perform actions on behalf of the user, and gain access to the user's sensitive data.
- **Cross-site request forgery (CSRF)** – Cross site request forgery, also known as CSRF or XSRF, is a type of attack in which attackers trick a victim into making a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. Execution of CSRF attack consists of mainly two-part.
  - Trick the victim into clicking a link or loading a web page
  - Sending a crafted, legitimate-looking request from victim's browser to the website

  Tricking the victim into clicking a link or loading a web page is done through social engineering or using the malicious link. While sending a crafted request, attackers send the value chosen by themselves and include the cookies, if any, which is associated with the origin of the website. A CSRF attack exploits the fact that the browser sends the cookies to the website automatically with each request. CSRF attacks take place only in the case of an authenticated user. This means that the victim must be logged in for the attack to succeed.

- **SQL injection (SQLi)** – SQL Injections are among the most common threat to data security.SQL injection, also known as SQLi, is a type of injection attack that exploits the weakness of application security and allows attackers to execute malicious SQL statements. Malicious SQL statements execution enable attackers to access or delete records, change an application data-driven behavior. Attackers can use SQL Injection vulnerabilities to bypass application security measures. Any website or web application, which uses a relational database such as MySQL, Oracle, SQL Server etc., can be impacted by SQL Injection. Attackers use SQLi to gain access to unauthorized information, modify or create new user permissions, or otherwise manipulate or destroy sensitive data. Attackers use specially-crafted input to trick an application into modifying the SQL queries that the application asks the database to execute. Let us understand it by example.
  Suppose that a developer has developed a web page to show the account number and balance into account for the current user's ID provided in a URL, Code snippet in java might be as below:

```
String accountBalanceQuery =
  "SELECT accountNumber, balance FROM accounts WHERE account_owner_id = "
  + request.getParameter("user_id");
try
{
   Statement statement = connection.createStatement();
   ResultSetrs = statement.executeQuery(accountBalanceQuery);
   while (rs.next())
   {
      page.addTableRow(rs.getInt("accountNumber"), rs.getFloat("balance"));
   }
}
catch (SQLException e) { ... }
```

For normal operation user visit the URL as below:
https://securebanking/show_balances?user_id=1234
From the java code snippet, we can check that SQL query to find the account balance for user 1234 will be as below:

SELECT accountNumber, balance FROM accounts WHERE account_owner_id = 1234

The above query is a valid query, and it will return only the intended result to show the balance for the user id 1234.
Now suppose, an attacker has identified the application security vulnerability and change the parameter user_id as-

0 OR 1 = 1

Now the query will be as below:

SELECT accountNumber, balance FROM accounts WHERE account_owner_id = 0 OR 1=1

When the application asks this query to execute into the database, it will return the account balance for all the accounts into the database. The attacker now knows every user's account numbers and balances.

- **Denial-of-service (DoS)** – Denial-of-service attack is a type of attack which makes information systems, devices or other network resources non-responding. A denial-of-service condition is accomplished by flooding the targeted host or network with traffic until the target cannot respond or simply crashes, preventing access for legitimate users.DoS attack exploits a vulnerability in a program or website to force improper use of its resources or network connections, which also leads to a denial of service. There are two general methods of DoS attack: flooding services or crashing services. Flood attacks occur when the system receives too much traffic for the server to buffer, causing them to slow down and eventually stop.

- **Insecure Direct Object References (IDOR)** - Insecure direct object references (IDOR) are a cybersecurity issue that occurs when a web application exposes a reference to internal implementation. Internal implementation objects include files, database records, directories and database keys. For example, an IDOR vulnerability could happen if the URL of a transaction could be changed through client-side user input to show unauthorized data of another transaction. For example, let's consider that the web application displays transaction details using the following URL:

  https://www.example.com/transaction?id=7777

A malicious attacker could try to substitute the id parameter value 7778 with other similar values, for example:

https://www.example.com/transaction?id=7778

transaction Id 7778 could be a valid transaction but belonging to a different user. The malicious attacker should not be authorized to see the transaction for id 7778. However, if the developer made an error, the attacker would see this transaction, and hence web app would have an insecure direct object reference vulnerability.

12.4 SPRING SECURITY OVERVIEW

Security is a very crucial aspect for any application to avoid malfunctioning, unauthorized access or hacking of crucial data. Spring security modules provide us with a framework to easily develop a secure application with very few configurations. Spring Security is a very robust and highly customizable authentication and authorization access-control framework. Along with Authentication and Authorization in Java applications, it also protects the application against attacks such as cross-site request forgery, session fixation, clickjacking etc. In short, Spring Security refers to a series of processes that intercept requests and delegate security processing to the designated Spring Security process. Let us understand the three important concepts before getting dive into Spring Security.

Authentication

Authentication refers to the process of confirming user's identity whom he claims to be. It's about validating user's credentials such as Email/Username/UserId and password. The system validates whether the user is one whom he claims to be using user's credentials. The system authenticates the user identity through login ID and password. Authentication using login Id and password is the usual way to authenticate; still, there are other ways to authenticate. Various methods for authentication are as follows:

- Login Form
- HTTP authentication
- HTTP Digest
- X.509 Certificates
- Custom Authentication Methods

Authorization

Authorization action takes place once a system has authenticated the user's identity. Authorization refers to the rules that determine who is allowed to do what. E.g. John may be authorized to create and delete databases, while Bob is only authorized to read. Authorization is a process of permitting required privileges to the user to access a specific resource in an application such as files, location, database etc. In simple terms, authorization evaluates a user's ability to access the system and up to what extent. There may be different approaches to implement authorization into an application. Different approaches to authorization include:

- **Token-based:** Users are granted a cryptographically signed token that specifies what privileges the user is granted and what resources they can access.
- **Role-Based Access Control (RBAC):** Users are assigned role/roles, and these roles specify what privileges users have. Users roles would restrict what resources they have access to.

- **Access Control List (ACL):** An ACL specifies which users have access to a particular resource. For instance, if a user wants to access a specific file or folder, their username or details should be mentioned in the ACL in order to be able to access certain data.

Various methods for authorization are as follows:

- Access controls for URLs
- Secure Objects and methods
- Access Control List (ACL)

Servlet Filters

A Servlet filter is an object that can intercept HTTP requests. Filter object is invoked at preprocessing and postprocessing of a request. Filters are mainly used to perform cross-cutting concerns such as conversion, logging, compression, encryption, decryption, input validation, security implementation etc.

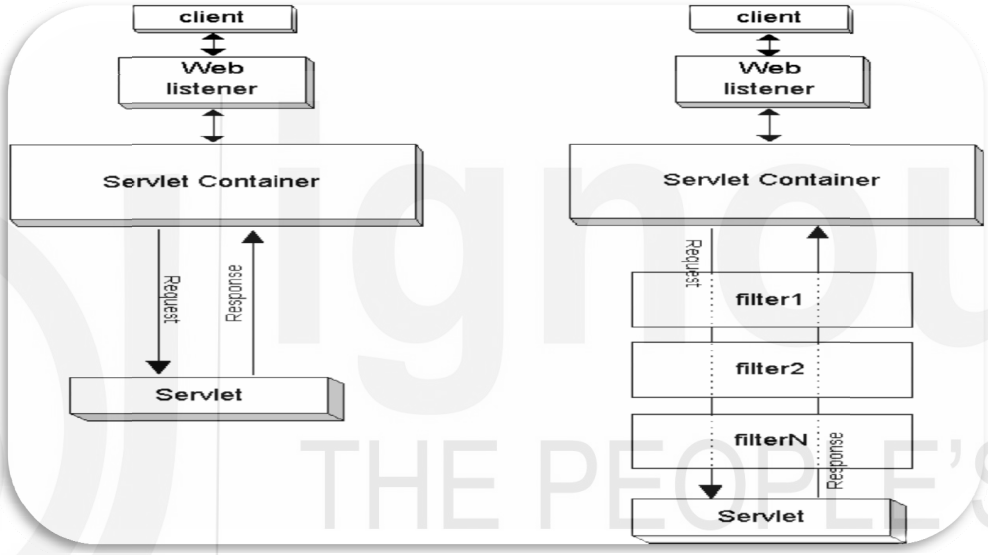


Figure 12.1: Servlet Filters

On the left of above image (Figure 12.1), we can see that there is no preprocessing and postprocessing of request, and thus request is directly reaching to servlet. If we want to implement the security, before request being processed by the servlet, DispatcherServlet in the context of spring, can be implemented using filters as shown above onthe right side of the image.

In the context of Spring Application, no security implementation in DispatcherServlet and no one will like to implement security in all the controllers. Ideally, authentication and authorization should be done before a request hits the controller. Spring security uses various filters to implement security in Spring Applications.

Spring 4 Security

The Spring Security stack is shown in the diagram(Figure 12.2)

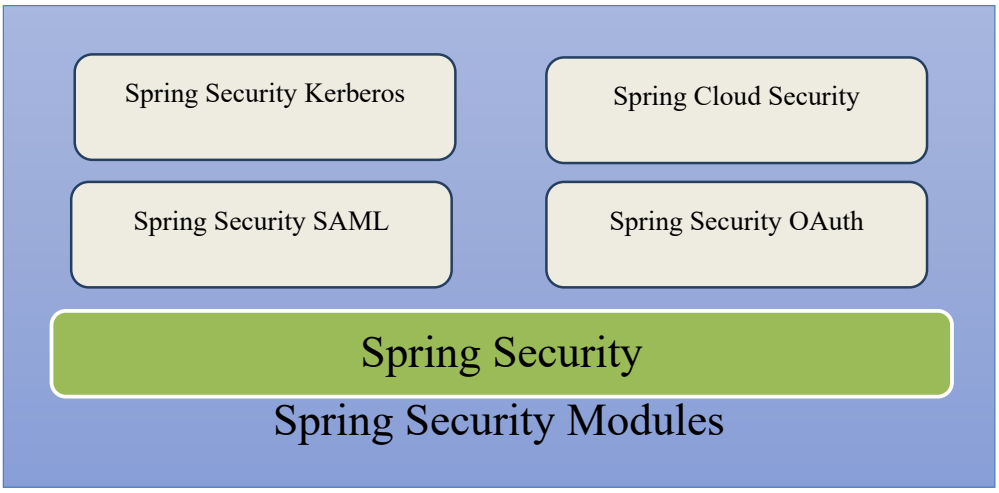


Figure 12.2: Spring Security Stack

Spring 4 Framework has the following modules to provide Security to the Spring-Based Applications:

- Spring Security
- Spring Security SAML
- Spring Security OAuth
- Spring Security Kerberos
- Spring Cloud Security

In Spring Framework, "Spring Security" module is the base module for the rest of the Spring Security modules. The following sections will outline some basics of **"Spring Security"** module.

Spring Security is one of the Spring Framework's security modules. It is a Java SE/Java EE Security Framework to provide Authentication, Authorization, SSO and other Security features for Web Applications or Enterprise Applications.

Spring 4 Security Features

Spring Security Official website: https://projects.spring.io/spring-security/
Spring Security Documentation website: https://docs.spring.io/spring-security/site/docs/
Spring 3.x Security Framework provides the following Features as per Spring Security Official website:

- Authentication and Authorization.
- Supports BASIC, Digest and Form-Based Authentication.
- Supports Cross-Site Request Forgery (CSRF) Implementation.
- Supports "Remember-Me" Feature through HTTP Cookies.
- Supports SSO (Single Sign-On) Implementation.
- Supports LDAP Authentication.
- Supports OpenID Authentication.
- Supports Implementation of ACLs
- Supports "Channel Security" that means automatically switching between HTTP and HTTPS.
- Supports I18N (Internationalization).
- Supports WS-Security using Spring Web Services.

- Supports Both XML Configuration and Annotations. Very Less or minimal XML Configuration.

Spring 4.x Security Framework supports the following new features:

- Supports WebSocket Security.
- Supports Spring Data Integration.
- CSRF Token Argument Resolver.

Spring 4 Security Levels

Spring Security supports the following two Levels of Authorization-

- Method Level Authorization
- URL Level Authorization

NOTE

Spring Security supports "Method Level Security" by using AOP (Aspect-Oriented Programming) that means through Aspects. Spring Security supports "URL Level Security" by using Servlet filters.

Spring 4 Security Advantages

Spring 4 Security Framework provides the following Advantages:

- Open Source Security Framework
- Comprehensive support for authentication and authorization
- Flexible, Easy to Develop and Unit Test the applications
- Protection against common tasks
- Declarative Security Programming
- Easy of Extendibility
- Ease of Maintenance
- CSRF protection
- We can develop Loosely-Coupled Applications.

☞Check Your Progress 3

1) What is Spring Security? Explain in detail with its feature and benefit.

.....
.....
.....
.....

2) What is Authentication (AuthN)? Write down various methods for AuthN.

.....
.....
.....
.....

3) Howdoes Authorization (AuthZ) make the system secured and restricted? List down various methods.

.....
.....
.....
.....

12.5 JAVA BASED CONFIGURATION

Spring security uses Servlet Filter to implement the security in java applications. It creates a Servlet Filter named as *SpringSecurityFilterChain*. *SpringSecurityFilterChain* takes care of all the security concerns such as validating submitted username and passwords, application URLs protection, redirecting to the log in form to unauthenticated users, etc. Spring Security provides an abstract class *AbstractSecurityWebApplicationInitializer* that will ensure the springSecurityFilterChain Filter get registered for every URL in the application.

SpringSecurityFilterChain can be registered into spring application with the initialization of *AbstractSecurityWebApplicationInitializer* as:

```
import org.springframework.security.web.context.*;

public class SecurityWebApplicationInitializer
        extends AbstractSecurityWebApplicationInitializer
{

}
```

With the latest Spring Security and/or Spring Boot versions, Spring Security is configured by having a class that:

- Is annotated with @EnableWebSecurity.
- Extends WebSecurityConfigurerAdapter, which basically provides a convenient base class for creating a WebSecurityConfigurer instance. The implementation allows customization by overriding methods. Here's what a typical WebSecurityConfigurerAdapter looks like:

```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter{ // (1)

  @Override
  protected void configure(HttpSecurity http) throws Exception {  // (2)
     http
.authorizeRequests()
.antMatchers("/home").permitAll() // (3)
.anyRequest().authenticated() // (4)
        .and()
.formLogin() // (5)
.loginPage("/login") // (5)
.permitAll()
        .and()
.logout() // (6)
.permitAll()
       .and()
.httpBasic(); // (7)
  }
}
```

1. A POJO class extending *WebSecurityConfigurerAdapter* and normal Spring @Configuration with the @EnableWebSecurity annotation
2. Overriding of WebSecurityConfigurerAdapter's ***configure(HttpSecurity)*** method enables you to configure the FilterChain using DSL.
3. permitAll() allows all the users for the configured URI. User does not have to be authenticated. antMatcher allows to use wildcards such as (*, **, ?) in the string. The request "/home" is allowed to all users without authentication.

4. All other requests except "/home", the user to be authenticated first, i.e. the user needs to login.
5. As described, authentication methods in section 12.4, form login with a custom loginPage (/login, i.e. not Spring Security's auto-generated one) is being used in the configuration. Login page should be accessible to anyone. Thus "/login" URI has been permitted to all using permitAll().
6. The default spring logout feature is configured and has been permitted to all.
7. Apart from form login, the configuration is also allowing for Basic Auth, i.e. sending in an HTTP Basic Auth Header to authenticate.

The above configuration creates SpringSecurityFilterChain which is accountable for security concerns such as validating submitted username and passwords, application URLs protection, redirecting to the log in form to unauthenticated users, etc. Java Configuration do the following for our application.

- Register required authentication for every URL
- Creates a login form
- Allow a user to be authenticated using form-based authentication as well as HTTP Basic Auth.
- Allow to logout
- Prevent from CSRF attack

Securing the URLs

The most common methods to secure the URL are as below:

- authenticated(): This protects the URL which requires user to be authenticated. If the user is not logged in, the request will be redirected to the login page.
- permitAll(): This is applied for the URL's which requires no security, such as css, javascript, login URL etc.
- hasRole(String role): This is used for authorization with respect to user's role. It restricts to single role. "ROLE_" will be appended into a given role. So if role value is as "ADMIN", comparison will be against "ROLE_ADMIN". An alternative is hasAuthority(String authority)
- hasAuthority(String Authority): This is similar to hasRole(String Role). No prefix will be appended in given authority. If authority is as "ADMIN", comparison will be against "ADMIN"
- hasAnyRole(String… roles): This allows multiple roles. An alternative is hasAnyAuthority(String… authorities)

HTTP Security

To enable HTTP Security in Spring, we need to extend the *WebSecurityConfigurerAdapter*. It allows to configure web-based security for specific http requests. By default all request will be secured but it can be restricted using requestMatcher(RequestMatcher) or other similar methods. The default configuration in the configure(HttpSecurity http) method is as:

```
protected void configure(HttpSecurity http) throws Exception
{
    http.authorizeRequests()
    .anyRequest().authenticated()
    .and().httpBasic();
}
```

Form Login

Based on the features that are enabled in spring security, it generates a login page automatically. It uses standard values for the URLs such as /login and /logout to process the submitted login form and logout the user.

```
protected void configure(HttpSecurity http) throws Exception
{
    http.authorizeRequests()
    .anyRequest().authenticated()
    .and().formLogin()
    .loginPage("/login").permitAll();
}
```

Authentication using Spring Security

Authentication is all about validating user credentials such as Username/User ID and password to verify the user's identity. We can either use in-memory-authentication or jdbc-authentication.

In-Memory Authentication

In-Memory authentication can be configured using AuthenticationManagerBuilder as below:

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception
{
    auth.inMemoryAuthentication()
    .withUser("user").password(passwordEncoder().encode("password")).roles("USER")
    .and()
    .withUser("admin").password(passwordEncoder().encode("password")).roles("USER",
    "ADMIN");
}
```

JDBC Authentication

To implement JDBC authentication, all you have to do is to define a data source within the application and use it as below:

```
@Autowired
private DataSourcedataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception
{
    auth.jdbcAuthentication().dataSource(dataSource)
    .withDefaultSchema()
    .withUser("user").password(passwordEncoder().encode("password")).roles("USER")
    .and()
    .withUser("admin").password(passwordEncoder().encode("password")).roles("USER",
    "ADMIN");
}
```

Aswe can see, we're using the autoconfigured *DataSource*.
The *withDefaultSchema* directive adds a database script that will populate the default schema, allowing users and authorities to be stored. DDL statements are given for the HSQLDB database. The default schema for users and authorities is as below:

```
create table users
```

```
(
username varchar_ignorecase(50) not null primary key,
password varchar_ignorecase(50) not null,
enabled boolean not null
);
create table authorities
(
username varchar_ignorecase(50) not null,
authority varchar_ignorecase(50) not null,
constraint fk_authorities_users foreign key(username) references users(username)
);
```

Default Schema will not work for other databases except HSQLDB database. Check the spring.io website for schema corresponding to a database. The database schema can be checked at:
https://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/appendix-schema.html

Customizing the search Queries

If Spring application is not using default schema for security, we need to provide custom queries to find user and authorities details as below:

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception
{
  auth.jdbcAuthentication()
  .dataSource(dataSource)
  .usersByUsernameQuery("select uuid,password,enabled "
  + "from users "
  + "where uuid= ?")
  .authoritiesByUsernameQuery("select uuid,authority "
  + "from authorities "
  + "where uuid= ?");
}
```

Customization of a search query in spring security is very easy, and two methods for that has been provided as ***usersByUsernameQuery*** and ***authoritiesByUsernameQuery***.

12.6 CREATE SPRING INITIALIZER CLASS

AbstractAnnotationConfigDispatcherServletInitializer class has simplified the Spring initialization. We don't need to manually create contexts but just set appropriate config classes in getRootConfigClasses() and getServletConfigClasses() methods.
Let us consider that we're developing a web application, and we're going to use Spring MVC, Spring Security and Spring Data JPA. For this simple scenario, we would have at least three different config files. A WebConfig contains all our web-related configurations, such as ViewResolvers, Controllers, ArgumentResolvers etc. RepositoryConfig may define datasource, EntityManagerFactory, PlatformTransactionManager etc.
For gluing all these together,we have two options. First, we can define a typical hierarchical ApplicationContext, by adding RepositoryConfig and SecurityConfig in root context and WebConfig in their child context:

```
public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
{
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
      return new Class<?>[] { RepositoryConfig.class, SecurityConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses()
    {
      return new Class<?>[] { WebConfig.class };
    }
    @Override
    protected String[] getServletMappings()
    {
      return new String[] { "/" };
    }
}
```

If we have a single DispatcherServlet, we can add the WebConfig to the root context and make the servlet context empty:

```
public class ServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
{
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
      return new Class<?>[] { RepositoryConfig.class, SecurityConfig.class,
      WebConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses()
    {
      return null;
    }
    @Override
    protected String[] getServletMappings()
    {
      return new String[] { "/" };
    }
}
```

12.7 CREATE CONTROLLER AND VIEW

Controllers and Mapping

Spring ***@Controller*** annotation is used to mark a class as web request handler in the Spring MVC application. While ***@RestController*** is a convenience annotation to mark a class as request handler for RESTful web services,***@RestController*** is annotated with @Controller and @ResponseBody.

A simple example of **@Controller** annotated Class responsible to handle a Get Request method.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
```

```
@Controller //(1)
@RequestMapping(value="/") //(2)
public class HomeController
{
  @GetMapping //(3)
  public ModelAndViewindex()
  {
    ModelAndView mv = new ModelAndView();
    mv.setViewName("index");
    mv.getModel().put("message", "User, Spring Security Implemented Successfully !!");
    return mv;
  }
}
```

1. @Controller annotation is used to mark the class responsible to handle the web request forwarded from Dispatcher Servlet.
2. @RequestMapping annotation has been used to mark at class level to make class HomeController responsible to all web request for the URL pattern "/"
3. @GetMapping annotation has been used to mark the method index() to be invoked for Method Get and URL pattern "/"

Views and Configuration

To render the view, *ViewResolver* configuration is required in Spring MVC Application or Spring Boot Application. For JSP, *ViewResolver* configuration is required in Spring MVC/Spring Boot application to resolve the location of jsp files. There are two ways to configure view resolver as-

- Add entries as shown below in application.properties

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

OR

- Configure InternalResourceViewResolver by extending WebMvcConfigurerAdapter

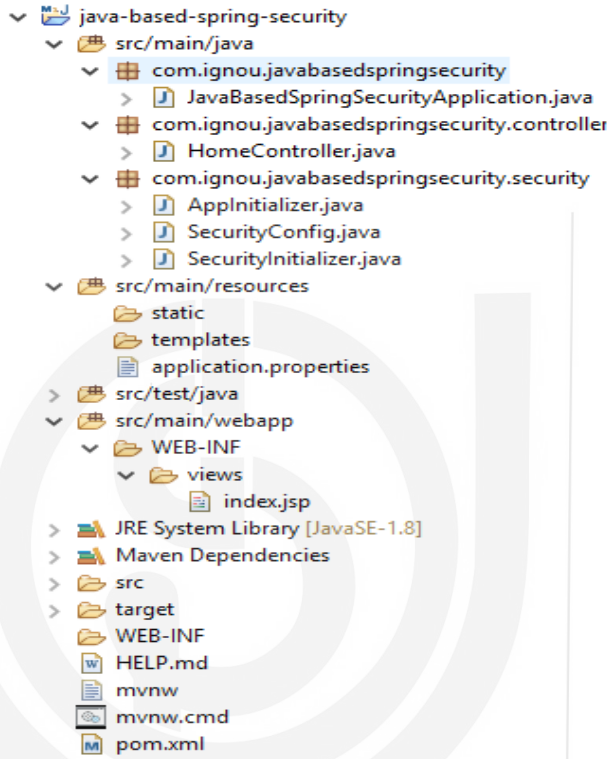
```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan
public class MvcConfiguration extends WebMvcConfigurerAdapter
{
  @Override
  public void configureViewResolvers(ViewResolverRegistry registry)
  {
InternalResourceViewResolver resolver = new InternalResourceViewResolver();
resolver.setPrefix("/WEB-INF/views/");
resolver.setSuffix(".jsp");
resolver.setViewClass(JstlView.class);
registry.viewResolver(resolver);
  }
}
```

12.8 RUN APPLICATION

This section will explain the steps required to run the Spring Boot application with Spring Security. Tools and software required to write the code easily, manage the spring dependencies and execute the application is as -

- Java 8 or above is required
- Maven
- Eclipse IDE

Step 1: Create a web maven project and add the required dependencies such as



spring-boot-starter-web, spring-boot-starter-security in pom.xml. Directory Structure for Spring Project in eclipse will be as below –

Step 2: Spring Boot Application Initializer

```
packagecom.ignou.javabasedspringsecurity;

@SpringBootApplication
publicclassJavaBasedSpringSecurityApplicationextendsSpringBootServletInitializer
{
    @Override
    protectedSpringApplicationBuilder configure(SpringApplicationBuilderapplication)
    {
      returnapplication.sources(JavaBasedSpringSecurityApplication.class);
    }
publicstaticvoid main(String[] args)
    {
```

Figure 2: Project Folder Structure In Eclipse

```
      SpringApplication.run(JavaBasedSpringSecurityApplication.class, args);
    }
}
```

Step 3: Spring Security Configuration

```
packagecom.ignou.javabasedspringsecurity.security;

@EnableWebSecurity
publicclassSecurityConfigextendsWebSecurityConfigurerAdapter
{
  @Autowired
  PasswordEncoderpasswordEncoder;
@Bean
  publicPasswordEncoderpasswordEncoder()
  {
    returnnewBCryptPasswordEncoder();
  }
  @Override
  protectedvoid configure(AuthenticationManagerBuilderauth) throws Exception
  {
    auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
    .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
    .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");
  }
  protectedvoid configure(HttpSecurityhttp) throws Exception
  {
    http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();
  }
}
```

Step 4: Initialize Spring Security

```
packagecom.ignou.javabasedspringsecurity.security;

publicclassSecurityInitializerextendsAbstractSecurityWebApplicationInitializer
{
        // No Code Needed Here
}
```

Step 5: Include the spring security into App Initializer

```
packagecom.ignou.javabasedspringsecurity.security;
publicclassAppInitializerextendsAbstractAnnotationConfigDispatcherServletInitializer
{
        @Override
        protected Class<?>[] getRootConfigClasses()
        {
                returnnew Class[] { SecurityConfig.class };
        }
        @Override
        protected Class<?>[] getServletConfigClasses()
        {
                returnnull;
        }
        @Override
        protected String[] getServletMappings()
        {
                returnnew String[] { "/" };
        }
}
```

Step 6: Controller Class and View

```
packagecom.ignou.javabasedspringsecurity.controller;
@Controller
@RequestMapping(value="/")
publicclassHomeController
{
  @GetMapping
  publicModelAndView index()
  {
    ModelAndViewmv = newModelAndView();
    mv.setViewName("index");
    mv.getModel().put("message", "User, Spring Security Implemented Successfully !!");
    returnmv;
  }
}
```

For the view we will use a simple jsp as below

```
<!DOCTYPEhtml>
<%@taglibprefix="spring"uri="http://www.springframework.org/tags"%>
<htmllang="en">
<body>
<div>
<div>
<h1>Spring Boot Security Example</h1>
<h2>Hello ${message}</h2>
</div>
</div>
</body>
</html>
```

Step 7: Spring Boot Jsp**ViewResolver** configuration

Add the below properties into application.properties file

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

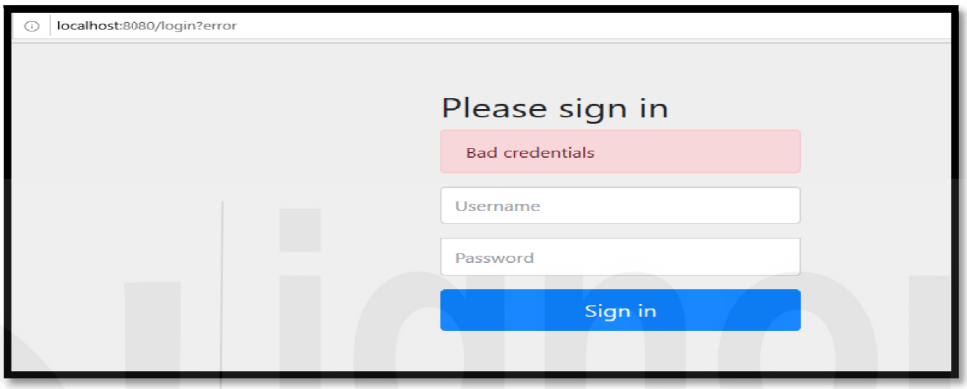
Step 8: Run the file created into Step 2 named asJavaBasedSpringSecurityApplication.java to start the application. This file can be executed using java command or from eclipse IDE itself. After successful execution of the application, security configured application can be accessed using browser on localhost:8080

In Step 3 we configured the spring security and defined the ***testuser*** with credentials as ***user@123*** and ***testadmin*** with credentials as ***admin@123***.

Execution Output:

If user tries to access **localhost:8080** and user is not logged in, spring will redirect to user on **localhost:8080/login** as shown below. This login page is spring security in-built.

If user provides **wrong credentials**, alert from spring security will be as below:



After **Successful login** user will get the below screen:

☞Check Your Progress 4

1) What is SecurityFilterChain in Spring Security?

.....

.....

.....

.....

2) Write the Spring Security Configuration class code for below mentioned criteria-

http://www.securesite.com/static	Open to everyone – CSS, JavaScript
http://www. securesite.com/login	Open to everyone
http://www. securesite.com/user/	ROLE_USER or ROLE_ADMIN
http://www. securesite.com/admin/	ROLE_ADMIN only

3) Consider that Spring Authentication with default schema attribute ***username*** is not suitable for the application. Email attribute as the primary key is being used instead of username. Write the Spring Security Configuration class code to adapt the queries for the new schema as below:

```
CREATE TABLE users
(
  name VARCHAR(50) NOT NULL,
  email VARCHAR(50) NOT NULL,
  password VARCHAR(100) NOT NULL,
  enabled TINYINT NOT NULL DEFAULT 1,
  PRIMARY KEY (email)
);

CREATE TABLE authorities
 (
  email VARCHAR(50) NOT NULL,
  authority VARCHAR(50) NOT NULL,
  FOREIGN KEY (email) REFERENCES users(email)
);
```

4) How to secure the URLs in Spring Security? Explain with available methods to make URL secure.

12.9 SUMMARY

In this unit, many concepts related to web security have been explained. You got to know about securing data at-rest and in-transit. In the unit JCA section described many useful concepts such as Cipher, Message Digest, Message Authentication Code, Signature. Using JCA API, the authenticity and integrity of the message can be verified. Further SSL and other useful concepts to secure the data in-transit is explained. The Spring Security provides a convenient and easy way to incorporate security in spring applications with minimum configuration. The concept of Authentication and Authorization has been explained to make application secure. Various methods such as authenticate(), permitAll(), hasRole(), hasAuthority() have been written to make URL secure in Spring Security. In-memory authentication and JDBC authentication has been described with the custom search query to incorporate user-defined schema for users and authorities.In the last section of this unit, you learned code to integrate spring security into the spring boot application.

12.10 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

☞Check Your Progress 1

1) JCA has been described with the core classes and interface in section 12.2.1
2) Comparison of Message Digest and Message Authentication Code

Message Digest	Message Authentication Code
A message digest algorithm takes a single input a message and produces a "message digest" (aka hash)	A Message Authentication Code algorithm takes two inputs, a message and a secret key, and produces a MAC
It allows you to verify the integrity of the message	It allows you to verify the integrity and the authenticity of the message
Any change to the message will (ideally) result in a different hash being generated.	Any change to the message **or** the secret key will (ideally) result in a different MAC being generated.
An attacker that can replace the message and digest is fully capable of replacing the message and digest with a new valid pair.	Nobody without access to the secret should be able to generate a MAC calculation that verifies the integrity and authenticity of the message

3) Man-in-the-middle attack is possible. Chuck, a man in the middle, attacks and intercepts all the messages sent by Alice. Chuck performs below-
   - Instead of forwarding message m, he forwards message n. Instead of the signature on m with Alice's private key, he forwards a signature on n with his private key.
   - Instead of Alice's public key he sends his public key to Bob. Bob will never see the difference as he does not know Alice's public key beforehand.

   A flaw in the approach is that the public key should not be transferred along with the message. Public key should be available to the person prior to verification of the signature.

☞Check Your Progress 2

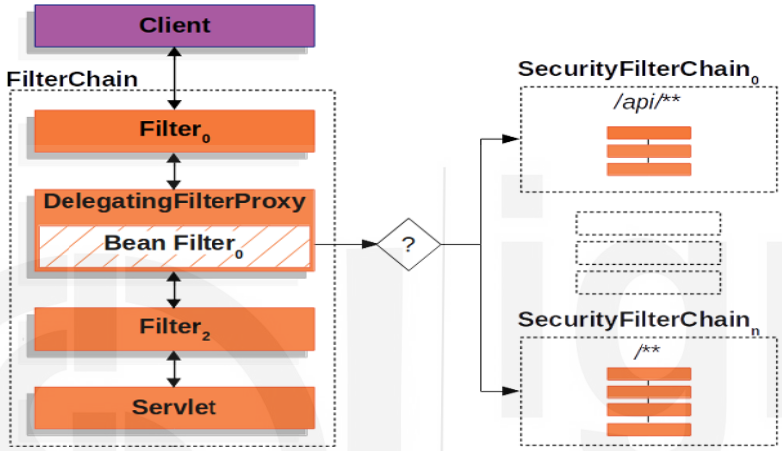
1) JSSE with the advantages has been written in section 12.2.2
2) java.net.URL class demonstrates the easiest way to add SSL to your applications. This approach is useful, but it is not flexible enough to let you create a secure application that uses generic sockets. Using JSSE API to implement SSL gives us full control over the protocols that need to be supported.
3) If javax.net.ssl.SSLHandshakeException occurs, it indicates that there is a handshake_failure. We haven't created or established the SSL certificate that the client and server should be referencing. Normally a certificate is obtained from a trusted certificate authority for a public site, but for testing and development purposes, it's common practice to create a self-signed certificate, which just means you are the issuer. These are typically issued to the localhost domain, so they'll function locally but not be trusted elsewhere.
   Refer section 12.2.2 for keytool to generate keypair and how to use this to solve javax.net.ssl.SSLHandshakeException issue.

☞Check Your Progress 3

1) Refer section 12.4, Spring Security has been explained in detail.
2) Refer Authentication in Section 12.4
3) Refer Authorization in Section 12.4

☞Check Your Progress 4

1) SecurityFilterChain is a FilterChain component which has zero or more Security Filters in an order.



Refer Section 12.5 for more details for SecurityFilterChain.

2) **Hint:** configure method definition in Spring Security Configuration will be as -

```
@Override
   protected void configure(HttpSecurity http) throws Exception
{
    http
    .authorizeRequests()
    .antMatchers("/static","/register").permitAll()
    .antMatchers("/user/**").hasAnyRole("USER", "ADMIN") // can pass
    multiple roles
    .antMatchers("/admin/**").hasRole("ADMIN")
    .anyRequest().authenticated()
    .and()
    .formLogin()
    .loginUrl("/login")
    .permitAll();
}
```

- /static and /register is allowed to everyone so permitAll() is used.
- /user/** must be accessible to either "User" or "Admin" roles. Thus hasAnyRole("USER", "ADMIN") is used.
- /admin/** must be accessible to the only Admin role. Thus hasRole("ADMIN") is used
- /login must be allowed for all so permitAll() is used.

3) We need to customize the search query. Spring security can adapt the customize search queries very easily. We simply have to provide our own

SQL statements when configuring the AuthenticationManagerBuilder. Two methods for customization has been provided as ***usersByUsernameQuery*** and ***authoritiesByUsernameQuery***. Refer to Section 12.5 for customizing the search query.

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
  throws Exception
{
     auth.jdbcAuthentication()
     .dataSource(dataSource)
     .usersByUsernameQuery("select email,password,enabled "
     + "from users "
     + "where email = ?")
     .authoritiesByUsernameQuery("select email,authority "
     + "from authorities "
     + "where email = ?");
}
```

4) Refer Securing the URLs in section 12.5

12.11 REFERENCES/FURTHER READING

- Craig Walls, "Spring Boot in action" Manning Publications, 2016. (https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf)
- Christian Bauer, Gavin King, and Gary Gregory, "**J**ava Persistence with Hibernate",Manning Publications, 2015.
- Ethan Marcotte, "Responsive Web Design", Jeffrey Zeldman Publication, 2011(http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf)
- Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018
- https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html
- https://docs.*oracle*.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html
- https://docs.spring.io/spring-security/site/docs/3.1.x/reference/springsecurity.html
- https://spring.io/guides/topicals/spring-security-architecture
- http://tutorials.jenkov.com/java-cryptography/index.html
- https://www.cloudflare.com/learning/security/what-is-web-application-security/
- https://www.baeldung.com/java-ssl
- https://www.baeldung.com/java-security-overview
- https://www.baeldung.com/spring-security-login
- https://dzone.com/articles/spring-security-authentication