Signed Code Sources

- Code signing is a security mechanism that performs digital signing of Java scripts and executables using cryptography algorithms to prevent malicious activities
- The object `java.security.CodeSource` includes a piece of code, `SecureClassLoader` (along with its subclasses) and `CodeSource` class are related with generating, modifying and handling code source objects
- The `CodeSource` class represents an original code execution point that includes (applets) codebase to encapsulate URL locations along with certificates used for verification of signed code
- Some implementation methods of the `CodeSource` class:

```
public CodeSource(URL url, Certificate certs[])
```

  - Creates a codesource for a set of certificates in a specified location
  - Creates a codesource for a set of code signers in a specified code location

Insecure Code for Signed Code Sources

- The `jarClassLoader` verifies the signature using the public key contained in a JAR file although the authenticity of this signature is still insufficient

JarRunner.java

```
1  package com.demo;
2
3  import java.io.IOException;
4  import java.lang.reflect.InvocationTargetException;
5  import java.lang.reflect.Method;
6  import java.lang.reflect.Modifier;
7  import java.net.JarURLConnection;
8  import java.net.URL;
9  import java.net.URLClassLoader;
10 import java.util.jar.Attributes;
11
12 public class JarRunner {
13 @SuppressWarnings("resource")
14 public static void main(String[] args)
15 throws IOException, ClassNotFoundException,
16 NoSuchMethodException, InvocationTargetException {
17 URL httpUrl = new URL(args[0]);
18 // Create class loader for the application jar file
19 JarClassLoader jcl = new JarClassLoader(httpUrl);
20 // Get the application's main class name
21 String sname = jcl.getMainClassName();
22 // Get arguments for the application
23 String[] argsts = new String[args.length - 1];
24 System.arraycopy(args, 1, argsts, 0, argsts.length);
25 // Invoke application's main class
26 jcl.invokeClass(sname, argsts);
```

```
27 }
28 }
29 final class JarClassLoader extends URLClassLoader {
30 private URL httpUrl;
31 public JarClassLoader(URL httpUrl) {
32 super(new URL[] { httpUrl });
33 this.httpUrl = httpUrl;
34 }
35 public String getMainClassName() throws IOException {
36     URL url = new URL("jar", "", httpUrl + "!/");
37     JarURLConnection juconn = (JarURLConnection) url.openConnection();
38     Attributes attrb = juconn.getMainAttributes();
39     return attrb != null ?
40     attrb.getValue(Attributes.Name.MAIN_CLASS) : null;
41     }
42     public void invokeClass(String sname, String[] args)
43     throws ClassNotFoundException, NoSuchMethodException,
44     InvocationTargetException {
45     Class<?> c1 = loadClass(sname);
46     Method m1 = c1.getMethod("main", new Class[] { args.getClass() });
47     m1.setAccessible(true);
48     int modifs = m1.getModifiers();
49 if (m1.getReturnType() != void.class || !Modifier.isStatic(modifs)
50 ||!Modifier.isPublic(modifs)) {
51     throw new NoSuchMethodException("main");
52 }
53 try {
54 m1.invoke(null, new Object[] { args });
55 } catch (IllegalAccessException e) {
56 System.out.println("Access denied");
57     } } }
```

Secure Code for Signed Code Sources

1. Secure code using Jarsigner

- In the code, `-verify` option (jarsigner) is used explicitly to check the JAR file signature at the command line
- `jarsigner -verify signed-updates-jar-file.jar`

2. Secure code using Certificate Chain

- Invoke class method is used to verify the signature i.e., by obtaining a chain of certificates from CodeSource class

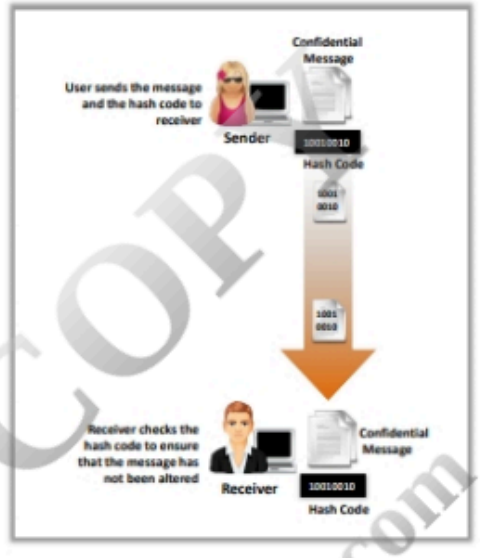
JarRunner.java

```
67@public void invokeClass(String sname, String[] args)
68 throws ClassNotFoundException, NoSuchMethodException,
69 InvocationTargetException, GeneralSecurityException,
70 IOException {
71 Class<?> c1 = loadClass(sname);
72 Certificate[] certf =
73 c1.getProtectionDomain().getCodeSource().getCertificates();
74 if (certf == null) {
75 // return, do not execute if unsigned
76 System.out.println("No signature!");
77 return;
78 }
79 KeyStore kyst = KeyStore.getInstance("JKS");
80 kyst.load(new FileInputStream(System.getProperty("user.home"+ File.sepa
81 "loadkeystorepassword".toCharArray());
82 // user is the alias
83 Certificate pubCertf = kyst.getCertificate("user");// check with th
84 certf[0].verify(pubCertf.getPublicKey());
```


Hashing

Hashing

- Hashing is one of the forms of **cryptography** that transforms the information into a **fixed-length value** or key that represents the original information
- The hashing technique ensures the **security of information** by checking the **integrity of information** on both the sender and receiver sides
- Checking the integrity of information:
  - The sender of the message creates a **hash code** of it and sends the message to the **receiver** along with its **hash code**
  - The receiver again creates a **hash code** for the same messages at the **receiver side** and compares both the hash codes; if it is a match, then the message is not tampered in the **transmission**. If they do not match, then message is tampered with by third parties during transmission



Hashing Algorithms

- To implement hashing in Java include **java.security.MessageDigest** class
- **MD2, MD5, SHA-1, SHA-265, SHA-384, SHA-512, PBKDF2, bcrypt,** and **scrypt** are some of the hashing algorithms used

Example: Hashing using MD5

Hashing.java

```
1  package com.demo;
2
3  import java.security.MessageDigest;
4
5  public class Hashing {
6  public static void main(String[] args) {
7
8      String passwordToHash="Password@123";
9      String GeneratedPassword=null;
10     try {
11         MessageDigest md=MessageDigest.getInstance("MD5");
12         md.update(passwordToHash.getBytes());
13         byte[] bytes=md.digest();
14         StringBuilder sb=new StringBuilder();
15         for(int i=0;i<bytes.length;i++)
16         {
17             sb.append(Integer.toString((bytes[i]&0xff)+0x100,16).substring(1));
18         }
19         GeneratedPassword=sb.toString();
20         System.out.println("Hashed password="+GeneratedPassword);
21
22     } catch (Exception e) {
23         e.getMessage();
24     }}}
```

Securing Hashed Password with Salt

- Hashing the same password results in the same hash every time which makes it **vulnerable** to **Dictionary** and **Brute Force Attacks**
- Rainbow tables, lookup tables and reverse lookup tables are used to **crack Hash codes**
- By adding a random string (**salt**) before implementing hashing results in generating random hash for the same string (**password**)

Do's and Don'ts while Implementing Salting

- Do not use **hard-coded salt**
- Re-Generate a **unique salt** each time
- Ensure that the size of the salt is big enough to generate numerous possible salts, so that the attacker cannot generate a Lookup table to **crack the passwords**
- Do not implement **double hashing**

Securing Hashed Password with Salt (Cont'd)

Example: Implementing Salting before Hashing

Hashing.java

```
1  package com.domo;
2
3  import java.security.MessageDigest;
4  import java.security.NoSuchAlgorithmException;
5  import java.security.SecureRandom;
6
7  public class Hashing {
8  public static void main(String[] args) throws NoSuchAlgorithmException {
9
10     String passwordToHash="Password@123";
11     String GeneratedPassword=null;
12     byte[] salt = getSalt();
13     try{
14         MessageDigest md=MessageDigest.getInstance("MD5");
15         md.update(salt);
16         byte[] bytes=md.digest();
17         StringBuilder sb=new StringBuilder();
18         for(int i=0;i<bytes.length;i++)
19         {
20             sb.append(Integer.toString((bytes[i]&0xff)+0x100,16).substring(1));
21         }
22         GeneratedPassword=sb.toString();
23         System.out.println("Hashed password="+GeneratedPassword);
24
25     } catch (Exception e) {
26         e.getMessage();
27     }}
28
29 private static byte[] getSalt() throws NoSuchAlgorithmException {
30     SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
31     byte[] salt = new byte[16]; sr.nextBytes(salt);    return salt;
```

Implementing Hashing with Salt in Spring Security

- Spring uses the following interfaces for password hashing and salting
  - `org.springframework.security.providers.encoding.PasswordEncoder`
  - `org.springframework.security.authentication.dao.SaltSource`

Vulnerable Bean Configuration

spring-security.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">


   <bean id="passwordEncoder" class="org.springframework.security.authentication.encoding.ShaPasswordEncoder">
   <constructor-arg value="160"/>
</bean>
<bean id="saltSource" class="org.springframework.security.authentication.dao.SystemWideSaltSource">
   <property name="systemWideSalt" value="#Saltvalue;" />
</bean>
</beans>
```

Implementing Hashing with Salt in Spring Security(Cont'd)

- Define a bean configuration in applicationContext.xml file to implement hashing
- ShaPasswordEncoder by default generates a 160 bit hash. Change the constructor-arg value to 256 to generate a 256 bit hash
- systemWideSalt property defines the value to be used as salt. Include a long salt for generating random salt

Secure Bean Configuration

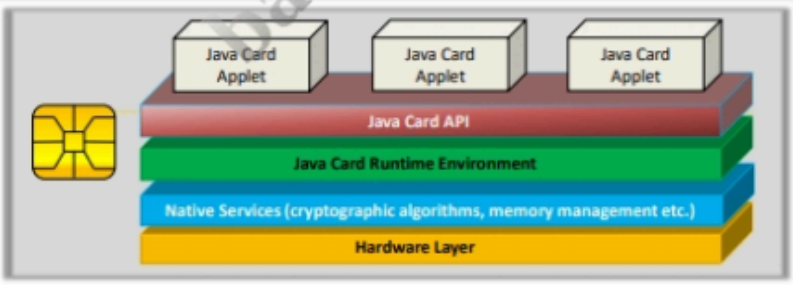
spring-security.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
<bean id="passwordEncoder" class="org.springframework.security.authentication.encoding.ShaPasswordEncoder">
<constructor-arg value="256"/>
</bean>
<bean id="saltSource" class="org.springframework.security.authentication.dao.SystemWideSaltSource">
<property name="systemWideSalt" value="#This,Is9087%49A3Salt*Value;" />
</bean>
</beans>
```

Java Card Cryptography

Java Card Cryptography

- Java Card specifications includes Java Card Virtual Machine specification(JCVM), Java Card Runtime Environment Specification (JCRE) and Java Card Application Programming Interface (API)
- Java Card Applet is a kind of Java application running on similar resource-constrained devices with a subset specifications of Java technology
- There are three major APIs defined in Java card platform:
  - javacard.framework
  - javacard.security
  - javacardx.crypto



Java Card Cryptography (Cont'd)

javacard.security

Interfaces

- AESKey
- DESKey
- DHKey
- DHPrivateKey
- DHPublicKey
- DSAKey
- DSAPrivateKey
- DSAPublicKey
- ECKey
- ECPrivateKey
- ECPublicKey
- HMACKey
- Key
- KoreanSEEDKey
- PrivateKey
- PublicKey
- RSAPrivateCrtKey
- RSAPrivateKey
- RSAPublicKey
- SecretKey
- SignatureMessageRecovery

Classes

- Checksum
- InitializedMessageDigest
- InitializedMessageDigest.OneShot
- KeyAgreement
- KeyBuilder
- KeyPair
- MessageDigest
- MessageDigest.OneShot
- RandomData
- RandomData.OneShot
- Signature
- Signature.OneShot

javacardx.crypto

Interfaces

- KeyEncryption

Classes

- AEADCipher
- Cipher
- Cipher.OneShot

Java Card Cryptography (Cont'd)

Generate Random Number in Applet on Smart Card

PasswordEncoding.java

```
27      {
28          //Generate Random Number
29          byte[] random = JCSystem.makeTransientByteArray((short)8, JCSystem.CLEAR_ON_DESELECT);
30          RandomData generator = RandomData.getInstance((byte)RandomData.ALG_FAST);
31          generator.setSeed(random, (short)0, (short)8);
32          generator.nextBytes(random, (short)0, (short)8);
```

Generate Message Digest in Applet on Smart Card

PasswordEncoding.java

```
11      public void numgen()
12      {
13      //Generate Random Number
14      byte[] digest = JCSystem.makeTransientByteArray((short)20, JCSystem.CLEA
15      MessageDigest alg = MessageDigest.getInstance(MessageDigest.ALG_SHA, fal
16      //buffer is the tmp data property (byte[]) in applet
17
18      alg.doFinal(buffer, (short)5, (short)8, digest, (short)0);
19      }
20
21  }
```

Java Card Cryptography (Cont'd)

Generate/Verify RSA signature in Applet on Smart Card

```
public boolean verifyRSAsign()
{
    //Create KeyPair Object
    KeyPair kp = new KeyPair(KeyPair.ALG_RSA, (short)512);
    //Generate KeyPair
    kp.genKeyPair();
    RSAPrivateKey privateKey = (RSAPrivateKey)kp.getPrivate();
    RSAPublicKey publicKey = (RSAPublicKey)kp.getPublic();
    //sign
    Signature sig = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
    sig.init(privateKey, Signature.MODE_SIGN);
    sig.sign(dataBytes, (short)0, (short)dataBytes.length, signedBytes, (short)0);
    //verify signature
    sig.init(publicKey, Signature.MODE_VERIFY);
boolean verify = sig.verify(signedBytes, (short)0, (short)signedBytes.length, buffer, (short)0, (short)(publicKey.getSize()/8));
    return verify;

}
```


Java Card Cryptography (Cont'd)

Generate / Verify ECC signature in Applet on Smart Card

PasswordEncoding.java

```
KeyPair kp = new KeyPair(KeyPair.ALG_EC_FP, KeyBuilder.LENGTH_EC_FP_192);
kp.genKeyPair();
ECPrivateKey privateKey = (ECPrivateKey)kp.getPrivate();
ECPublicKey publicKey = (ECPublicKey)kp.getPublic();
//sign
Signature sig = Signature.getInstance(Signature.ALG_ECDSA_SHA, false);
sig.init(privateKey, Signature.MODE_SIGN);
sig.sign(dataBytes, (short)0, (short)dataBytes.length, signedBytes, (short)0);
//verify signature
sig.init(publicKey, Signature.MODE_VERIFY);
boolean verify = sig.verify(signedBytes, (short)0, (short)signedBytes.length, buffer, (short)0, (short)(publicKey.getSize()/8));
```

.AES in Applet on Smart Card

PasswordEncoding.java

```
public void AESinApplet()
{
    //Create AESKey Object
    AESKey key = (AESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_AES, KeyBuilder.LENGTH_AES_128, false);
    //Create Cipher object
    Cipher cipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD, false);
    //encryption
    cipher.init(key, Cipher.MODE_ENCRYPT);
    cipher.doFinal(dataBytes, (short)dataOffset, (short)dataLength, encryptedBytes, (short)0);
    //decryption
    cipher.init(key, Cipher.MODE_DECRYPT);
    cipher.doFinal(encryptedBytes, (short)0, (short)encryptedLength, dataBytes, (short)0);
```

Crypto Module

- Spring crypto module provides classes for
  - symmetric encryption
  - key generation
  - password encoding

Encryptors

Set of classes for creating

- ByteEncryptor
  - Uses 256-bit AES using PKCS #5's PBKDF2 (Password-Based Key Derivation Function #2) for encryption
  - Returns encrypted data in raw byte[] form
- TextEncryptor
  - Implements standard BytesEncryptor for encryption and returns result as hex-encoded strings
- Queryable TextEncryptor
  - The initialization vector for encryption is constant and generates the same encryption result for the same text
  - It should be implemented when the previously generated hash needs to be matched with the generated hash for the same text

Spring Security Crypto Module: Key Generators

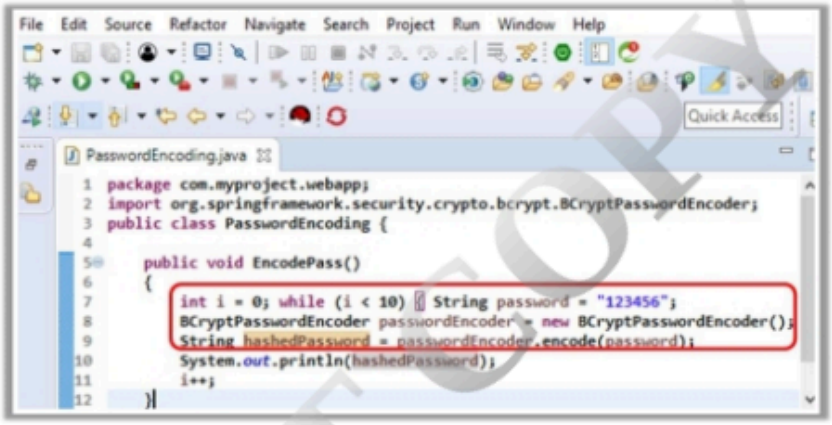
- Set of classes for creating following types of key generators
  - **BytesKeyGenerator**
    - generates **byte[]** keys
    - **KeyGenerators.secureRandom()** method generates a key of **8 bytes** by default
    - Specify keysize while implementing **KeyGenerators.secureRandom(16)** for generating large size keys
    - **KeyGenerators.shared()** method generates the **same key** every time
  - **StringKeyGenerator**
    - generates **hex-encoded string** keys
    - **KeyGenerators.string()** generates **8-byte** long string key by default

Spring Security Crypto Module: PasswordEncoder

- Provide password encoding feature by implementing **PasswordEncoder** interface
- **StandardPasswordEncoder**, **Md5PasswordEncoder** and **BCryptPasswordEncoder** are some of the password encoders supported in spring security
  - **StandardPasswordEncoder**
    - Combines a plain password with a site-wide secret and 8-byte random salt, and implements SHA-256 hashing algorithm with 1024 iterations
    - **Site-wide Key** should not be stored along with the passwords to prevent attackers from gaining access to the stored passwords using a brute force attack
    - 1024 iterations enables creating a unique and strong key
    - The **random salt** ensures that a unique hash is generated ~~on~~ when the same text is supplied multiple times
  - **BCryptPasswordEncoder**
    - **BCryptPasswordEncoder** implements bcrypt hashing function based on Blowfish cipher
    - **Bcrypt** generates random salt to generate unique encoded hash of length 60 characters for the same text

Implementing BCryptPasswordEncoder()

- Sample Code for Implementing **BCryptPasswordEncoder()**

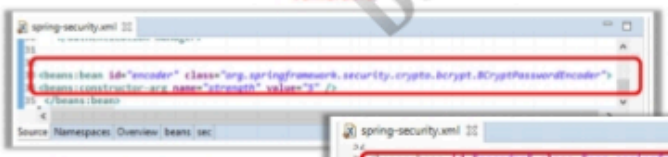


Configuring BCryptPasswordEncoder() in Spring Security

- Configure the authentication-provider to implement **BCryptPasswordEncoder** hashing function in **spring.security.xml**

- To secure the **BCryptPasswordEncoder** give a high value to the strength parameter of the BCryptPasswordEncoder

Vulnerable



Secure

JavaScript Object Signing and Encryption (JOSE)

JOSE is a framework for implementing Cryptography to JSON messages
Used for Client authentication and authorization
Supported by various programming languages

- JWS: JSON Web Signature
  - Securing JSON data using digital signature or Message Authentication Code (MAC) using base64 URL encoding
  - Used for Authorization headers and URI query parameters
  - Simple and compact
  - No canonicalization

  JWS Format: <Header> + <.> + Payload + <.> + Signature.
- JWE: JSON Web Encryption
  - Encryption of JSON
  - Complicated

  JWE Format: <Header> + <.> + Encrypted Key + <.> + Initialization Vector + <.> + Cipertext + <.> + Authentication Tag
- JWT: Java Web Token
  - JWT is a JWS or / and JWE with JSON claim (Name / value pair) transferred between clients
- JWK: JSON Web Key
  - Public cryptographic keys containing metadata
  - Can be added to JWS, JWE or JWT header
  - Published at HTTPS endpoint
  - Replacement of self signed certificate
  - Saved in files

- To implement JOSE in Spring include the following packages
  - org.springframework.security.oauth2.jwt
  - org.springframework.security.oauth2.jose
- JWA : JSON Web Algorithms
  - List of Crypto algorithms used for JWE / JWS
  - The "alg" parameter of JWS denotes the algorithm
  - The "alg" and "enc" parameter of JWE denotes the algorithm

List of Supported Algorithm - JWS

Alg parameter	Algorithm
HS256	HMAC implements SHA-256
HS384	HMAC implements SHA-384
HS512	HMAC using SHA-512
RS256	RSASSA-PKCS1-v1_5 using SHA-256
RS384	RSASSA-PKCS1-v1_5 using SHA-384
RS512	RSASSA-PKCS1-v1_5 using SHA-512
ES256	ECDSA using P-256 and SHA-256
ES384	ECDSA using P-384 and SHA-384
ES512	ECDSA using P-521 and SHA-512
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512
none	No digital signature or MAC value

Attacks against JWT, JWS and JWE

- **Signature Exclusion Attack:** The attacker removes the signature of a signed message, makes it a unsigned message and hence, the message is considered as valid
- **Countermeasures:** Implementing no algorithm specifies that the JWS payload is insecure or the payload is not secured by digital signature or MAC value

- **Key Confusion (Algorithm Substitution):** The attacker replaces the key with a known crypto key for implementing a known algorithm
- **Countermeasures:** Using an algorithm parameter in the signature verification function. This will determine the actually implemented algorithm and detect any change found

  Verify (string token , string algorithm , string verificationKey)

- **Bleichenbacher Million Message Attack:** Ciphertext attack on protocols based on the RSA encryption, where the attacker sends multiple ciphertexts to the server and studies the received response error messages, timing differences etc. to identify valid and invalid padded messages and extracts sensitive information in the payload
- **Countermeasures:** Implement alternative Optimal Asymmetric Encryption Padding (OAEP) technique. Respond consistently to all errors by checking the length of CEK and parity bits. Random Filling where the misformatted messages are considered properly PKCS-1 formatted

Implementing JWS using Jose4J

JOSE4J is an Open source library provided by Apache to implement JWS, JWT, JWK

Generating JWS using Jose4J

```
package joseImpl;

import org.jose4j.jwk.EcJwkGenerator;
import org.jose4j.jwk.PublicJsonWebKey;
import org.jose4j.jws.AlgorithmIdentifiers;
import org.jose4j.jws.JsonWebSignature;
import org.jose4j.keys.EllipticCurves;
import org.jose4j.lang.JoseException;

public class JoseImpl {

    public void Jose() throws JoseException {
    PublicJsonWebKey jwk = EcJwkGenerator.generateJwk(EllipticCurves.P256);
    JsonWebSignature jws = new JsonWebSignature();
    jws.setAlgorithmHeaderValue(AlgorithmIdentifiers.ECDSA_USING_P256_CURVE_AND_SHA256); jws.setPayload("USA #1!");
    jws.setKey(jwk.getPrivateKey()); jws.setKeyIdHeaderValue(jwk.getKeyId());
    String StrDenseSerialization = jws.getCompactSerialization();
    System.out.println(StrDenseSerialization);
    }
```

Consuming JWS using Jose4J

```
public void Jose1() throws JoseException
{
    JsonWebKey jwk = JsonWebKey.Factory.newJwk("{\"kty\":\"EC\",\" + \"kid\":\"my-first-key\",\" "
            + "+ \"\"x\":\"xJKTWTx76f190Zow4LMpDc3oHLC_vm-db7mdsFvO13Q\",\" + "
            + "\"\"y\":\"jXBG649Qqf7pf8HO_jcJ8Jrhy23hjD933i6QEVMkk\",\" +"
            + " \"\"crv\":\"P-256\"}");
    String StrDenseSerialization = "eyJhbGciOiJFUzI1NiIsImtpZCI6Im15LWZpcnN0LWtleSJ9." +
            "VVNBICMxIQ." +
            "QJGB_sMj-w3yCBunJs2uxKgvZgG2Hq9PA-TDQEbWdTm2mnj2sUSr8KZJAUREzF1FF25BbrgyohbGdGE1cB-hrA";
    JsonWebSignature jws = new JsonWebSignature();
    jws.setCompactSerialization(StrDenseSerialization);
    jws.setKey(jwk.getKey()); String payload = jws.getPayload(); System.out.println(payload);
}
```

Implementing JWE using Jose4J (Cont'd)

Generating JWE using Jose4J

```
public void Jose2() throws JoseException
{
    JsonWebEncryption jwe = new JsonWebEncryption();
    jwe.setPayload("I actually really like Canada");
    jwe.setKey(new PbkdfKey("don't-tell-p@ul|pam!"));
    jwe.setAlgorithmHeaderValue(KeyManagementAlgorithmIdentifiers.PBES2_HS256_A128KW);
    jwe.setEncryptionMethodHeaderParameter(ContentEncryptionAlgorithmIdentifiers.AES_128_CBC_HMAC_SHA_256);
    String StrDenseSerialization = jwe.getCompactSerialization();
    System.out.println(StrDenseSerialization);
}
```

Consuming JWE using Jose4J

```
public void Jose3() throws JoseException
{
    String StrDenseSerialization = "eyJhbGciOiJQQkVTMi1IUzI1NiIsImtBMTI4S1ciLCJlbmMiO"
            + "iJBMTI4QQ3DLUhTMjU2IiwicDJjIjo4MTkyLCJwMnMiOiJRa2JMUW5pS0xVVFFWUDR sIn0." +
            "g7s-MxHFnSWHCfO33hgWYiAtH1lB83TnuFWoaFIEujEYb14pqeH9Mg." + "Gh172lwv9VqemjMQMaVPdg."
            "YMg_FBaoT3ZByou3CURhKraGX1nc5QJDoJcWyUSyow0." + "Ie4iYLbdQCquMNJf37rEZg";
    JsonWebEncryption jwe = new JsonWebEncryption();
    jwe.setCompactSerialization(StrDenseSerialization);
    jwe.setKey(new PbkdfKey("don't-tell-p@ul|pam!"));
    String payload = jwe.getPayload(); System.out.println(payload);
}
```

Implementing JWK using Jose4J (Cont'd)

Generating JWK using Jose4J

JoseImpl.java

```
71  public void Jose04()
72  {
73      List<JsonWebKey> jwkList = new LinkedList<JsonWebKey>();
74      for (int kid = 4; kid < 7; kid++)
75          {
76              JsonWebKey jwk = EcJwkGenerator.generateJwk(EllipticCurves.P256);
77              jwk.setKeyId(String.valueOf(kid)); jwkList.add(jwk);
78          }
79      JsonWebKeySet jwks = new JsonWebKeySet(jwkList);
80      System.out.println(jwks.toJson(JsonWebKey.OutputControlLevel.PUBLIC_ONLY));
81      }
```

Consuming JWK using Jose4J

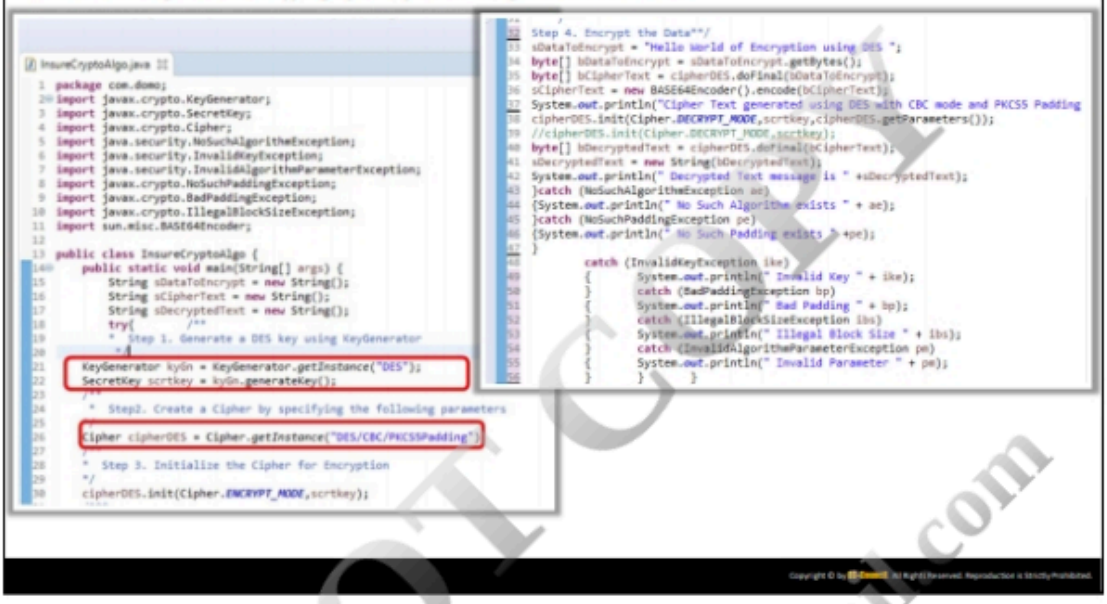
JoseImpl.java

```
83
84  public void Jose05() throws JoseException
85  {
86      String jwksJson = "{\"keys\":[\n" +
87              + " \" {\"kty\":\"EC\",\n\"kid\":\"4\",\n\" +
88              + "\" \"x\":\"LX-7aQnJRAx3jDDTioNssbODUfID_6XvZP8NsGzMlRo\", n\" +"
89              + " \" \"y\":\"d3bHEoeHzeZPYuz6qJKJoRVLks7X8-BJXbevfyoJQ-A\",n\" + "
90              + "\" \"crv\":\"P-256\"},n\" + \" {\"kty\":\"EC\",n\"kid\":\"5\",n\" + "
91              + "\"x\":\"f83013D2xF1Bg8vub9tLe1gHMiW76e8Tus9uPHvRVEU\",n\" +"
92              + " \" \"y\":\"x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a8\",n\" +"
93              + " \" \"crv\":\"P-256\"},n\" + \" {\"kty\":\"EC\",n\"kid\":\"6\",n\" +"
94              + " \" \"x\":\"J82J37uc12Y3AzArSdqIj4Qgr0CCFuZ18WI77jsi50B\",n\" +"
95              + " \" \"y\":\"StTavax8aRMMJ4unKdKsV0ucf3pOI3OG771gOo45uBU\",n\""
96              + " + \" \"crv\":\"P-256\"}n\" + \"]}";
97      JsonWebKeySet jwks = new JsonWebKeySet(jwksJson);
98      JsonWebKey jwk = jwks.findJsonWebKey("5", null, null, null);
99      System.out.println(jwk.getKey());
```


Dos and Don'ts in Java Cryptography

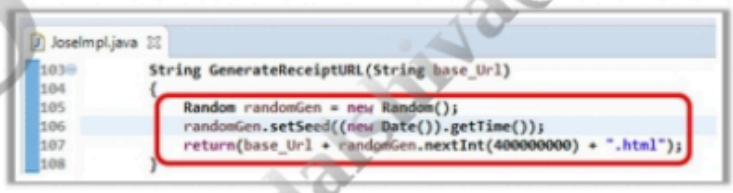
Dos and Don'ts: Avoid using Insecure Cryptographic Algorithms

1. The following code uses cryptographically insecure algorithms such as DES

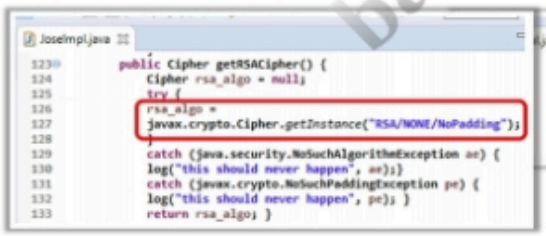


Dos and Don'ts: Avoid using Statistical PRNG, Inadequate Padding and Insufficient Key Size

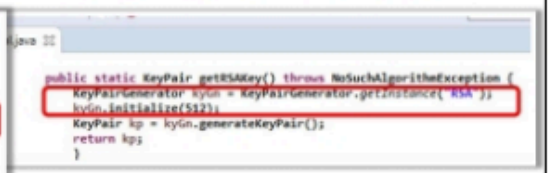
2. This code uses statistical PRNG to create a URL for a receipt which remains active for a certain period of time after purchase



3. Don't use inadequate/inappropriate RSA padding



4. Don't use insufficient/inappropriate key size



Dos and Don'ts: Implement Strong Entropy

1. The following code uses strong entropy for key generation using Java cryptographic extensions

```
149
150
151⊖    public static void main(String[] args) {
152         try {
153         // Initialize a secure random number generator
154         SecureRandom secRan = SecureRandom.getInstance("SHA1PRNG");
155         // Method 1 - Calling nextBytes method to generate Random bytes
156         byte[] b = new byte[512];
157
158         secRan.nextBytes(b);
159         // Printing the SecureRandom number by calling secRan.nextDouble()
160         System.out.println("Secure Random num generated by calling nextBytes() is " + secRan.nextDouble());
161         // Method 2 - Using setSeed(byte[]) to reseed a Random object
162         int ByteCount_seed = 10;
163         byte[] bseed = secRan.generateSeed(ByteCount_seed);
164          System.out.println(" Seed value is " + new BASE64Encoder().encode(bseed));
165         secRan.setSeed(bseed);
166         System.out.println(" Secure Random num generated using setSeed(byte[]) is  " +
167         secRan.nextDouble());
168         } catch (NoSuchAlgorithmException ae)
169         {
170         System.out.println(" No Such Algorithm exists " + ae);
```


Dos and Don'ts: Implement Strong Algorithms

2. The following code uses strong algorithm using Java cryptographic extensions

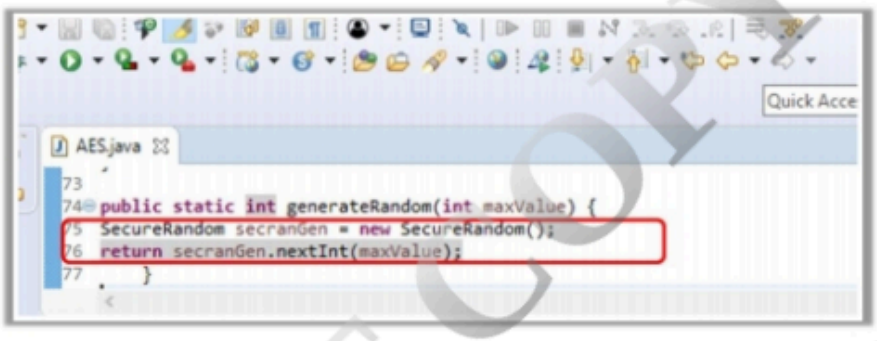
```
package com.domo;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
import java.security.InvalidAlgorithmParameterException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import sun.misc.BASE64Encoder;
public class AES {
public static void main(String[] args) {
String sDataToEncrypt = new String();
String sCipherText = new String();
String sDecryptedText = new String();
try{
/**
*  Step 1. Generate an AES key using KeyGenerator
*  Initialize the keysize to 128 */
    KeyGenerator kyGn = KeyGenerator.getInstance("AES");
    kyGn.init(128);
    SecretKey scrtkey = kyGn.generateKey();
    /**
    *  Step2. Create a Cipher by specifying the following parameters
    */
    Cipher aesCipher = Cipher.getInstance("AES");
    /**
    * Step 3. Initialize the Cipher for Encryption */
    aesCipher.init(Cipher.ENCRYPT_MODE,scrtkey);
    /**
    *  Step 4. Encrypt the Data */
    sDataToEncrypt = "Hello World of Encryption using AES ";
    byte[] bDataToEncrypt = sDataToEncrypt.getBytes();
    byte[] bCipherText = aesCipher.doFinal(bDataToEncrypt);
    sCipherText = new BASE64Encoder().encode(bCipherText);
    System.out.println("Cipher Text generated using AES is " +sCipherText);
```

```
    /**
    *  Step 5. Decrypt the Data
    */
    aesCipher.init(Cipher.DECRYPT_MODE,scrtkey,aesCipher.getParameters());
    byte[] bDecryptedText = aesCipher.doFinal(bCipherText);
    sDecryptedText = new String(bDecryptedText);
    System.out.println(" Decrypted Text message is " +sDecryptedText);
    }
    catch (NoSuchAlgorithmException ae)
    {
    System.out.println(" No Such Algorithm exists " + ae);
    }
    catch (NoSuchPaddingException pe)
    {
    System.out.println(" No Such Padding exists " + pe);
    }
    catch (InvalidKeyException ike)
    {
    System.out.println(" Invalid Key " + ike);
    }
    catch (BadPaddingException bp)
    {
    System.out.println(" Bad Padding " + bp);
    }
    catch (IllegalBlockSizeException ibs)
    {
    System.out.println(" Illegal Block Size " + ibs); }
    catch (InvalidAlgorithmParameterException pe)
    {
    System.out.println(" Invalid Parameter " + pe);
    }
```

Dos and Don'ts: Implement Strong Algorithms (Cont'd)

3. Use Java's SecureRandom class

- This code uses Java's SecureRandom class to generate cryptographically strong pseudo-random number

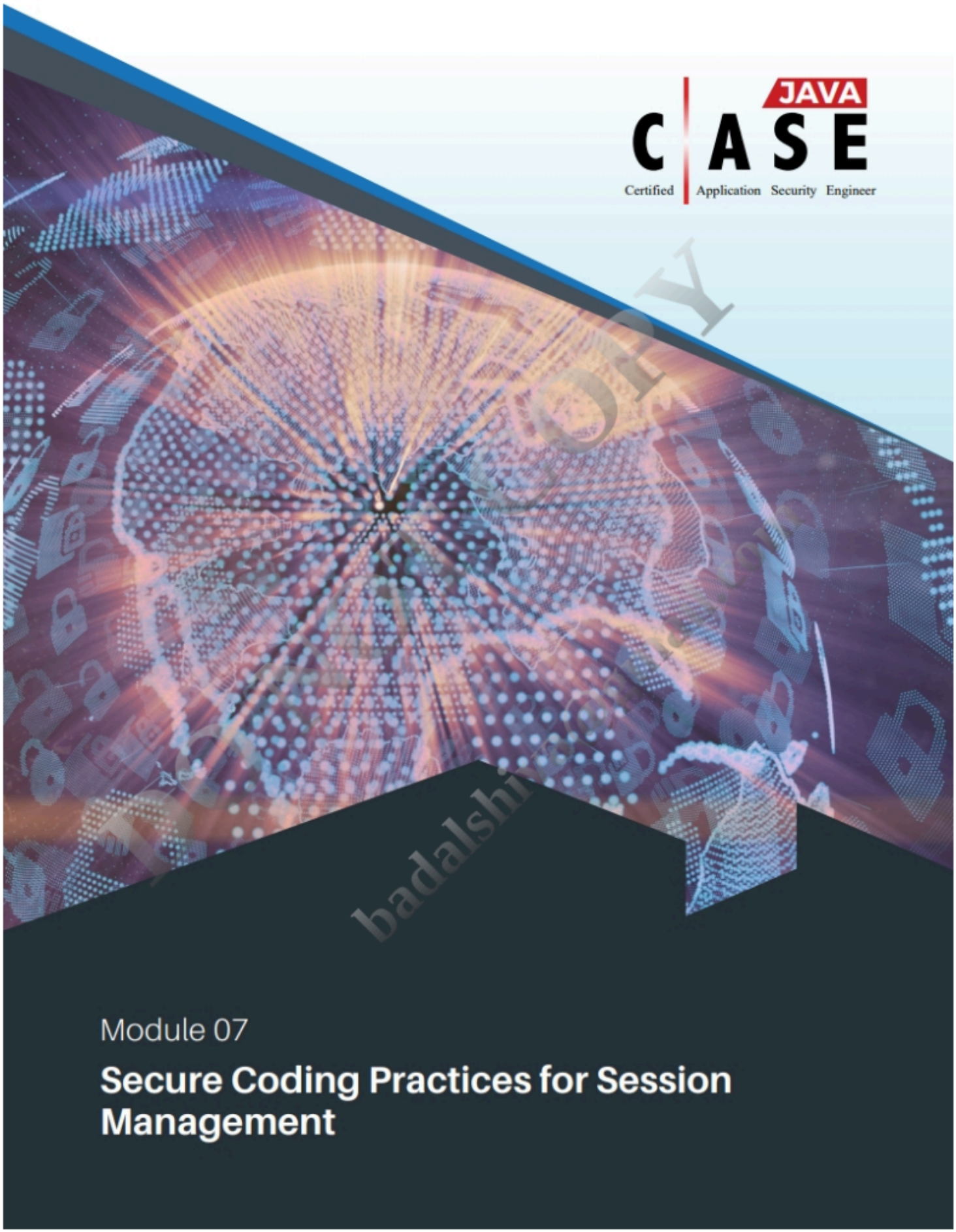


Best Practices for Java Cryptography

- ✓ Securely manage keys that are used for code signing and sealing
- ✓ Use existing crypto libraries rather than creating your own cryptographic protocols
- ✗ Don't use pseudorandom number generators instead use cryptographically secured random numbers
- ✗ Don't store encrypted keys and passwords that are dependent on garbage collector in memory; rather use configuration files with authorized entities
- ✓ Securely manage objects that are transmitted across the network with signing and sealing
- ✓ Ensure that the key length used by algorithms is at least 128 bits (Eg, RSA key 512 bits)
- ✗ Don't store encryption keys, keystore name, the alias and password details in the source code
- ✗ Don't store encryption keys, keystore name, password, alias, etc., in unprotected external files

Module Summary

- Cryptography deals with security issues in accordance with privacy, integrity, authentication and nonrepudiation
- Java platform offers cryptographic operations using APIs such as Java Cryptography Architecture and Java Cryptography Extension
- The Cipher Class in `javax.crypto` package performs the main functionality of cryptographic cipher using encryption and decryption
- JCA provides a specific framework for digital signatures with `java.security.Signature` class that provides functionality of signing and verifying digital signatures
- Digital certificate includes User (entity) Information, User's public key, Digital signature of the CA, Issue and expiry date
- Java Archive (JAR) is a file format that contains multiple files .i.e., class files and subsidiary resources associated with applets and applications
- Code signing is a security mechanism that performs digital signing of Java scripts and executables using cryptography algorithms to prevent malicious activities



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

This page is intentionally left blank.

Module Objectives

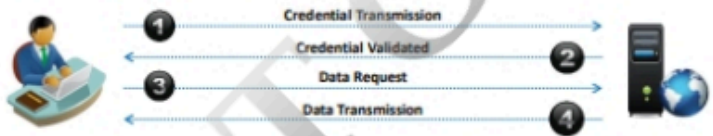
1. Introduction to Session Management in Java
2. Discuss Session Management in Spring Framework
3. Discuss Session Vulnerabilities and their Mitigation Techniques
4. Learn Best Practices and Guidelines for Secure Session Management

Session Management

- A session can be defined as a number of **requests** made by a particular client over a period of time
- Session management can be used to keep track of the information of a **web user** in a session, such as number of requests etc.
- Java session management is helpful in **storing** the information of user, application security and for timing out a session
- Improper authentication and session management results in disclosure of the users' identity by stealing passwords, keys and session tokens

Session Tracking

1. A session can be defined as a conversation between a **server** and a **client**
2. When there is a series of continuous requests and responses from the same client to a server, the server cannot identify from which client it is getting requests as **HTTP is a stateless protocol**
3. When there is a need to maintain the conversational state, **session tracking is needed**
4. Different methods of session tracking are **Cookies**, **URL Rewriting**, **Hidden Fields**, **Session Objects** etc.



Session Tracking Methods: HttpSession

- HttpSession helps to store the full session object for a specific client
- HttpSession provides some methods which help to store attributes and remove attributes from a session

The following code shows how to create a new HttpSession object

SessionTracking.java

```
35
36
37  HttpSession session=request.getSession();
38  session.setAttribute("user", uname);
39
```

- This code creates a **session object** with a name of "session" and an attribute user with value name
- The session can be invalidated with the `void invalidate()` method

List of the HttpSession methods to help manage session

- **long getCreationTime()**: It will return session creation time in milliseconds
- **String getId()**: It will return the unique identifier of the session
- **long getLastAccessedTime()**: Returns the time of the last request for the session
- **int getMaxInactiveInterval()**: Returns the session active time
- **void invalidate()**: This method will destroy the session
- **boolean isNew()**: Returns value true or false of session creation
- **void setMaxInactiveInterval(int interval)**: Active time of session in container

Session Tracking Methods: Cookies

- A cookie, also known as an **HTTP cookie**, **web cookie**, or **browser cookie**, is a piece of data stored by a website within a browser, and then subsequently sent back to the same website by the browser
- Cookies were designed to be a **reliable mechanism** for websites to remember things that a browser had done there in the past

The following code shows how to create a new cookie

SessionTracking.java

```
Cookie user = new Cookie("username","Jummy");
user.setMaxAge(3600);
response.addCookie(user);
```

- This code creates a cookie with a name of "username" and a value of "Jummy"
- The cookie's expiration date is set with the `setMaxAge()` method to 3,600 seconds from the time the browser receives the cookie

The following code demonstrates how you would retrieve the value for a specific cookie

SessionTracking.java

```
String user1 = "";
Cookie[] ck = request.getCookies();
if (ck != null) {
for (int x = 0; x < ck.length; x++) {
if (ck[x].getName().equals("user"))
user1 = ck[x].getValue();
```

- In this code, an array of cookies is retrieved from the `HttpServletRequest` object using the `getCookies()` method
- The array is walked through until a cookie with the name of "user" is returned by the `getName()` method
- Once the cookie is found, the `getValue()` method is called to retrieve the value of the cookie

Setting a Limited Time Period for Session Expiration

- Configure `session-timeout` attribute for setting session duration
- If the time period of session is set for a shorter duration, then the attacker gets less time to steal the **cookies**, thus reducing the risk of **stolen cookies**
- To implement session-timeout in JAVA configuration use **javax.servlet.http.HttpSession** API

Vulnerable Code

SessionTracking.java | web.xml

```
<session-config>
<session-timeout>60</session-timeout>
</session-config>
</web-app>
```

Design | Source

Secure Code

SessionTracking.java | web.xml

```
<session-config>
<session-timeout>10</session-timeout>
</session-config>
</web-app>
```

Design | Source

Setting Session Duration using JAVA Configuration

Vulnerable

SessionTracking.java

```
HttpSession session=request.getSession();
session.setAttribute("user", uname);
session.setMaxInactiveInterval(60*60);
```

Secure

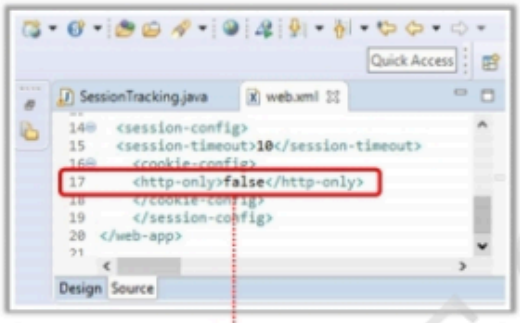
SessionTracking.java

```
HttpSession session=request.getSession();
session.setAttribute("user", uname);
session.setMaxInactiveInterval(10*60);
```

Preventing Session Cookies from Client-side Scripts Attacks

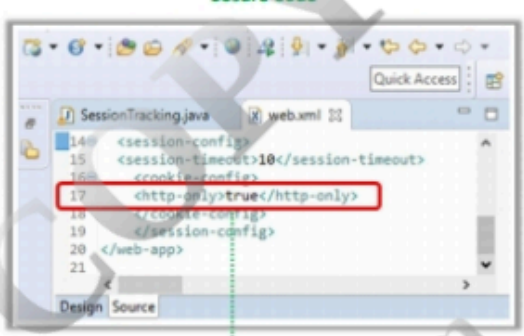
- To prevent the client script from accessing the session cookie, set the `httpOnlyCookie` attribute to `true` in the **web.xml** file

Vulnerable Code



- Setting `http-only` to `false`, then cookie is accessible from client script

Secure Code



- Setting `http-only` to `true`, then cookie is not accessible from client script

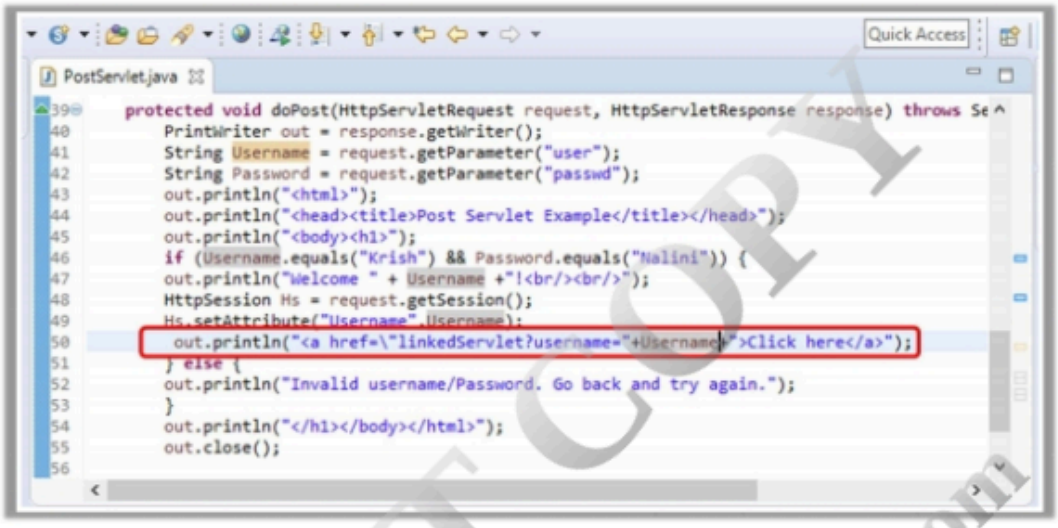
Session Tracking Methods: URL Rewriting

I. URL rewriting is used where either a **browser** or a **wireless device** doesn't accept cookies. In this case, session tracking isn't possible by using cookies

II. When the web server detects that the browser is not accepting cookies then the session ID is **encoded** into the hyperlinks on the **webpages**

III. Web server **extracts** the **ID** from the **URL address** and traces out the HTTP session using the **getSession()** method

Example Code for URL Rewriting



```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws Se
    PrintWriter out = response.getWriter();
    String Username = request.getParameter("user");
    String Password = request.getParameter("passwd");
    out.println("<html>");
    out.println("<head><title>Post Servlet Example</title></head>");
    out.println("<body><h1>");
    if (Username.equals("Krish") && Password.equals("Nalini")) {
    out.println("Welcome " + Username +"!<br/><br/>");
    HttpSession Hs = request.getSession();
    Hs.setAttribute("Username",Username);
     out.println("<a href=\"linkedServlet?username="+Username+">Click here</a>");
    } else {
    out.println("Invalid username/Password. Go back and try again.");
    }
    out.println("</h1></body></html>");
    out.close();
```

Session Tracking Methods: Hidden Fields

- Hidden form fields are basically used for session maintenance and also used for **tracking sessions**
- It supports almost all the browsers and does not need any special **server requirements**
- Limitation of the hidden field is, it works only for a sequence of dynamically **generated forms**

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws Ser
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HEAD><TITLE>Shopping Cart  Present</TITLE></HEAD>");
    out.println("<BODY>");
    // Cart objects are being passed as the item parameter.
    String[] objects = request.getParameterValues("item");
    // Print the current cart objects.
    out.println("You currently have the following objects in your cart:<BR>");
    if (objects == null) {
    out.println("<B>None</B>");
    }
    else {
    out.println("<UL>");
    for (int a = 0; a < objects.length; a++) {
    out.println("<LI>" + objects[a]);
    }
    out.println("</UL>");
    }
    // Ask if the user wants to add more objects or check out.
    // Include the current objects as hidden fields so they'll be passed on.
    out.println("<FORM ACTION=\"/servlet/ShoppingCart\" METHOD=POST>");
    if (objects != null) {
    for (int a = 0; a < objects.length; a++) {
    out.println("<INPUT TYPE=hidden NAME=object VALUE=\"" +
    objects[a] + "\">");
    } }
    out.println("Would you like to<BR>");
    out.println("<INPUT TYPE=submit VALUE=\" Add More Items \">");
    out.println("<INPUT TYPE=submit VALUE=\" Check Out \">");
    out.println("</FORM>");
    out.println("</BODY></HTML>");
```

Session Tracking Methods: Session Objects

1. Session object is used for the maintenance of a user session related information on the **server side**
2. Session object is used to **store**, **retrieve** and **remove information** based on program logic
3. Each session object created for a user persists until the user **logs out** or the user remains idle for the **session expiration time**
4. Basically on an average a typical session lasts for **10 minutes** by **default**

Session Management in Spring Framework

Spring Session Management

The following options can be used to implement an `HttpSession` in spring applications

1. **Injecting HttpSession**
   - Inject **HttpSession** wherever required
   - Retrieving and saving data in the session has to be managed **manually**

2. **Accept it as a parameter**
   - Useful for **web tire applications**

Example: Injecting HttpSession

ShopingCartService.java

```
4
5  import org.springframework.beans.factory.annotation.Autos
6  import org.springframework.stereotype.Service;
7
8  @Service
9  public class ShopingCartService {
10 @Autowired HttpSession httpsesion;
```

Example: Accept as Parameter

MyShoppingCartController.java

```
1  package com.myproject.webapp;
2
3  import javax.servlet.http.HttpSession;
4
5  import org.springframework.stereotype.Controller;
6  import org.springframework.web.bind.annotation.RequestMapping;
7
8  @Controller public class MyShoppingCartController {
9      @RequestMapping("/addToMyCart")
10
11     public String addCart(long prodId, HttpSession httpSession) {
12
```

Spring Session Management (Cont'd)

3. **Creating Bean with session scope**
   - Retrieving and saving changes to the Bean is taken care by spring
   - Spring lifecycle methods(**@PostConstruct** or **@PreDestroy** annotated methods) defined in the Bean will be implemented automatically

4. **Using @SessionAttributes annotation**
   - Include **@SessionAttributes** annotation for model attributes to be stored in the session
   - **@SessionAttributes** copies the specified model attributes to HttpSession before rendering the view

Example: Creating Session Scope Bean

MyShoppingCartService.java

```
3
4  import org.springframework.context.annotation.Scope;
5  import org.springframework.context.annotation.ScopedProxyMode;
6  import org.springframework.stereotype.Component;
7
8  @Component
9  @Scope(proxyMode=ScopedProxyMode.TARGET_CLASS, value="session")
10 public class MyShoppingCartService {
```

Example: Using @SessionAttribute

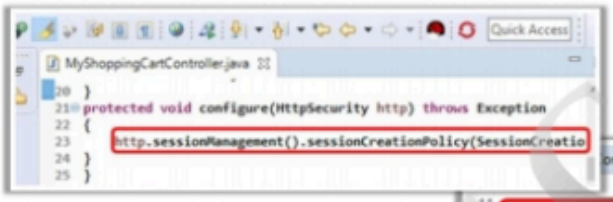
MyShoppingCartController.java

```
15 @SessionAttributes("myRequestObject")
16 public class MyShoppingCartController {
17     @RequestMapping("/addToMyCart")
18
19     public String addCart(long prodId, HttpSession httpSession)
20     {
21
22         return null;
```

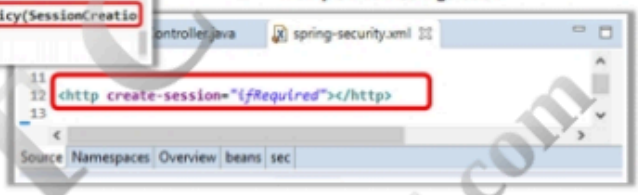
Session Management using Spring Security

- To control session creation using spring security, configure create-session attribute with the following options
  - **Always:** Always creates a session if it does not exist
  - **ifRequired (Default):** Creates session when required
  - **Never:** Spring security will not create sessions but will use session created by application
  - **Stateless:** Application will not create or use session

Example : JAVA Configuration



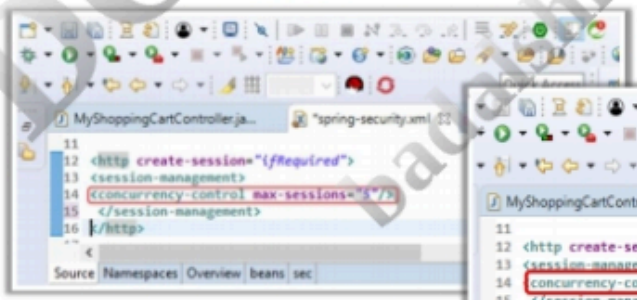
Example : XML Configuration



Restricting Concurrent Sessions per User using Spring Security

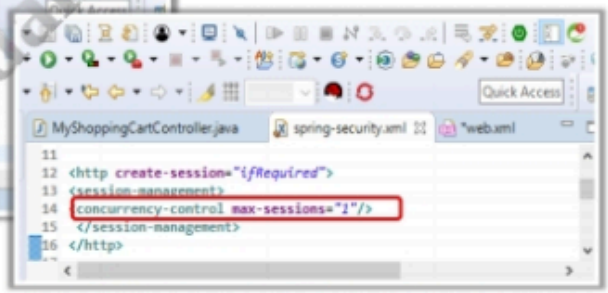
- Concurrent sessions for each user can be controlled using Spring Security

Vulnerable Code



- Enables attackers to login at the same time as the user and perform attacks

Secure Code



- Prevent attackers to perform attacks when a user is logged in

Restricting Concurrent Sessions per User using Spring Security (Cont'd)

- To enable concurrent **session-control**
- Enabling concurrent sessions makes sure to notify the spring security session registry that the session is destroyed

Enabling Concurrent Session-control

- Include the following listener in the web.xml

```
<listener>
<listener-class>org.springframework.security.web.session.HttpSessionEventPublisher</listener-class>
</listener>
```

Example : Control Concurrent Sessions JAVA Configuration

```
protected void configure(HttpSecurity http) throws Except
{
    http.sessionManagement().maximumSessions(2);
}
```

Example: Bean for enabling Concurrent Session-control

```
@Bean
public HttpSessionEventPublisher httpSessionEventPublisher() {
    return new HttpSessionEventPublisher();
}
```

Controlling Session Timeout

```
<http create-session="ifRequired">
<session-management>
<concurrency-control max-sessions="1" expired-url="UserLogin.html"/>
</session-management>
</http>
```

Redirecting user to a configured URL on sending a request with an expired session id

```
<session-management invalid-session-url="login.html">
<concurrency-control max-sessions="1" expired-url="UserLogin.html" />
</session-management>
```

Redirecting user to a configured URL on sending a request with an Invalid session id

Configuring session-fixation-protection in XML

```
@Override
protected void configure(HttpSecurity http) throws Exception
{
    http.sessionManagement()
    .invalidSessionUrl("Relogin.html")
    .maximumSessions(1).expiredUrl("UserLogin.html");
}
```

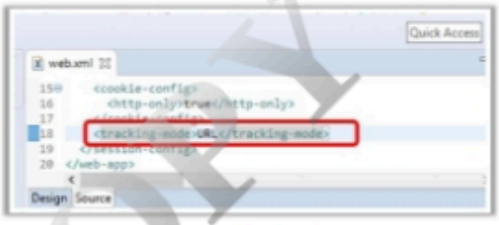
Java configuration for redirecting user to a configured URL on requesting a page with expired / invalid session id

Prevent using URL Parameters for Session Tracking

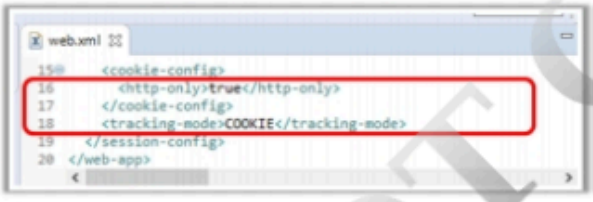
- A URL with a session tracking parameter can be easily used to steal the session
- Disable URL rewriting to prevent appending **JESSIONID** to the URL
- Set `disable-url-rewriting="true"` in the `<http>` namespace (Servlet 3.0 onwards)

Vulnerable Code

- Enables attackers to login at the same time as the user and perform attacks

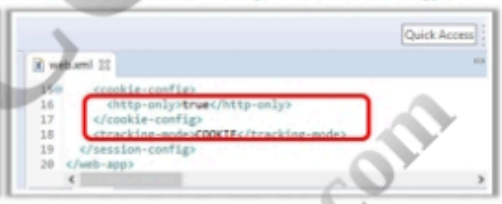


Configuring Session Fixation Protection in web.xml



Secure Code

- Prevents attackers from performing attacks when a user is logged in



Prevent Session Fixation with Spring Security

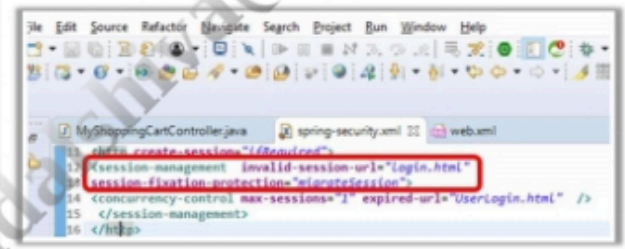
Spring Framework provides Session Fixation Protection

- Configure **session fixation protection** attribute to prevent session fixation
- **Session fixation protection** attribute determines the behavior of the existing session when an already authenticated user tries to login again

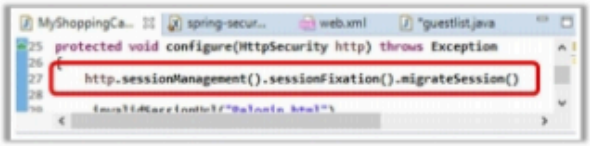
The following values can be assigned:

- **`migrateSession-(Default)`:** Creates a new http session and invalidates the old session after copying the attributes of the old session
- **`None`:** Old session remains and is not invalidated
- **`newSession`:** New http session is created without copying the attributes of the old session

Configuring Session Fixation Protection in XML



JAVA Configuring for Session Fixation Protection



Prevent Session Fixation with Spring Security (Cont'd)

Vulnerable Code

- Enables attackers to login at the same time as the user and perform attacks

MyShoppingCartController.java | spring-security.xml | web.xml

```
<session-management  session-fixation-protection="none">
<concurrency-control max-sessions="1" expired-url="UserLogin.html"
 </session-management>
</http>
```

Secure Code

- Prevents attackers from performing attacks when a user is logged in

MyShoppingCartController.java | spring-security.xml | web.xml

```
<http create-session="ifRequired">
<session-management  invalid-session-url="Login.html"
session-fixation-protection="migrateSession">
<concurrency-control max-sessions="1" expired-url="UserLogin.html"  />
 </session-management>
</http>
```

Use SSL for Secure Connection

- Always configure SSL for web applications in order to protect session IDs from prying eyes
- Set the `SSLEnable="true"` element to true in the server.xml file

Implement SSL in server.xml

MyShoppingCartController.java | spring-security.xml | server.xml

```
<Connector
clientAuth="true" port="8443" minSpareThreads="5" maxSpareThreads="75"
enableLookups="true" disableUploadTimeout="true"
scheme="https" secure="true" SSLEnabled="true"
keystoreFile="/Users/mporges/Desktop/tomcat-ssl/final/server.jks"
keystoreType="JKS" keystorePass="password"
truststoreFile="/Users/mporges/Desktop/tomcat-ssl/final/server.jks"
truststoreType="JKS" truststorePass
SSLVerifyClient="require" SSLEngine=
/>
```

Implement SSL in spring-security.xml by setting force Https property to true

spring-security.xml

```
<beans:bean id="authenticationProcessingFilterEntryPoint" class="org.springframe
<beans:constructor-arg name="strength" value="12" />
<beans:property name="forceHttps" value="true"></beans:property>
<beans:property name="loginUrl" value="true"></beans:property>
 </beans:bean>
```

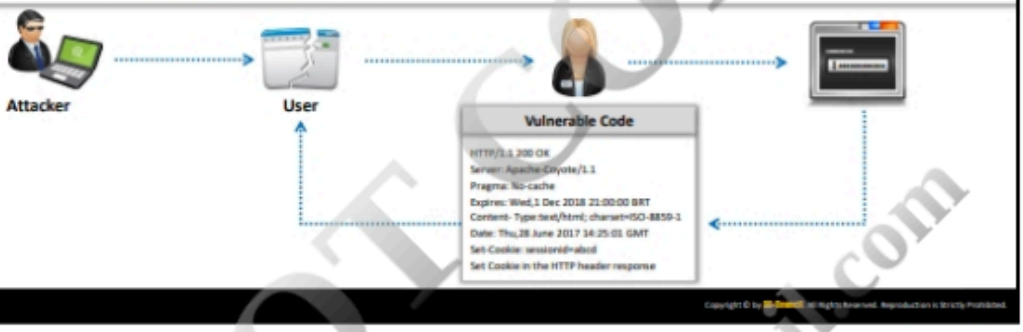
Session Vulnerabilities and their Mitigation Techniques

Session Vulnerabilities

- Session hijacking includes exploiting the **web session** control mechanism that is usually managed by the **session token**
- In a **session hijack attack**, the attacker compromises a session by obtaining a session token either through stealing or predicting and gains access to the webserver
- Different ways of compromising a session token are:
  - Predictable session token
  - Session sniffing
  - Client-side attack (XSS, malicious codes, trojans)
  - Man-in-the-middle attack
  - Man-in-the-browser attack

Types of Session Hijacking Attacks

1. **Attack using client-side scripting**
   - Attackers can fix the session ID value by changing the victim's cookie and inserting a **JavaScript code** in the **URL** that will be executed in the victim's browser
   - http://website.kom/<script>document.cookie="sessionid=abcd";</script>
2. **Attack using <META> tag**
   - Attackers use **code injection attacks** by injecting **malicious code** into the URL and send it to the victim
   - http://website.kon/<meta http-equiv=Set-Cookie content="sessionid=abcd">
3. **Attack through HTTP header response**
   - An attacker creates a **cookie** value of the Session ID, **modifies** the **server response** and **intercepts** the **packages** exchanged between the client and the web application by inserting the **Set-Cookie parameter**



Types of Session Hijacking Attacks (Cont'd)

4. **Session fixation**
   - The session fixation attack is a kind of session hijacking, which **steals** the **established session** between the client and the web server once the user logs in
   - The session fixation attack fixes an established session on the **victim's browser**, to begin attack before the user logs in

Types of Session Hijacking Attacks (Cont'd)

5. **Stealing session ID through HTTP GET request**
   - Attackers embed the session ID information in the URL and it is received by the application through **HTTP GET requests** when the client clicks on the **links**
   - **Example**: http://wxyz.com/wxyz.jsp?article=1234;session id=ER69045454

Stealing session ID using HTTP POST command

```
FORM METHOD=POST ACTION=?/cgi-bin/news.pl?>
<INPUT TYPE =?hidden? Name =?sessionid?
VALUE=?ER69045454?>
<INPUT TYPE =?hidden? Name =?allowed? VALUE=?true?>
<INPUT TYPE =?submit? Name =?Read News Article?>
```


Countermeasures for Session Hijacking

- Regularly clear the **history** and **offline content**
- See that session is **expired** after users log out
- Make sure that **cookies** and **sessions** are removed from the browser
- **Life span** of the session or a cookie should be reduced
- Always implement **https** over http
- Clicking on **unknown links** should be avoided
- Session key has to be a **long random number** or **string**, so that it avoids guessing of a **valid session** and even avoids the chances of a **brute force attack**
- **Session ID** has to be **regenerated** in order to **avoid session fixation** and it is not possible for the attacker

Countermeasures for Session ID Protection

1. If implementing SSL is not possible due to a performance issue or any other reason then protect session IDs using other ways
2. Alter the Session IDs frequently to reduce the validity of a session ID
3. Do not include session IDs as part of URLs, as they can be cached by the browser, sent in the referer header, can be traced from the logs, or can be forwarded to a 'friend' by mistake
4. Session IDs should be long, complicated, random numbers that cannot be easily guessed
5. Modify the session IDs while switching to SSL, authenticating, or other major transitions. Never accept session IDs chosen by a user
6. Protect the entire session using SSL. This ensures that the session ID (e.g., session cookie) cannot be stolen using man-in-the-middle attack, which is a major threat to session IDs

Best Practices and Guidelines for Secure Session Management

Best Coding Practices for Session Management

1. Session Invalidate

- Session invalidate can be defined as removing the **HttpSession** object and its values from the system

Example of Session Invalidate

PostServlet.java

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletE
    response.setContentType("text/html");
    HttpSession Hsession = request.getSession();
    Hsession.removeAttribute(Login);
    Hsession.removeAttribute(password);
    Hsession.invalidate(); response.sendRedirect("http://ec.com/servlets/login.html");
    }
```

Best Coding Practices for Session Management (Cont'd)

2. Session Validate

- Each session should have a defined **timeout** when there is no activity
- If needed, a session hard limit can also be used in some cases in order to **kill** a **session** irrespective of session activity
- Such settings can be done by either in **configuration** files or directly in the code itself

Session Validate Sample Code

web.xml

```
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <session-config>
    <session-timeout>30</session-timeout>
    <cookie-config>
      <http-only>true</http-only>
```

Design Source

Best Coding Practices for Session Management (Cont'd)

3. Use POST not GET

- As a part of best practice it is always recommended to use POST instead of GET, as all the major browsers use GET by default for **form submission**
- In the HTML forms POST should be specified like this **<form method ="POST"...>**

Vulnerable Code

Login.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charse
<title>Register</title>
</head>
<body>
<form action="LoginController" method="get"></form>
</body>
```

Secure Code

Login.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Register</title>
</head>
<body>
<form action="LoginController" method="Post"></form>
```


Best Coding Practices for Session Management (Cont'd)

4. Customize the Session Cookie Name, Path and Domain

- As a part of best practice it is always recommended to change the session cookie name from **JSESSIONID**
- This will protect the cookie from overwriting values from some other application

Vulnerable Code

MyShoppingCartController.java | spring-security.xml | web.xml

```
}
@Bean
public CookieSerializer cookieserializer()
{
    DefaultCookieSerializer serializer = new DefaultCookieSerializer(
    serializer.setCookieName("JSESSIONID");
    serializer.setCookiePath("/");
    serializer.setDomainNamePattern("^.+?\\.(\\w+\\.[a-z]+)$");
    return serializer;
}
```

Secure Code

MyShoppingCartController.java | spring-security.xml | web.xml

```
@Bean
public CookieSerializer cookieserializer()

    DefaultCookieSerializer serializer = new DefaultCookieSerializer();
    serializer.setCookieName("MYAppCookie");
    serializer.setCookiePath("/");
    serializer.setDomainNamePattern("^.+?\\.(\\w+\\.[a-z]+)$");
    return serializer;
```

Configuring Default Cookie Name in web.xml

```
<session-descriptor><cookie-name>MYWEBAPIID<cookie-name></session-descriptor>
```

Best Coding Practices for Session Management (Cont'd)

5. Generate a High Entropy Session ID

- Attackers can guess the session ID if it is of low entropy which is **completely predictable**
- Generate a **complex** and **less** predictable session ID

Vulnerable Code

LowEntropy.java

```
1  package com.demo;
2
3  import java.math.BigInteger;
4  import java.security.SecureRandom;
5
6  public class LowEntropy {
7  public static int TOKEN_SIZE=2;
8  static SecureRandom SECURE_RANDOM=new SecureRandom();
9  byte[] token;
10 public void EntropicToken()
11 {
12     token=new byte[TOKEN_SIZE];
13     SECURE_RANDOM.nextBytes(token);
14     System.out.println(new BigInteger(token).toString(TOKEN_SIZE));
15 }
16 public static void main(String[] args) {
17     LowEntropy loE=new LowEntropy();
18     loE.EntropicToken();
19     }
```

- If the size of the random key generated and stored is small, it is easy to guess

Secure Code

LowEntropy.java

```
1  package com.demo;
2
3  import java.math.BigInteger;
4  import java.security.SecureRandom;
5
6  public class LowEntropy {
7  public static int TOKEN_SIZE=16;
8  static SecureRandom SECURE_RANDOM=new SecureRandom();
9  byte[] token;
10 public void EntropicToken()
11 {
12     token=new byte[TOKEN_SIZE];
13     SECURE_RANDOM.nextBytes(token);
14     System.out.println(new BigInteger(token).toString(TOKEN_SIZE));
15 }
16 public static void main(String[] args) {
17     LowEntropy loE=new LowEntropy();
18     loE.EntropicToken();
19     }
```

- This is a complex key which is not predictable

Best Coding Practices for Session Management (Cont'd)

6. Do not Store Non-serializable Objects into a Session

- Non-serializable session objects can not be **replicated**
- For better reliability and performance it is recommended that **session data is serializable**

Vulnerable Code

BestClassInitialize.java

```
1  package com.demo;
2
3  import javax.servlet.http.HttpSession;
4
5  public class BestClassInitialize  {
6
7      String globName;
8      String globValue;
9
10     public void addToSession(HttpSession session)
11     {
12         session.setAttribute("glob", this);
13
14     }
```

Secure Code

BestClassInitialize.java

```
1  package com.demo;
2  import java.io.ObjectInputStream;
3  import java.io.ObjectOutputStream;
4
5  import java.io.Serializable;
6
7  import javax.servlet.http.HttpSession;
8
9  public class BestClassInitialize implements Serializable {
10     private static final long serialVersionUID = 1L;
11     String globName;
12     String globValue;
```

Checklist to Secure Credentials and Session IDs

Following is the checklist to find out the flaws in user credentials and session IDs

Check security aspects of all the stored credentials using **hashing** or **encryption**
Check if the credentials can be **guessed** or **overwritten**
Check if the session IDs are at all vulnerable to attacks such as **session fixation** etc.
Check whether the session IDs are **exposed** in the **URL**
Check whether the passwords, session IDs and other credentials are sent via **TLS connections** or not
Check if SSL is used by the **login page**
Check if the audit is logged with **who, when, from, where** and **what data**
Check if **inbuilt session management** is used or not
Check whether the **new, preset** or **invalid session ids** are **rejected**
Check the security levels of **password reset** and **Question** and **answer clues**

Guidelines for Secured Session Management

Guidelines

Password Strength

- Ensure that the password is of a enough length and complexity, using **combinations** of alphabet, numbers and non alphanumeric characters

Password Use

- Users should be restricted by number of **login attempts**
- Avoid the display of the passwords provided during failed login attempts as there are chances of **exploitation**
- Ensure that the user is provided with all the **login details** along with **date** and **time**

Password Storage

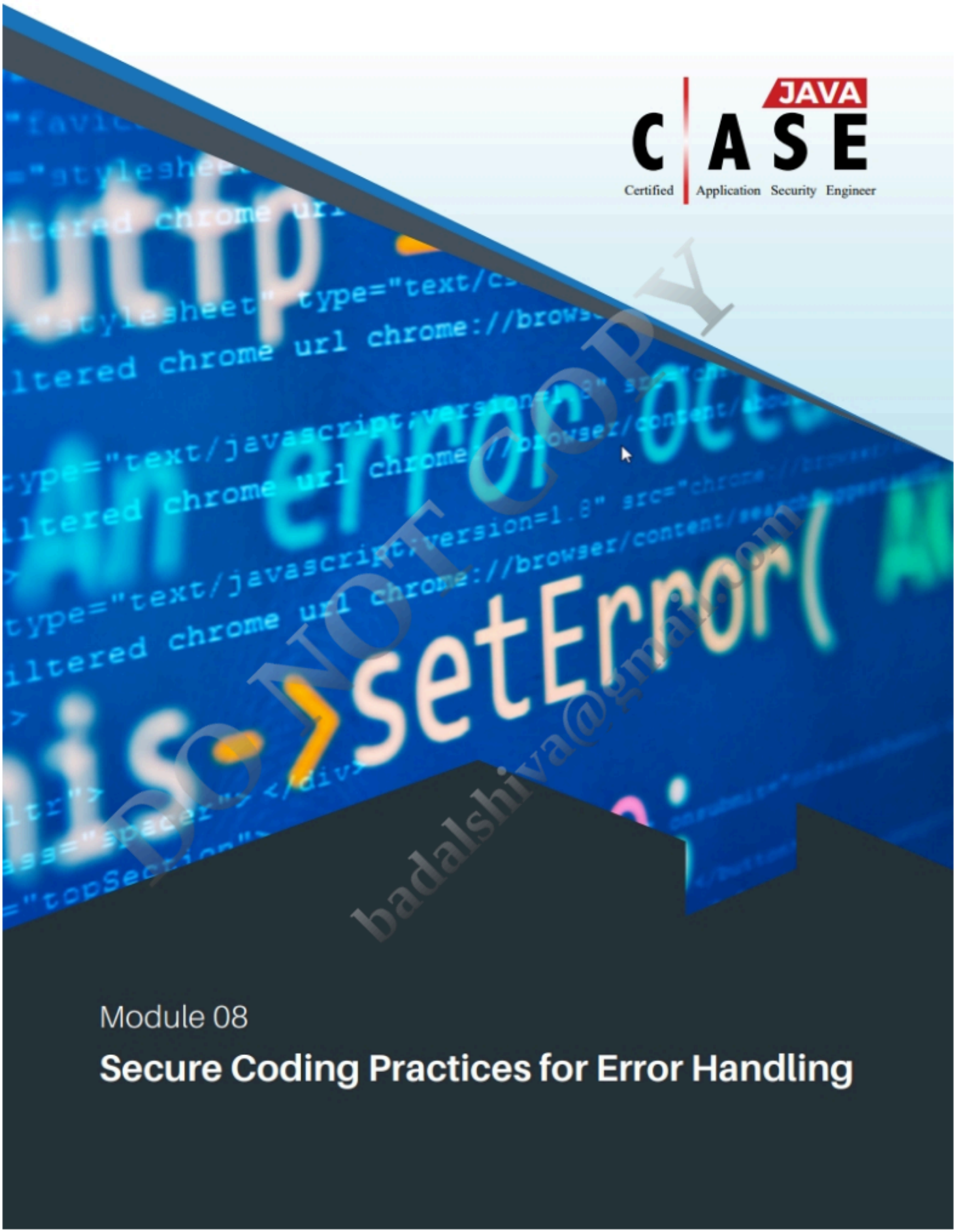
- It is always better to store the passwords either in **hashed** or **encrypted** format to avoid exposure
- Even **decryption keys** should be strongly **protected**

Browser Caching

- Make sure that all the **authentication pages** are marked with **no cache tag** in order to prevent usage of previously typed user credentials
- Latest browsers have feature called **AUTOCOMPLETE=OFF flag** to prevent credentials and sensitive information in autocomplete caches

Module Summary

- In Java, session tracking is done with the help of Cookies, URL Rewriting, Hidden Fields, Session Objects, etc.
- Improper session management results in disclosure of the users' identity by stealing passwords, keys and session tokens
- Attacker compromises a session by obtaining a session token either through stealing or predicting and gains access to the webserver
- Session IDs should be long, complicated, random numbers that cannot be easily guessed
- A URL with a session tracking parameter can be easily used to steal the session



This page is intentionally left blank.

Module Objectives

1	Introduction to Exception and Error Handling in Java
2	Discuss Erroneous Exceptional Behaviors
3	Learn Dos and Don'ts in Error Handling
4	Discuss Spring MVC Error Handling
5	Discuss Exception Handling in Struts 2
6	Learn Best Practices for Error Handling
7	Introduction to Logging in Java
8	Discuss Logging using Log4j
9	Learn Secure Coding in Logging
10	Learn Best Practices for Logging

Introduction to Exceptions

Exception and Error Handling

- Exception and error occur when an **unexpected** or **abnormal event** takes place during the execution of the application
- In Java, exceptions and errors belong to the parent class **Throwable**
- An exception can be classified into two types namely, checked (**compile time**) exceptions and unchecked (**runtime**) exceptions

Checked Exceptions

- Belong to the base class **Exception**
- They occur when there is an **error** found in **code**
- These exceptions are identified by the **compiler** and can be handled by programmers

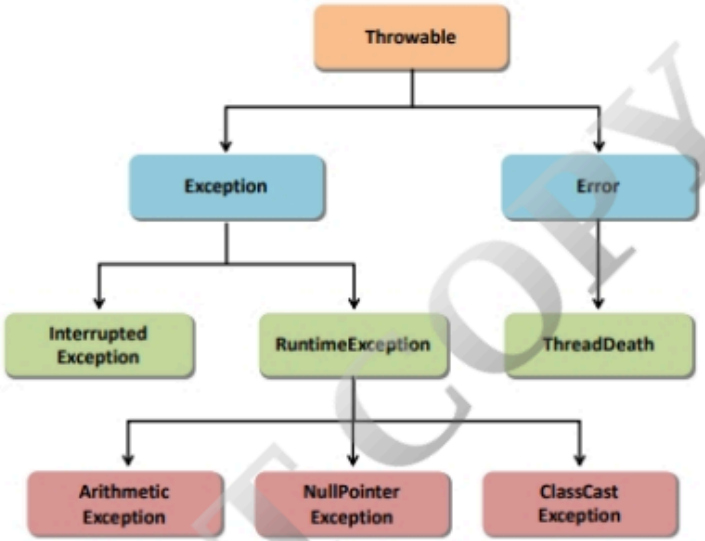
Unchecked Exceptions

- Belong to the base class **RuntimeException**
- They occur when an **error** arises during the execution of the **program**
- These errors cannot be handled by programmers

Exception and Error Handling (Cont'd)

- Errors in Java are caused due to **irrecoverable conditions** such as memory leak, LinkageError, etc. that cannot be handled in programs
- Developers should ensure that exceptions are **properly handled** and do not disclose any information to the user/attacker
- Attackers can make use of information gained from error messages to perform **attacks**
- Information leakage can lead to **social engineering exploits**
- Unhandled exceptions may result in **abnormal termination** of the application

Exception and Error Handling (Cont'd)



Example of an Exception

- Given below is a sample vulnerable code that throws an arithmetic exception
- An exception message that is thrown is shown below

```
public class DivByZeroEg {

        public static void main(String args[]) {
        System.out.println(10/0);
        System.out.println("Print this line");

        }
```

```
public class DivByZeroEg {

        public static void main(String args[]) {
        System.out.println(10/0);
        System.out.println("Print this line");

        }
}
```

```
<terminated> DivByZeroEg [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.e
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivByZeroEg.main(DivByZeroEg.java:6)
```

Handling Exceptions in Java

Exceptions in Java are handled using 'try-catch-finally' blocks

Try

- The lines of code that may cause an exception are placed **inside** a try block
- If the try block encounters an exception, the remaining **statements** in the try block are **not executed**

Catch

- The exceptions that are thrown in the try block are **caught** and **handled** in the catch block

Finally

- The code in the finally block is always executed regardless of the **occurrence** of **exceptions**
- The finally block generally includes code that **frees resources**, **closes connections**, etc.

Syntax of try-catch-finally Block

Handling Exceptions in Java (Cont'd)

Handling exceptions using 'try-with-resources' blocks

- Ensures resources are closed automatically
- Doesn't require **finally** clause to close resources
- To use **try-with-resources** implement **AutoCloseable** (java.lang.AutoCloseable) interface

Example of try-with-resources Block

Handling Exceptions in Java (Cont'd)

try-with-resource vs try-catch-finally Blocks

try-with-resource Block	try-catch-finally Block
Resources are closed on completing the try block	Need to explicitly close the resources in the finally block
On encountering an error while closing the resource, the exception from the try block is thrown	On encountering an error while closing the resource from the finally block, the main relevant exception from the try block is hidden and the exception from the finally block is thrown

Exception Classes Hierarchy

Throwable		
Error	1. LinkageError	2. VirtualMachineError
Exception	1. ClassNotFoundException	2. IllegalAccessException
	3. InstatntiationException	4. InterruptedException
	5. IOException	EOFException
		FileNotFoundException
	6. RuntimeException	ArithmeticException
		ArrayStoreException
		ClassCastException
		IllegalArgumentExeption
		IllegalMonitorStateException
		NegativeArraySizeException
		NullPointerException
		SecurityException

Exceptions and Threats

Exception Name	Description of Information Leak or Threat
`java.io.FileNotFoundException`	Discloses file system structure, user name enumeration
`java.sql.SQLException`	Database structure, user name enumeration
`java.net.BindException`	Enumeration of open ports when untrusted client can choose server port
`java.util.ConcurrentModificationException`	May provide information about thread-unsafe code
`javax.naming.InsufficientResourcesException`	Insufficient server resources (may aid DoS)
`java.util.MissingResourceException`	Resource Enumeration
`java.util.jar.JarException`	Underlying file system structure
`java.security.acl.NotOwnerException`	Owner Enumeration
`java.lang.OutOfMemoryError`	Denial-of-Service
`java.lang.StackOverflowError`	Denial-of-Service

Erroneous Exceptional Behaviors

Erroneous Exceptional Behaviors

- Suppressing/ignoring checked exceptions
- Disclosing sensitive information
- Logging sensitive data
- Restoring objects to prior state, if a method fails
- Avoid using statements that suppress exceptions
- Prevent access to untrusted code that terminates JVM
- Never catch NullPointerException
- Never allow methods to throw RuntimeException, Exception, or Throwable
- Never throw undeclared checked exceptions
- Never let checked exceptions escape from finally block

Erroneous Exceptional Behaviors: Suppressing or Ignoring Checked Exceptions

- java.lang.InterruptedException is thrown when a thread is interrupted while sleeping or waiting
- The run() method of Runnable interface cannot throw a checked exception but should handle InterruptedException

Vulnerable Code

- In the below example, code catches and suppresses InterruptedException
- The run()method caller fails to check that an interrupted exception has occurred

Demo.java

```
1
2  public class Demo implements Runnable {
3
4      @Override
5      public void run() {
6          try {
7              Thread.sleep(1000);
8          } catch (InterruptedException e) {
9
10             e.printStackTrace();
11         }
12     }
13 }
```

Secure Code

- In the below code, run() method appropriately catches the InterruptedException
- It restores the status of the current thread by promptly calling interrupt() method

Demo.java

```
1
2  public class Demo implements Runnable {
3
4      @Override
5      public void run() {
6          try {
7              Thread.sleep(1000);
8          } catch (InterruptedException e) {
9
10             Thread.currentThread().interrupt();
11         }
12     }
13 }
```

Erroneous Exceptional Behaviors: Suppressing or Ignoring Checked Exceptions (Cont'd)

Vulnerable Code

Demo.java

```
1
2  public class Demo implements Runnable {
3
4      @Override
5      public void run() {
6          try {
7              Thread.sleep(1000);
8          } catch (InterruptedException e) {
9
10             e.printStackTrace();
11         }
12     }
13 }
```

- The code prints the exception's **stack trace**
- Exception's stack trace is used for **debugging purposes** but it may also results in suppressing the exception
- Printing the stack trace can provide information to attacker about the **structure** and **state** of the **process**
- This code snippet does not evaluate the **expressions** or **statements** that occur after the **try block** throws exceptions

Secure Code

Demo.java

```
9
10     public static void main(String args[]) throws FileNotFoundException
11     {
12         String fileName = "temp.txt";
13         String line = null;
14             FileReader fileReader = new FileReader(fileName);
15                 BufferedReader bufferedReader =  new BufferedReader(file
16
17                 try {
18                     while((line = bufferedReader.readLine()) != null) {
19                         System.out.println(line);
20                     }
21                 } catch (IOException e) {
22
23                     e.printStackTrace();
24                 }
```

- The secure code uses **FileNotFoundException**
- It requests the user to specify the **desired file name**

Erroneous Exceptional Behaviors: Disclosing Sensitive Information

- Throwing exceptions without filtering sensitive information may result in **information leaks**
- It may provide attackers with information that can be used to perform **further exploits**

Vulnerable Code

- The example code provides a user with **contents** and **layout** of the **file system**

*Demo.java

```
4
5      public static void main(String args[]) throws FileNotFoundException
6      {
7          try {
8              FileInputStream finSt = new FileInputStream(System.getenv("APPDATA") +args[0]);
9              } catch (FileNotFoundException x)
10             {
11             System.out.println("Invalid file");
12             return;
13             }
14     }
```

Secure Code

- The code implements the policy that only files in the **c:\homepath** can be accessed by the user
- **File.getCanonicalFile()** method is also used to **canonicalize** the file subsequent path name

Demo.java

```
10
11
12     public static void main(String args[]) throws FileNotFoundException
13     {
14         File f1 = null;
15     try {
16     f1 = new File(System.getenv("APPDATA") +
17     args[0]).getCanonicalFile();
18             if (!f1.getPath().startsWith("c:\\homepath"))
19             {
```

Erroneous Exceptional Behaviors: Disclosing Sensitive Information (Cont'd)

Vulnerable Code

```
Demo.java
public static void main(String args[]) throws IOException
{
    try {
        FileInputStream fInSt = new FileInputStream(System.getenv("APPDATA") +args[0]);
        } catch (FileNotFoundException x)
        {
            Logger logger = Logger.getAnonymousLogger();

            logger.log(Level.SEVERE, "thrown Exception", x);

            throw new IOException("Unable to retrive file",x);
```

- The example code throws a general exception after a logged exception **leaks the file system layout information** to the attacker

Secure Code

```
public class Demo {

    public static void main(String args[]) throws IOException
    {try {
        FileInputStream fInSt =null;
        switch(Integer.valueOf(args[0])) {
        case 1:
            fInSt=new FileInputStream("c:\\homepath\\file1");
        case 2:
            fInSt=new FileInputStream("c:\\homepath\\file2");
            default:
                System.out.println("Invalid Option");
                break;
        }
        } catch (Throwable t) {
        MyExceptionReporter.report(t);
        }
```

- The code implements the policy that only files in the **c:\homepath\file1** and **c:\homepath\file2** can be accessed by the user

Erroneous Exceptional Behaviors: Logging Sensitive Data

- Throwing exceptions while logging a process can sometimes **disrupt the process**, and failure to throw exceptions can create **security vulnerabilities**
- Programmers should ensure that the logging process should not be disrupted by **throwing incorrect exceptions**

Vulnerable Code

- The code snippet below writes **SecurityException** to the standard error stream that can be inadequate for the logging process
- Standard error stream can be **exhausted or closed** and may further prevent exceptions from recording
- Error streams used for certain **security-related exceptions** or errors may not provide adequate security
- If any input/output errors occur while writing security exceptions, the catch block may throw an **IOException** leaking security information to the attacker
- Using **System.out.print*()**, Console.printf() or Throwable.printStackTrace() to print may leak security information

Secure Code

- The code snippet below uses **java.util.logging.Logger** (default logging API in JDK 1.4)
- It attempts to simplify the entire logging process by using only one **logger**


```
Demo.java | MyExceptionReporter.java
    try {
        FileInputStream fInSt = new FileInputStr
        } catch (SecurityException x)
        {
            System.err.println(x);
        }
```

```
Demo.java | MyExceptionReporter.java
    try {
        FileInputStream fInSt = new FileInputStream(System.getenv("APPDATA") +args[0]);
        } catch (SecurityException x)
        {
            Logger logger =Logger.getAnonymousLogger();

            logger.log(Level.SEVERE, "thrown Exception", x);
```

Erroneous Exceptional Behaviors: Restoring Objects to Prior State, if a Method Fails

- Objects (general and security-related) should be maintained in a **consistent manner** even if exceptions arise

Objects consistent methods include:

1. Using **rollback** during failure events
2. Rearrange **exceptional condition mechanism** so that the exception condition executes before the modification of the object
3. Any **changes** to the **object** should be avoided
4. Input validation
5. Operations that require execution should be implemented on a temporary **copy** of the **object** rather than the original object and committing those **changes** to the **original object**

Erroneous Exceptional Behaviors: Restoring Objects to Prior State, if a Method Fails (Cont'd)

Vulnerable Code

- The code example uses dimension class with three **internal attributes** length, width and height of a rectangular box
- The **getVolumePackage()** method is used for returning the total volume of the box after accounting for packaging material
- getVolumePackage() method returns volume
- Input validation rejects **negative values**
- Input values cannot be more than 10, if an input is more than 20 then **illegalArgumentException** is **thrown**

Demo.java | MyExceptionReporter.java

```
public class Demo {

    private int length;
    private int width;
    private int height;
    static public final int PADDING=2;
    static public final int MAX_DIMENSION = 10;

    public Demo(int length, int width, int height) {
        this.length=length;
        this.width=width;
        this.height=height;
    }

    protected int getVolumePackage(int weight) {
        length+= PADDING;
        width+= PADDING;
        height += PADDING;
        try {
        if(length <= PADDING || width <= PADDING || height <= PADDING ||
        length > MAX_DIMENSION + PADDING || width > MAX_DIMENSION + PADDING ||
        height > MAX_DIMENSION + PADDING || weight <= 0 || weight > 20)
        {
            throw new IllegalArgumentException();
        }

        int volume = length * width * height;
        length -= PADDING; width -= PADDING; height -= PADDING;
        return volume;
        } catch (Throwable t)
        {
            MyExceptionReporter mer = new MyExceptionReporter();
            mer.report(t);
        }
        return -1;

}
    public static void main(String[] args) {
    Demo d = new Demo(10, 10, 10);
    System.out.println(d.getVolumePackage(21)); // Prints -1 (error)
    System.out.println(d.getVolumePackage(19)); // Prints 2744 instead of 1728
```

Erroneous Exceptional Behaviors: Restoring Objects to Prior State, if a Method Fails (Cont'd)

Secure Code

Example with **Input Validation**

- The secure code performs **input validation** before modifying the object state
- The try block in the code considers only those statements that **throw exceptions** removing others outside the try block

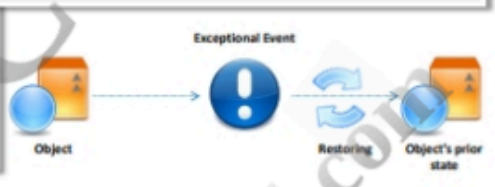
```
Demo.java | MyExceptionReporter.java

protected int getVolumePackage(int weight) {
    length+= PADDING;
    width+= PADDING;
    height += PADDING;
    try {
    if(length <= PADDING || width <= PADDING || height <= PADDING ||
    length > MAX_DIMENSION + PADDING || width > MAX_DIMENSION + PADDING ||
    height > MAX_DIMENSION + PADDING || weight <= 0 || weight > 20)
    {
        throw new IllegalArgumentException();
    }

    int volume = length * width * height;
    length -= PADDING; width -= PADDING; height -= PADDING;
    return volume;
```

Secure Code

Example with **Rollback**

- The code example uses a catch block replacing the **getVolumePackage()** method
- This code aims at restoring prior **object state** if any exceptional event occurs

```
Demo.java | MyExceptionReporter.java

        int volume = length * width * height;
        length -= PADDING; width -= PADDING; height -= PADDING;
        return volume;
        } catch (Throwable t)
        {
            MyExceptionReporter mer =new MyExceptionReporter();
            mer.report(t);
        }
        return -1;
}
```



Erroneous Exceptional Behaviors: Avoid using Statements that Suppress Exceptions

- Use of **return, break, continue** or **throw statements** is restricted in a finally block
- When a program executes a try or catch block , the finally block in the try block is executed, sending the program to **normal completion**
- Statements used in the finally block may cause the try block to **complete abruptly** and suppress any exceptions thrown from the try or catch blocks

Vulnerable Code

```
Demo.java | MyExceptionReporter.java

public static boolean PerformLogic()
{
  try {

      throw new IllegalStateException();
      }
   finally {
       System.out.println("finally Logic Performed..");
   return true;
   }
}
```

- In the code, **return** statement used in the finally block completes abruptly

Secure Code

```
Demo.java | MyExceptionReporter.java

public class Demo {
public static boolean PerformLogic()
{
    try {
        throw new IllegalStateException();

    finally {
        System.out.println("finally Logic Performed..");
    }

  //return true;
    //return statements must go here if any;
    //applicable only when exception is thrown conditionally
}
}
```

- In the below code, return statement is removed from the finally block

Erroneous Exceptional Behaviors: Prevent Access to Untrusted Code that Terminates JVM

System.exit() when invoked can even terminate the Java Virtual Machine (JVM) resulting in the termination of currently running programs as well as threads that might even lead to **Denial-of-Service (DoS) attacks**

Vulnerable Code

Demo.java | MyExceptionReporter.java

```
1  public class Demo {
2
3      public static void main(String args[])
4      {
5          //.................
6          //.........
7
8          System.exit(1);// Exit Abruptly
9          System.out.println("This will never Execute");
10
11     }
12 }
```

- The above code shows how the JVM is shutdown and the running process is terminated by using **System.exit()**
- The code also does not have **security manager** to check if the caller is valid or not to invoke the **System.exit()**

Erroneous Exceptional Behaviors: Prevent Access to Untrusted Code that Terminates JVM (Cont'd)

Secure Code

- The code contains security manager "**passwordSecurityManager**" that overrides the **checkExit()** method of the **SecurityManager** class
- This installed security manager tracks if **exit** is permitted or not
- It also does necessary **cleanup action** as well as **logging** the **exception**

PasswordSecurityManager.java

```
1
2  public class PasswordSecurityManager extends SecurityManager {
3      private boolean Flag;
4      public PasswordSecurityManager(){
5          super();
6          Flag = false;
7      }
8      public boolean ExitAllowed(){
9          return Flag;
10
11     }
12     @Override
13     public void checkExit(int status) {
14         if (!ExitAllowed()) {
15             throw new SecurityException();
16         }
17         super.checkExit(status);
18     }
19     public void AllowedExit(boolean b) {
20         Flag = b;
21     }
22 }
23 class TerminateExit{
24     public static void main(String[] args) {
25         PasswordSecurityManager PSM=PasswordSecurityManager();
26         System.setSecurityManager(PSM);
27         try {
28             //..
29             System.exit(1);
30         } catch (Throwable t) {
31             if(t instanceof SecurityException)
32             {
33                 System.out.println("Interupted System.exit(1)");
34                 //Log exception
35             }else {
36                 //handover to Exception Handler
37             }
```

Erroneous Exceptional Behaviors: Never Catch java.lang.NullPointerException

- **java.lang.NullPointerException** and **RuntimeException** should not be caught by the programs
- A NullPointerException indicates about the **existence of null pointer** dereference by throwing an exception at runtime
- **Bugs** in the program are indicated by the Runtime exceptions need to be fixed by the programmer

Vulnerable Code

- The following code gives an example based on `isName()` method
- In this method a String argument is taken and if the given string is a valid name then it returns true
- When this method fails to check whether the given string is null or not and instead catches `NullPointerException`, then it returns `false`

Secure Code

- The following code checks if the given string is null or not instead of catching the `NullPointerException`

Demo.java

```
public class Demo {

    boolean isName(String str) {
        try {
            String sNames[] = str.split(" ");
            if(sNames.length != 2) {
                return false;
            }
            return (isCapitalized(sNames[0]) && isCapitalized(sNames[1]));
        } catch (NullPointerException e) {
            return false;
        }
    }
```

```
boolean isName(String str) {
    try {
        if (str == null) {
            return false;
        }
        String sNames[] = str.split(" ");
        if(sNames.length != 2) {
            return false;
        }
        return (isCapitalized(sNames[0]) && isCapitalized(sNames[1]));
    } catch (NullPointerException e) {
        return false;
    }
```

Erroneous Exceptional Behaviors: Never Allow Methods to Throw RuntimeException, Exception, or Throwable

- **RuntimeException**, **Exception**, or **Throwable** should not be thrown or else it may lead to various errors

Vulnerable Code

- The following code uses `toUpperCase()` method
- This method accepts a string and **returns true** if the string is a capital letter followed by lowercase letters
- When a null string argument is passed this method throws a `RuntimeException`

Secure Code

- The following code throws a `NullPointerException` in order to denote exceptional condition

test.java

```
package Test;

public class test {
boolean isName(String str)
{
  if(str==null)
  {
    throw new RuntimeException("Null String");

  }
  if(str=="")
  {return true;}
  String begin=str.substring(0, 1);
  String next=str.substring(1);
  next.equals(next.toLowerCase());
  return begin.equals(begin.toUpperCase());
}
```

Demo.java

```
public class Demo {

    boolean isName(String str) {

        if (str == null) {
            throw new NullPointerException();
        }
        if(str=="")
        {
            return true;
        }
        String begin=str.substring(0, 1);
        String next =str.substring(1);
        return (begin.equals(begin.toUpperCase()));

    }
```

Erroneous Exceptional Behaviors: Never Throw Undeclared Checked Exceptions

1. In Java, there are a few techniques that permit **throwing** the undeclared checked exceptions at **runtime**
2. These techniques do not allow the usage of **throws clause** by weakening the ability of **caller methods**
3. These techniques should never be used to throw undeclared checked exceptions

Vulnerable Code

- The given vulnerable code **throws undeclared checked exceptions**
- This code uses the **undelcaredThrow()** method that takes a **Throwable** argument and then invokes a function that throws the argument without declaration

Demo.java

```
1  public class Demo {
2          private static Throwable t;
3          private Demo() throws Throwable {
4            throw t; }
5          public static synchronized void undeclaredThrow(Throwable
6            // These exceptions must not be passed
7          if (t instanceof IllegalAccessException ||
8              t instanceof InstantiationException) {
9            // Unchecked, declaration not required
10           throw new IllegalArgumentException(); }
11         Demo.t = t;
12         try {
13       //Throwable argument passed above is thrown in next line
14       // even though it throws clause of class.newInstance fail
15           // to declare that this may happen
16           Demo.class.newInstance();
17       } catch (InstantiationException ie) { /* unreachable */
18       } catch (IllegalAccessException ie) { /* unreachable */
19         } finally { // Avoid memory leak
20           Demo.t = null;
21         } }
22
23
24         public static class UndeclaredException {
25           public static void main(String[] args) {
26             // No declared checked exceptions
27             Demo.undeclaredThrow(
28                 new Exception("Some checked exception"));
29       }
30       }
31 }
```

Erroneous Exceptional Behaviors: Never Throw Undeclared Checked Exceptions (Cont'd)

Secure Code

- This solution code uses **java.lang.reflect.Constructor.newInstance()** instead of **Class.newInstance()**
- The **Constructor.newInstance()** process shawls any exceptions thrown from within the constructor into a checked exception known as **Invocation Target Exception**

Demo.java

```
37
38         public static synchronized void undeclaredThrow1(Throwable t) throws NoSuchMethodException, SecurityException {
39               // These exceptions should not be passed
40               if (t instanceof IllegalAccessException ||
41                   t instanceof InstantiationException) {
42                 // Unchecked, no declaration required
43                 throw new IllegalArgumentException(); }
44               Demo.t = t;
45               try {
46                 Constructor<Demo> constr =Demo.class.getConstructor(new Class<?>[0]);
47                 constr.newInstance();
48               } catch (InstantiationException ie) { /* unreachable */
49               } catch (IllegalAccessException ie) { /* unreachable */
50               } catch (InvocationTargetException ie) {
51                 System.out.println("Exception thrown: "+ ie.getCause().toString());
52               } finally { // Avoid memory leak
53                 Demo.t = null;
54               } }
```

Erroneous Exceptional Behaviors: Never Let Checked Exceptions Escape from Finally Block

- Methods invoked from within a **finally** block throw an **exception**
- If such exceptions are not caught and handled then it results in termination of entire **try block**

Vulnerable Code

Invoke.java

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.IOException;
4
5  public class Invoke {
6  public static void doInvoke(String file) {
7  // ... code to check or set character encoding ...
8      try {
9        BufferedReader rdr =
10           new BufferedReader(new FileReader(file));
11       try {
12         // Do operations
13       } finally {
14         rdr.close();
15         // ... Other cleanup code ...
16       }
17     } catch (IOException ix) {
18       // Forward to handler
19     } } }
```

- This vulnerable code has a finally block which **closes reader object**
- It is wrongly assumed by the programmer that the **statements** of the **finally block** do not throw exceptions
- This results in the **failure** of handling **exceptions**

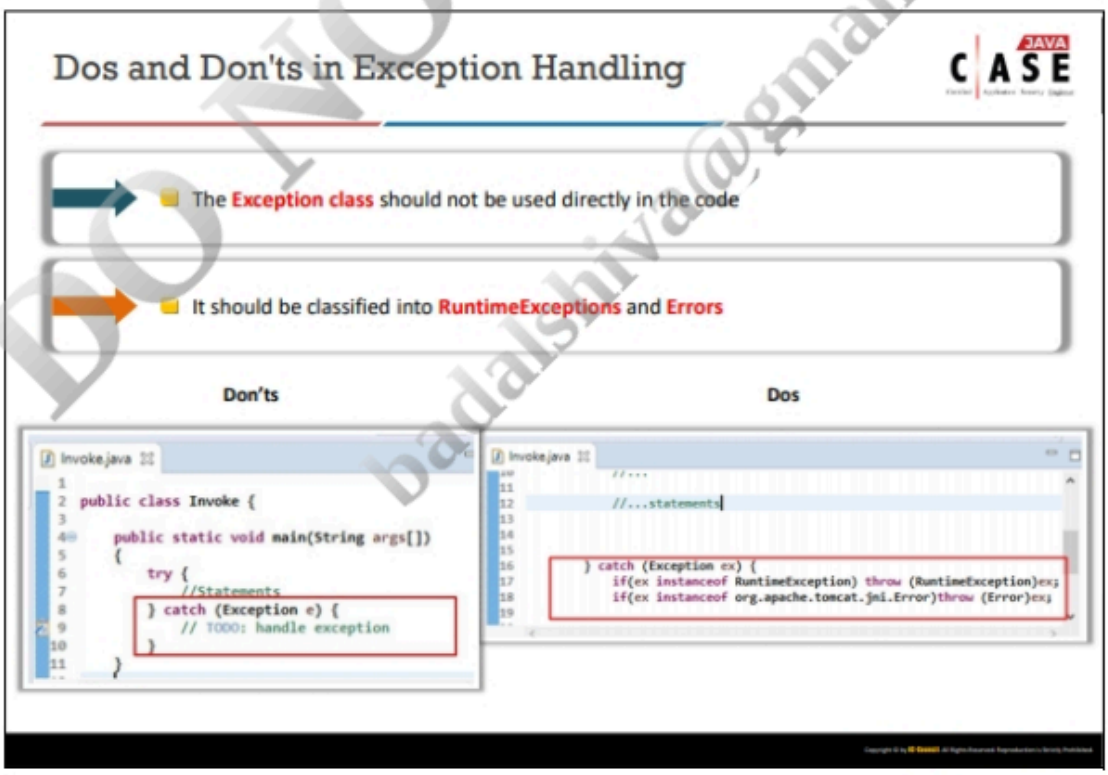
Erroneous Exceptional Behaviors: Never Let Checked Exceptions Escape from Finally Block (Cont'd)

Secure Code

Invoke.java

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.IOException;
4
5  public class Invoke {
6      public static void doInvoke(String file) {
7          // ... code to check or set character encoding ...
8          try {
9              BufferedReader rdr = new BufferedReader(new FileReader(file));
10             try {
11                 // Do operations
12             } finally {
13                 try {
14                     rdr.close();
15                 } catch (IOException ie) {
16                     // Forward to handler }
17                     // .........Other clean-up code ...
18                 }
19             }
20         } catch (Exception e) {
21             // TODO: handle exception
```

- The code enfolds the `close()` method request in a **try-catch block** in the finally block
- Therefore, the potential `IOException` is held without allowing it to circulate anymore



Dos and Don'ts in Exception Handling (Cont'd)

- The printStackTrace() function should not be used in the code as it reveals all the exception information to the user
- An attacker can misuse the information to perform attack on the application
- The exception thrown by an application should be stored in the logger file using a logger application

Don'ts

Invoke.java

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

Dos

```
try {
}catch(Exception ex){
    Logger logger=Logger.getLogger("Stack-trace-logger");
    logger.log(level, "logger", ex);
}
```

Dos and Don'ts in Exception Handling (Cont'd)

- The vulnerable code throws an exception handler function with the same name from all user functions (isA, isB and isC)
- This is not a good practice, because when an the error is encountered, it becomes difficult for the developer to identify the exact function that has thrown the error
- It is a good practice to code separate error functions for every user function

Don'ts

MyDomainEg.java

```
public interface MyDomainEg {
    void isA()
    throws
     MyDomainException;
    void isB()
     throws
    MyDomainException;
    void isC()
     throws MyDomainException;
}
```

Dos

MyDomainEg.java

```
public interface MyDomainEg {
void isA() throws SomeMyDomainException,OtherMyDomainException;

void isB()throws SomeMyDomainException,OtherMyDomainException;
}
```

Dos and Don'ts in Exception Handling (Cont'd)

- Clean up code such as releasing resources, closing input I/O streams, and deleting files should be done in the **finally** block

Don'ts

DemoForException.java

```
1  import java.io.BufferedReader;
2  import java.io.FileNotFoundException;
3  import java.io.FileReader;
4  import java.io.IOException;
5
6  public class DemoForException {
7
8      public void Readfile()
9      {
10     FileReader fr = null;
11     String file=null;
12     try {
13         fr = new FileReader(file);
14         BufferedReader br = new BufferedReader(fr);
15         String line = null;
16         fr.close();
17     } catch (FileNotFoundException e) {
18
19         e.printStackTrace();
20     }
```

- Clean up code should not be kept in **try** or **catch** block

Dos

DemoForException.java

```
10     FileReader fr = null;
11     String file=null;
12     try {
13         fr = new FileReader(file);
14         BufferedReader br = new BufferedReader(fr);
15         String line = null;
16
17     } catch (FileNotFoundException e) {
18
19         e.printStackTrace();
20     } finally {
21       if (fr != null) {
22           try {
23               fr.close();
24           } catch (IOException e) {
25
26               e.printStackTrace();
27           }
28       }
```

- Clean up code should be kept only in **finally** block

Dos and Don'ts in Exception Handling (Cont'd)

- Exceptions should be logged properly using a **logger file**

Don'ts

Invoke.java

```
1
2  public class Invoke {
3
4      public static void Test()
5      {
6          try {
7              int i=5;
8              i=i/0;
9
10         } catch (Exception ex) {
11             ex.printStackTrace();
12         }
13     }
```

Dos

Invoke.java

```
1  import java.util.logging.Level;
2  import java.util.logging.Logger;
3
4  public class Invoke {
5
6      public static void Test()
7      {
8          try {
9              int i=5;
10             i=i/0;
11
12         } catch (Exception ex) {
13             Logger logger=Logger.getLogger("Stack-Trace-Logger");
14             logger.log(Level.SEVERE, "thrown Exception", ex);
15
16             throw new ArithmeticException("Devide By zero");
17
18
19         }
20     }
21     public static void main(String[] args) {
22         Test();
23     }
```

Avoid using Log Error and Throw Exception at Same Time

- Throwing and logging exception at same time creates multiple **logs**

The following example shows the improper and proper approach

Improper Approach

Invoke.java

```
1  import org.apache.log4j.Logger;
2  import org.apache.log4j.LogManager;
3
4  public class Invoke {
5
6      public static void Test()
7      {
8          final Logger logger = Logger.getLogger(Invoke.class);
9          try {
10             int i=5;
11             i=i/0;
12
13         } catch (Exception ex) {
14             logger.error("The Error is", ex);
15
16         }
17     }
```

catch **Exception** class and log an Exception

Proper Approach

```
import org.apache.log4j.Logger;
import org.apache.log4j.LogManager;

public class Invoke {

    public static void Test()
    {
        final Logger logger = Logger.getLogger(Invoke.class);
        try {
            int i=5;
            i=i/0;

        } catch (ArithmeticException ex) {
            System.out.println("The Error is.."+ex.getMessage());
        }
    }
```

Handled only **Exception** instead log an error

Spring MVC Error Handling

Spring MVC Error Handling

The following are different exception handling **approaches** in Java Spring MVC Framework

Controller Level Exception Handler

- Controller exception can be handled by:
  - Adding **@ExceptionHandler** annotation
  - Implementing **HandlerExceptionResolver**

Global Exception Handler

- Adding **@ControllerAdvice** annotation to define global exceptions
- Using **HandlerExceptionResolver** interface for creating global exception handlers

Custom Errors Mapping

- Adding **@ResponseStatus** annotation to map custom errors to exception status code

Handling Controller Exceptions with @ExceptionHandler Annotation

- **To Implement @ExceptionHandler annotation**
  - Add **@ExceptionHandler** annotation to the method
  - Include a list of exceptions to be handled by the method
  - When handling multiple exceptions, **specific exceptions** are thrown ignoring **general exceptions**
  - Return error response as string, Modelview object, ResponseEntity or @Responsebody

Vulnerable Code: Unhandled Exception

ErrorHandling.java

```
@RequestMapping("/addNewUser")
public String newUser(Model model) throws Unauthor
{
model.addAttribute("username", new User());
List<String> prefContactMethods=new ArrayList<Stri
prefContactMethods.add("E-mailaddress");
prefContactMethods.add("CallNo");
model.addAttribute("prefContctName", prefContctMet
List<String> sexType=new ArrayList<String>();
sexType.add("Male");
sexType.add("Female");
model.addAttribute("sexcat", sexcat);
if(true)
throw new UnauthorizedAccessException();
return "newUser";
}
```

Secure Code: Handling Exception using @ExceptionHandler Annotation

AdminCourseController.java | RestController.java

```
package com.ECouncil.CASE;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@RestController
@RequestMapping("/admincourses")
public class AdminCourseController {
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<Object> get(){ throw new RuntimeException("courses not yet supported");
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<Error> handle(RuntimeException ex){
    System.out.println("controller local exception handling @ExceptionHandler");
    Error error = new Error(ex.getMessage());
    return new ResponseEntity<Error>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Handling Controller Exceptions with HandlerExceptionResolver

- **To implementing HandlerExceptionResolver Interface**
  - Override the resolveException() method
  - It will handle all the exceptions of the controller
  - Use instanceof to get the exception type

Example: Controller Exception Handling using HandlerExceptionResolver

ECEmployee.java

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.http.HttpStatus;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.view.json.MappingJackson2JsonView;
public class ECEmployee implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler,
            Exception ex) {
        System.out.println("exception handling HandlerExceptionResolver"); response.reset();
        response.setCharacterEncoding("UTF-8");
        response.setContentType("text/json");
        ModelAndView model=new ModelAndView(new MappingJackson2JsonView());
        if(ex instanceof RuntimeException)
        {
            response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            model.addObject("errorcode", HttpStatus.INTERNAL_SERVER_ERROR.value());
            model.addObject("errormessage", ex.getMessage());
                    }
        else
        {   response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            model.addObject("errcode", HttpStatus.INTERNAL_SERVER_ERROR.value()); model.addObject("errmessage", ex.getMessage());
```

Spring MVC: Global Exception Handling

- For global exception handling use **@ControllerAdvice** annotation along with **@ExceptionHandler** method annotation for any class

Example: Global Exception Handling using @ControllerAdvice

MyProjectExceptionHandler.java | ControllerAdvice.java

```
1  package com.ECouncil.CASE;
2
3  import org.springframework.http.HttpStatus;
4  import org.springframework.http.ResponseEntity;
5  import org.springframework.web.bind.annotation.ExceptionHandler;
6
7  @ControllerAdvice
8  public class MyProjectExceptionHandler extends ResponseEntityExceptionHandler {
9
10
11     @ExceptionHandler(RuntimeException.class)
12     public ResponseEntity<Error> handle(RuntimeException ex){
13         Error error = new Error(ex.getMessage());
14         return new ResponseEntity<Error>(error, HttpStatus.INTERNAL_SERVER_ERROR);
15     }
16
17 }
```

Global Exception Handling: HandlerExceptionResolver

- Create a custom global error handler by implementing **HandlerExceptionResolver** interface
- It enables mapping exceptions to a view name

Example: MappingExceptionResolver - with Java Configuration

MyProjectExceptionHandler.java

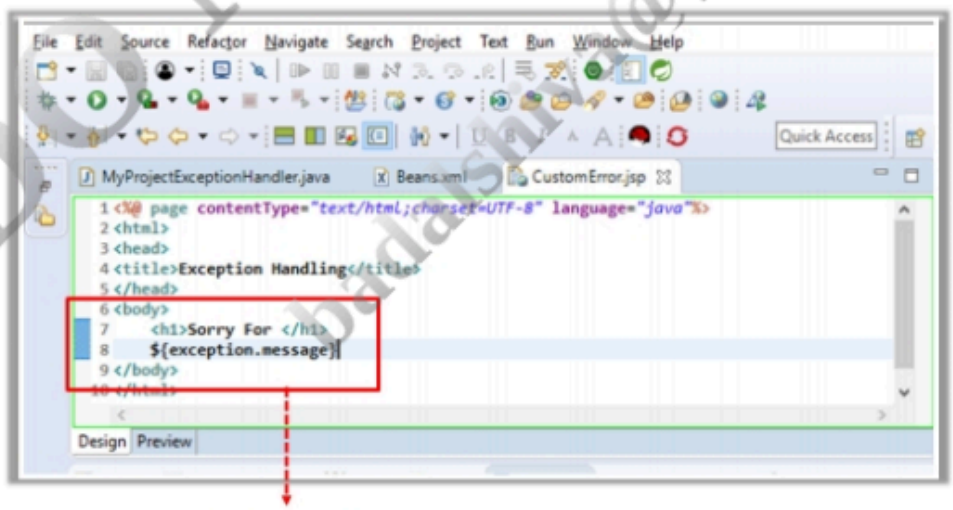
```
@Bean
public ExampleMappingExceptionResolver exceptionResolver()
{
    ExampleMappingExceptionResolver resolver = new ExampleMappingExceptionResolver();
    Properties exceptions = new Properties();
    exceptions.put(ArithmeticException.class, "error");
    resolver.setExceptionMappings(exceptions);
    return resolver;


}
}
```

Example: MappingExceptionResolver - with XML Configuration

Beans.xml

```
<bean class="org.springframework.web.servlet.handler.ExampleMappingExceptionResolver">
<property name="exceptionMappings"> <props>
<prop key="java.lang.ArithmeticException">error</prop> </props>
</property>
  </bean>
```

Global Exception Handling: HandlerExceptionResolver (Cont'd)



Sample Custom Error Page

Mapping Custom Exceptions to Statuscode with @ResponseStatus

- Adding **@ResponseStatus** annotation enables mapping custom exceptions with default spring status codes
- **ResponseStatusExceptionResolver** automatically handles all thrown exceptions

Default Exception Handling Status Codes

Status Code	Exception
404	Not Found–>NoSuchRequestHandlingMethodException etc.
400	Bad Request–>BindException, HttpMessageNotReadableException etc.
406	Not Acceptable–>HttpMediaTypeNotAcceptableException
415	Unsuppported Media Type–>HttpMediaTypeNotSupportedException
405	Method Not Allowed–>HttpRequestMethodNotSupportedException
500	Internal Server Error–>HttpMessageNotWritableException

Example: Status Code Mapping to Custom Exception

Beans.xml | CustomError.jsp | ControllerA... | EmployeeNotA...

```
1  package com.ECouncil.CASE;
2
3  import org.springframework.http.HttpStatus;
4  import org.springframework.web.bind.annotation.ResponseStatus;
5
6  @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "")
7  public class EmployeeNotAvailableException extends RuntimeException
8  {
9      private static final long serialVersionUID = 1L;
10     EmployeeNotAvailableException()
11     {
```

Configure Custom Error Page in Spring MVC

- By default, Spring MVC will raise an exception while trying to access unauthorized resources. Such exceptions should be handled correctly by customizing **error page configuration** in Spring MVC
- It is also possible to configure custom error pages using Java configuration with the help of **@EnableWebSecurity** annotation and **WebSecurityConfigurerAdapter**

XML configuration

web.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <http>
4     <intercept-url pattern="/admin/*" access="hasAnyRole('ROLE_ADMIN')"/>
5         <access-denied-handler error-page="/my-error-page" />
6 </http>
```

Design | Source

Note: If an unauthorized user try to access a page then response will be redirected to "/my-error-page"

Exception Handling in Struts 2

Exception Handling: Struts 2

Global Exception Handling

- To implement **global exception handling** include the given struts.xml file. This will map exceptions of all action classes

Global Exception Configuration

```
struts.xml
33 <global-results>
34 <result name="globalerror">/Errorpage.jsp</result>
35 </global-results>
36 <global-exception-mappings>
37     <exception-mapping exception="java.lang.Exception" result="globalerror"/>
38 </global-exception-mappings>
39
40 </package>
```

Action Level Exception handing

- To implement **action level exception handling** add <exception-mapping> elements within the <action> element of the **struts.xml** file

Action Level Exception Configuration

```
struts.xml
23 </global-exception-mappings>
24
25     <action name="EmployeeAction" class="Com.ECProject.Info.EmployeeAction">
26     <exception-mapping result="databaseError" exception="java.sql.SQLException"></exception-mapping>
27     <result name="success">/DbConnection.jsp</result>
28     <result name="databaseError">/DbConnection.jsp</result>
29     </action>
```

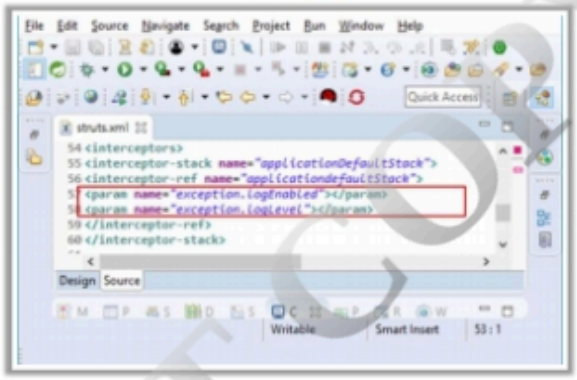
Will redirect to DBError.jsp when action "connectDB" thows any exception

Exception Handling: Struts 2 (Cont'd)

Logging Exception

- Log exceptions to page and/or log files
- To enable logging declare an interceptor as shown:

Enable Logging



Note: The code snippet should be included in struts.xml

Best Practices for Error Handling

Best Practices for Handling Exceptions in Java

1. Throw exceptions that are relevant to the interface

- A function should throw exceptions that are relevant to its **interface**

  Example: `java.io.FileInputStream.FileInputStream(String name)` throws `FileNotFoundException`
- The above code is used for opening a file on disk for reading bytes
- The given constructor may throw a `FileNotFoundException` and not `IndexOutOfBoundsException` that is completely irrelevant to the code
- Developers should make sure to throw **exceptions** that are **relevant** to the interface

2. The reason for exception should be specified

- The exceptions should always **specify what caused** them to occur
- Exceptions can be handled in either of the following two ways:
  - Generating a **separate exception** for each and every event
  - Creating a **generic exception** and mentioning what caused that exception
- The best practice is to use an exception class when handling specifically with **separate code**
- Use a single generic exception class for handling exceptions in a **default way**

Best Practices for Handling Exceptions in Java (Cont'd)

3. Encapsulation

- In a scenario where a function throws an exception that is received from another function, it should be encapsulated in a locally generated exception class
- In the sample code, `FileNotFoundException` is converted into `ActionException`
- When the code encounters the `FileNotFoundException` inside the `try` block, the `catch` clause throws an `ActionException`
- The execution thread is suspended and the exception is reported. The function `getCause()` can be made use of to find the root exception cause

Example Code

General.java

```
5
6  public class General {
7      public static FileOutputStream main(String sname) thr
8      {
9
10         try {
11             String fName = sname;
12             return new  FileOutputStream(fName);
13             } catch (FileNotFoundException fe) {
14             throw new ActionException(fe);
15             }
16
17     }
18 }
```

Best Practices for Handling Exceptions in Java (Cont'd)

4. Balance the catch blocks

- There are two approaches that can be used to balance the `catch` blocks:

The first approach is to use a single `catch` block for all the events.

The second approach is to use multiple `catch` blocks .i.e., each exception will have a separate `catch` block

For example: Many catch blocks one for each Exception

General.java

```
1  package com.ec.web;
2
3  public class General {
4      public static void main(String args[]) throws Command
5      try {
6
7      } catch (Throwable t) {
8          throw new CommandExecutorException(t);
9      }
10
11     }
12 }
```

General.java

```
4      try {
5
6          System.out.println("Here Statements");
7      }
8      catch (ArrayIndexOutOfBoundsException  ex) {
9
10     }
11     catch (ArithmeticException a) {
12
13     }catch(RuntimeException x) {}
14
15     }
```

- This approach is not a good practice as it does not give control over the exceptions being caught
- Developers should follow this approach as it is easy to execute various operations depending on the exceptions thrown

Best Practices for Handling Exceptions in Java (Cont'd)

5. Create custom error pages

- A generic error page should be used for all exceptions
- This prevents an attacker from retrieving internal response to errors

This can be done using the following ways:

1. Declarative Exception Handling using Struts

struts-config.xml

```
17
18 </action-mappings>
19 <global-exceptions>
20 <exception
21  type="ECProject.Info.EmployeeNotFoundException"
22  path="/Sorry.jsp"
23  key="EC.proj.SorryAct">
24 </exception>
25 </global-exceptions>
26
27 </struts-config>
```

Design Source

- The code should be included in struts-config.xml

2. Using Java Servlets

web.xml

```
3 <error-page>
4 <exception-type>EmployeeNotFoundException</exception-type>
5 <location>Sorry.jsp</location>
6 </error-page>
```

Design Source

- The code should be implemented in web.xml

Best Practices for Handling Exceptions in Java (Cont'd)

6. Exception names should be meaningful

- The exception names must be **meaningful** and should **express** the **action** that caused them
- Programmers should make sure to use **specific exceptions** and not generic exceptions
- Example: **java.io.FileNotFoundException** can be used as it implies that file is not found
- **java.lang.Exception** should not be used as it is very **generic**

7. Throw early and catch late

- This principle should be followed by all **programmers**
- The exception should be **thrown early** and **caught late**

Introduction to Logging