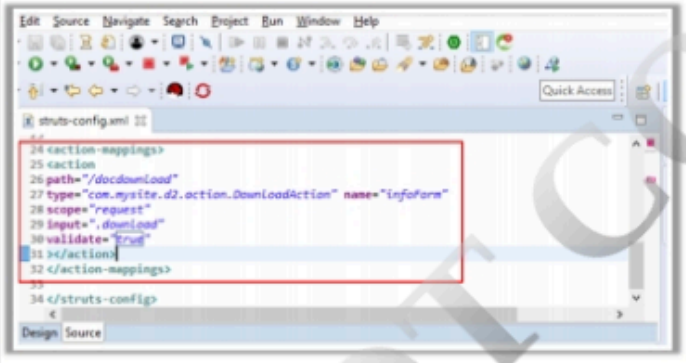
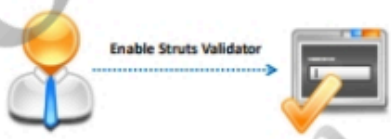
Enable the Struts Validator

- The action form mapping for the validate() method should always be given the value true
- The form disables the struts validator if the action form mapping is disabled

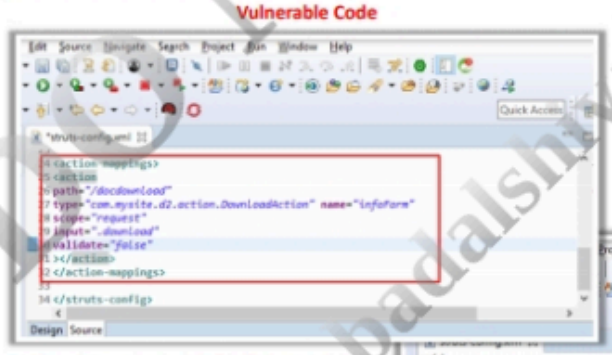
The following code shows how to enable action form mapping





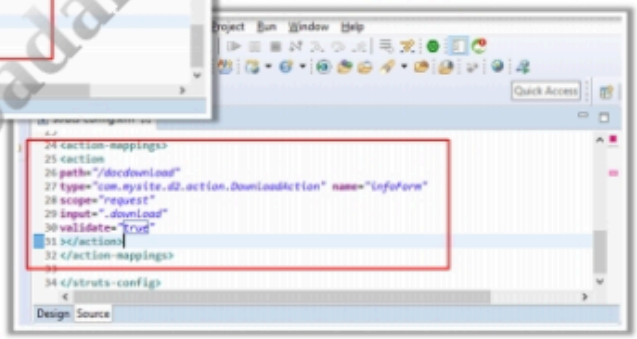
Secure and Insecure Struts Validator Code

Vulnerable Code



- The code is vulnerable as it deactivates the struts validator as the validate parameter is set to false

Secure Code



- The code is secure as it deactivates the struts validator as the validate parameter is set to true

Struts 2 Framework Validator

- Struts 2 implements **XWork framework** for input validations
- Struts 2 validation framework separates the **actual validation** logic and **application code**
- It can handle both client side and server side validations
- For configuring the validation framework **validators.xml** file or **annotation** is used
- If **validators.xml** file is **not present** in **class path**, the **default.xml** present in com/opensymphony/xwork2/validator/validators/folder is used
- Data of the action class is validated using **validate()** method

Struts 2 Framework Validator (Cont'd)

Validators Types

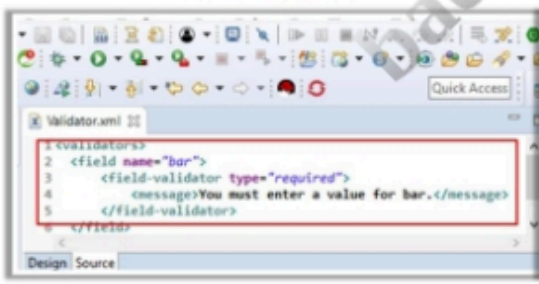
Field Validators

- Implemented on **fields accessible** using action
- Field validators are bound to individual fields and can display specific error messages for individual fields

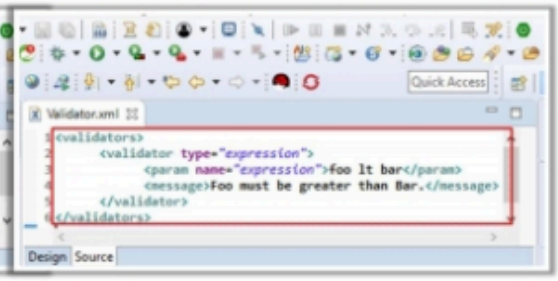
Non Field Validators

- Implemented at **action level**
- Non field validators are bound to action and display action specific messages

Example : Field Validator



Example : Non Field Validator

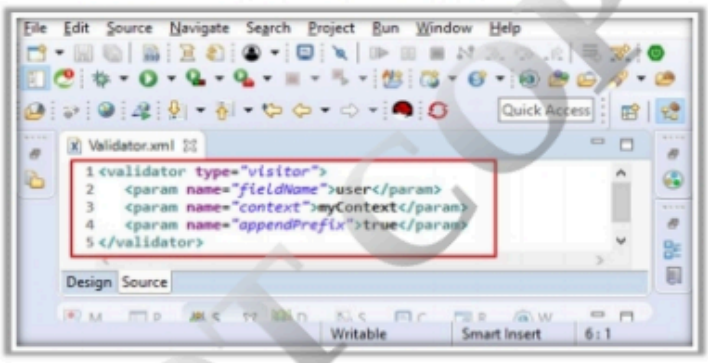


Struts 2 Framework Validator (Cont'd)

Visitor Field Validators

- Implemented on **model properties** of the action
- Can validate simple object properties, collection of objects, arrays etc.

Example : Visitor Field Validator



Struts 2 Framework: Built-in Data Validators

List of Some Built-in Data Validators

Validator Name	Class Name	Validation
required	RequiredFieldValidator	If field is not null
requiredstring	RequiredStringValidator	If string not-null and contains length > 0
int	IntRangeFieldValidator	If integer within a certain range
stringlength validator	StringLengthFieldValidator	If String field is of a certain length based on minlenth and maxlength specified
email validator	EmailValidator	If field contains valid email address
date validator	DateRangeFieldValidator	If the date field is within a specified parameter range
conditionalvisitor	ConditionalVisitorFieldValidator	Conditional validator will forward the request to the conditionalfield validator if the expression is true
Expression	ExpressionValidator	Validated based on regular expression supplied
fieldExpression	FieldExpressionValidator	Validates a field based on OGNL expression
Double	DoubleRangeFieldValidator	if double within a certain range
url	URLValidator	If the field value is valid URL
regex	RegexFieldValidator	Validates a field based on regular expression

Struts 2 Framework Annotation Based Validators

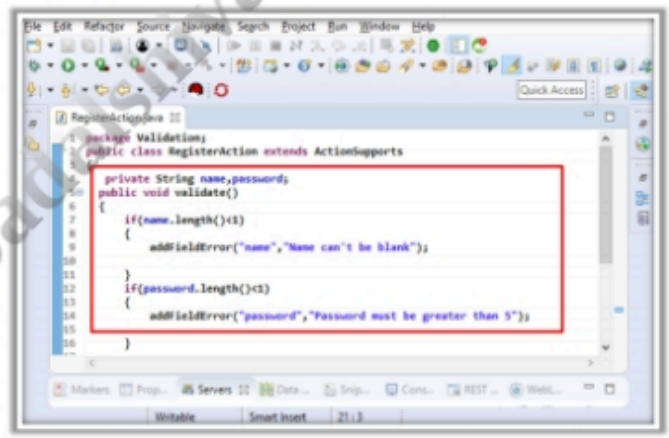
List of Annotation Based Validators

Annotation Name	Validation
@RequiredStringValidator	If the string is not null and length >0
@RequiredFieldValidator	If the string is not null
@EmailValidator	If the field is valid email address
@RegexFieldValidator	If the field matches the provided regular expression
@IntRangeFieldValidator	If the numeric field is within specified range
@StringLengthFieldValidator	If the field length is within the specified minimum and maximum length
@DateRangeFieldValidator	If the date field is within specified range
@FieldExpressionValidator	If the field matches the provided OGNL expression
fieldExpression	FieldExpressionValidator
Double	DoubleRangeFieldValidator
url	URLValidator
regex	RegexFieldValidator

Struts 2 Custom Validation: Workflow Interceptor

- Implement **Validateable** and **ValidationAware** interface in the action class to create custom validations
- The **validate()** method of Validateable interface should be overridden to implement custom logic
- **Workflow interceptor** is responsible for extracting the defined error details and messages of the action class
- **Field** and **Action** level validations can be implemented by the **ValidationAware** interface

Example: Action class implementing custom validation logic by overriding the validate()



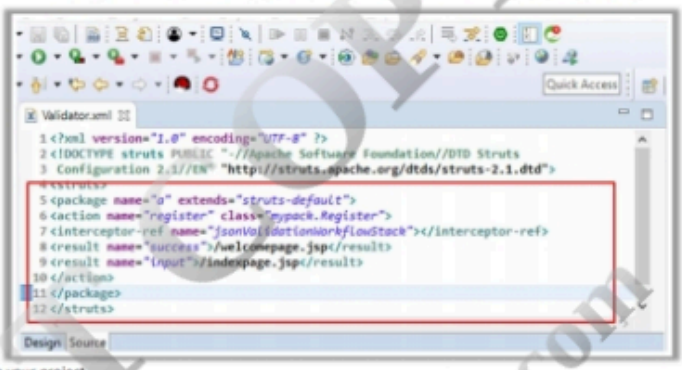
Struts 2 AJAX Validation: jsonValidation Interceptor

- To implement AJAX validation use **jsonValidation** interceptor along with validation interceptor
- jsonValidationWorkflowStack package contains **jsonValidation**, validation and workflow interceptors and basicstack to implement AJAX Validations

Steps to Implement AJAX Validation

- Design input from
- Inherit the ActionSupport class in the action
- Implement validation in validation.xml file
- Define error message
- Register jsonValidationWorkflowStack in Validator.xml

Example: Validator.xml with Result and jsonValidationWorkflowStack Interceptor



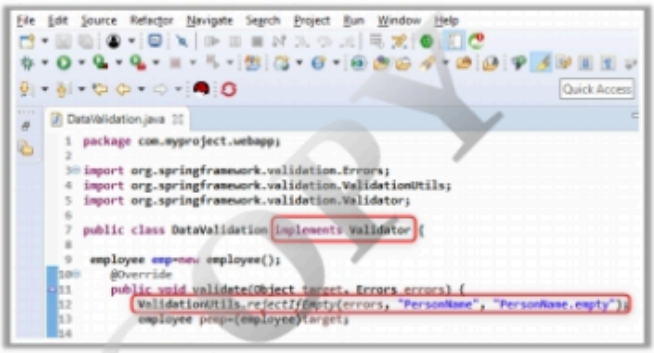
Note: To implement AJAX validation include dojo plugin in your project

Data Validation: Spring Framework

Spring Validator

- **Spring Validator** supports various validation methods, easy to **customize** and **implemented** as plugin to support any validator
- The spring validator interface **org.springframework.validation.Validator** is used to validate objects on server. It contains two methods
  - Support (returns Boolean): Checks if the target class can be validated by the validator
  - Validate : Performs the validation
- In spring the validator interface uses the error object for providing error report of validation
- While validating **error object reports** about failure validation
- Annotations are also used to validate spring application like **@Validator**

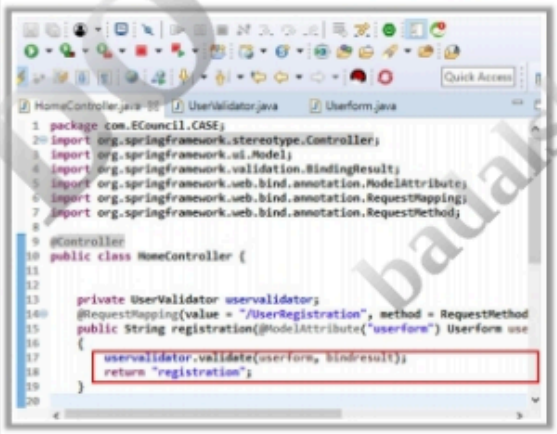
Example using Spring Validator



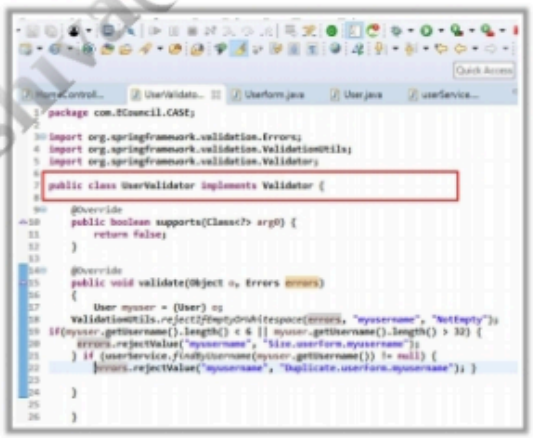
```
package com.myproject.webapp;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class DataValidation implements Validator {

 employee emp=new employee();
   @Override
   public void validate(Object target, Errors errors) {
       ValidationUtils.rejectIfEmpty(errors, "PersonName", "PersonName.empty");
        employee emp=(employee)target;
```

Data Validation: Spring MVC Framework

Example: Spring MVC Data Validation



```
package com.ECouncil.CASE;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {

    private UserValidator uservalidator;
    @RequestMapping(value = "/UserRegistration", method = RequestMethod
    public String registration(@ModelAttribute("userform") Userform use
    {
        uservalidator.validate(userform, bindresult);
        return "registration";
    }
```

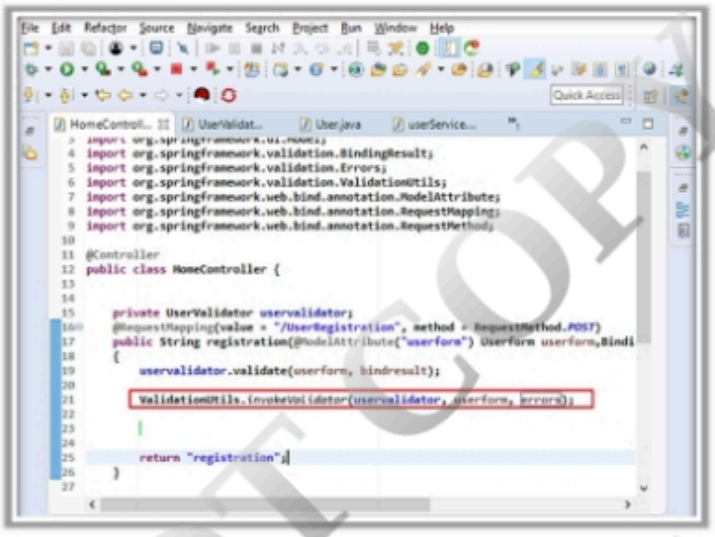


```
package com.ECouncil.CASE;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class UserValidator implements Validator {

    @Override
    public boolean supports(Class<?> arg0) {
        return false;
    }

    @Override
    public void validate(Object o, Errors errors)
    {
        User myuser = (User) o;
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "myusername", "NotEmpty");
if(myuser.getUsername().length() < 6 || myuser.getUsername().length() > 32) {
        errors.rejectValue("myusername", "Size.userform.myusername");
    } if (userService.findByUsername(myuser.getUsername()) != null) {
        errors.rejectValue("myusername", "Duplicate.userform.myusername"); }

    }

    }
```

- **Validate()** method of the userValidator class validates userform and binds the result
- The class **UserValidator** implements Validator interface at model class and this class object is used in the controller

Implementing Validator

- Validator can be implemented using **invokeValidator()** method in the controller code



JSR 380 Bean Validator API

- JSR 380 is the **Java API** specification for **Bean Validation**
- It is used to validate model properties using annotations

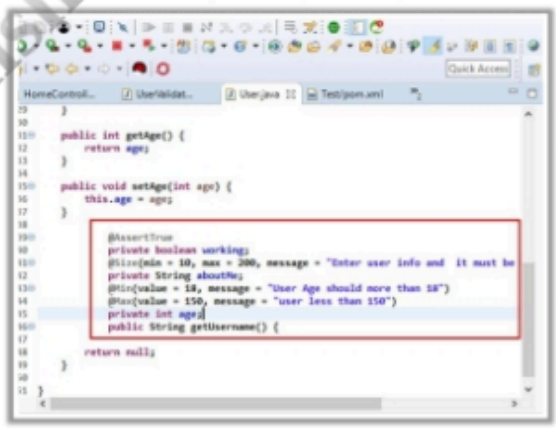
- **Standard JSR Annotations**
  - **@NotNull:** It will validate value is not null
  - **@AssertTrue:** It will validate Boolean value
  - **@Size:** This validates the size of the attribute like for string size min or max
  - **@Min:** It validates minimum size value of the attribute
  - **@Max:** It validates maximum size value attribute
  - **@Email:** validates the Email address
  - **@NotEmpty:** It validates Not null property
  - **@NotBlank**
- **Defining Error message globally**

```
javax.validation.constraints.Min.message=Value
must be at least {value}
```

- **Defining Error message as property**

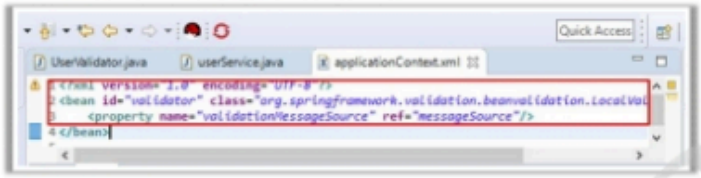
```
@Max(value=120, message="{age.too.old}")
//Annotation
age.too.old=Age must be under 120.// Message
bundle property
```

Manually Passing Error Message



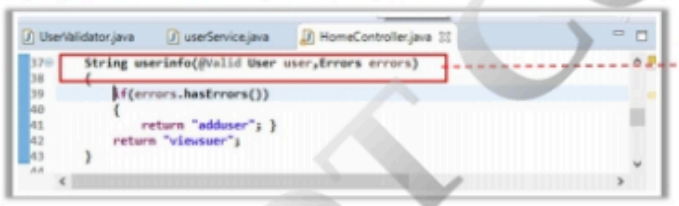
Configuring JSR 380

Configure validator in the applicationContext



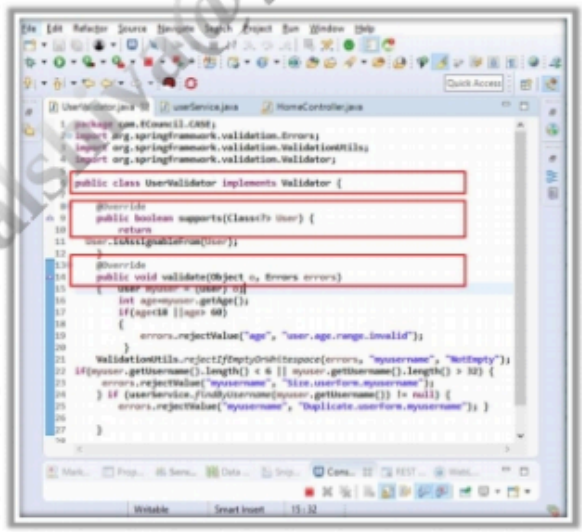
***-servlet.xml**

Spring MVC will automatically validates inputs from the controller



Custom Validator Implementation in Spring

- Custom validation is required for some special business parameter values, for example validating age between 18 to 60
- Custom validator can be implemented using **Validator** interface
- Override the **supports**(Class mycls) method
- Override **validate**(Object o, Errors errors) method



Spring Validation and Security

1. Spring validation is done to prevent attacks caused through **unchecked input**
2. All the inputs should be checked using the **spring validator** before they are processed by the application
3. Each and every field included in the form should be validated in the **corresponding validation form**
4. Spring support groups and roles in application
5. Vulnerabilities in input validation may lead to **buffer overflow attacks, cross-site scripting, SQL injection**, etc.

Input Validation Errors

Input Validation Errors: Improper Sanitization of Untrusted Data

- Web applications accept data from **untrusted sources, contradicted users** and **network connections** and then transfer data from one trusted boundary to another trusted domain to execute
- The data travels across domains in string format with some **internal syntactic structure** that the subsystem should **parse**
- If the subsystem cannot handle the **malicious input** properly it may lead to different types of injection attacks such as SQL injection, XML injection, etc.

Vulnerable Code for SQL injection

- The vulnerable code is an example of SQL injection vulnerability that occurs because the **SQL statement** used in the code **sqlString** accepts unsanitized input arguments

```
ValidateUser.java
3  import java.sql.Connection;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.sql.Statement;
7
8  public class ValidateUser {
9
10     public void doPrivilegedAction(String Uname, char[] pwd)
11 throws SQLException {
12 Connection conn = getConnection();
13 if (conn == null) { // handle error
14 }
15 try {
16 String passwd = hashPasswd(pwd);
17 String sqlString = "SELECT * FROM db_user WHERE username = '"
18 +Uname+ "' AND password = '" + passwd + "'";
19 Statement statmt = conn.createStatement();
20 ResultSet rSet = statmt.executeQuery(sqlString);
21 if (!rSet.next()) {
22 throw new SecurityException("Incorrect User name or password");
23 }
24 }catch (Exception e) {
25     // Logger handle exception
```


Input Validation Errors: Improper Sanitization of Untrusted Data (Cont'd)

- For sanitizing untrusted input variables, JDBC library is provided with an API for building SQL commands
- The **java.sql.PreparedStatement** class when used properly can escape input strings preventing SQL injection
- The **set*()** method of **preparedstatement** class performs strong type checking of user inputs

Secure Code for SQL Injection

- In the following code, **doPriviledgedAction()** method uses **preparedstatement** instead of **java.sql.statment**
- This method also **validates** the **length** of username argument, preventing it from executing an absurdly long username intentionally submitted by an attacker

```
ValidateUser.java
7  public class ValidateUser {
8
9      public void callme(String Uname, char[] pwd) throws SQLException {
10
11
12         Connection conn = getConnection();
13             if(conn == null) {
14         }
15         try {
16             String passwd = hashPassword(pwd);
17             if (Uname.length() > 8) {
18             }
19             String sqlString ="select * from db_user where Uname=? and pwd=?";
20             PreparedStatement prdstmt = conn.prepareStatement(sqlString);
21             prdstmt.setString(1, Uname);
22             prdstmt.setString(2, passwd);
23             ResultSet rSet = prdstmt.executeQuery();
24             if (!rSet.next()) {
25                 throw new SecurityException("Incorrect username or password");
26             }
27         }
```

Input Validation Errors: Improper Sanitization of Untrusted Data (Cont'd)

Vulnerable Code for XML Injection

```
ValidateUser.java
1  package Validation;
2
3  import java.io.BufferedOutputStream;
4  import java.io.IOException;
5
6  public class ValidateUser {
7
8      private void createXMLStream(BufferedOutputStream bufOpStrm, String quantity) throws
9
10         String xmlStr = "<item>\n<description>Widget</description>\n" + "" + "<price>500
11                 + quantity + "</quantity></item>";
12         bufOpStrm.write(xmlStr.getBytes());
13         bufOpStrm.flush();
14     }
15
```

- In the code, for sending an XML query to a server, a simple **string concatenation** is used by a client method
- The client method used however **does not perform input validation** properly leading to an XML injection

Secure Code for XML Injection

```
5  import java.util.regex.Pattern;
6
7  public class ValidateUser {
8
9      private void createXMLStream(BufferedOutputStream bufOpStrm, String quantity) throws
10
11         if(!Pattern.matches("[0-9]+", quantity)) {
12
13         }
14         String xmlStr = "<item>\n<description>Widget</description>\n" +""
15                 + "<price>500.00</price>\n" +"<quantity>"+quantity+"</quantity></item>";
16         bufOpStrm.write(xmlStr.getBytes());
17         bufOpStrm.flush();
18         }
```

- In the code snippet **whitelisting method** is used to sanitize inputs
- The method **strictly validates** quantity field input numbers between 0 and 9

Input Validation Errors: Improper Validation of Strings

- Web applications that accept input strings from untrusted sources perform **filtering** and **validation mechanisms** based on the strings' character data
- Unicode standards are followed for character information in Java by:
  - Checking if two strings are **equivalent** to each other
  - Transforming a particular unicode normalization form to either **canonical** or **compatibility** equivalence

Normalization Forms

Form	Description
Normalization Form D (NFD)	Canonical Decomposition
Normalization Form C (NFC)	Canonical Decomposition, followed by Canonical Composition
Normalization Form KD (NFKD)	Compatibility Decomposition
Normalization Form KC (NFKC)	Compatibility Decomposition, followed by Canonical Composition

- Using normalization forms KC and KD for arbitrary input strings may sometimes remove **formatting distinctions** that are important for text semantics
- NFKC converts the input strings into an equivalent **canonical form** without altering formatting distinctions to the required input form
- The **iNormalize method** s used to convert **Unicode text** into an equivalent **composed** or **decomposed form** making sorting and searching of text easier

Input Validation Errors: Improper Validation of Strings (Cont'd)

- In the below code, String is **validated before normalizing** and it fails to detect any arbitrary inputs
- Validation logic also fails to detect inputs as a check for angle brackets does not detect alternate **Unicode representations**

Vulnerable Code

```
String Str = "\uFE64" + "script" + "\uFE65";
Pattern ptrn = Pattern.compile("[<>]");
Matcher mtchr = ptrn.matcher(Str);
if (mtchr.find()) {
    throw new IllegalStateException();
} else {
}
Str = Normalizer.normalize(Str, Form.NFKC);
```

- In the below code, validating is performed after normalizing string into **canonical angle brackets**
- Input validation mechanism throws an **IllegalStateException** if it detects any malicious inputs

Secure Code

```
String Str = "\uFE64" + "script" + "\uFE65";
Str = Normalizer.normalize(Str, Form.NFKC);
Pattern ptrn = Pattern.compile("[<>]");
Matcher mtchr = ptrn.matcher(Str);
if (mtchr.find()) {
    throw new IllegalStateException();
} else {
}
```

Input Validation Errors: Improper Logging of User Inputs

- Log injection attacks may occur if a log contains **malicious user inputs** due to improper validation mechanism or improper sanitization
- Attackers can split a log entry into two log entries using **carriage return** and **line feed** (CRLF) **sequences** or insert a fake log entry
- Also, logging malicious inputs that are not sanitized can result in **information leakage**

Vulnerable Code

```
if (loginCorrect) {
    logger.severe("User login succeeded for: " + username);
} else {
    logger.severe("User login failed for: " + username);
}
```

- In the code, username is logged for **invalid user request** without proper validation

Secure Code

```
if((!Pattern.matches("[A-Za-z0-9_]+", username)))
{
    logger.severe("User login failed for unauthorized user");
}
else if (loginCorrect) {
logger.severe("User login succeeded for: " + username);
}
else {
logger.severe("User login failed for: " + username);
}
```

- In the code, validation is performed before **logging username**

Input Validation Errors: Improper Incorporation of Malicious Inputs into Format Strings

- In Java, **format strings** implementation is **stricter** than in any other languages
- If any conversion argument **fails** to **match** with the corresponding **format specifier**, implementation method throws an **exception**
- This method aims at **limiting malicious exploits**; although a malicious user input can yet exploit format strings and cause denial of service or data leaks
- It is advisable to not **incorporate** any **input strings** of untrusted sources into format strings

Vulnerable Code

```
1  package Validation;
2
3  import java.util.Calendar;
4  import java.util.GregorianCalendar;
5  public class ValidateUser {
6      static Calendar cal=new GregorianCalendar(1995, GregorianCalendar.MAY, 23);
7          public static void main(String[] args)
8          {
9
10             // args[0] is the credit card expiration date
11             // args[0] can contain either %1$tm, %1$te or %1$tY as malicious arguments
12             // First argument prints 05 (May), second prints 23 (day)   and third prints
13             // Perform comparison with cal, if it doesn't match print the following line
14         System.out.printf(args[0] +
15                 " did not match! HINT: It was issued on %1$terd of some month", cal);
16         }
```

- The code describes an **information leak** issue
- It accepts the expiration date of a credit card and incorporates it within the format string

Secure Code

```
1  package Validation;
2
3  import java.util.Calendar;
4  import java.util.GregorianCalendar;
5  public class ValidateUser {
6      static Calendar cal=new GregorianCalendar(1995, GregorianCalendar.MAY, 23);
7          public static void main(String[] args)
8          {
9
10             // args[0] is the credit card expiration date
11             // Perform comparison with cal, if it doesn't match print the following line
12         System.out.printf("%s did not match! " + " HINT: It was issued on %1$terd of some month", args[0],cal);
13     }
14 }
```

- In the code, it is ensured that user inputs are not **incorporated** in the **format string**

Input Validation Errors: Inappropriate Use of Split Characters in Data Structures

- The Java language string occupies **16 bits**
- Java char cannot represent various possible **Unicode characters**
- Splitting a char or byte array risks splitting a **multi byte character**
- Attackers can bypass the input validation checks when characters are split between two data structures

Vulnerable Code

```
1  package Validation;
2
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.net.Socket;
6
7  public class ValidateUser {
8
9      public final int MaxSize = 1024;
10     public String read_Bytes(Socket skt) throws IOException {
11         InputStream ins = skt.getInputStream();
12         byte[] data = new byte[MaxSize+1];
13         int offset = 0;
14         int bytesRead = 0;
15         String str = new String();
16         while ((bytesRead = ins.read(data, offset, data.length - offset)) != -1) {
17         offset += bytesRead;
18         str += new String(data, offset, data.length - offset, "UTF-8");
19          if (offset >= data.length) {
20          throw new IOException("Too much input");
21             }
22          }
23          ins.close();
24          return str;
25         }
26 }
```

- The code fails to interact between characters of **multibyte encoding** and **boundaries** between the **loop iterations**

Input Validation Errors: Inappropriate Use of Split Characters in Data Structures (Cont'd)

- The code eliminates the splitting of multi byte encoded characters in the buffers
- It also defers the construction of the result string until the data is read completely

Secure Code

ValidateUser.java

```
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;

public class ValidateUser {

    public final int MaxSize = 1024;
    public String read_Bytes(Socket skt) throws IOException {
        InputStream ins = skt.getInputStream();
        byte[] data = new byte[MaxSize+1];
        int offset = 0;
        int bytesRead = 0;
        while ((bytesRead = ins.read(data, offset, data.length - offset)) != -1) {
        offset += bytesRead;
        if (offset >= data.length) {
        throw new IOException("Too much input");
        }
        }
        String str = new String(data, "UTF-8");
        ins.close();
        return str;

        }
}
```


Input Validation Errors: Improper Validation of Non-character Code Points

- Attackers can bypass the validation checks if the input validation is done without the removal of non-character code points

Vulnerable Code

- The code below shows how only valid ASCII characters are accepted
- It also shows deletion of non-ASCII characters validation without deleting non-ASCII characters
- It allows an attacker to bypass the validation checks by changing the <script> tag

ValidateUser.java

```
package Validation;

import java.io.BufferedOutputStream;
import java.io.IOException;
import java.text.Normalizer;
import java.text.Normalizer.Form;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ValidateUser {

    private void createXMLStream(BufferedOutputStream bufOpStrm, String quantity) throws IOException {

        String str = "<scr" + "\uFEFF" + "ipt>";
        str = Normalizer.normalize(str,Form.NFKC);
              Pattern ptrn = Pattern.compile("<script>");
        Matcher mtchr = ptrn.matcher(str);
        if (mtchr.find()) {
          System.out.println("Found black listed tag");
        } else {

        }
        str = str.replaceAll("^\\p{ASCII}]","");
```

Input Validation Errors: Improper Validation of Non-character Code Points (Cont'd)

Secure Code

- The code below gives example of how a Unicode sequence \uFFD is replaced with some unknown characters
- <script> tag is checked before performing sanitation
- This process avoids malicious input from bypassing the filters

ValidateUser.java

```
4  import java.io.IOException;
5  import java.text.Normalizer;
6  import java.text.Normalizer.Form;
7  import java.util.regex.Matcher;
8  import java.util.regex.Pattern;
9
10 public class ValidateUser {
11
12@   private void createXMLStream(BufferedOutputStream bufOpStrm, String quantity)
13           throws IOException {
14
15       String str = "<scr" + "\uFEFF" + "ipt>";
16       str = str.replaceAll("^\\p{ASCII}]", "\uFFFD");
17       str = Normalizer.normalize(str,Form.NFKC);
18             Pattern ptrn = Pattern.compile("<script>");
19       Matcher mtchr = ptrn.matcher(str);
20       if (mtchr.find()) {
21         System.out.println("Found black listed tag");
22       } else {
23
24       }
25       str = str.replaceAll("^\\p{ASCII}]","");
```

Input Validation Errors: Improper Use of String Modification

Vulnerable Code for XML Injection

- String should not be modified after validation. By doing that it allows to attacker to bypass validation
- With the help of Unicode perform the modification of the string to validate properly

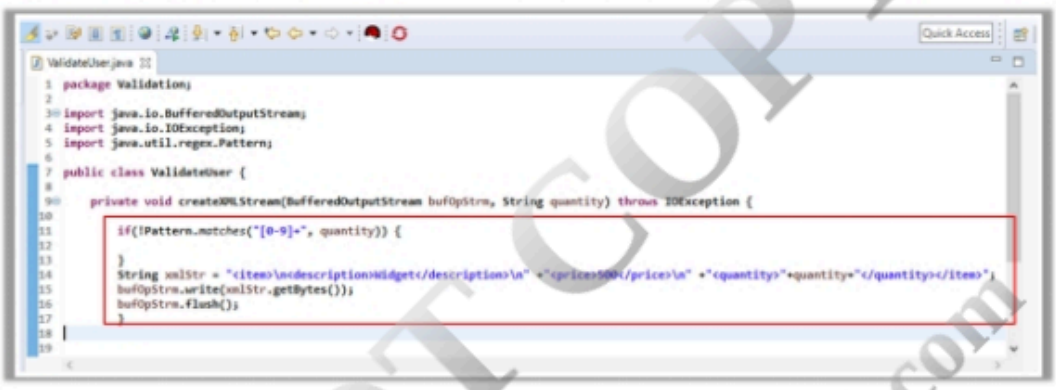
ValidateUser.java

```
1  package Validation;
2
3@ import java.io.BufferedOutputStream;
4  import java.io.IOException;
5
6  public class ValidateUser {
7
8@   private void createXMLStream(BufferedOutputStream bufOpStrm, String quantity) throws IOException {
9        String xmlStr;
10       xmlStr = "<item>\n<description>Widget</description>\n" +
11       "<price>500.0</price>\n" +
12       "<quantity>" + quantity + "</quantity></item>";
13       bufOpStrm.write(xmlStr.getBytes());
14       bufOpStrm.flush();
15   }
```

Input Validation Errors: Improper Use of String Modification (Cont'd)

Secure Code for XML Injection

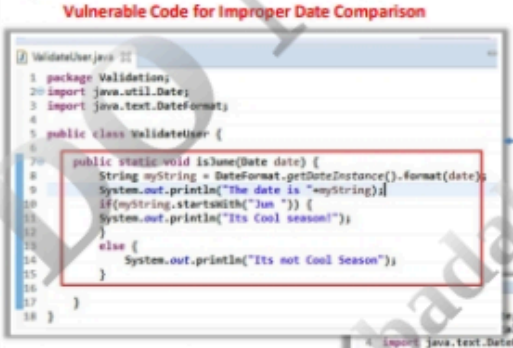
- In the code snippet the **whitelisting method** is used to sanitize inputs
- The method **strictly validates** quantity field input numbers between 0 and 9



```
ValidateUser.java
1  package Validation;
2
3  import java.io.BufferedOutputStream;
4  import java.io.IOException;
5  import java.util.regex.Pattern;
6
7  public class ValidateUser {
8
9      private void createXMLStream(BufferedOutputStream bufOpStrm, String quantity) throws IOException {
10
11         if(!Pattern.matches("[0-9]+", quantity)) {
12
13         }
14         String xmlStr = "<item>\n<description>Widget</description>\n" +"<price>500</price>\n" +"<quantity>"+quantity+"</quantity></item>";
15         bufOpStrm.write(xmlStr.getBytes());
16         bufOpStrm.flush();
17         }
18
19
```

Input Validation Errors: Improper Comparison of Locale-dependent Data

Vulnerable Code for Improper Date Comparison



```
ValidateUser.java
1  package Validation;
2  import java.util.Date;
3  import java.text.DateFormat;
4
5  public class ValidateUser {
6
7      public static void isJune(Date date) {
8          String myString = DateFormat.getDateInstance().format(date);
9          System.out.println("The date is "+myString);
10         if(myString.startsWith("Jun ")) {
11         System.out.println("Its Cool season!");
12         }
13         else {
14             System.out.println("Its not Cool Season");
15         }
16
17     }
18 }
```

- Ensure the date specification as per correct locales
- Improper comparison may lead to unexpected results and bypass of input validations

Secure Code for Locale Comparison

```
import java.text.DateFormat;

public class ValidateUser {

    public static void isJune(Date date) {
        Date rightNow = null;
        @SuppressWarnings("null")
        String myString = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.US).format(rightNow.getTime());
        System.out.println("The date is "+myString);
        if(myString.startsWith("Jun ")) {
        System.out.println("Its Cool season!");
        }
        else {
            System.out.println("Its not Cool Season");
        }
    }
}
```

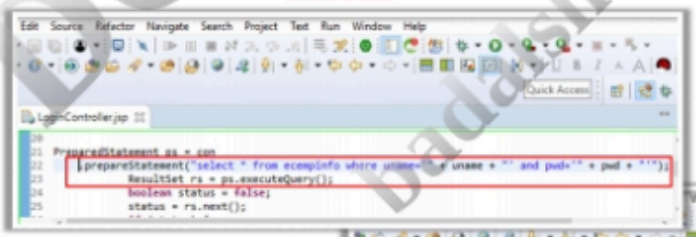
- The code snippet implements US Local while defining the date parameters

Common Secure Coding Practices

SQL Injection

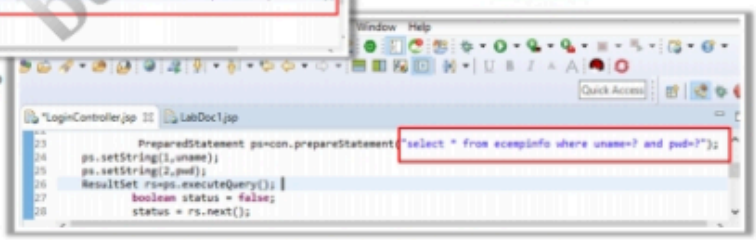
- SQL injection is a code injection technique that uses the **security vulnerability** of a database for its attacks
- Attackers inject a **malicious code** into the strings that are later on passed on to SQL server for execution

Insecure Code



- In the code, **concatenating strings** are used to form an SQL query resulting in SQL injection attacks

Secure Code

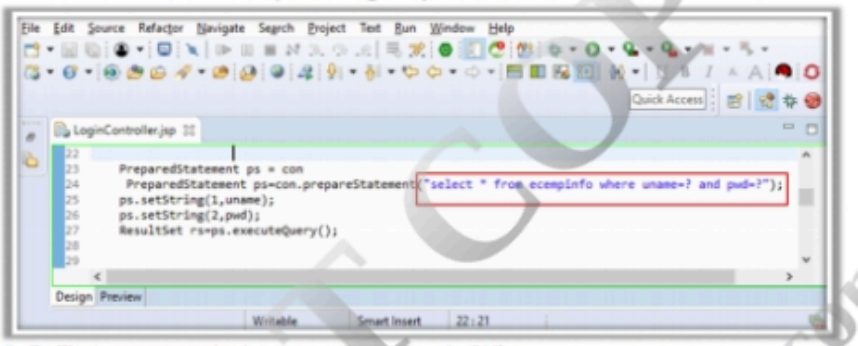


- In the code, **parameterized queries** are used to prevent SQL injection attacks

Prepared Statement

- Prepared statements define **methods** and **properties** that are used to send SQL commands to retrieve data from the database
- They can be used with or without **parameters**
- Increased performance of the code in case of repeated SQL statements
- They are also known as **pre-compiled statements**
- Prepared statements are supported by various DBMS like **Oracle, MySQL, DB2, SQL Server** and **PostgreSQL**

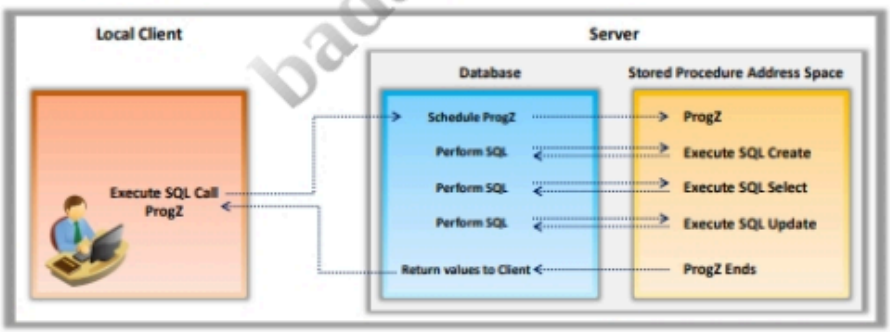
Example: Creating a Prepared Statement



- The '?' is the **parameter marker** that represents the parameters (in JDBC)

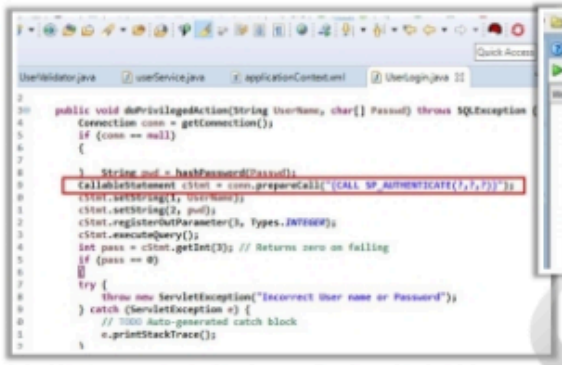
Stored Procedures

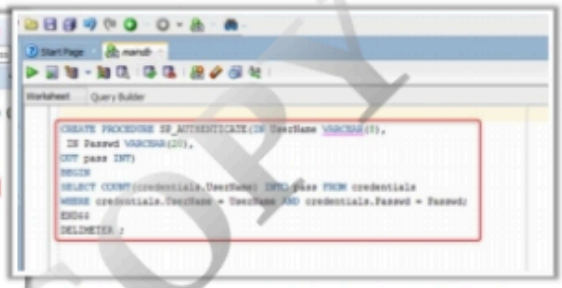
- Stored procedures are user written programs that are stored in **database servers** and can be invoked by client applications
- Stored procedures can **execute SQL statements** at the server, as well as application logic level, for a specific function
- The SQL CALL statement is part of the **SQL ISO/ANSI standard** for invoking stored procedures from database management systems
- Client applications can invoke a stored procedure by using a single **SQL CALL statement**
- Client programs can **pass** and **receive** parameters from stored procedure



Vulnerable and Secure Code for Stored Procedures

Example: Implementing Stored Procedure





- java.sql.CallableStatement API is used to execute SQL Stored Procedures. CallableStatement is not vulnerable to SQL injection unless the called stored procedure code is vulnerable.
- The SELECT statement restricts the size of parameters and length of input variables

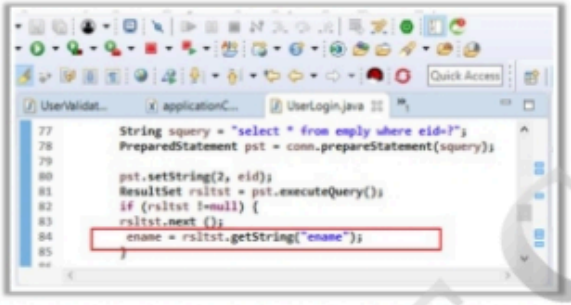
Stored Procedures for Securing Input Validation

1. Stored procedures enable a user to split the application logic between the client and server
2. This technique is used to prevent the client applications from manipulating the contents of the server data
3. Stored procedures are secured through encoding in the database and require admin privileges to execute the program
4. Stored procedures perform input validation of data by verifying text, characters and numeric data
5. Stored procedures prevent SQL injection attacks by performing thorough validation on any user input before processing the data
6. Stored procedures use a safe interface such as a callable statement in JDBC (Java Database Connectivity)
7. Stored procedures also manage network traffic where a single SQL statement can be used to send and receive a series of SQL statements

Cross-site Scripting (XSS)

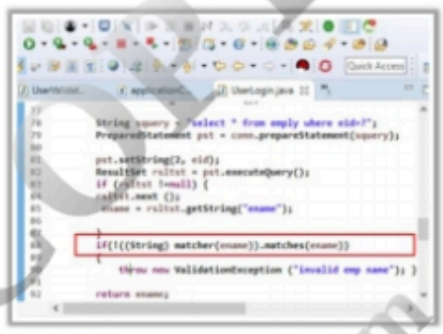
- Cross-site scripting ('XSS' or 'CSS') attacks exploit vulnerabilities in **dynamic environment web pages** that enable attackers to inject client-side script into web pages viewed by other users

Insecure Code



- The insecure code snippet queries a database for an employee id **without proper input validation mechanism** and prints the employee name

Secure Code



- The secure code snippet **uses input validation** and **output encoding** to prevent attackers from executing any malicious scripts

Whitelisting vs Blacklisting

Whitelisting

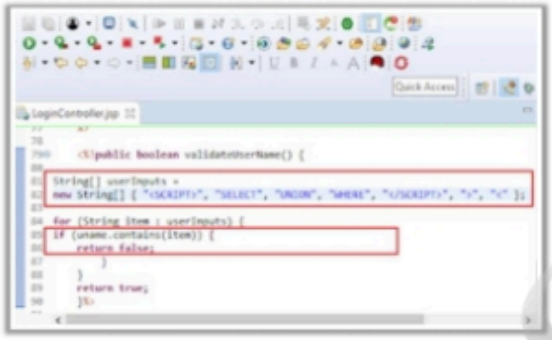
- Whitelisting is a process to **verify** the **inputs** against possible expected known values
- The **maintenance costs** of blacklist coded applications are more than the whitelist coded one
- As a whitelist expects only **known good desired values** and rejects every thing else, therefore it is a good approach in input validation
- **Regular expressions** help in creating a whitelist for the **secured code**
- Before **implementing** regular expressions in code, **test** the regular expressions itself

Blacklisting

- Testing input values against a set of negative expected inputs is called **blacklisting**
- Checking all the input values received with that of the known **bad** or **negative conditions** is called blacklisting
- Developers cannot think of every possible bad input **permutation** and insert it in the code
- Therefore a blacklist works well only with the list created by the developer, but all new **vectors** will be missed

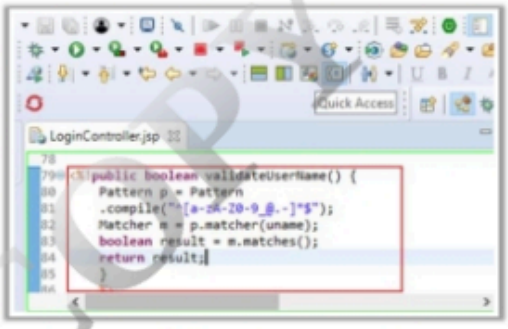
Vulnerable and Secure Code for Blacklisting & Whitelisting

Vulnerable Code



- In the code, the validateUserName function uses the blacklisting approach

Secure Code



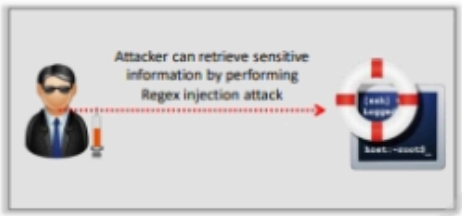
- In the code , the validateUserName function uses the whitelisting approach

Vulnerable and Secure Code for Blacklisting & Whitelisting (Cont'd)

Vulnerable Code

Regular Expressions

- A regular expression provides a **concise** and **flexible** means to "match" (specify and recognize) strings of text, such as particular **characters**, **words**, or **patterns** of characters
- There are even chances of modifying the input of a regular expression by the attacker; such type of an attack is known as a **regex injection attack**
- Proper input validation helps in avoiding such attacks



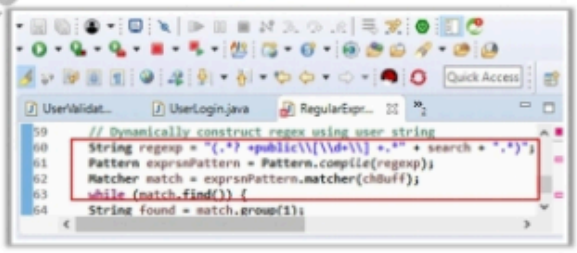
Expression	Description
(013)	A single digit 0, 1, or 3
(0-9) (0-9)	Any two-digit number from 00-99
(0-9 && (^4 5 6 7)	A single digit that is 0, 1, 2, 3, 8, or 9
(a-z 0-9)	A single character that is either a lowercase letter or a digit
(a-z A-z) (a-z A-Z 0-9_$)	A valid Java identifier consisting of alphanumeric characters, underscores and dollar signs with the first character being an alphabet
(wb) (ad I eed)	Matches wad, weed, bad, and beed
(AZ I CA I CO) (0-9) (0-9)	Matches AZxx, CAxx and COxx where x is a single digit

Vulnerable and Secure Code for Regular Expressions

Vulnerable Code

- The sample code regularly loads a log file into memory and clients are allowed to get the keyword search suggestions after passing the keyword as an argument to suggestSearches()
- This code might also lead to Regex injection attacks

Secure Code



- This secure code filters the non-alphanumeric characters excluding 'space' and 'single quote' from the search string. This code prevents regexp injection

Character Encoding

- In character encoding systems, code is paired with each character from a presumed **repertoire** that may include bit patterns, octets and number sequences
- **American Standard Code for Information Interchange** (ASCII) and **Unicode** are the recognized standards used in character encoding systems
- Character encoding is related to input/output where text is transmitted between different systems using **encoding schemes**

Vulnerable and Secure Code for Character Encoding

Vulnerable Code

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class CharacterEncoding {

    public static void main(String args[]) {

        FileInputStream fInStr = null;
        try {
        fInStr = new FileInputStream("AnyFile");
        DataInputStream dInStr = new DataInputStream(fInStr);
        byte[] bData = new byte[1024];

        dInStr.readFully(bData);
        String result = new String(bData);
```

- In the code, the difference between **default encoding** and **byte array** produced during encoding causes **undefined behavior** for inputs

Secure Code

```
import java.io.FileInputStream;
import java.io.IOException;

public class CharacterEncoding {

    public static void main(String args[]) {

        FileInputStream fInStr = null;
        try {
        fInStr = new FileInputStream("AnyFile");
        DataInputStream dInStr = new DataInputStream(fInStr);
        byte[] bData = new byte[1024];

        dInStr.readFully(bData);
        String encoding="UTF-16LE";
        String result = new String(bData,encoding);
```

- The code explicitly specifies intended **character encoding** in the **String constructor**

Checklist for Character Encoding

- Strictly implement standard **ASCII** or **Unicode character** encoding techniques to prevent **injection attacks**
- Sanitize rejected data and don't store erroneous log data in applications
- Create rules for maximum length and usage of special characters for inputs from external sources
- Use **HTML encoding** and **URL encoding** for user inputs to restrict script injection attacks from running in the browser
- Use compatible encodings on both sides of a communication channel to eliminate risks of data corruption
- Create **rules for data validation** by listing input characters from strongest to weakest such as:
  1. Limit exact matches
  2. Accept desirable known
  3. Reject known undesirables
  4. Sanitize known undesirable matches

Cross-site Scripting (XSS) Countermeasures

1. Use HTML and URL encoding to ensure all the **input values** are validated before being echoed back to the user
2. Take necessary precautions while using **dynamic environments** that execute script and retrieve data
3. Configure the **web browser** to disable scripting
4. Use **filter techniques** that store and process input variables on the server
5. Use properly designed **error handling mechanisms** for reporting input errors
6. Implement **character encoding techniques** for web pages such as ISO-8859-1 or UTF 8

HTML Encoding

I. HTML encoding is used to **protect unsafe output** from attackers

II. The variables found in an application should be **validated** and **encoded** before the output is sent to the HTML pages

III. Converting JavaScript and HTML tags through encoding protects the code from **cross-site scripting vulnerability**

IV. User data that does not implement HTML encoding may result in **markup injection**

V. HTML encoding is done by converting the html entities into encodings as "<script>", and is displayed as "<script>"

Examples:

- / --> /
- & --> &
- ' --> '
- < --> <
- " --> "
- > --> >

Vulnerable and Secure Code for HTML Encoding

Vulnerable Code

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859
<title>Sample Captcha</title>
</head>
<body>
String result=request.getParameter("result");
Student Result: <%=result%>
</body>
</html>
```

- The given code **requests** input from the user
- The code is **executed correctly** when the user gives valid input
- As there is no input validation, the code becomes **vulnerable** if the user enters **invalid input**

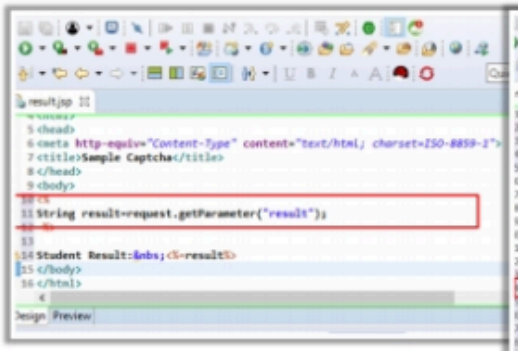
Secure Code

```
public static String HTMLEncode(String TagFragment){
final StringBuffer res = new StringBuffer();
final StringCharacterIterator itr = new StringCharacterIterator(TagFragment);
    char chrctr =  itr.current();
while (chrctr != StringCharacterIterator.DONE )
  {
    if (chrctr == '<')
        res.append("&lt;");
    else if (chrctr == '>')
        res.append("&gt;");
    else if (chrctr == '\"')
        res.append("&quot;");
    else if (chrctr == '\'')
        res.append("&#039;");
    else if (chrctr == '\\')
        res.append("&#092;");
    else if (chrctr == '&')
        res.append("&amp;");
    else {
//add it to the result as is
        res.append(chrctr);
    }
    chrctr = itr.next();
  }
  return res.toString();
}
```

- The given code ensures **HTML entities** are encoded
- Each character is **passed dynamically** to the encoding function

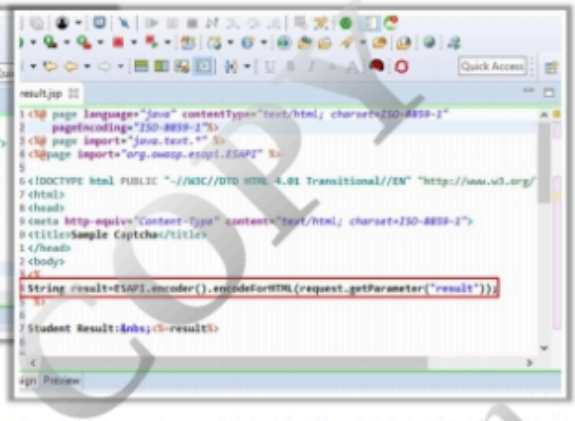
HTML Encoding using ESAPI Encoder

Vulnerable Code



- The given code **requests** input from the user
- The code is **executed correctly** when the user gives valid input
- As there is no input validation, the code becomes **vulnerable** if the user enters **invalid input**

Secure Code

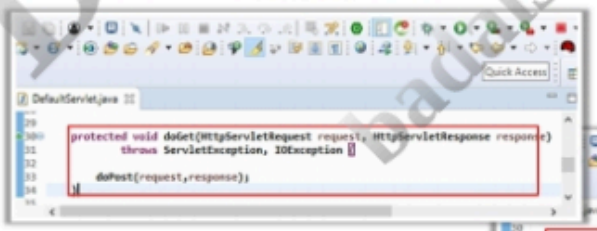


- The given code implements **ESAPI.Encoder** encode for HTML method to encode data to display as HTML

Cross-site Request Forgery (CSRF)

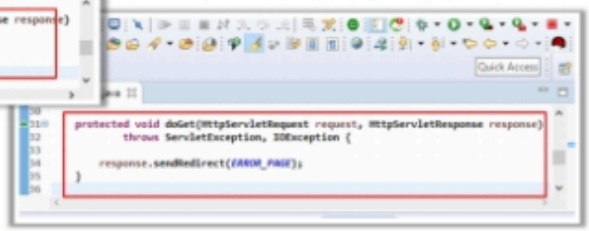
- Cross-site Request Forgery (CSRF) attacks **exploit web application vulnerabilities** where an attacker sends malicious requests to third party sites through an unsuspecting user's browser
- Usually, vulnerabilities in web application (**GET/POST/Path parameters**) confront this type of an attack, not the victim's browser or CRPF hosting site

Insecure Code



- In the code, attackers can **manipulate** the **GET request** to view any unauthorized data

Secure Code



- In the code, the programmer writes code to send **error page** if user **modifies GET requests**

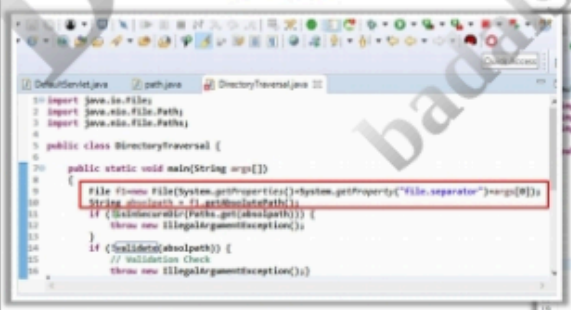
Cross-site Request Forgery (CSRF) Countermeasures

1. Web applications should use strong **authentication methods** such as cookies, http authentication, etc.
2. Use page tokens such as **time tokens** that change with every http or https page request
3. Appropriately use **GET** and **POST requests**
4. Check the referrer such as HTTP "**referer**" or **referrer** to mitigate this type of attacks
5. Implement OWASP **CSRFGuard library** that uses synchronizer token patterns to minimize the risk of CSRF attacks

Directory Traversal

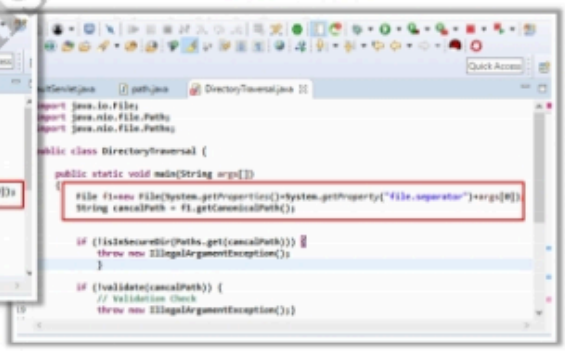
- Directory traversal is an HTTP exploit where an attacker can **access restricted files** and **directories** stored on the web server by bypassing the validation process
- Attackers can manipulate variables of **absolute** or **relative path names** that contain file links or can use "**. . /**" sequences with file names and access files and directories

Insecure code



- The code snippet uses command line argument for file path and uses **File.getAbsolutePath()** method to find the absolute path

Secure Code



- The **getCanonicalPath()** method is used to resolve any aliases, symbolic links and shortcuts

Directory Traversal Countermeasures

1. Use **strong filter** mechanisms for user data
2. Use filters for form input such as **GET** and **POST** parameters
3. Update **server software** at regular intervals
4. Define access rights to the **protected areas** of the website
5. Apply **checks/hot** fixes to prevent exploitation

HTTP Response Splitting

- HTTP response splitting is a web application vulnerability that can compromise an application and its environment with input values mostly through **HTTP requests**
- This attack creates **CR (Carriage Return)** and **LF (Line Feed)** sequences in the **HTTP header** and controls the remaining header and body of the response that the application intends to send and also can create another HTTP message
- Attacker splits the HTTP response by:
  - HTTP Header Splitting
  - HTTP redirect
  - HTTP cookie header

Insecure Code

- The code shows **cookie header response** split

Secure Code

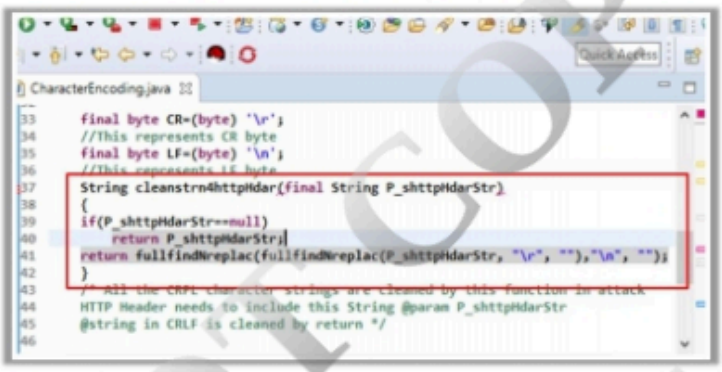
```
1: HTTP/1.1 200 OK
2: (etc...)
3: Set-Cookie: username=ECuser
4: HTTP/1.1 200 OK
5: (malicious content here)
```

- The code uses CR/LF to prevent any HTTP response splitting attack

HTTP Response Splitting Countermeasures

- Avoid direct placing of user **input variables** in the HTTP header
- Disallow and Filter **CR/LF** characters
- Use encoding for **validating header input** and output

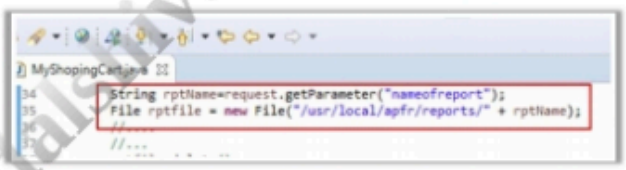
Example: Secure Code for HTTP response Splitting Attack



Parameter Manipulation and Countermeasures

- Use strong **input validating mechanisms** for user data inputs
- Implement strict **application security** routines and updates
- Implement **standards** for minimum and maximum allowable length, characters, patterns and numeric ranges
- Use strictly configured **firewalls** to block and **identify parameters** that are defined in a web page

Insecure Code

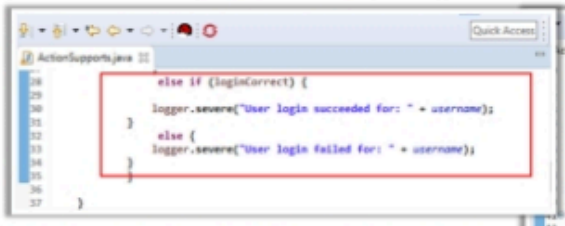


- An attacker can **manipulate** the file name or **delete** desired one

Protecting Application from Log Injection Attack

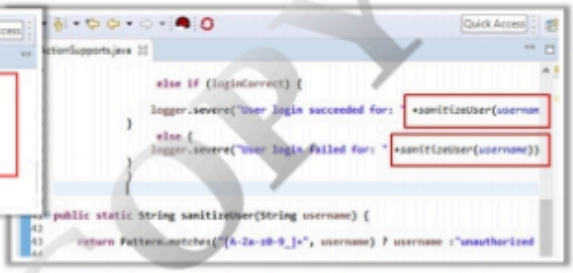
- Unsanitized user input may leak sensitive data of the application
- **Log Entry** may be split by attackers with a character of carriage return or line feed that might mislead the auditor

Vulnerable Code



- If users do not include secure sanitization then it is vulnerable to Log injection

Secure Code



- In the code whitelisting shows only character, number and space allowed

Log Containing Misleading Data

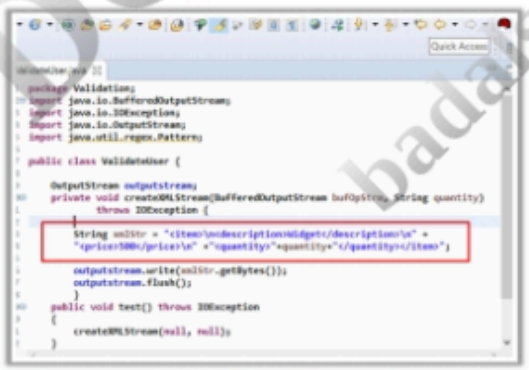
```
java.util.logging.LogManager$RootLogger log
SEVERE: login for: guest
java.util.logging.LogManager log
SEVERE: Login for: administrator
```

- There are different ways of handling log injection attacks
  1. Whitelisting
  2. Blacklisting

XML Injection

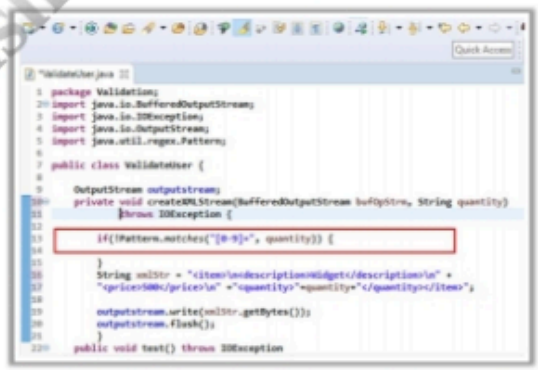
- XML injection is an **attack technique** where an attacker injects **XML Tags** in the SOAP message to change the structure of XML

Insecure Code



- This code uses string **concatenation** to build an XML query, but **fails** to **validate** the input resulting in an XML attack

Secure Code



- In the code, a **definite number** (0-9) is assigned by the coder to validate inputs

Command Injection

- Command injection is an attacking method in which a hacker **alters the contents** of the webpage by using HTML code into the input mechanism and by identifying the form fields that lack valid constraints

Insecure Code

```
String bktype = request.getParameter("backuptype");
String command = new String("cmmnd.exe /k\"c:\\util\\myOb.bat"+ bktype + "&&c:\\util\\cleanup.bat\"");
Runtime.getRuntime().exec(command);
```

- In the code, attackers can execute **multiple commands** separated by two ampersands passing a string "**&& del c:\\dbms*.***"

Secure Code

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletExc
    final int Maxsname = 50;
    final String f_regExp="[a-zA-z]{1,"+Maxsname+"}";
    final Pattern bkpattrn = Pattern.compile(f_regExp);
    final String bksname=f_regExp;
    ValidBksname(bksname,bkpattrn);

String bktype = request.getParameter("backuptype");
String command = new String("cmmnd.exe /k\"c:\\util\\myOb.bat"+ bktype + "&&c:\\util\\cleanup.bat\
Runtime.getRuntime().exec(command);
    doGet(request, response);
}
```

- The code snippet uses **whitelisting to prevent command injections**

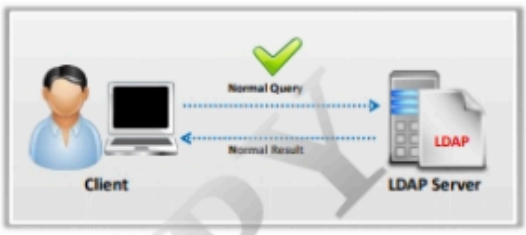
LDAP Injection

- An LDAP (Lightweight Directory Access Protocol) injection attack is done by exploiting the vulnerabilities of **input validation** to run arbitrary LDAP statements against information directories
- In the code snippet, **`searchRecord()`** method is used for searching the directory using LDAP protocol. However, the string filter used in the method does not perform adequate filtering resulting in executing specially crafted inputs

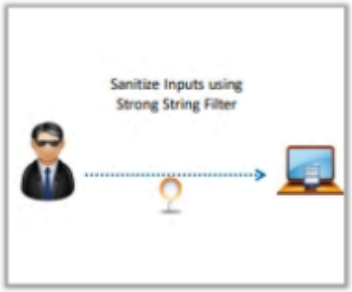
Insecure Code

```
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchControls;
import javax.naming.directory.SearchResult;
import javax.xml.xpath.XPathExpression;

public class path {// String UsrsName = "5*"; // Invalid
    // String Usrpwd = "*"; // Invalid

    private void searchRecord(String UsrsName, String Usrpwd) throws NamingExcep
    Hashtable<String, String>environment = new Hashtable<String, String>();
    environment.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFact
    try {
    DirContext dirctx = new InitialDirContext(environment);
    SearchControls srchctrl = new SearchControls();
    String[] attrbfltr = {"cnt", "email"};
    srchctrl.setReturningAttributes(attrbfltr);
    srchctrl.setSearchScope(SearchControls.SUBTREE_SCOPE);
    String bsc = "dircon=example, dircon=com";
    // This results to (&(strm=5*)(Usrpwd=*))
    String fltr = "(&(strm=" + UsrsName + ")(Usrpwd=" + Usrpwd + "))";
    NamingEnumeration<?> rslts = dirctx.search(bsc, fltr, srchctrl);
    while (rslts.hasMore()) {
    SearchResult schrs = (SearchResult) rslts.next();
    Attributes atrbs = schrs.getAttributes();
    Attribute attr = atrbs.get("cnt");
    System.out.println(attr.get());
    attr = atrbs.get("email");
    System.out.println(attr.get()); }
    dirctx.close();
    }
    catch(Exception e) {
```

LDAP Injection (Cont'd)



Secure Code




```
path.java
9
10⊖   public static XPathExpression compile(String string) {
11
12
13        String UsrsName = "Sherlock Holmes"; // Valid
14        String Usrpwd = "secret2"; // Valid
15
16        srchctrl.setSearchScope(SearchControls.SUBTREE_SCOPE);
17        String bsc = "dircon=example, dircon=com";
18        if(!UsrsName.matches("[\\w\\s]*") || !Usrpwd.matches("[\\w]*")) {
19        throw new IllegalArgumentException("Invalid input");
20        }
21        String fltr = "(&(strn = " + UsrsName + ")(userPassword=" + Usrpwd + "))";
22
```

- The code uses **whitelisting technique** to sanitize inputs using strong String filter that contains valid characters

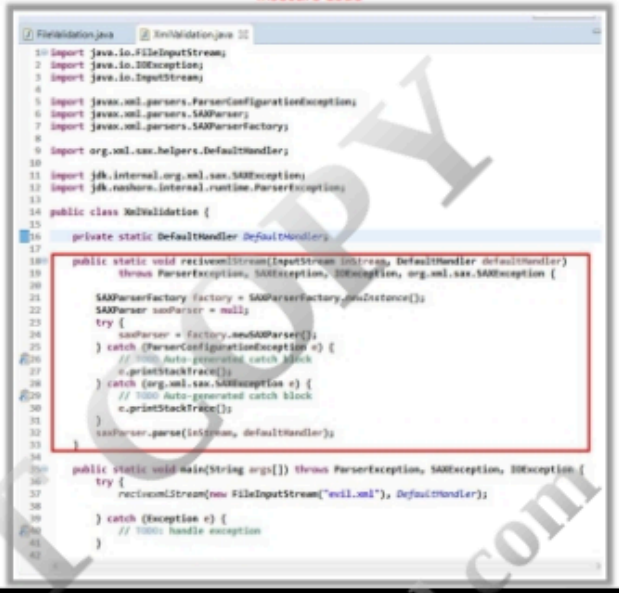
XML External Entity Attack

1. An **XML External Entity attack** enables the attacker to access local or remote contents resulting in information disclosure and denial of service attacks
2. The external entities use SYSTEM Keyword; the XML parser reads the document from the URI specified
3. This value can be replaced by the attacker to read any local or remote file

XML External Entity Attack (Cont'd)

- Prevent **XML Extrenal Entity attacks** by implementing org.xml.sax.EntityResolver resolveEntity() method
- Contains customized handler using whitelist for checking external entites
- resolveEntity() method resolves the external entities of XML document while parsing

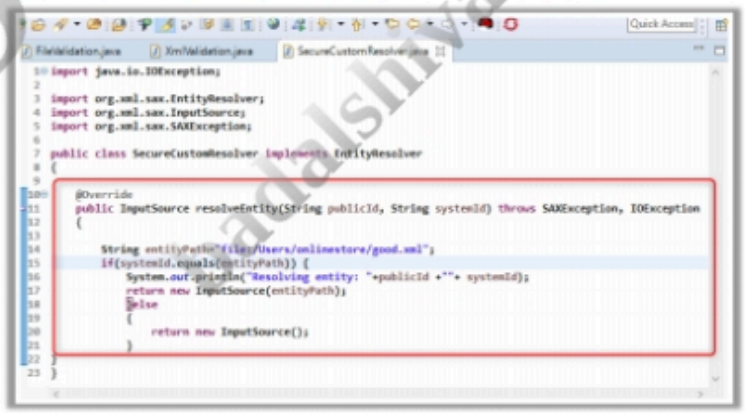
Insecure Code



XML External Entity Attack (Cont'd)

Secure Code

- The below code snippet implements **org.xml.sax.EntityResolver** resolveEntity() **method**



resolveEntity() method Syntax

```
InputSource resolveEntity(String publicId, String systemId) throws SAXException, IOException
```

Unrestricted File Upload Attack

- The **File Upload** feature exposes the application to upload malicious files. This results in both, **client side** and **server side** attacks
- **While uploading files restrict**
  - Uploading harmful content like exe
  - Overwriting existing files
  - Uploaded file size to prevent Denial of Service attack

Prevent Unrestricted File Upload: Validate File Extension

Secure Code

- The code **validates file extension** using **regular expression**

FileValidation.java

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class FileValidation {


    private static Pattern fileExtnPtrn = Pattern.compile("([^\\s]+(\\.(?i)(txt|doc|csv|pdf))$)");

    public static boolean validateFileExtn(String userName){

        Matcher mtch = fileExtnPtrn.matcher(userName);
        if(mtch.matches()){
            return true;
        }
        return false;
    }
    public static void main(String a[]){
        System.out.println("Is 'java2novice.pdf' allowed file? "
                        +validateFileExtn("java2novice.pdf"));
        System.out.println("Is 'cric.doc' allowed file? "
                        +validateFileExtn("cric.doc"));
        System.out.println("Is 'java.gif' allowed file? "
                        +validateFileExtn("java.gif"));
        System.out.println("Is 'novice.mp3' allowed file? "
                        +validateFileExtn("novice.mp3"));
        System.out.println("Is 'java_2.jpeg' allowed file? "
                        +validateFileExtn("java_2.jpeg"));
    }

}
```

Injection Attacks Countermeasures

Avoid implementing **custom parsers** as there are chances of ignoring **attack vectors**. So it is always better to use **tested** and **secured** parsers
While using **DOM** based parser, validate the XML stream size against a maximum before parsing
Defend against Denial of Service attacks by using **SAX** based parsing
Validate input to find the presence of malicious characters to avoid **XML injection attacks**
A thorough **input validation** needs to be performed before a request is sent to the **database**
Differentiate input strings from executable commands by using **parameterized stored procedures** to access the database or use **SQL parameters** to build SQL commands
It is always **suggested** to use less **privileged** accounts to access the database
Replace all **single quotes** with two single quotes
Error reports should not be displayed to the outsiders or else the same errors are used by attackers for **further attacks / malicious purpose**
Database server and **webserver** should be isolated and located on different machines

CAPTCHA

- CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart)

- Used to **authenticate** user input
- Consists of randomly generated image with **distorted letters/numbers**
- Determines whether the input is given by a **real user** or a computer program
- Used to **mitigate attacks** caused by automated tools
- Implemented on applications using third party software like **jcaptcha, simplecaptcha, recaptcha**

Sample Code for Creating CAPTCHA

The creation of CAPTCHA is given in the Servlet codeCaptchaServlet.java


```
import java.io.OutputStream;
import java.util.Random;

import javax.imageio.ImageIO;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.awt.Color;
import java.awt.Font;
import java.awt.GradientPaint;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;

/**
 * Servlet implementation class CaptchaServlet
 */
@WebServlet("/CaptchaServlet")
public class CaptchaServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void executeReq(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
            final int wd = 200;
            final int ht = 50;
            char cdata[][] = new char[1][];
            cdata[0] = getRandomNumber(8).toCharArray();
            final BufferedImage buffrdImage = new BufferedImage(wd, ht,
            BufferedImage.TYPE_INT_RGB);
            final Graphics2D G2D = buffrdImage.createGraphics();
            final Font f = new Font("calibri", Font.BOLD, 18);
            RenderingHints rHints = new RenderingHints(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
            rHints.put(RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);
            G2D.setRenderingHints(rHints);
            G2D.setFont(f);
            G2D.setColor(new Color(255, 255, 0));
```

Source: http://techbycaltech.com

Sample Code for Creating CAPTCHA (Cont'd)

```
            final GradientPaint grad_Paint = new GradientPaint(0, 0, Color.black, 0, ht / 2,
            Color.black, true);
            G2D.setPaint(grad_Paint);
            G2D.fillRect(0, 0, wd, ht);
            G2D.setColor(new Color(255, 153, 0));
            Random rndno = new Random();
            final String captcha = String.copyValueOf(cdata[0]);
            request.getSession().setAttribute("captcha", captcha);
            int xCordinate = 0;
            int yCordinate = 0;
            for (int a = 0; a < cdata[0].length; a++) {
            xCordinate += 10 + (Math.abs(rndno.nextInt()) % 15);
            if (xCordinate >= wd - 5) {
            xCordinate = 0;
            }
            yCordinate = 20 + Math.abs(rndno.nextInt()) % 20;
            G2D.drawChars(cdata[0], a, 1, xCordinate, yCordinate);
            }
            G2D.dispose();
            response.setContentType("image/png");


            final OutputStream outStr = response.getOutputStream();
            ImageIO.write(buffrdImage, "png", outStr);
            outStr.close(); }
            protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
            executeReq(request, response);
            }
            protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
            executeReq(request, response); }
            public static String getRandomNumber(int length) {
            String chcode = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890"
            Random randl = new Random();
            char[] buf = new char[length];
            for (int i = 0; i < length; i++) {
            buf[i] = chcode.charAt(randl.nextInt(chcode.length()));
            }
            return new String(buf);
```

Source: http://techbycaltech.com

Sample Code for CAPTCHA Verification

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/Captcha")
public class CaptchaVarification extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public CaptchaVarification() {
        super();

    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        final String vCaptcha = (String) request.getSession().getAttribute("captcha");
        final String code = (String) request.getParameter("code");
        if (vCaptcha != null && code != null) {
        if (vCaptcha.equals(code)) {
        request.setAttribute("result","Congratulations, You Passed The Captcha Test");
        } else {
        request.setAttribute("result",
        "Sorry, You Failed The Captcha Test");
        }
        request.getRequestDispatcher("/result.jsp").forward(request,
        response);

    }
```

Sample Code for Displaying CAPTCHA

The CAPTCHA is displayed using JSP

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Sample Captcha</title>
</head>
<body>

<form method="post" action="verifyCaptcha.do">
<table >
<tr>
<td align="center">
<img src="samplecaptchaServlet.do">
<br />
<br />
<input type="text" name="capatchacode">
</td>
</tr>
</table>
<input type="submit" value="submit">
</form>
<br>
<br>

</body>
</html>
```

result.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
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Sample Captcha</title>
</head>
<body>
<%=request.getAttribute("result")%>
</body>
</html>
```

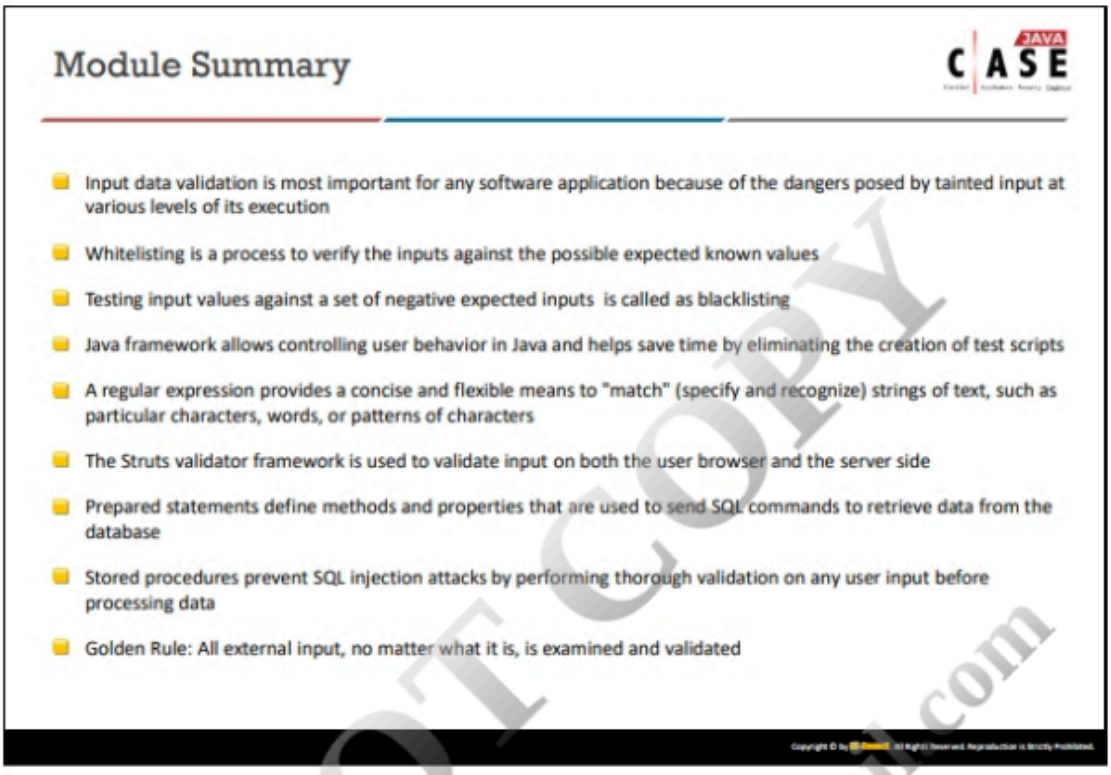
Best Practices for Input Validation

Golden Rule: All external inputs, no matter what they are, should be examined and validated
Do not log unvalidated user inputs, as it may result in leaking sensitive data
Ensure that a data validation mechanism is present
Ensure that the data validation occurs on the server side
Ensure that proper length checks on all inputs exist
Examine where data validation occurs, and if a centralized model or decentralized model is used
Ensure that all fields, cookies, HTTP headers/bodies & form fields are validated
Ensure that there are no backdoors in the data validation model

Best Practices for Input Validation (Cont'd)

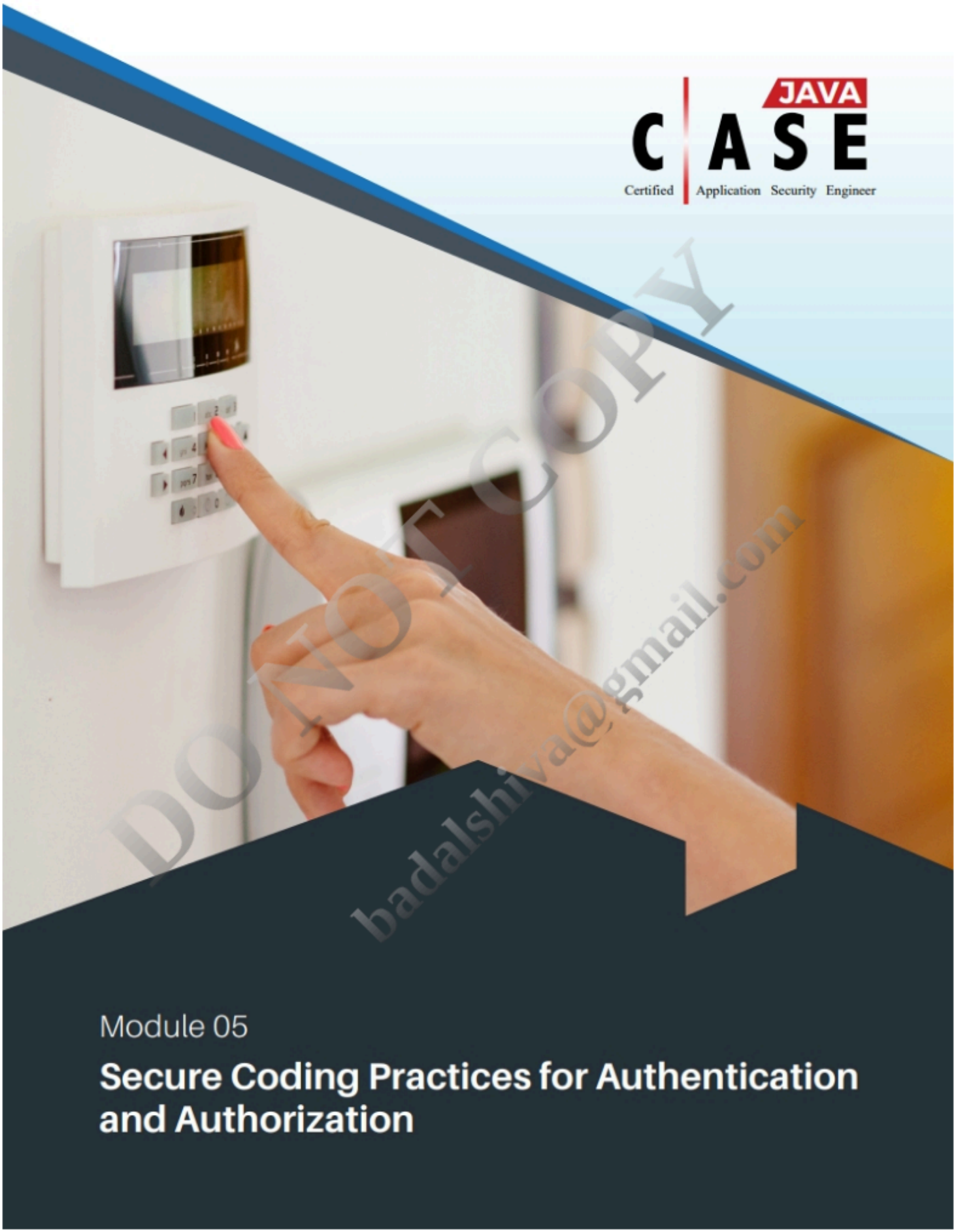
Apply known good values, i.e., whitelists, wherever it is feasible
Do not incorporate input strings of untrusted sources into format strings
Ensure that all inputs are canonicalized
Perform filtering and validation mechanisms based on the strings' character data
Ensure that inputs follow correct syntax and content restrictions
Use java.sql.PreparedStatement class to prevent SQL injection attacks
Ensure that all inputs that can (and will) be modified by a user such as HTTP headers, input fields, hidden fields, etc., are properly validated
Ensure that the data is well-structured and contains only known good characters, if possible

Module Summary



This page is intentionally left blank.

JAVA CASE
Certified Application Security Engineer



Module 05

Secure Coding Practices for Authentication and Authorization

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	Understand Authentication Concepts
2	Discuss Authentication Implementation in Java
3	Discuss Authentication Weaknesses and Prevention
4	Understand Authorization Concepts
5	Discuss Access Control Model
6	Discuss EJB Authorization
7	Discuss Java Authentication and Authorization (JAAS)
8	Discuss Authorization Common Mistakes and Countermeasures
9	Discuss Java EE Security
10	Discuss Authentication and Authorization in Spring Security Framework
11	Learn Defensive Coding Practices against Broken Authentication and Authorization

Introduction to Authentication

Introduction to Authentication

- The main purpose of authentication is to verify a **user's access** to a protected segment of the web application
- Authentication is of two types, namely **basic authentication** and **form-based authentication**

The responsibilities of authentication systems are:

- Verify the user and provide a safe communication passage
- Gather information about the way the user accesses the system

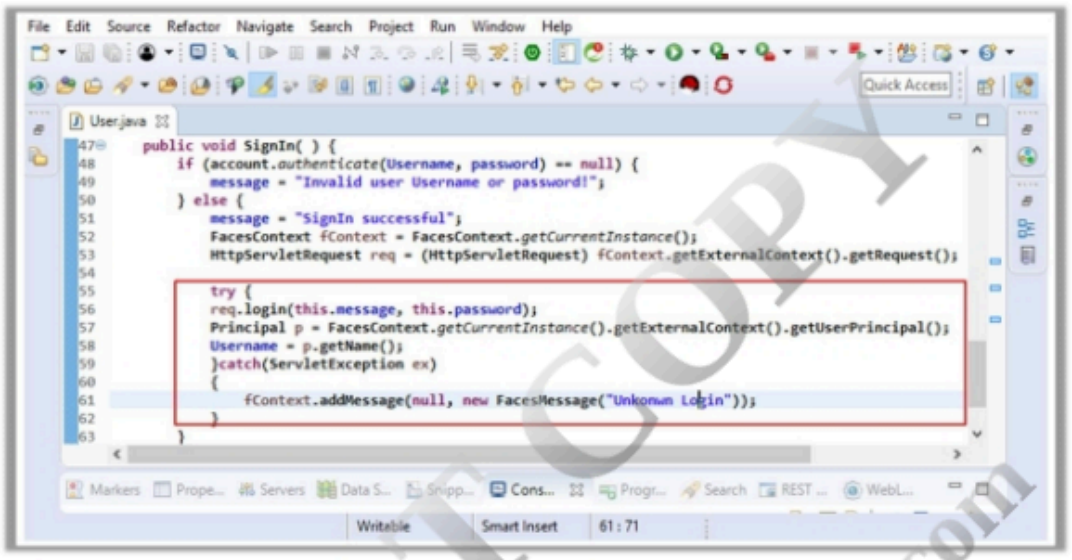
J2EE supports four types of authentication mechanisms

- HTTP Basic Authentication
- Form-based Authentication
- Client/Server Mutual Authentication
- HTTP Digest Authentication

Java Container Authentication

- Authentication in a web application is **role-based**, i.e., a user needs to be assigned a role in order to access the web application, e.g., a customer, developer or manager
- A role gets access only when the identity of the user is identified by the container, and for all other cases, access is denied, resulting in **HTTP 401 status code**
- Information about the role and user that is already stored in the **database** of the web application is compared with that of the user who wants to access the web application
- After **authentic validation** of the user, his roles are assigned by the web application through one of the following methods
  - JDBC Login Module
  - LDAP Login Module
  - Windows Login Module
  - Custom JAAS Login Module
- In Java, container authenticates web application, i.e., **tomcat, jboss,** performs the process of authentication
- These results of the authentication are passed on to the web application, thus avoiding the need to implement authentication code such as **HTTP Basic** or **NTLM**

Authentication Mechanism Implementation



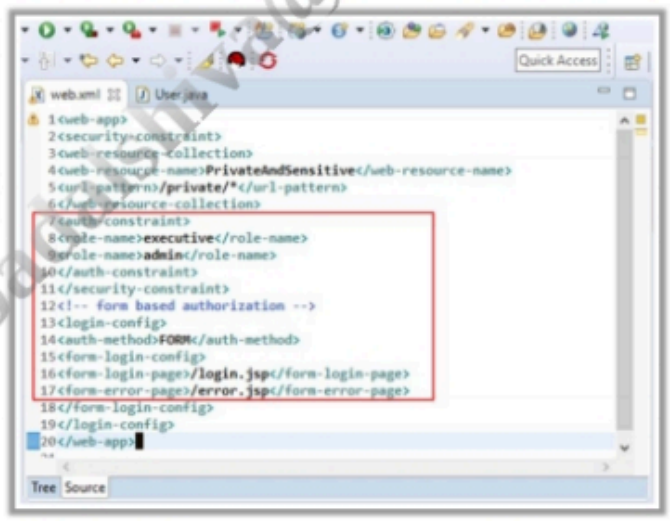
Types of Authentication

Declarative vs Programmatic Authentication

- Sometimes, declarative security alone is not sufficient; in such cases, programmatic security enhances **security** by giving complex, dynamic **rules** and **policies** for application components
- In programmatic security, business methods that are invoked help J2EE container to determine whether a caller should be given **privileges** or not, i.e., **access** or **denial** of **permissions** to a resource
- Enhanced security can be provided by programmatic mechanism supplementing declarative security for **role membership** and **user identity**
- J2EE container services provide **application tiers** and **components** with authentication and authorization facilities identifying service providers and callers
- J2EE container services support **declarative** and **programmatic authentication**
- Rules and permissions play a key role in declarative security model defined in the deployment descriptor document that is bundled with **application component**
- These **rules** and **permission** are assigned by application deployer according to **deployment descriptor**

Declarative Security Implementation

- **Servlet containers** are used to implement declarative security
- Program control is through deployment descriptor **web.xml file**
- Declarative security along with servlets and JSP give a **robust protection**



Programmatic Security Implementation

LoginServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.annotation.security.DeclareRoles;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@DeclareRoles("ECCUser")

@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public LoginServlet() {
        super();
    }

    protected void processRequest(HttpServletRequest req, HttpServletResponse resp)throws ServletException,IOException
    {

            resp.setContentType("text/html;charset=UTF-8");
            PrintWriter pw = resp.getWriter();
            try {
            String UsrName = req.getParameter("txtUserName");
            String passwd = req.getParameter("txtPassword");
            pw.println("Before Login" + "<br><br>");
            pw.println("IsUserInRole?.."
            + req.isUserInRole("ECCUser")+"<br>");
            pw.println("getRemoteUser?.." + req.getRemoteUser()+"<br>");
            pw.println("getUserPrincipal?.."
            + req.getUserPrincipal()+"<br>");
            }catch (Exception e) {
```

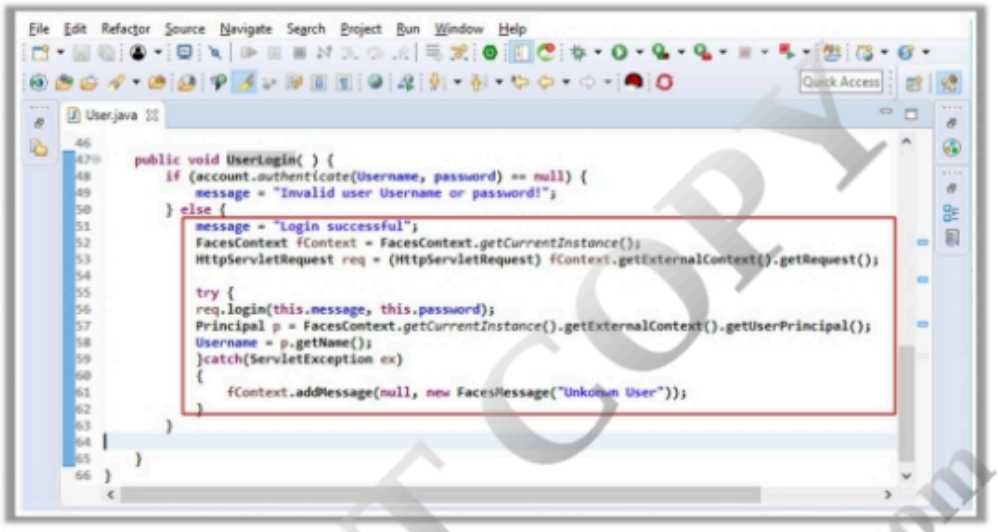
Programmatic Security Implementation (Cont'd)

LoginServlet.java

```
try {
req.login(UsrName, passwd);
} catch(ServletException se) {
pw.println("Login Failed with a ServletException.."
+ se.getMessage());
return;
}
pw.println("After Login..."+"<br><br>");
pw.println("IsUserInRole?.."
+ req.isUserInRole("ECCUser")+"<br>");
pw.println("getRemoteUser?.." + req.getRemoteUser()+"<br>");
pw.println("getUserPrincipal?.."
+ req.getUserPrincipal()+"<br>");
pw.println("getAuthType?.." + req.getAuthType()+"<br><br>");
req.logout();
pw.println("After Logout..."+"<br><br>");
pw.println("IsUserInRole?.."
+ req.isUserInRole("ECCUser")+"<br>");
pw.println("getRemoteUser?.." + req.getRemoteUser()+"<br>");
pw.println("getUserPrincipal?.."
+ req.getUserPrincipal()+"<br>");
pw.println("getAuthType?.." + req.getAuthType()+"<br>");
```


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.

Java EE Authentication Implementation Example



Basic Authentication

- Basic authentication uses **standard HTTP basic security**
- Authentication is through SSL using dialog box instead of HTML form, requiring no changes in **JSP pages** or **servlets.** SSL/TLS should be explicitly enabled else the credentials will be transmitted in plain text
- Easier to setup since it requires no creation and **configuration** of **login** or **login error pages**
- Mostly, **intranet applications** use basic authentication

How to Implement Basic Authentication?

tomcat-users.xml

```
48  <role rolename="engineer" />
49  <role rolename="developer" />
50  <role rolename="tester" />
51  <role rolename="admin" />
52  <user username="john" password="test@123" roles="developer,admin" />
53  <user username="thomas" password="abcdefg" roles="tester,admin" />
54  <user username="joseph" password="dream123" roles="tester,engineer" />
55  <user username="julie" password="123456" roles="developer,admin" />
56 </tomcat-users>
```

- Define roles and setup users with usernames, passwords, and roles

web.xml

```
21  <security-constraint>
22  </security-constraint>
23 <login-config>
24    <auth-method>BASIC</auth-method>
25    <realm-name>MyRealm</realm-name>
26 </login-config>
27 </web-app>
```

- Activating BASIC authentication in the server and designation of Realm

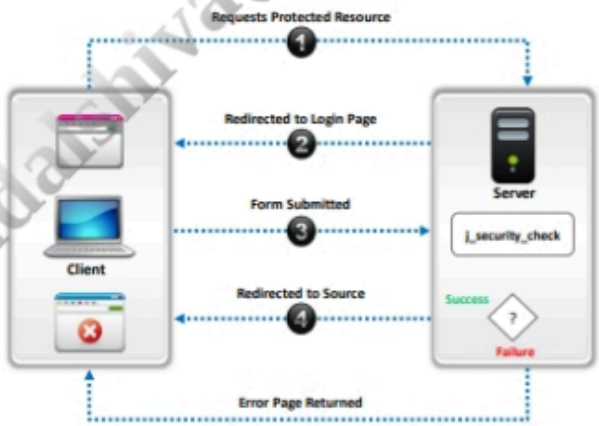
tomcat-users.xml | web.xml

```
39    <display-name></display-name>
40    <user-data-constraint><description></description>
41    <transport-guarantee>CONFIDENTIAL</transport-guarantee
42    </user-data-constraint>
```

- Implementing BASIC Authentication with SSL

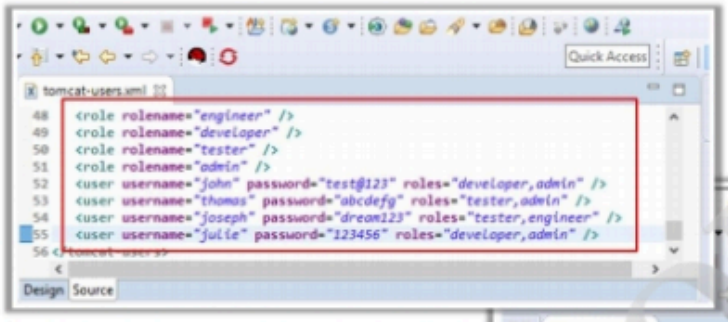
Form-based Authentication

- Form-based authentication is an HTML-based form that uses declarative security
- Easy implementation through control over GUI of the authentication form
- User credentials and roles are checked for their correctness, if found incorrect, error page appears
- Container handles the entire process of authentication
- SSL can be added to part or whole of the web application

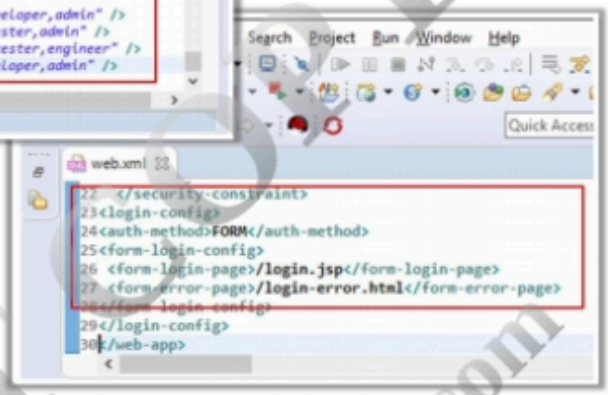


Form-Based Authentication

Form-based Authentication Implementation



- Define **roles** and **setup** users with **usernames**, passwords, and roles



- Activating **Form-based authentication** in the server

Implementing Kerberos-Based Authentication

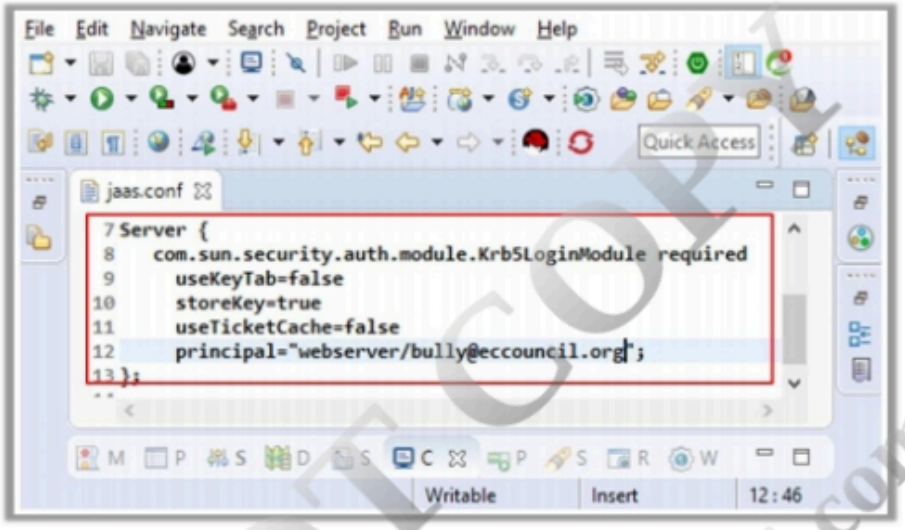
- Kerberos V5 is mainly used as an **authentication security protocol** within a domain
- It verifies a user as well as the server providing information; this is known as **mutual authentication**

Working Overview of Kerberos

1. This mechanism **issues tickets** containing user identity, encrypted password, and encrypted data
2. Working process is **invisible** to the **user**
3. Key Distribution Centre in Kerberos **stores** account **information** and client **passwords**

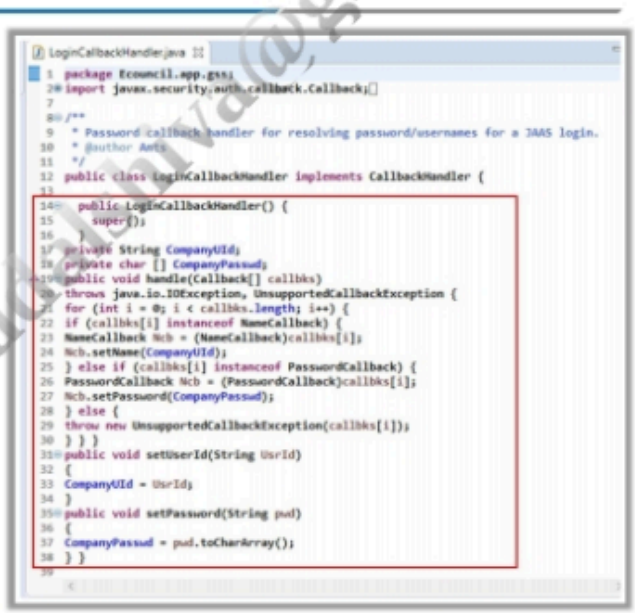
Secured Kerberos Implementation

Example for Kerberos Implementation with jaas.conf



Secured Kerberos Implementation (Cont'd)

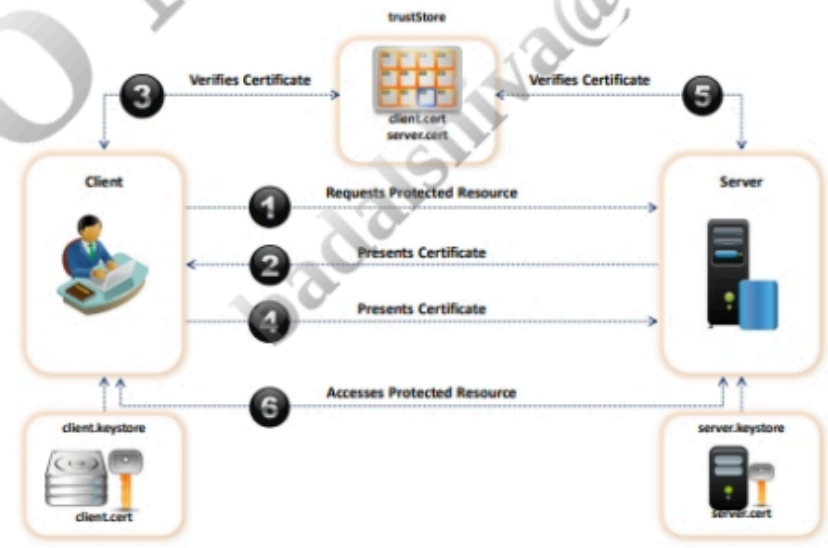
- Implementing Kerberos with *CallbackHandler.java
- LoginCallbackHandler implements CallbackHandler



Client Certificate Authentication

- Client authentication **ensures** whether the users are **legitimate** or not
- In HTTPS, browsers and webservers communicate over secured SSL encrypted connection
- Servers determine a visitor's **identity** using the following methods
  - If tomcatAuthentication="**false**", then it retrieves the username and assumes all authentication is done
  - If tomcatAuthentication="**true**", CLIENT-CERT results in the value of `org.apache.catalina.authenticator.SSLAuthenticator` is input to the application
- The client and the server present each other their certificates from the keystores in authentication process
- Authentic client and server certificate copies are stored in each of the keystores for secured SSL communication

Client Certificate Authentication (Cont'd)



Certificate Generation with Keytool

- `Keystore` class is a memory collection of keys and certificates
- **Keytool** is a command line tool provided by Java for generating **keystores**
- Every keystore has a unique **private key** that cannot be exported but can be copied easily
- A keytool helps in **sharing public** version of the **private key** with the server

Certificate Generation with Keytool (Cont'd)

Generating the Client and Server `Keystores`

```
1: keytool -genkeypair -alias serverkey -keyalg RSA -dname "CN=Web Server,OU=Application
Development,O=Highwinds,L=Winter Park,S=FL,C=US" -keypass password -keystore
server.jks -storepass password
2: keytool -genkeypair -alias clientkey -keyalg RSA -dname "CN=client,OU=Application
Development,O=Highwinds,L=Winter Park,S=FL,C=US" -keypass password -storepass password
-keystore client.jks
```

Exporting the Client's Public Certificate and Importing it in to the Server's `Keystore`

```
1: keytool -exportcert -alias clientkey -file client-public.cer -keystore client.jks -
storepass password
2: keytool -importcert -keystore server.jks -alias clientcert -file client-public.cer -
storepass password -noprompt
3: # view the contents of the keystore (use -v for verbose output)
4: keytool -list -keystore server.jks -storepass password
```

Certificate Generation with Keytool (Cont'd)

- **Export the Server's Public Certificate and import it in to the Client's `Keystore`**

```
C:\WINDOWS\system32\cmd.exe
C:\                    >keytool -exportcert -alias serverkey -file s
erver-public.cer -keystore server.jks -storepass password
```

```
C:\WINDOWS\system32\cmd.exe
C:\                    >keytool -importcert -keystore client.jks -alias
servercert -file server-public.cer -storepass password -noprompt
```

```
# view the contents of the keystore (use -v for verbose output)
```

```
C:\WINDOWS\system32\cmd.exe
C:\                    >keytool -list -keystore client.jks -storepass p
assword
```

Certificate Generation with Keytool (Cont'd)

- **Configure Tomcat for SSL using the Server `Keystore`**

server.xml | catalina.pr... | project.pro... | project.xml | genfiles.pr...

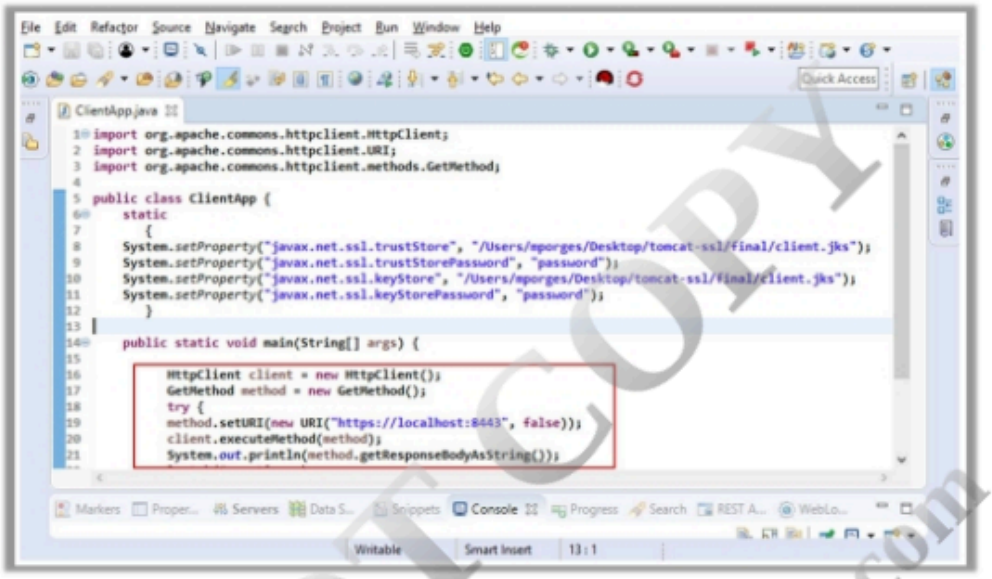
```
89    <Connector
90  clientAuth="true" port="8443" minSpareThreads="5" maxSpareThreads="75"
91  enableLookups="true" disableUploadTimeout="true"
92  acceptCount="100" maxThreads="200"
93  scheme="https" secure="true" SSLEnabled="true"
94  keystoreFile="/Users/mporges/Desktop/tomcat-ssl/final/server.jks"
95  keystoreType="JKS" keystorePass="password"
96  truststoreFile="/Users/mporges/Desktop/tomcat-ssl/final/server.jks"
97  truststoreType="JKS" truststorePass="password"
98  SSLVerifyClient="require" SSLEngine="on" SSLVerifyDepth="2" sslProtocol="TLS"
99 />
```

Design | Source

Note:

- Appropriate paths and passwords substituted for **truststorePass**, **truststoreFile**, **keystorePass**, and **keystoreFile** in `{tomcat.home}/conf/server.xml`
- **clientAuth attribute** enquires clients to provide authentication certificates

Implementing Encryption and Certificates in Client Application



Authentication Weaknesses and Prevention

Authentication Weaknesses and Prevention: Brute Force Attack

It is a continuous **trial** and **error method** of attempts to login to a web application using dictionary and other permutation combined characters

Preventive Measures

- Mandate web application accepts only increased length of **user id credentials**, especially password
- Impose web application that accepts only user id credentials that contain all valid characters including **special characters** like **!**, **@**, **#**, **$**, etc.
- Impose **time delay** between failed attempts
- Lock out targeted account access after a certain **restricted failed attempts**
- Impose a **password aging policy**

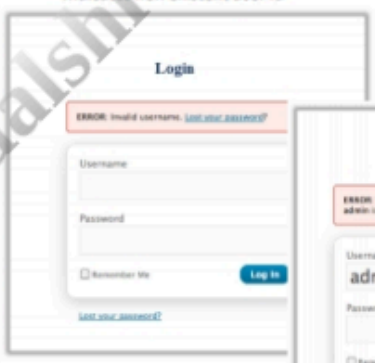
Authentication Weaknesses and Prevention: Web-based Enumeration Attack

Attacker experiments with various **user ID credentials** and analyses the responses to find valid ones through **error message**

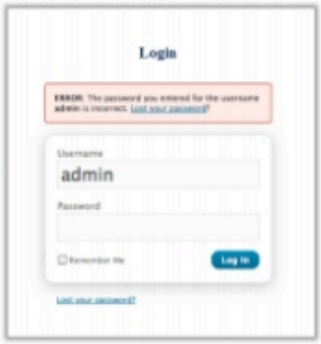
Web-based Enumeration Attack Prevention

- Web applications need to respond with similar error messages to all **authentication failures**
  - Credential submitted are not valid
- Analyze **URLs** and their responses during security testing to authentication failures and **prevent** unnecessary **information leakage**
  - http://www.foo.com/err.jsp?User=baduser&Error=0
  - http://www.foo.com/err.jsp?User=gooduser&Error=2
- Analyze web page titles and their responses during **authentication failures** and prevent unnecessary information leakage
  - Invalid user
  - Invalid authentication

Indicates non existent user id



Indicates user ID is correct but password is not



Authentication Weaknesses and Prevention: Weak Password Attacks

- Attackers **crack** weak passwords by guessing, using default user credentials, tools and techniques

Weak Password Attack Prevention

1. Mandate web application that accepts only **increased length** of user ID credentials, especially password
2. Mandate passwords to be at least **6 characters** long with a combination of uppercase and lowercase letters, digits, and special characters
3. Impose a **password aging policy**
4. **Incorrect** authentication failure messages should be avoided
5. Implement **account lockout policy**
6. Highly critical applications need **multi-factor authentication**

Introduction to Authorization

Introduction to Authorization

- Authorization is the process that controls **access rights** of principals to system resources that include:
  - Access to users
  - Access to processes
  - Access to machines
- Main goals of authorization is to provide information security attributes, i.e., **Confidentiality**, **Integrity** and **Availability** (CIA)
- Implementation of authorization in Java is through JAAS that determines authenticated user's **purview** of given **rights**

Introduction to Authorization (Cont'd)

- It is the process of defining **access policies** for confidential resources
- Authorization shows "who is **executing**" "where is the code **residing**" "who is the **owner** of the code"

Steps to implement authorization

1. Defining **roles** to users
2. Define **security roles** of an application to roles defined in memory realm
3. Define the **resource** and **HTTP** method constraints for an application in web.xml file
4. Apply the constraints that are accessible by role
5. Check for user **authentication** for the application
6. Allow the server to **process** the **request** when the details are verified

JEE Based Authorization

- JEE security handles roles for **managing suitable access** to appropriate resources by **users** and **groups**
- Authorization in JEE is of two forms, **programmatic** and **declarative** authorization

Declarative

- In declarative authorization, the security information is listed in **deployment descriptors**
- These deployment descriptors are accessed by application servers and determine whether a role is assigned to a user and also decides if defined role is able to access a particular **resource**

Programmatic

- Programmatic authorization is performed by using **APIs** and **functions** such as `isUserInRole()`

Example code for Declarative Authorization

LoginServlet.java | web.xml

```
16
17    <auth-constraint>
18 <description>Must authenticate before querying the
19       <role-name>Admin</role-name>
20    </auth-constraint>
21    <user-data-constraint>
```

Example code for Programmatic Authorization

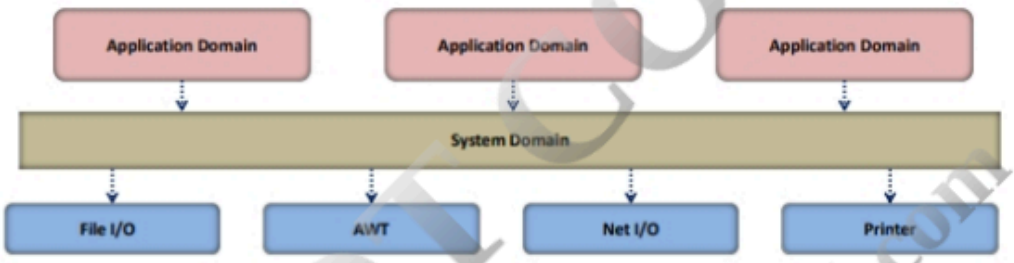
LoginServlet.java

```
25        if(req.isUserInRole("Admin"))
26        {
27            System.out.println("welcome");
28        }
29
```

Access Control Model

Access Control Model

- All the protected resources in Java applications are accessed only through system domain
- Protected resources include keyboard, AWT, printing systems, networking facilities and file systems
- Permissions to running programs are checked by calling the method `SecurityManager.checkPermission()`, ensuring proper permissions are given to the objects
- While checking permissions, `SecurityManager.checkPermission()` invokes `AccessController.checkPermission()` to verify the permissions

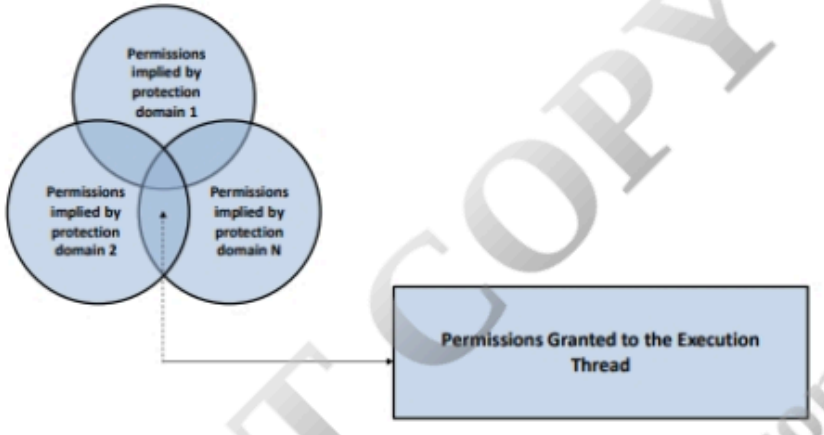


Access Control Model (Cont'd)

- The `AccessController.checkPermission()` method logic ensures each class in the thread stack gets `ProtectionDomain`
- All these `ProtectionDomains` are composed into a `java.security.AccessControlContext` object
- `implies()` method is invoked along with permission object on the `ProtectionDomain` to check as a parameter
- In case if checked permissions are invoked by `ProtectionDomain` for each stack frame, this continues even for the next frame on the stack
- Testing continues until stack ends, concluding that all the thread's classes have proper permission for the operation's success

Access Control Model (Cont'd)

- Even if one of the `ProtectionDomain` in the thread's stack does not have **checkPermission**, then access to resource is denied along with a `SecurityException` thrown



Access Control Model (Cont'd)

- Both authentication and authorization play an important role in **information security** and together form access control

Access control comprises three models:

1. **Discretionary** access control (DAC)
2. **Mandatory** access control (MAC)
3. **Role – based** access control (RAC)

Discretionary Access Control (DAC)

- This works based on the **notion of ownership**
- If an individual creates a file then he/she becomes the owner of the file and identified as **data owner**
- For example, Bob creates a file called student.txt. He wants Alice to read the file so he provides access to read the file but do nothing else such as **editing** or **deleting**
- In this scenario, Bob is the data owner and Alice is the user who is **authorized** to view the file created by Bob
- DAC is prescribed through **Access Control Lists** (ACL's) that stores rules of access to resources
- ACLs contain subjects (users) and objects (resources)

- The subjects are allowed to **access objects** by defining a set of privileges
- DAC can **grant** and **revoke** the rights to resources
- **Capability list** stores a list of rights an object has for each subject called capabilities
- Authentication happens when **capabilities** are **granted**
- Limitation of DAC is that it is very **hard** and **complex** to implement in big enterprise scenarios
- It is **tedious** to define rights or privileges per individual as it leads to large access control rules for each file

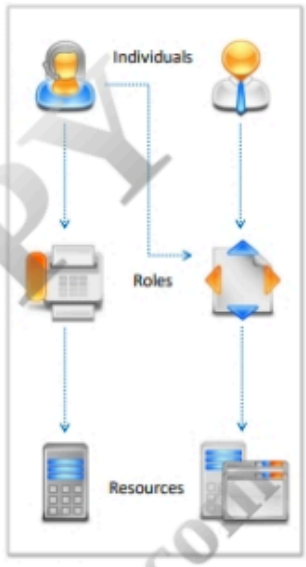
Mandatory Access Control (MAC)

- This model relies on the principle of **limiting information dissemination**, and policies imposed by every data item in the system are governed by **Central Authority** throughout lifetime
- The creator of object (file) does not necessarily know who has **authorized access** to it
- The access control policies are imposed on policy, **specific** to the **user**
- As opposed to discretionary access control (DAC), there is no concept of data owner, **confidentiality** is a sign of access control that guards the ability to read data
- Policies may vary depending on each action being performed on the resources, i.e., functions for a file in a file system are **read**, **write**, **delete**, **create**, etc.

- Objects (file which contains data) in the system are assigned **clearance levels** that are specified as unclassified, classified, secret and top secret
- A user with clearance level of classified is allowed to read **unclassified information** but not secret and top secret information
- Some examples of MAC are **official** reports, **accounting** reports, **medical reports statistical analysis**, **e-business applications**, etc.
- Access control must be mandatory whenever possible from system end user's perspective, and it protects the system from **operational security errors** and also makes the system simpler

Role-based Access Control (RBAC)

- This is the **popular** access control model
- A user has access to resources based on the **role assigned**; roles are allocated depending on job function
- Permissions are distinct on job authority and responsibilities contained in job function. They also determine **interaction** between subject and object
- It functions on the concept of user **roles** and **information** accessibility
- An organization has different departments, and roles are assigned based on **requirements**; RBAC provides only access and privileges needed by an individual
- For **example**, the Human Resource Department does not have a need to view account department's information; the role of HRD is to view human resources information



Servlet Container

1. A component of web server that facilitates **loading**, **executing** and **initializing** servlets in standalone (without a web server) or by running on another host
2. Examples of servlet container are **tomcat**, **Glassfish**, etc.
3. A web server uses Hyper Text Transfer Protocol (HTTP) to **communicate** with end users, which are generally web browsers
4. A Java-based web server communicates through HTTP messages by using two main classes, **java.net.Socket** and **java.net.ServerSocket**
5. Servlet container **manages** large number of **servlets** ; hence, it can handle many requests
6. For each Http method (`Get`, `Post`), **HttpServlet** class has a corresponding method (`doGet()`, `doPost()`, `doPut()`)
7. Servlet container calls the **`service()` method** of servlet and provides an instance of ServletRequest and ServletResponse that enables in identifying remote user, parameters of HTTP `POST`

Authorizing Users by Servlets

- To permit access to authorized users, servlets can be configured by:

Defining Roles

- An application function can be defined by **security** roles that are made up of users, groups or both users and groups
- The relationship between users and groups is identified based on the implementation used
- Roles can be defined in Java EE deployment descriptor file as **web.xml** and corresponding mapping to roles in the web server deployment descriptor file as **serv-web.xml**

Defining Constraints

- Access permissions can be defined using auth-constraint element in **web.xml** file
- Web containers impose authorization at **page level**
- Once the user is authenticated, the auth-constraint defined in the **deployment descriptor** checks if the user belongs to any one of the roles

Authorizing Users by Servlets (Cont'd)

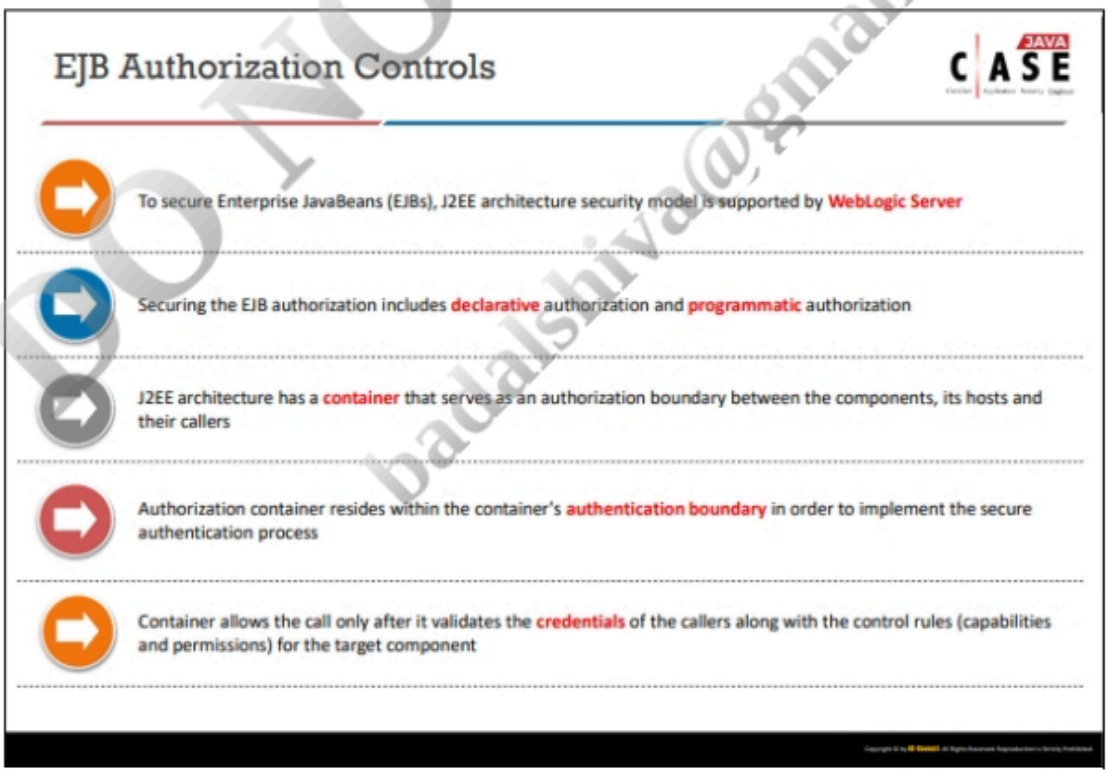
Example for defining roles

- Each element of **security-role-mapping** in **web.xml** file maps role names to principals and groups that are allowed by web applications
- The role-name must match with role-name of security-role element of the equivalent **web.xml** file

Example of Defining Constraints for Servlet Authorization Servlets

LoginServlet.java | web.xml

```
<security-constraint>
<web-resource-collection>
<web-resource-name>LoginPages</web-resource-name>
<description> access  authorized users only </description>
<url-pattern>/manager/*</url-pattern>
<http-method>GET</http-method>
</web-resource-collection>
<auth-constraint>
<description>The roles who can access are</description>
<role-name>clerk</role-name>
</auth-constraint>
</security-constraint>

  <security-role-mapping>
<role-name>clerk</role-name>
<principal-name>john</principal-name>
<principal-name>web</principal-name>
<group-name>HR</group-name>
</security-role-mapping>
<security-role-mapping>
<role-name>manager</role-name>
<principal-name>Sinha</principal-name>
</security-role-mapping>
</web-app>
```



Declarative Security with EJBs

1. EJB or the Enterprise JavaBeans Technology is Java EE's server-side component architecture. It helps in developing **portable**, **secure**, **traditional** and **distributed** applications
2. Deployment descriptors such as **ejb-jar.xml** and **weblogic-ejb-jar.xml** are used in the EJBs to implement declarative security
3. These descriptors are used for the purpose of **defining security** specifications
4. The EJB container uses such **security definitions** at runtime to enforce the requirements
5. Descriptors also map the application's security specifications with its **runtime definitions**

Declarative Security with EJBs (Cont'd)

- **Following are the steps to configure security in the EJB deployment descriptors:**

1. Use a text editor to create **ejb-jar.xml** and **weblogic-ejb-jar.xml** deployment descriptor files
2. Define constraints for security role name, EJB name and the method name in **ejb-jar.xml** file
3. Security role names are **case sensitive**; follow some restrictions and conventions when specifying role names
4. In **weblogic-ejb-jar.xml** descriptor file, define security role name and associate it to one or multiple principals (users or groups) in a security realm

Declarative Security with EJBs (Cont'd)

- The following code shows the mapping of security role names to a security realm by using ejb-jar.xml and weblogic-ejb-jar.xml files

ejb-jar.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <assembly-descriptor>
3 <security-role>
4 <role-name>clerk</role-name>
5 </security-role>
6 <security-role>
7 <role-name>bob</role-name>
8 </security-role>
9 <method-permission>
10 <role-name>clerk</role-name>
11 <role-name>bob</role-name>
12 <method>
13 <ejb-name>accountsPayable</ejb-name>
14 <method-name>getReceipts</method-name>
15
16 </method>
17 </method-permission>
18 <!-- ...-->
19 </assembly-descriptor>
20 <!-- ...
21 weblogic-ejb-jar.xml entries: -><security-role-assignment>
22 <role-name>clerk</role-name>
23 <principal-name>ec</principal-name>
24 <principal-name>John</principal-name>
25 <principal-name>Jacob</principal-name>
26 </security-role-assignment>
```

Design Source

Programmatic Security with EJBs

- javax.EJBContext.getCallerprincipal()and javax.ejb.EJBContext.isCallerInRole() methods can be used to implement programmatic security in EJBs

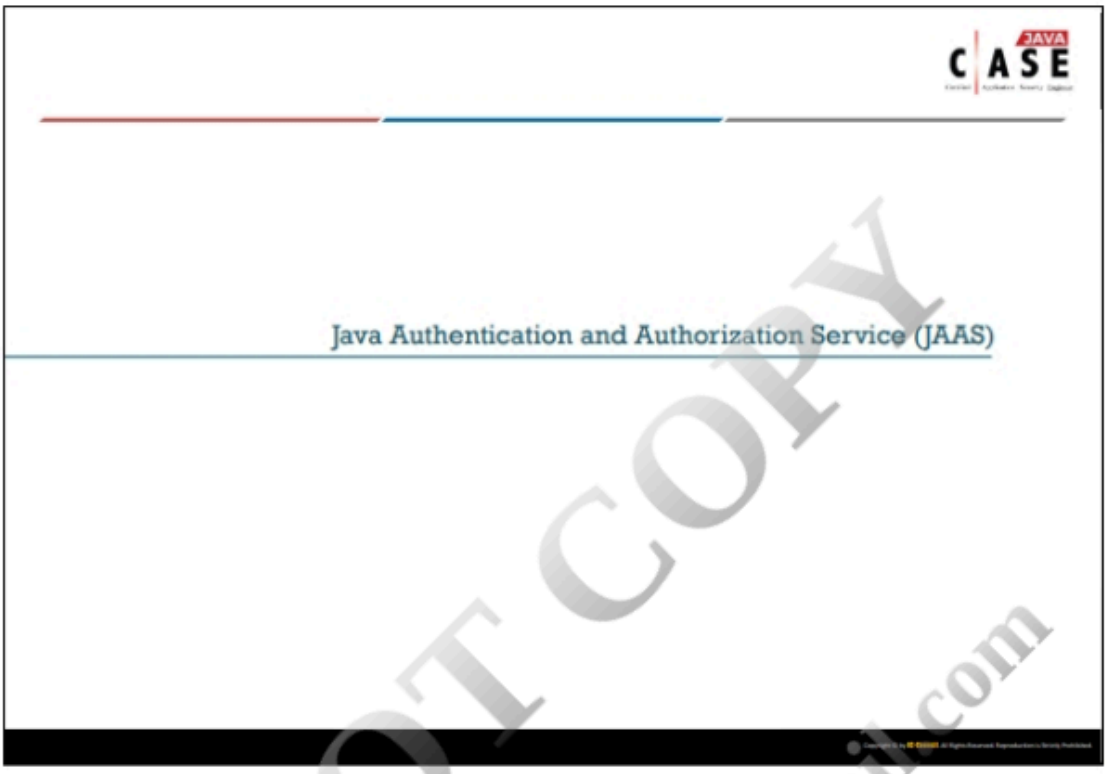
getCallerPrincipal

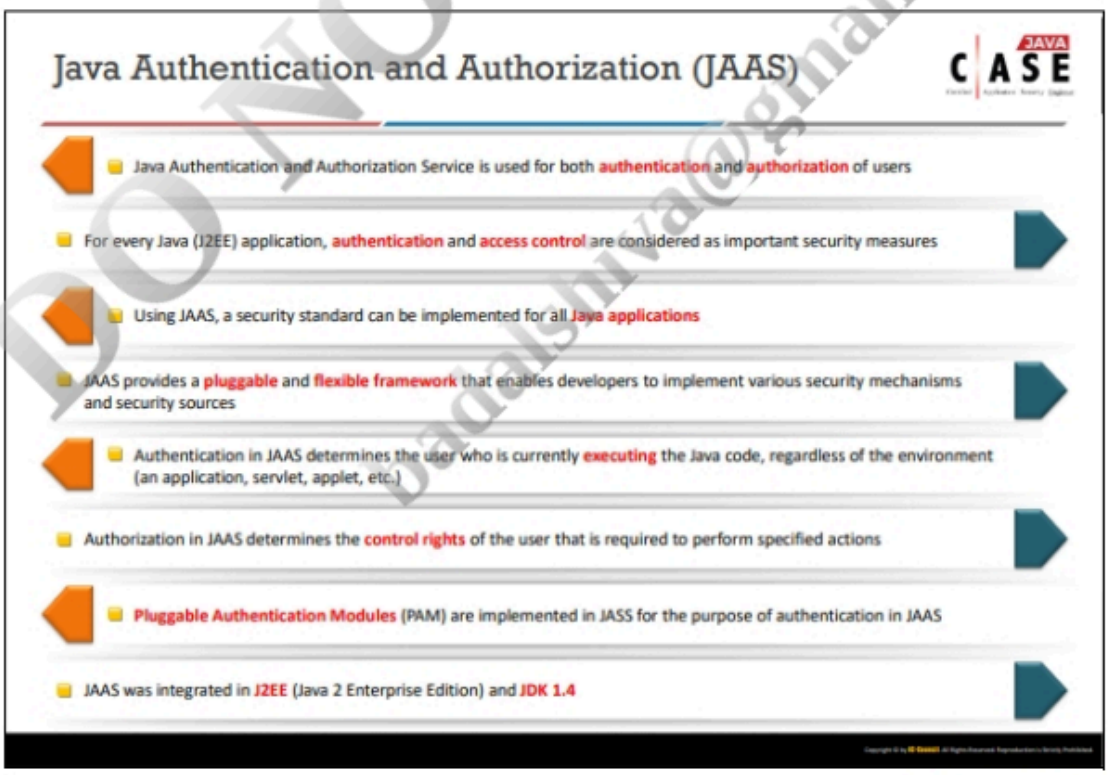
- By using the method `getCallerPrincipal()`, the EJB caller can be determined
- If one exists in the `subject` of calling user, WLSUser Principal is returned by method `javax.ejb.EJBContext.getCallerPrincipal()`
- If there are multiple WLSUser Principals, the first one in `Subject.getPrincipals().iterator()` method is returned
- In case there are no WLSUser Principals, the first non-WLSGroup Principal will be returned by method `getCallerPrincipal()`
- If principals are not present or all principals belong to WLSGroup, the method returns `weblogic.security.WLSPrincipals.getAnonymousUserPrincipal()`

isCallerInRole

- This method determines whether the caller is assigned a security role and authorized to perform actions on the resource of WebLogic in the executing thread

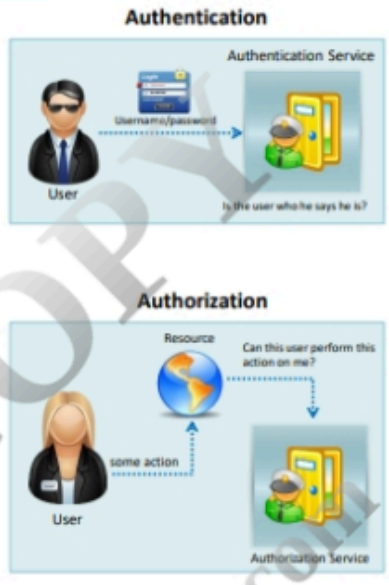
Example, `javax.ejb.EJBContext.isCallerInRole("admin")` returns true if the current user has admin privileges





JAAS Features

- JAAS is implemented using pure Java
- Authentication of users is done through PAM framework
- JAAS supports single sign-on for login authentication in J2EE applications
- JAAS provides centralized role-based control that includes hierarchical roles
- JAAS provides access control policies for user-based, group-based, and role-based authorization



JAAS Architecture

- The application layer initially deals with LoginContext
- Next to LoginContext are one or more LoginModules that are dynamically configured
- These LoginModules make use of appropriate security infrastructure to achieve authentication
- JAAS also provides reference LoginModule implementations such as JndiLoginModule
- JAAS is stackable in addition to being pluggable
- Single login implementation works through ordered security module stacks, featured one over the other, interacting with the Java architecture

A High-level Overview Of JAAS

Pluggable Authentication Module (PAM) Framework

1. The **authentication** framework of JAAS is based on PAM
2. PAM enables authentication in a pluggable fashion that supports the addition of more authentication **modules** to the application
3. Java application functions independently regardless of the implemented authentication **technologies**
4. New authentication technologies can be implemented without the need of **modifying** the application
5. PAM enables different **web applications** to authenticate users against its own data store
6. PAM is the feature that helps programmers to implement single sign-on in web applications that are deployed on the same **application server**

JAAS Classes

JAAS Common Classes

- **Subject** (`javax.security.auth.Subject`)
- **Principal** (`java.security.Principal`)
- **Credential**

JAAS Authentication Classes

- **Callback** (`javax.security.auth.callback.Callback`)
- **CallbackHandler** (`javax.security.auth.callback.CallbackHandler`)
- **Configuration** (`javax.security.auth.login.Configuration`)
- **LoginContext** (`javax.security.auth.login.LoginContext`)
- **LoginModule** (`javax.security.auth.spi.LoginModule`)

JAAS Authorization Classes

- **Policy** (`java.security.Policy`)
- **AuthPermission** (`javax.security.auth.AuthPermission`)
- **PrivateCredentialPermission** (`javax.security.auth.PrivateCredentialPermission`)