Logging in Java

- In Java, logging frameworks are provided by the package **java.util.logging** and used to write log messages on to a central location
- The log messages can be **error messages, informational messages** or **configuration messages**

Logging is divided into three parts:

Logger	Receives the message that is then passed to the **logging framework** The framework then calls the **formatter** including the message
Formatter	Formats the **message** for output The framework then **passes** the **output** to the handler
Handler (Appender)	The appropriate **handler dispositions** the output Examples of Java Logging Frameworks: Apache Commons Logging, Java Logging API, Log4J, SLF4J

Example for Logging Exceptions

- **The code below gives an example of logging an exception to a log file**

Invoke.java

```
7        Logger.getLogger(SimpleClass.class.getName());
8     public void writeLog() {
9        logger.info("Enable logging...");
10       try {
11          throw new Exception("Simulating an exception");
12       } catch (Exception e){
13          logger.log(null, e.getMessage());
14       }
15    }
16     public static void main(String[] args) {
17
18       SimpleClass simpleObj = new SimpleClass();
19       simpleObj.writeLog();
20       logger.info("Done...");
21    }
```

Logging Levels

- **Logging plays a major role in web applications as it is the only way to track the server's activities. Logging levels based on Java log4j are as follows:**

DEBUG
- The lowest restricted Java logging level
- The messages logged in this level are used to debug the application
- It is used only in the development and testing environment

INFO
- This level is more restricted than the DEBUG Java logging level
- The messages logged in this level are used to give information to the programmers
  E.g. The server has started, incoming and outgoing messages

WARN
- This level is more restricted than the INFO level
- The messages logged in this level are used to give warning messages
  E.g. Connection lost between client and server, Database connection lost etc.

ERROR
- This level is more restricted than the WARN level
- Error and exception messages are logged in this level
- ERRORS should always be printed in Java

FATAL
- This level indicates severe error conditions that leads the application to abort
- After this, the application may eventually crash or get stopped

OFF
- This level has the highest possible rank and it is intended to turn off logging in Java

Logging using Log4j

Log4j and Java Logging API

	log4j	java.util.logging
Manager Class	org.apache.log4j.LogManager	java.util.logging.LogManager
Logger Objects	org.apache.log4j.Logger	java.util.logging.Logger
Named Loggers	Supported	Supported

Logging Levels	• Levels are declared in the org.apache.log4j.Level class
	• log4j has five pre-defined levels:
	  – FATAL
	  – ERROR
	  – WARN
	  – INFO
	  – DEBUG
		• Levels are declared in the java.util.logging.Level class
		• There are seven pre-defined levels:
		  – SEVERE
		  – WARNING
		  – INFO
		  – CONFIG
		  – FINE
		  – FINER
		  – FINEST

Log Event Object	org.apache.log4j.spi.LoggingEvent	java.util.logging.LogRecord
Output Destinations	Appender classes - these classes implement the org.apache.log4j.Appender interface	Handler classes - these classes extend java.util.logging.Handler

Log4j and Java Logging API (Cont'd)

	log4j	java.util.logging
Output Message Filtering	Filter classes. These classes extend org.apache.log4j.spi.Filter	Filter classes. These classes must implement the java.util.logging.Filter interface
Output Message Formatting	Layout classes extend org.apache.log4j.Layout	Formatter classes extend java.util.logging.Formatter
Programmatic Configuration	Supported	Supported
Property File Configuration	Supported	Supported
XML Configuration	Supported. The XML configuration file must comply with log4j.dtd	Not supported
Email Notification	SMTPAppender class	SMTPHandler class (third-party extension)

Java Logging using Log4j

- Given below is a sample code used to log files using Log4j

*Invoke.java

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

public class Invoke {
    private static final Logger logger =
        Logger.getLogger(SimpleClass.class.getName());
    public void writeLog() {
        logger.debug("Here Debug message");
        logger.info("Enable logging...");
        logger.warn("Warning Message");
        logger.error("Here Error message");
        logger.fatal("fatal error message");


        try {
            throw new Exception("Simulating an exception");

        } catch (Exception e){
            logger.log(null, e.getMessage());
        }
    }
    public static void main(String[] args) {

        SimpleClass simpleObj = new SimpleClass();
        simpleObj.writeLog();
        logger.info("Done...");
    }
```

Secure Coding in Logging

Vulnerabilities in Logging

1. It is not a good practice to log and then throw the errors

DemoForException.java

```
import java.io.BufferedReader;

public class DemoForException {
    private static final Logger logger =
            Logger.getLogger(SimpleClass.class.getName(
public  void test()
 {
try {
    int i=5;
    i=i/0;
} catch (Exception ex) {

    logger.error("Error",ex);
    throw ex;

    }
 }
```

2. Either log exceptions or throw exceptions; both should not be done together

Example: Log Exception

DemoForException.java

```
public  void Info() throws MyException
{
try {
    int i=5;
    i=i/0;
} catch (ArithmeticException ex) {

    logger.error("Error",ex);
    throw new MyException("Error",ex);
```

Example: Throw Exception

```
    try {

    }
catch (ArithmeticException ex) {

    ex.printStackTrace();
    throw new MyException("Error",ex);

    }
```

Vulnerabilities in Logging (Cont'd)

3. Exceptions should be thrown instead of returning null values are returned only in a normal use case

DemoForException.java

```
    try {

    }
catch (ArithmeticException ex) {

    ex.printStackTrace();
    return;

    }
```

4. Null values are returned only in a normal use case

```
22    public String map()
23    {try{
24
25    }catch(Exception e)
26    {
27      return null;
28    }
```

Logging: Vulnerable Code and Secure Code

1. Group log messages regardless of the level

- The vulnerable code below may look fine in a test case, but when it is displayed in the log file of an application server with 500 threads running parallel, the two messages may be spaced up with 1000 lines in-between in the log file

Vulnerable Code

```
DemoForException.java
57 try {
58         logger.debug("Using cache policy A");
59         logger.debug("Using retry policy B");
60     }
61 catch (ArithmeticException ex) {
62
```

Secure Code

```
DemoForException.java
57 try {
58         logger.debug("Using cache policy A, using retry policy B");
59
60     }
61 catch (ArithmeticException ex) {
62
```

Logging: Vulnerable Code and Secure Code (Cont'd)

2. Parameterized logging using SLF4J

Vulnerable Code

```
DemoForException.java
57     }
58
59 try {
60     if(logger.isDebugEnabled())
61     {
62         logger.debug("created user[" + user + "] with role [" + role + "]");
63
64     }
65
66     }
67 catch (ArithmeticException ex) {
68
69     logger.error("message",ex);
70     return;
```

Secure Code

```
61
62 try {
63     slf4jLogger.debug("created user[{}] with role [{}]",user,role);
64
65     }
66 catch (ArithmeticException ex) {
67
68     slf4jLogger.error("message",ex);
69     return;
70     }
```

Logging: Vulnerable Code and Secure Code (Cont'd)

3. Log the exception stacktrace

Vulnerable Code

```
DemoForException.java
57      }
58
59  try {
60      int i=5;
61      i=i/0;
62      }
63  catch (ArithmeticException ex) {
64
65      logger.error("Error in application"+ex);
66      }
67
68  }
69  }
```

Secure Code

```
DemoForException.java
57      }
58
59  try {
60      int i=5;
61      i=i/0;
62      }
63  catch (ArithmeticException ex) {
64
65      logger.error("Error in Application", ex);
66      }
67
68  }
69  }
```

Logging: Vulnerable Code and Secure Code (Cont'd)

4. Log exceptions in proper order

Vulnerable Code

```
DemoForException.java
33  {
34  try {
35      int i=5;
36      i=i/0;
37  } catch (Exception lolevelException) {
38
39
40      throw new ArithmeticException();
41
42      }
```

Secure Code

```
DemoForException.java
36      int i=5;
37      i=i/0;
38  } catch (Exception lolevel) {
39
40          throw new HighLevelException("Error", lolevel);
41          }
```

Logging: Vulnerable Code and Secure Code (Cont'd)

5. Log exceptions when they are handled. Do not log messages before they are handled

Vulnerable Code

DemoForException.java

```
35 try {
36     int i=5;
37     i=i/0;
38 } catch (Exception hklvel) {
39
40         slf4jLogger.error("catch Error herer", hklvel);
41         }
```

Logging: Vulnerable Code and Secure Code (Cont'd)

6. Log uncaught exceptions

Vulnerable Code

- In web applications, uncaught exceptions can be sent to a JSP page and log it as follows:

web.xml

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

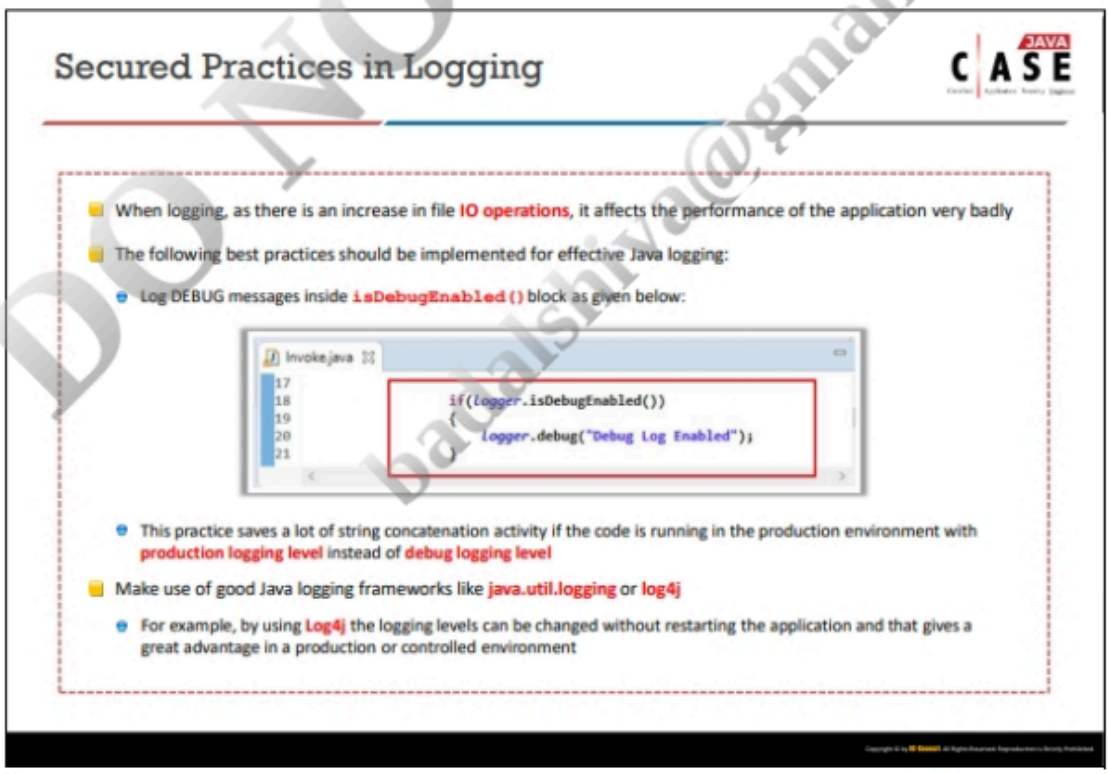
Design | Source

Secure Code

- The example code is as follows:

*DemoForExcepti... | MyDomainEg.java | FileValidation.j... | DivByZero8g.java

```
12⊖     public static void main(String args[]) throws IOException
13      {
14
15⊖         Thread.setDefaultUncaughtExceptionHandler(new UncaughtExceptionHandl
16⊖             @Override
17              public void uncaughtException(Thread t, Throwable e) {
18                  Logger log = org.slf4j.LoggerFactory.getLogger(t.getClass());
19                  log.error("Exception in thread [" + t.getName() + "]", t);
20              }
21          });
22
```

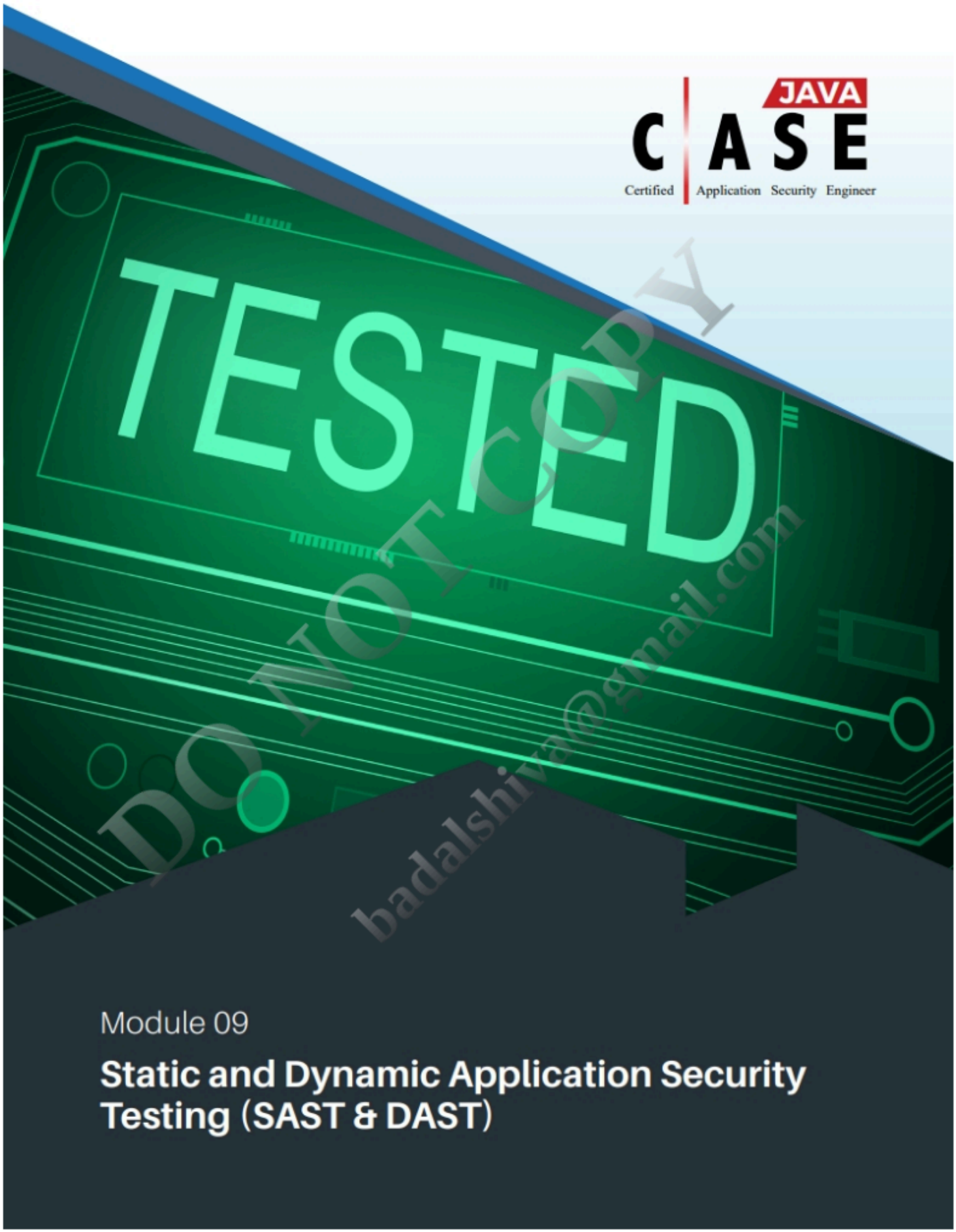


Secured Practices in Logging (Cont'd)

- Do not log sensitive data when logging in a **production environment**. Logging sensitive information may cause security risks. Example: e-mail address, phone numbers and account numbers should not be logged
- In server applications, all **bugs** and **errors** can be **tracked** only by using Java logs and too many logs will affect the application; therefore, select the appropriate logging level for every logging message
- Ensure that the format of the Java logging in the specified Java **logger** is included. In the case of **java.util.logging** API, the properties file should include the **logging formatter**. While printing logs the thread name and the Java class name should be specified
- Log messages consistently and the messages must be **informative**. The data should be **printed** with the **message** wherever required
- Use code level prefix when logging information (e.g. **database code** or **session code** etc.) so that it will be clear to identify which part of the code is **printing** the log message

Module Summary

- Exception and error occur when an unexpected or abnormal event takes place during the execution of the application
- Exceptions in Java are handled using 'try-catch-finally' blocks
- java.lang.InterruptedException is thrown when a thread is interrupted while sleeping or waiting
- Use of return, break, continue or throw statements is restricted in a finally block
- System.exit() when invoked can even terminate the Java Virtual Machine (JVM) resulting in the termination currently running programs as well as threads that might even lead to denial-of-service (DoS) attacks
- RuntimeException, Exception, or Throwable should not be thrown or else it may lead to various errors
- printStackTrace() function should not be used in code as it reveals all the exception information to the user
- In Java, logging frameworks are provided from package java.util.logging and used to write log messages on to a central location



Module 09

Static and Dynamic Application Security Testing (SAST & DAST)

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

This page is intentionally left blank.

Module Objectives

1. Introduction to Static Application Security Testing (SAST)
2. Discuss Manual Secure Code Review Techniques for Most Common Vulnerabilities
3. Introduction to Dynamic Application Security Testing
4. Discuss DAST using Automated Application Vulnerability Scanning Tools
5. Discuss DAST using Proxy-based Security Testing Tools

Static Application Security Testing

Static Application Security Testing (SAST)

1. SAST, also known as **secure code review**, is one of the software security assurance approach to identify security-related **weaknesses** in the code
2. It involves detailed systematic inspection of **source code** to detect vulnerabilities and design flaws
3. It should be performed toward the end of the source code development when application code is **stable** or **nearly completed**
4. It should always be performed in combination with **human effort (Manual)** and **technology support (Automated)**

Objectives of SAST

- SAST aims to detect **application security vulnerabilities** and their **root causes** when code is not running

Root cause analysis (Source to sink)
Source to Sink analysis (what is the source (input) and the sink (vulnerable code pattern) for any vulnerability) is carried out using **Data flow diagrams**

Design Analysis
Design flaws like **mishandling of nonuser inputs** and external integrations like **server-to-server** can be easily identified

Find all Instances of the Common Vulnerabilities
Security code review enumerates all instances of **certain vulnerability**

Identify Uncommon Security Flaws
Security code review being specific to the application may **highlight some flaws** like insecure termination of execution flow, synchronization error, etc.

Analyze Effectiveness of Existing Security Controls
Security code review enumerates **weakness** in the existing security control against known **threats**

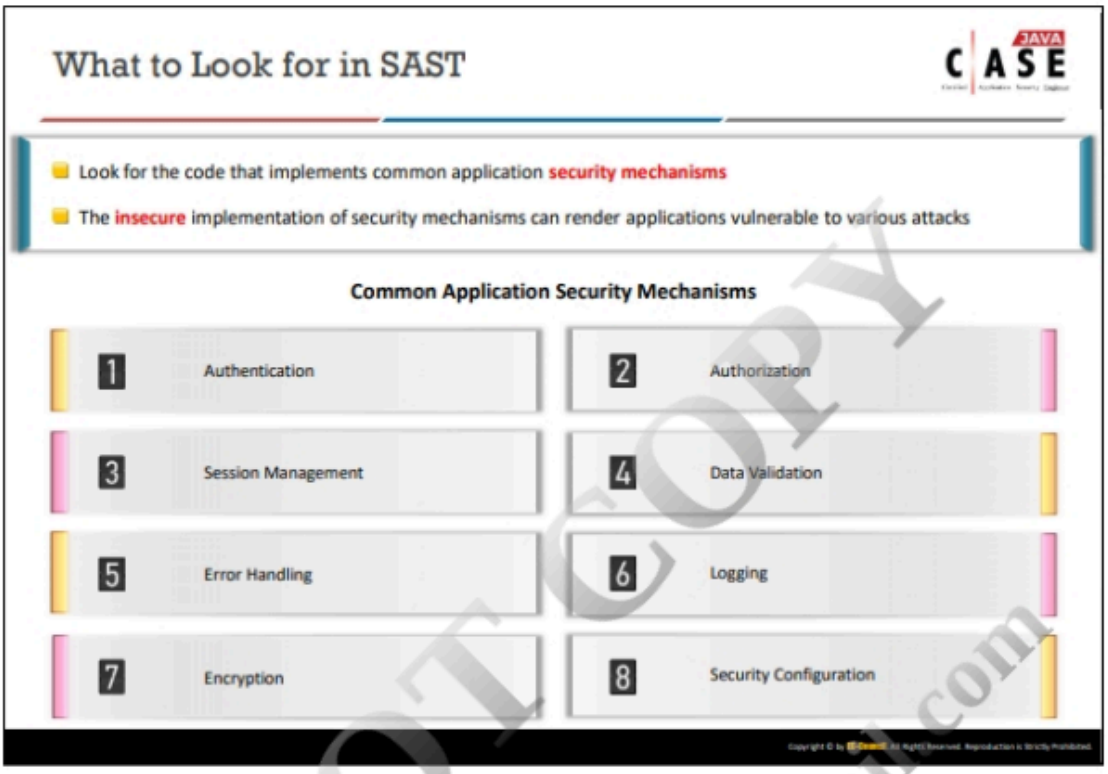
Effective Remediation Recommendations
Suggests **mitigation techniques** that can best suit the application instead of a generic one

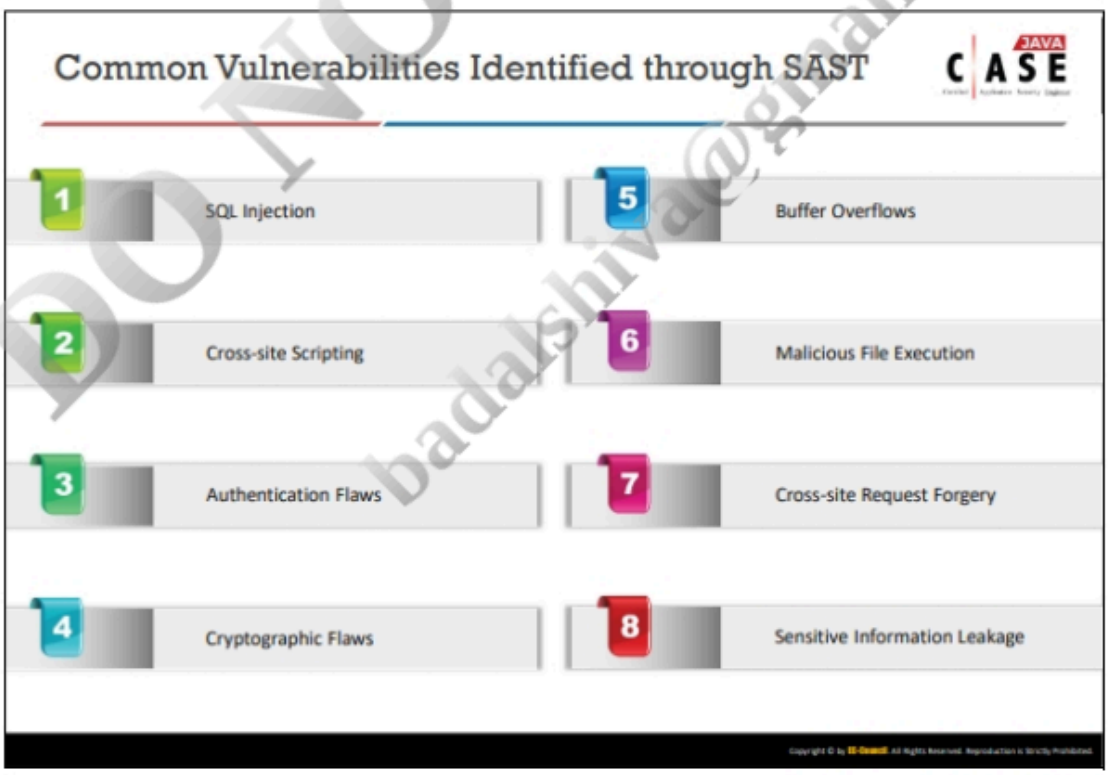
Why SAST

- Many serious security vulnerabilities can be addressed
- Decreases the remediation cost of vulnerabilities in later stages
- Deeper analysis is possible with less false positive and negative
- Help you to add best secure coding practices as per department coding standards which can prevent future security issues

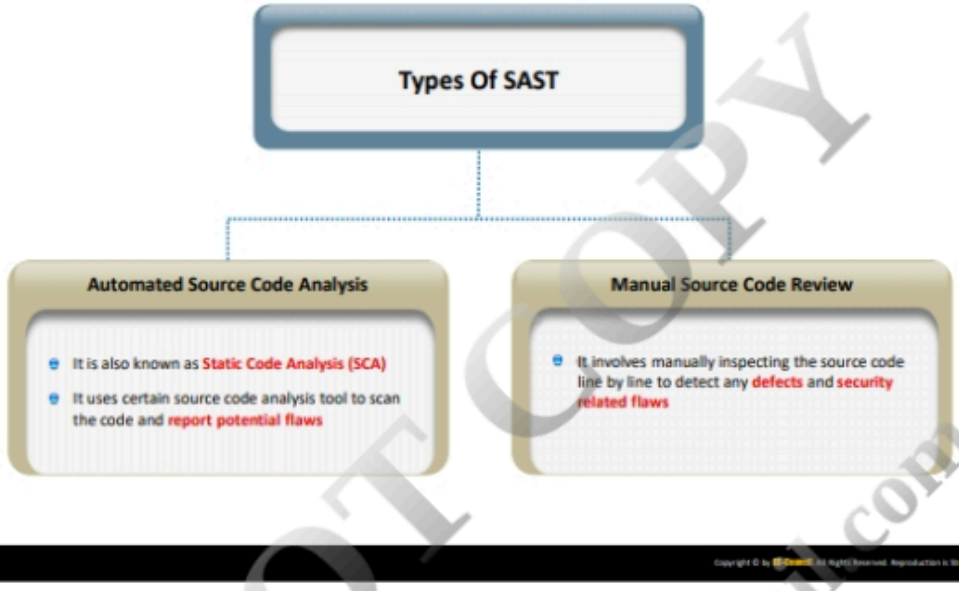
Skills Required for SAST

1. Able to think from the attacker's perspective
2. Working knowledge of the programming language under review
3. Knowledge of standard, secure coding techniques
4. Quickly able to detect insecure coding practices
5. Analytical skills

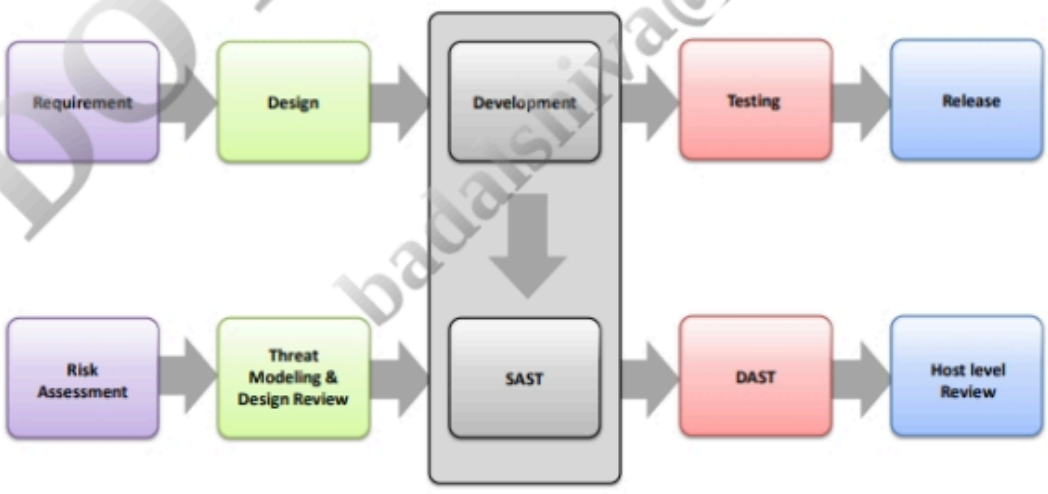




Types of SAST



Where does SAST Fit in SDLC?

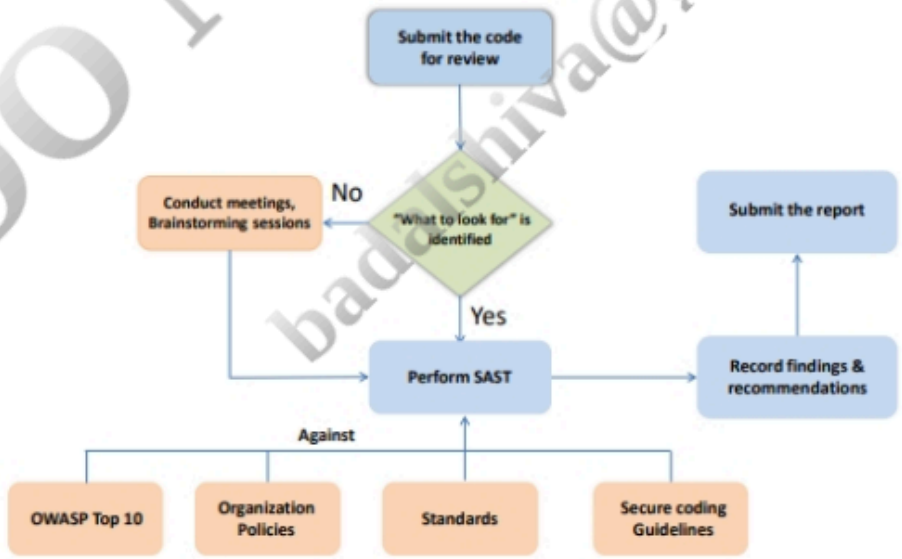


SAST Steps

1. **Perform an application walkthrough**
   - Understand the applications functionality
2. **Developers Interview**
   - Talk to **developers**; more you involve developers, more effective you secure code review will be
   - It will help you quickly
     - Determine the **high risk areas**
     - Understand **developer's logic** behind particular functionality
     - Establish **friendly** and **respectful** relationship between you and developers
3. **Analyze threat Model**
   - Figure out the major existing **security threats**
4. **Define secure code review objectives**
   - Based on threat model analysis, identify the objectives such as:
     - Code review for all major **security vulnerabilities**
     - Code review for **OWASP top 10**
     - Code review for **specific vulnerability**

5. **Define scope for secure code review**
   - Be clear on what to **look** for and what to **avoid**
   - Calculate the **time** and **efforts** required
   - Define the **budget constraints**
6. **Categorized the Objectives**
   - Prioritize the objective based on **severity** and **requires more attention**
7. **Conduct Static Code Analysis**
   - Use **multiple** source code analysis tools to perform secure source code analysis
   - Most common vulnerabilities and flaws are **identified**
8. **Conduct Manual Source code Review**
   - Inspect the application's code line by line in order to find the **use of insecure coding practices**
9. **Remediation/Recommendation**
   - Suggest secure coding practice for each **findings**
10. **Prepare Report**

SAST Activities-flow Chart



Recommendation for Effective SAST

CASE JAVA

01 Code review should always be done by a **third party person**, a **peer** or an **external entity**

02 Understand the application architecture and business logic and sort out the **possible security issues** that the application can be vulnerable to

03 Set clear set of objectives for "**what to look for**" based on security issues

04 Review code for security objectives only. Do not review code for other issues like performance, business logic, reliability, etc. Do it **separately**

05 Spend time to set **clear objectives**

06 Prioritize the security issues; higher-level security vulnerabilities should be **addressed first**

07 Set a **reasonable time limit** for the review

08 Do not lose track while going into detail of the **higher-level** security vulnerabilities

09 Do not spend too much time in not a high-priority area or objective, **address them later**

10 Always select small and manageable piece of code for review. It will help you to find a larger number of security vulnerabilities with **great focus** and within **less time**

11 Use multi-pass approach. Let the multiple people allow to review the **same code**

12 Detect secure coding mistakes that are repeatedly committed and add the corresponding secure coding practice in your **development department's coding standards**

SAST Deliverable

CASE JAVA

SAST report should include:

1. Types of vulnerabilities
2. Severity level/Impact
3. Location, line of code where security issue exists
4. Recommendations

Automated Source Code Analysis

- With the help of automated source code analysis, you can **identify** the code associated with most common application **security vulnerabilities**
- The **static code analysis** tools automate the entire process of secure code review; however, they scans the code **superficially**
- Multiple static code analysis tools should be used to conduct **secure code review**

Advantages:

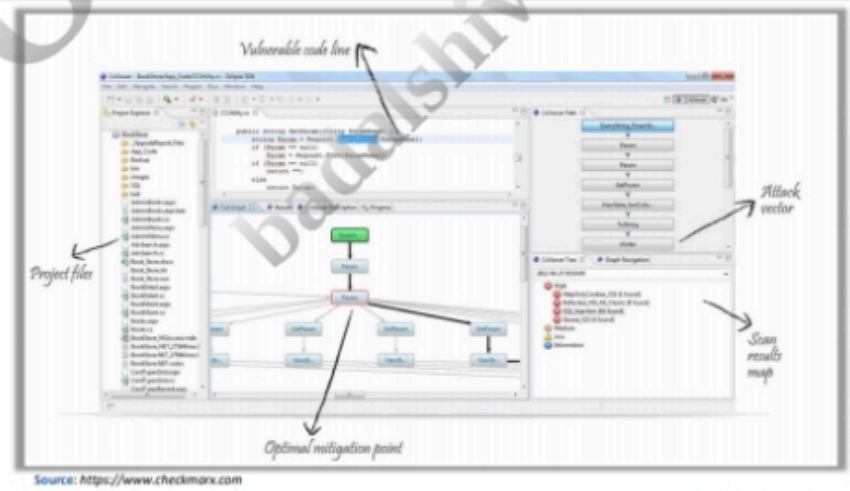
- Takes **very less time** to complete

Disadvantages:

- Unable to **uncover** every issue
- Good automated review tools are **expensive**
- High possibility of **false-positive** or **false-negative errors**
- Requires **human intervention** to deal with of false-positive or false-negative

Static Code Analysis using Checkmarx Static Code Analysis

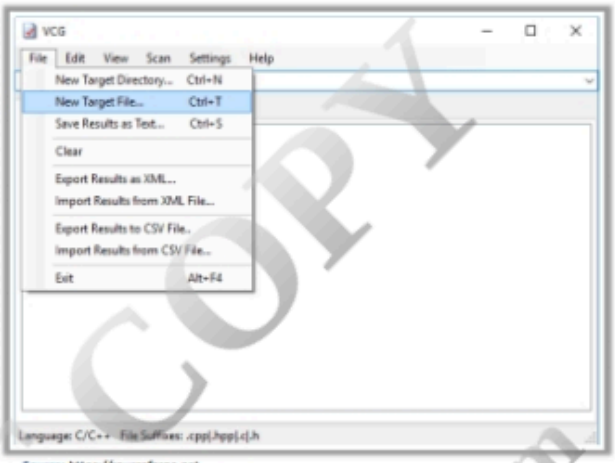
- **Checkmarx** static code analysis software, seamlessly integrated into development process which identifies hundreds of known code vulnerabilities



Source: https://www.checkmarx.com

Static Code Analysis using Visual Code Grepper (VCG)

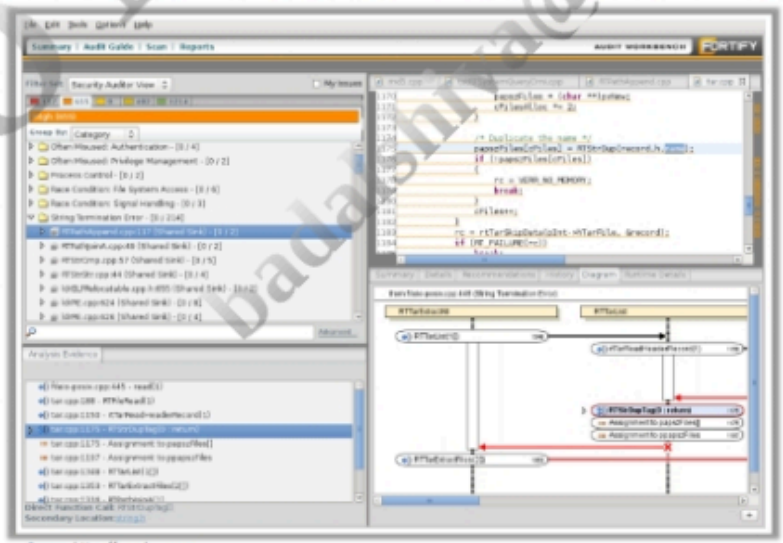
- VCG is an automated code security review tool for C++, C#, VB, PHP, Java and PL/SQL which is intended to drastically speed up the code review process by **identifying bad/insecure code**
- Steps for SCA:
  - **Install** and **run** the VCG tool
  - Select the **language** of your project
  - Browse the **project/file** of your interest and click **Scan**



Source: *https://sourceforge.net*

Static Code Analysis using HP Fortify

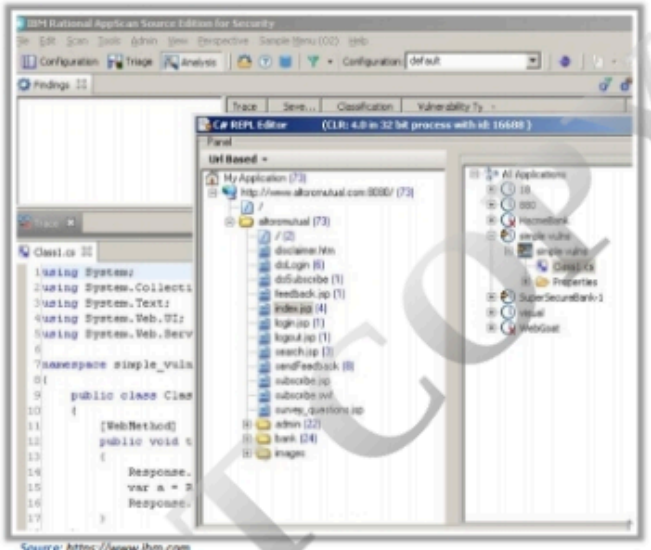
- Fortify Static Code Analyzer is an **automated static code analysis** to help developers eliminate vulnerabilities and build secure software



Source: *https://soas.hpe.com*

Static Code Analysis using Rational AppScan Source Edition

- AppScan Source is a static application security testing (SAST) solution which scans your application source code for **security vulnerabilities**



Source: https://www.ibm.com

Selecting a Static Analysis Tool

- You need to select the right tool while selecting the static analysis tool for secure code review
- The process of selecting the **right tool** depends on certain conditions:
  - The static analysis tool used for reviewing the code should support the **programming language** used for developing the application
  - The accuracy of the tool with respect to **false-negative** and **false-positive errors**
  - The tool needs to understand the **libraries** and **frameworks** used in the application
  - It should be easy to **integrate the tool** to the IDE selected for developing the application
  - The **time** and **effort** required for the set-up of the tool
  - The type of analysis preferred for code review process
  - The license cost incurred for the tool
- You should have a clear idea regarding the type of static code analysis required for performing **code review**

Manual Secure Code Review

With manual secure code review, you can analyze and detect many security vulnerabilities that are missed out during the **static code analysis**

Advantages

- Less chance of getting **false-positive** and **false-negatives**
- High percentage of **vulnerabilities** can be detected

Disadvantages

- Time consuming process
- Requires **good understanding** of the code used

Manual Secure Code Review for Most Common Vulnerabilities

Code Review for PCI DSS Compliance

- The PCI DSS compliancy is checked against a list of **PCI DSS requirements**
- Check the applications against a list of PCI DSS requirements

PCI DSS Requirement	Description
6.3.7	Review of the custom code for vulnerabilities before releasing the code to production or customers
6.3.5	Deletion of custom user accounts, usernames and passwords before releasing it to the customers
6.5	Review and prevention of common coding vulnerabilities in an application
6.6	Review of custom application codes and installation of application layer firewall for web applications

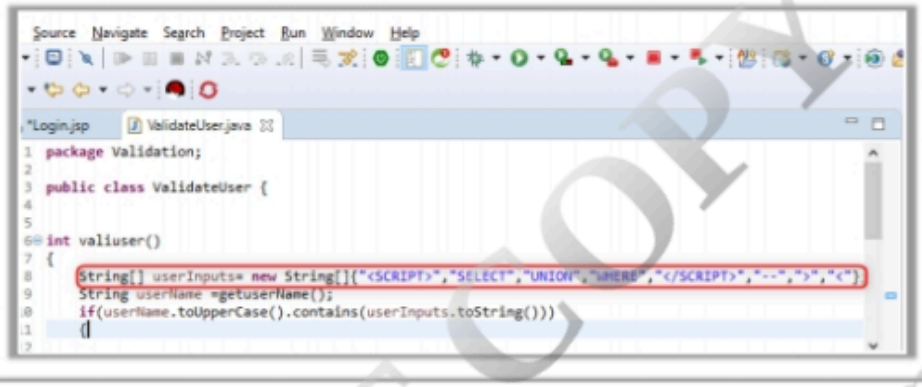
Code Review for PCI DSS Compliance (Cont'd)

- **The PCI DSS requirement 6.5 reviews and prevents common coding vulnerabilities such as:**

- 6.5.1 Unvalidated input
- 6.5.2 Broken access control
- 6.5.3 Broken authentication and session management
- 6.5.4 Cross-site scripting (XSS) attacks
- 6.5.5 Buffer overflows
- 6.5.6 Injection flaws
- 6.5.7 Improper error handling
- 6.5.8 Insecure storage
- 6.5.9 Denial of service
- 6.5.10 Insecure configuration management

Code Review for Blacklisting Validation Approach

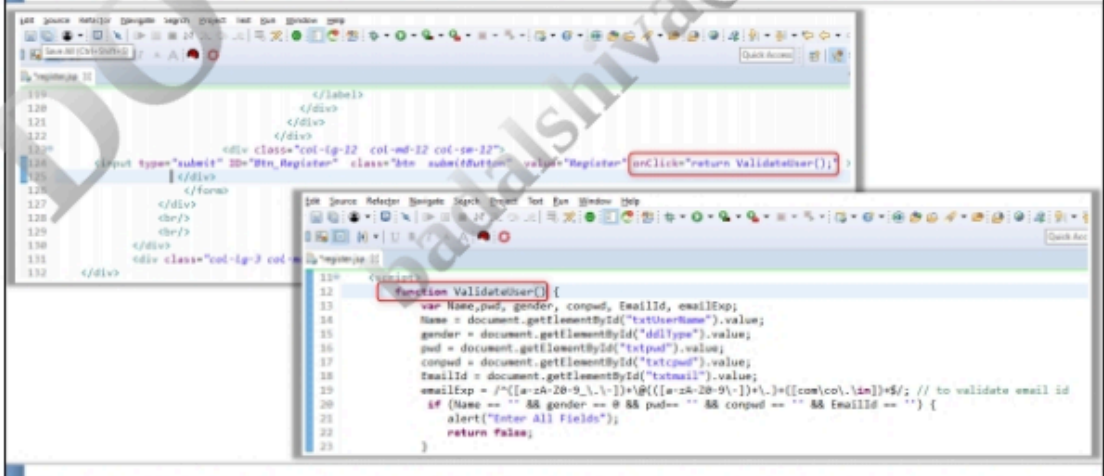
Code Pattern: Look for code where the developer is attempting to use a **blacklisting approach** to validate user inputs



Recommendation: Use a **whitelisting** based validation approach to validate your inputs

Code Review for Client-side Validation Approach

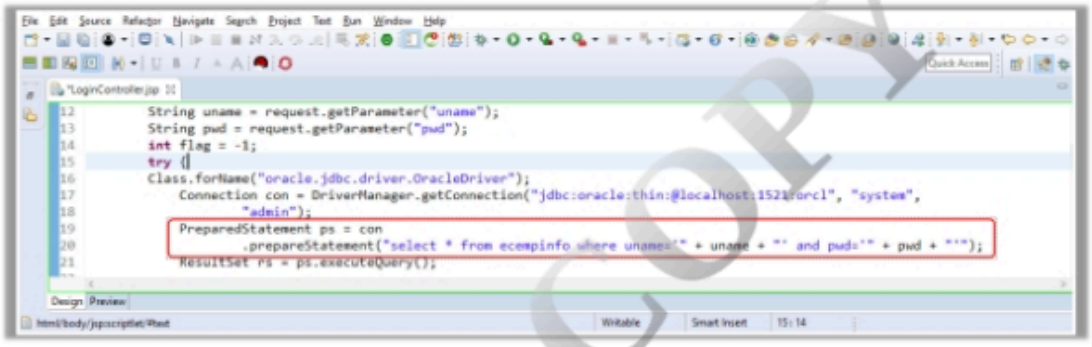
Code Pattern: Look for code where the developer is attempting to use only **client-side input validation script** to validate user inputs



Recommendation: Client-side validation can be bypassed easily. Use both **client** and **server-side validation** techniques to validate user inputs

Code Review for Non-parametrized SQL Query

Code Pattern: Look for code where the developer is attempting to write a **non-parametrized query**

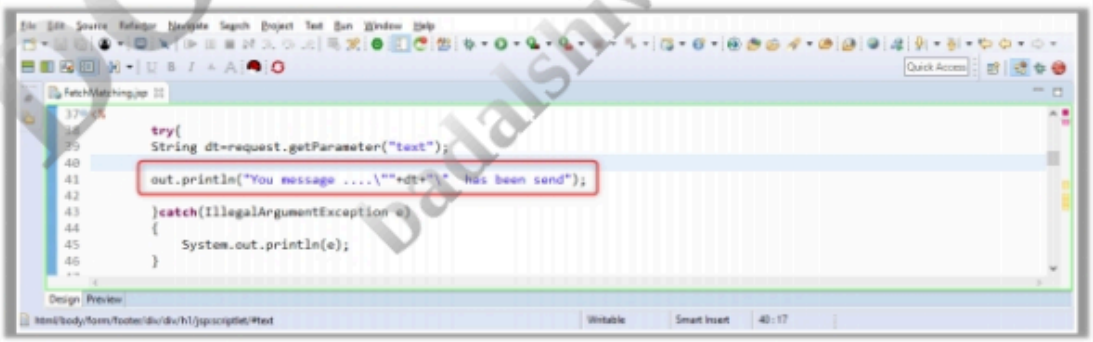


```
String uname = request.getParameter("uname");
String pwd = request.getParameter("pwd");
int flag = -1;
try {
Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "system",
            "admin");
    PreparedStatement ps = con
            .prepareStatement("select * from ecempinfo where uname='" + uname + "' and pwd='" + pwd + "'");
    ResultSet rs = ps.executeQuery();
```

Recommendation: Always use **parametrized** query to avoid **SQL injection attacks**

Code Review for XSS Vulnerability

Code Pattern: Look for code where the developer is attempting to **render output without encoding**

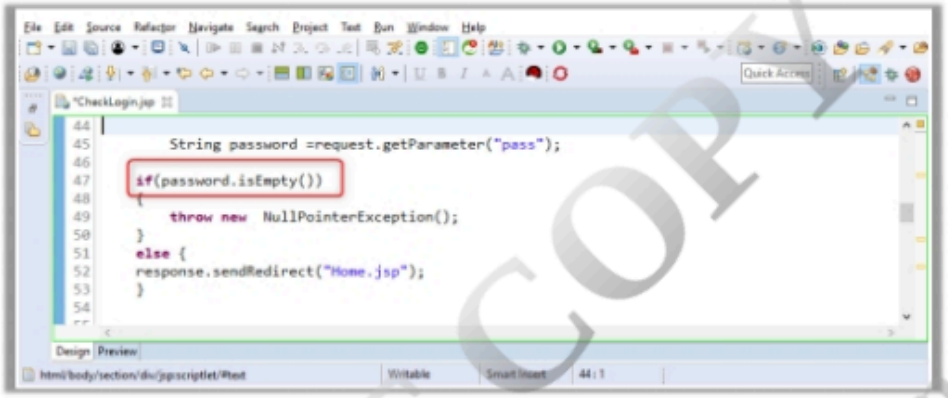


```
try{
String dt=request.getParameter("text");

out.println("You message ....\""+dt+"\" has been send");

}catch(IllegalArgumentException e)
{
    System.out.println(e);
}
```

Recommendation: Always **encode** your output to avoid **XSS**

Code Review for Weak Password Authentication

Code Pattern: Look for code where the developer is attempting to authenticate users with a **weak password condition**

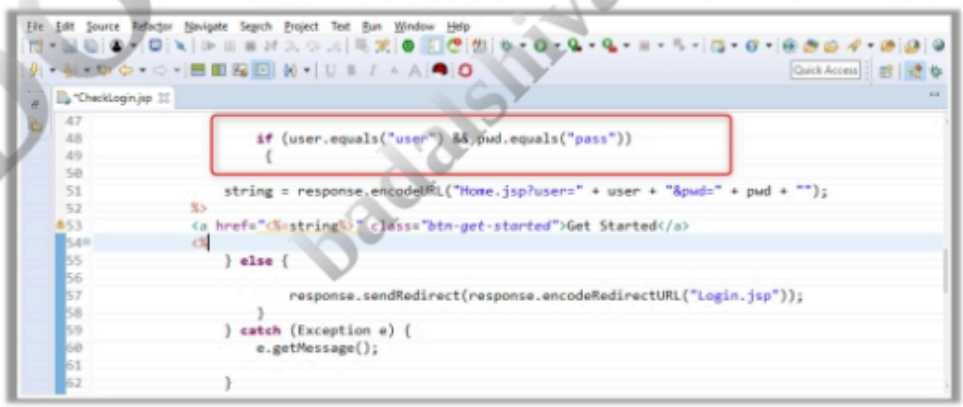


```
44
45        String password =request.getParameter("pass");
46
47    if(password.isEmpty())
48    {
49        throw new  NullPointerException();
50    }
51    else {
52    response.sendRedirect("Home.jsp");
53    }
54
```

Recommendation: Always use **strong password rules** or **conditions** to authenticate users

Code Review for Hard-coded Passwords

Code pattern: Look for code where the developer is attempting to use **hard-coded passwords**

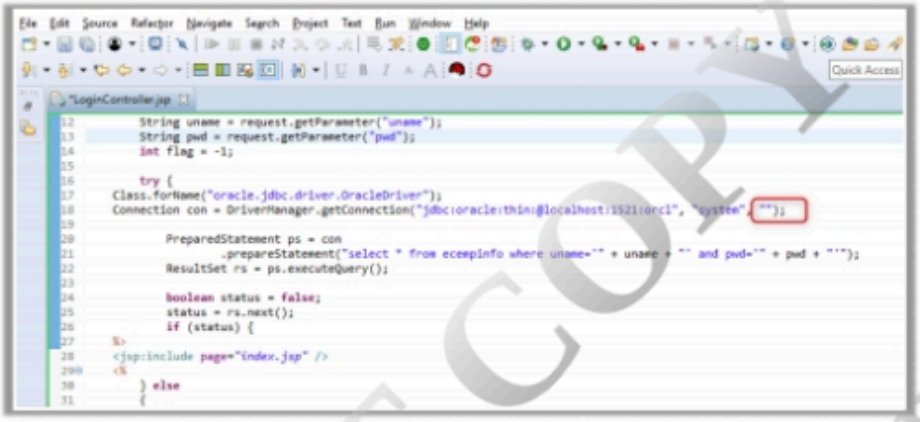


```
47
48                  if (user.equals("user") && pwd.equals("pass"))
49                  {
50
51                  string = response.encodeURL("Home.jsp?user=" + user + "&pwd=" + pwd + "");
52              %>
53              <a href="<%=string%>" class="btn-get-started">Get Started</a>
54              <%
55                  } else {
56
57                          response.sendRedirect(response.encodeRedirectURL("Login.jsp"));
58                      }
59                  } catch (Exception e) {
60                      e.getMessage();
61
62                  }
```

Recommendation: Never use hard-coded secrets such as **passwords** in coding. Hard-coded passwords can be an easy target for attackers

Code Review for Empty Password in Connection String

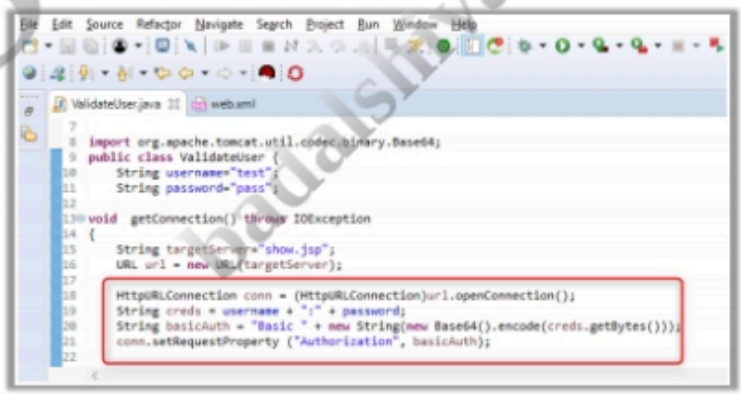
- **Code pattern**: Look for the connection string where the developer is attempting to use an **empty password** to connect to a remote database server



Recommendation: Never set **empty** or **plain text** for connection string as this will facilitate **dictionary attacks** or **brute force attacks**

Code Review for Insecure Basic Authentication

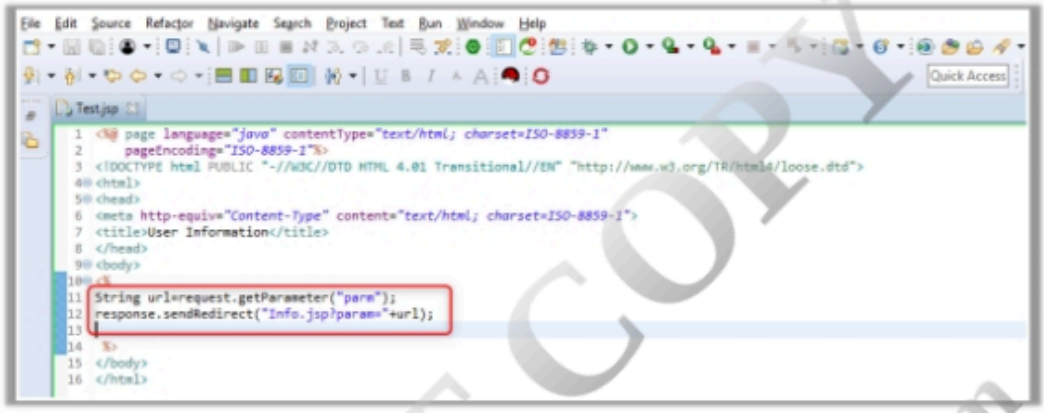
- **Code Pattern**: Look for code where the developer is attempting to use **basic authentication and HTTP connection** without using SSL



Recommendation: Use **SSL** for basic authentication so that credential can not be decoded easily

Code Review for Open Redirect

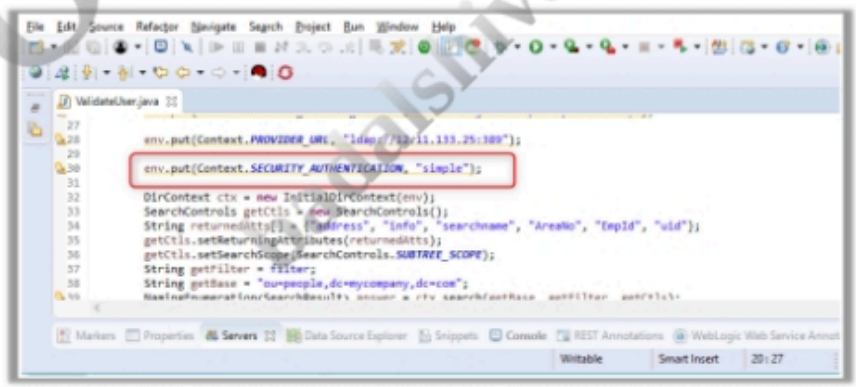
Code Pattern: Look for code where the developer is attempting to redirect to the path that is created from user supplied input



Recommendation: Always avoid using redirects or forwards. User input should not be used to construct the source path. If it is necessary to send a parameter then check if the parameter is valid. Ensure that the user is authorized to browse the constructed URL

Code Review for Insecure LDAP Authentication

Code Pattern: Look for code where the developer is attempting to use "simple" basic authentication which is weak; attackers can easily get cleartext



Recommendation: Always use secure mechanisms like those listed in LDAP. LDAP supports SASL mechanism, SASL provides a security layer for authentication between client and server

Code Review for Insecure Authorization Mechanism

Code Pattern: Look for code where the developer is attempting to use an **inadequate authorization mechanism** by which the code becomes complex and also hard to maintain. Even if the mistake is simple, an attacker can easily abuse the code

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>User Information</title>
</head>
<body>
<%
if (request.isUserInRole("admin")){
    response.sendRedirect("userinfo.jsp");
}
else if (request.isUserInRole("user")){
 response.sendRedirect("userinfo.jsp");
}
else {
```

Recommendation: Always use a **proper authorization** method which is secure and easy to be altered

Code Review for Weak Password Length

Code pattern: Look for code where the developer is attempting to set passwords with a **weak hashing** algorithm

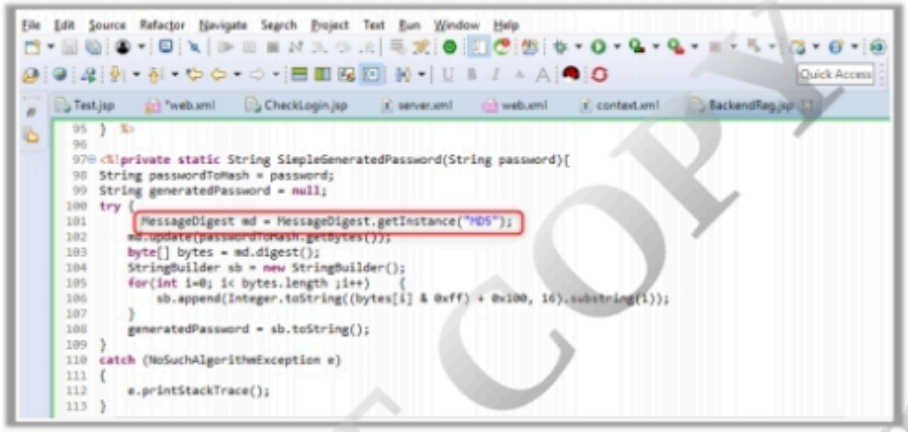
```
} %>

<%!private static String SimpleGeneratedPassword(String password){
String passwordToHash = password;
String generatedPassword = null;
try {
     MessageDigest md = MessageDigest.getInstance("SHA1");
    md.update(passwordToHash.getBytes());
    byte[] bytes = md.digest();
    StringBuilder sb = new StringBuilder();
    for(int i=0; i< bytes.length ;i++)    {
        sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
    }
    generatedPassword = sb.toString();
}
catch (NoSuchAlgorithmException e)
{
    e.printStackTrace();
}
```

Recommendation: Set passwords with a strong hashing function like **Bcrypt** to make it unbreakable

Code Review for Use of Weak Hashing Algorithm

CASE JAVA

Code Pattern: Look for code where the developer is attempting to set a **weak hashing algorithm**



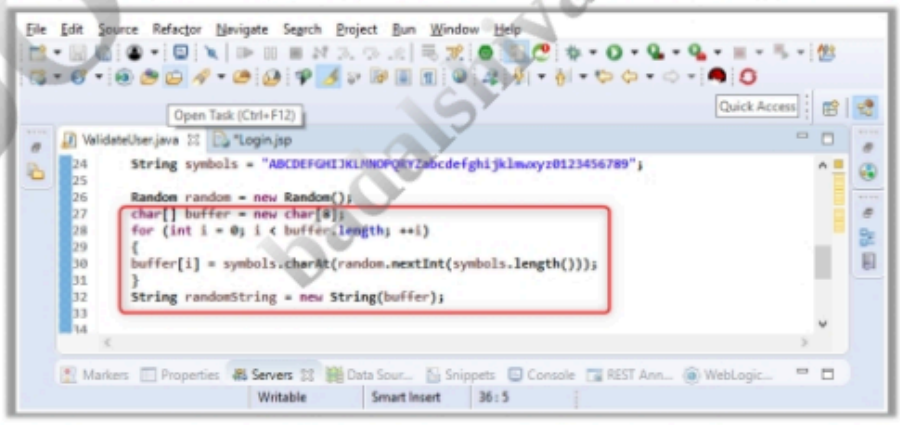
```
95  } %>
96
97  <%!private static String SimpleGeneratedPassword(String password){
98  String passwordToHash = password;
99  String generatedPassword = null;
100 try {
101     MessageDigest md = MessageDigest.getInstance("MD5");
102     md.update(passwordToHash.getBytes());
103     byte[] bytes = md.digest();
104     StringBuilder sb = new StringBuilder();
105     for(int i=0; i< bytes.length ;i++)    {
106         sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
107     }
108     generatedPassword = sb.toString();
109 }
110 catch (NoSuchAlgorithmException e)
111 {
112     e.printStackTrace();
113 }
```

Recommendation: Always use **strongest hashing algorithm**

Code Review for Use of Weak Random Number Generator

CASE JAVA

Code Pattern: Look for code where the developer is using a **weak random number generation method**. An 8 character string can be predicted by an attacker

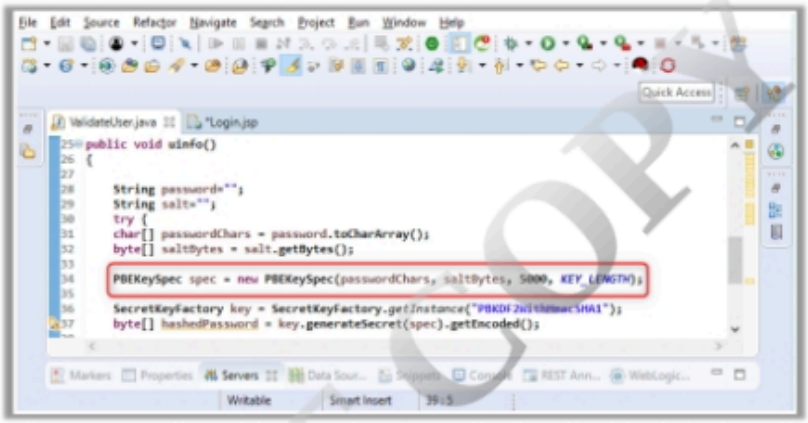


```
24      String symbols = "ABCDEFGHIJKLMNOPQRYZabcdefghijklmxyz0123456789";
25
26      Random random = new Random();
27      char[] buffer = new char[8];
28      for (int i = 0; i < buffer.length; ++i)
29      {
30      buffer[i] = symbols.charAt(random.nextInt(symbols.length()));
31      }
32      String randomString = new String(buffer);
```

Recommendation: To prevent an attacker from predicting random numbers, use a string with more than **8 characters** to generate random numbers

Code Review for Use of Insecure PBE Work Factor

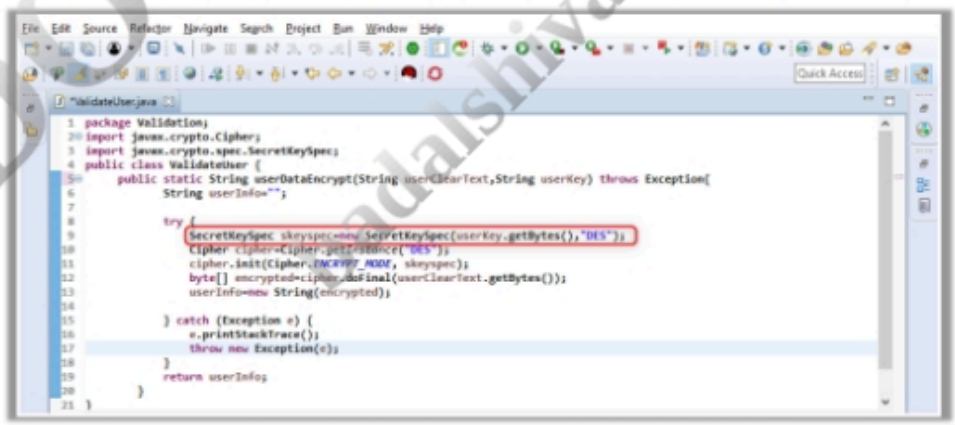
Code Pattern: Look for code the developer is using with a **work factor** of **5000** which is considered to be insecure



Recommendation: The developer should use **big iteration value** which generates complex hashing to prevent the password from being predicted

Code Review for Use of Weak Encryption Algorithm

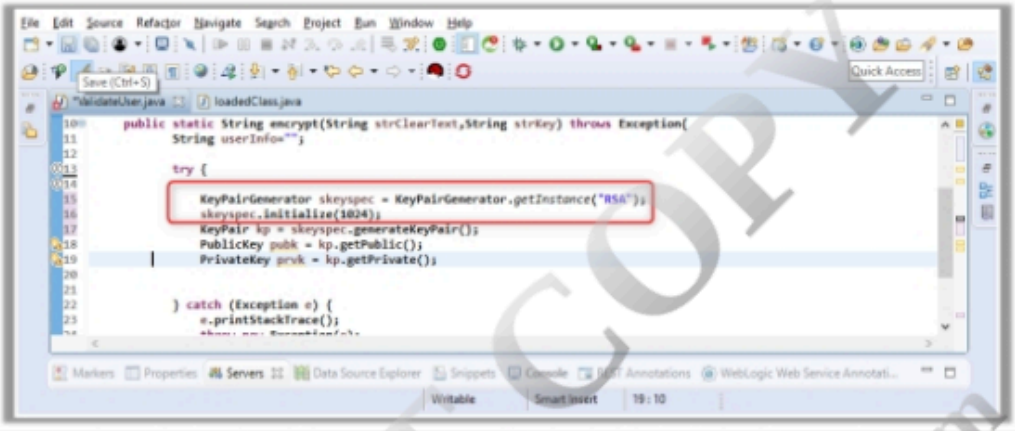
Code Pattern: Look for code where the developer is attempting to set a **weak encryption algorithm**



Recommendation: Always use the **strongest encryption algorithm**

Code Review for Use of Insufficient Encryption Key Size

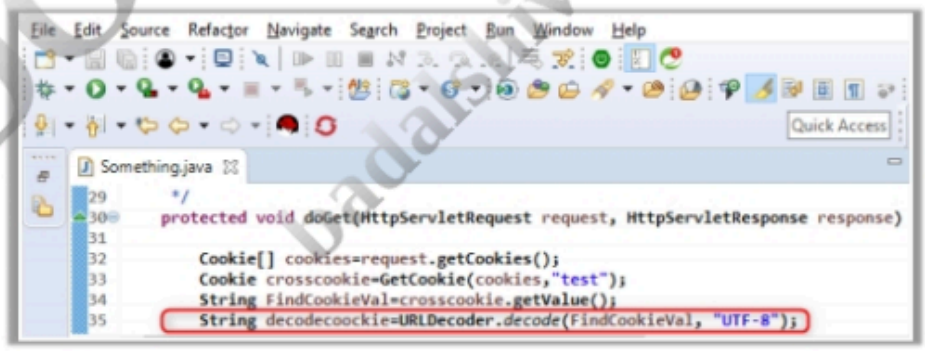
Code Pattern: Look for code where the developer is attempting to set an insufficient encryption **key size**



Recommendation: Always use a secure algorithm with sufficient key size for setting passwords

Code Review for Unsafe Decoding

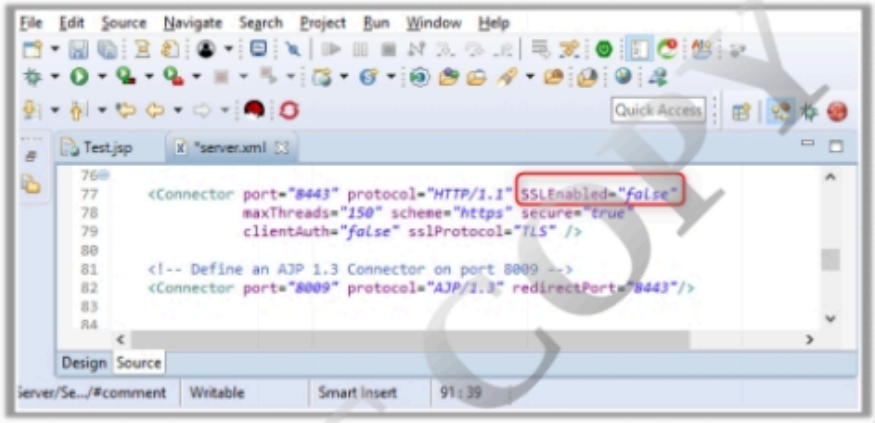
Code Pattern: Look for code where the developer is attempting to **decode** cookie value, if cookies are not safely decoded, attacker can easily abuse the code



Recommendation: Always cross check the **cookie value** before use and confirm whether it is double encoded

Code Review for Use of SSL

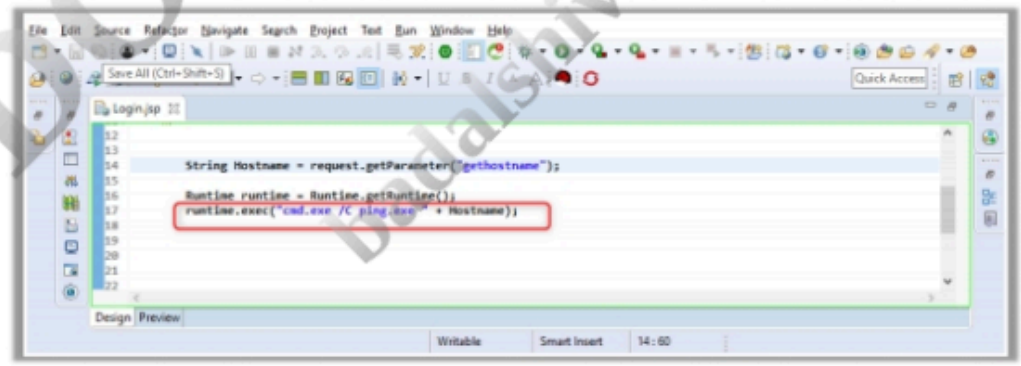
- **Code Pattern:** Look for configuration where the developer is attempting to set a **false** value to the **SSLEnabled** attribute in server.xml file



Recommendation: Always set the **SSLEnabled** attribute to **true**

Code Review for OS Command Injection

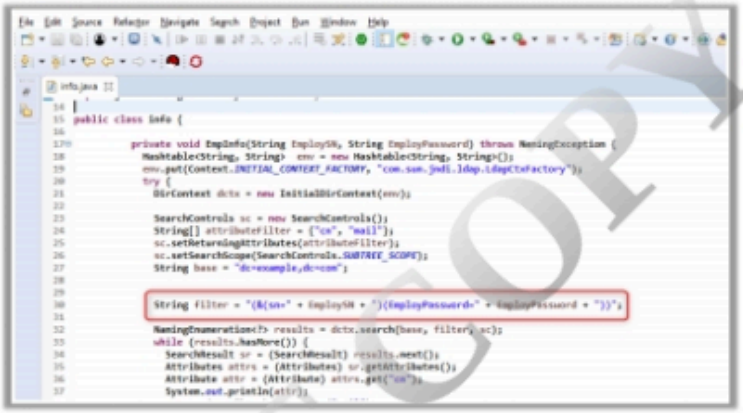
- **Code Pattern:** Look for code where the developer is allowing the **OS command** to execute from the application. Attackers can misuse this functionality and can execute any arbitrary command



Recommendation: Developers should avoid executing OS command using **runtime.exec.** Instead use **Java API's** to execute any OS related command

Code Review for LDAP Injection

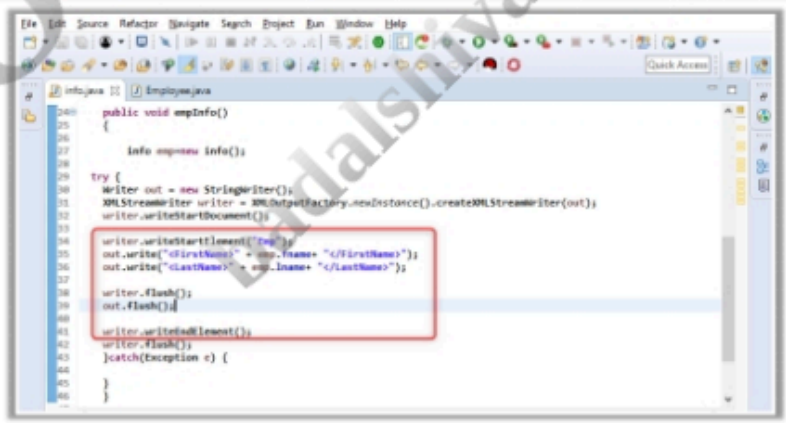
Code Pattern: Look for code where the developer allows the search filter without validation so attackers can input any malicious character and invoke the method. **Example**: EmpInfo()



Recommendation: Always use a **whitelist** to validate user input so the user input contains only valid characters such as passwords containing only alphanumeric characters

Code Review for XML Injection

Code Pattern: Look for code where the developer is attempting to use xml for writing data streams **write()** method which doesn't apply any **meta character encoding**



Recommendation: Always use **meta character encoding** while writing data into xml

Code Review for Unsafe use of request parameter to execute SQL Query

Code Pattern: Look for code where the developer is attempting to execute a query directly received as **request parameter**. Attackers can manipulate these request parameters and execute a malicious query

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<%
   String sql=request.getParameter("query");
   PreparedStatement ps=null,ps1=null;
   Connection con1=null,con=null;
   ResultSet rs=null,rs1=null;
   int maxid=0;
Class.forName("oracle.jdbc.driver.OracleDriver");
        con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","system","admin");
        ps=con.prepareStatement(sql);
        rs=ps.executeQuery();
        while(rs.next())
        {
                maxid=rs.getInt(1)+1;
```

Recommendation: Never use a **request parameter** for executing a query

Code Review for Insecure File Upload

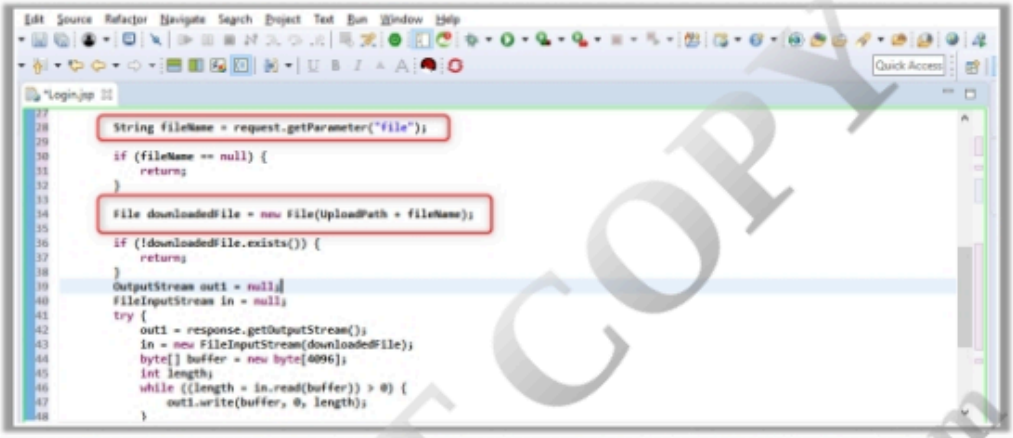
Code Pattern: Look for code where the developer is attempting to upload a **file without proper validation**. An attacker can send any file and execute a command from the file

```
try{
Part filePart = request.getPart("File");
String fileName = filePart.getName();
boolean fileUploaded = false;

if(filePart != null && fileName != null){

InputStream fileContent = filePart.getInputStream();
byte[] bytes = IOUtils.toByteArray(fileContent);


OutputStream out = new FileOutputStream(UploadPath + fileName);

out.write(bytes);
fileUploaded = true;

fileContent.close();
out.flush();
out.close();
}}
```

Recommendation: Always **validate file extensions** while uploading files

Code Review for Directory Traversal

Code Pattern: Look for code where the developer is attempting to read a file if the file exists. Attackers can manipulate path variables and can access **restricted files** and **directories**

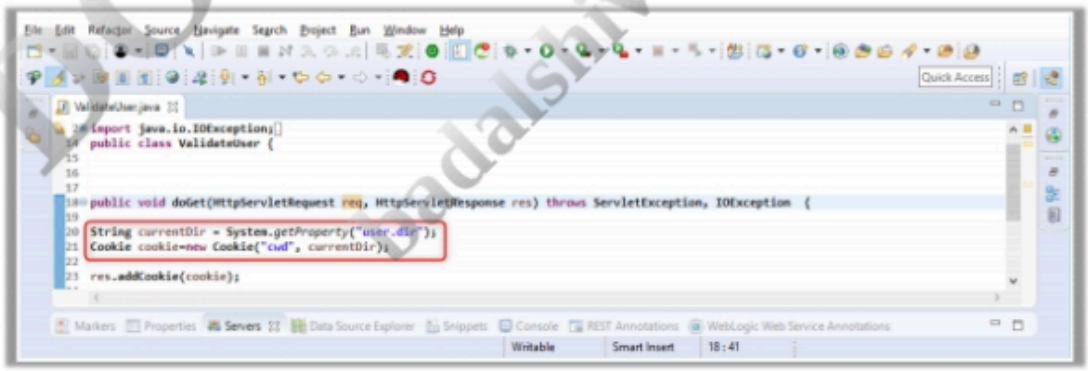


```
String fileName = request.getParameter("file");
if (fileName == null) {
    return;
}
File downloadedFile = new File(UploadPath + fileName);
if (!downloadedFile.exists()) {
    return;
}
OutputStream out1 = null;
FileInputStream in = null;
try {
    out1 = response.getOutputStream();
    in = new FileInputStream(downloadedFile);
    byte[] buffer = new byte[4096];
    int length;
    while ((length = in.read(buffer)) > 0) {
        out1.write(buffer, 0, length);
    }
```

Recommendation: Always **validate** the **location** of a file and read file from that location

Code Review for Sensitive Information Exposure

Code Pattern: Look for code where the developer is attempting to store sensitive information into a **cookie**. A cookie can be read by attackers

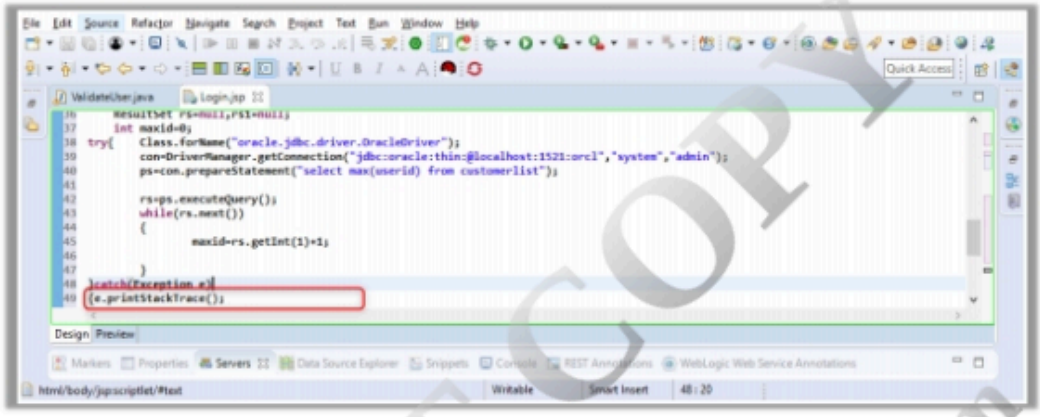


```
import java.io.IOException;
public class ValidateUser {

public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
String currentDir = System.getProperty("user.dir");
Cookie cookie=new Cookie("cwd", currentDir);
res.addCookie(cookie);
```

Recommendation: Do not store **sensitive information** in **cookies**

Code Review for Sensitive Information Leakage

Code Pattern: Look for code where the developer is using **printStackTrace** while handling exceptions. **printStackTrace** gives detailed information about the exception occurred

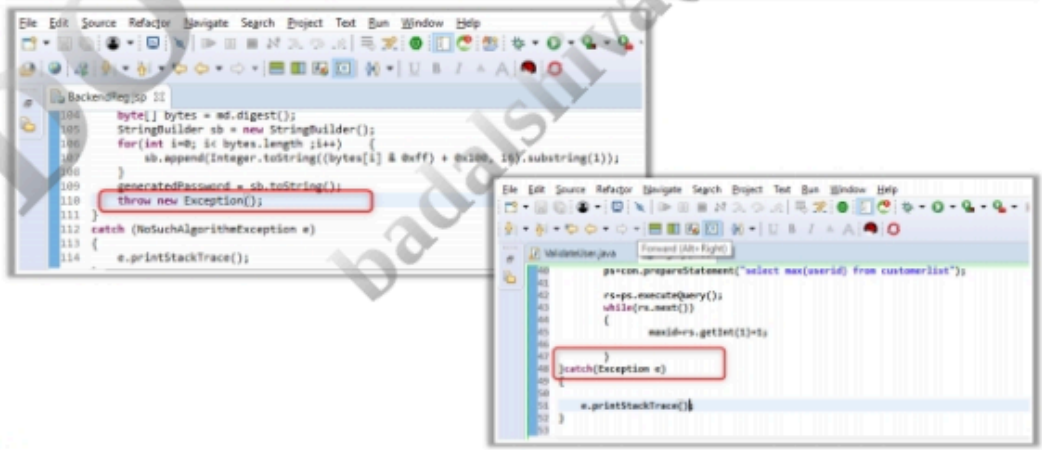


```
ResultSet rs=null,rs1=null;
int maxid=0;
try{    Class.forName("oracle.jdbc.driver.OracleDriver");
        con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","system","admin");
        ps=con.prepareStatement("select max(userid) from customerlist");

        rs=ps.executeQuery();
        while(rs.next())
        {
                maxid=rs.getInt(1)+1;

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

Recommendation: Do not print the **stack trace** when the exception is caught

Code Review for Generic Exception Throwing and Catching

Code Pattern: Look for code where the developer is attempting to **throw** and **catch** an exception with **super class**



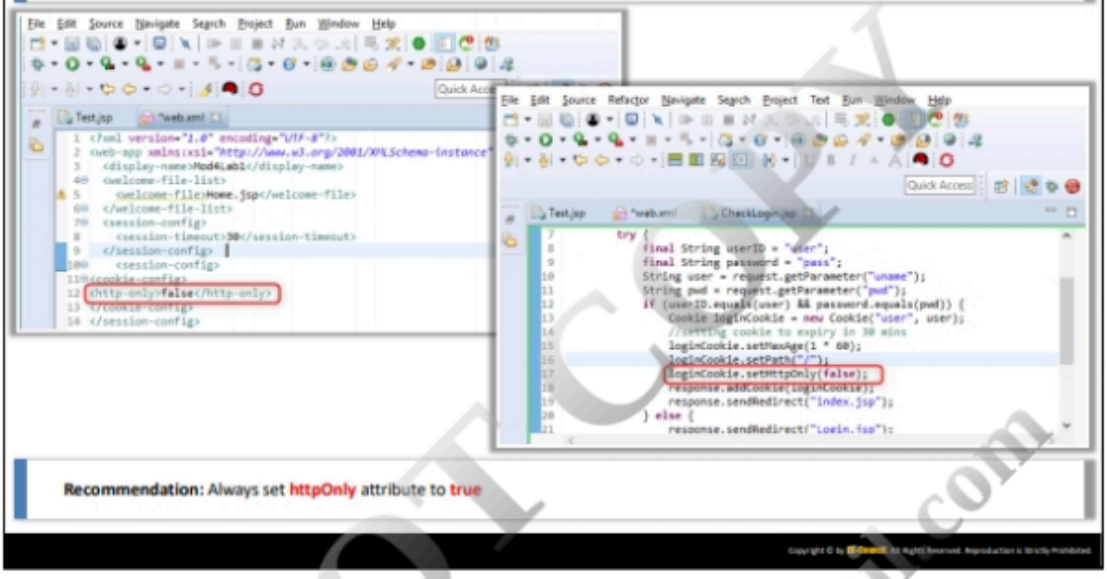
```
byte[] bytes = md.digest();
StringBuilder sb = new StringBuilder();
for(int i=0; i< bytes.length ;i++)    {
    sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
}
generatedPassword = sb.toString();
throw new Exception();
}
catch (NoSuchAlgorithmException e)
{
    e.printStackTrace();
```

```
ps=con.prepareStatement("select max(userid) from customerlist");

rs=ps.executeQuery();
while(rs.next())
{
        maxid=rs.getInt(1)+1;

}
}catch(Exception e)
{

    e.printStackTrace();
}
```

Recommendation : Always **throw** and **catch** exceptions with a specific type of **exception class**

Code Review for Cookies Vulnerable to Client-side Scripts Attacks

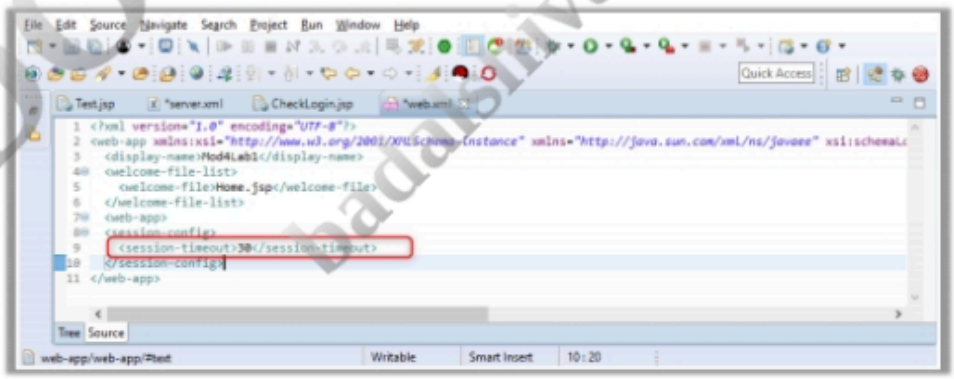
Code Pattern: Look for the configuration where the developer is attempting to set the **httpOnly** attribute to **false**



Recommendation: Always set **httpOnly** attribute to **true**

Code Review for Cookies Vulnerable to CSRF Attacks

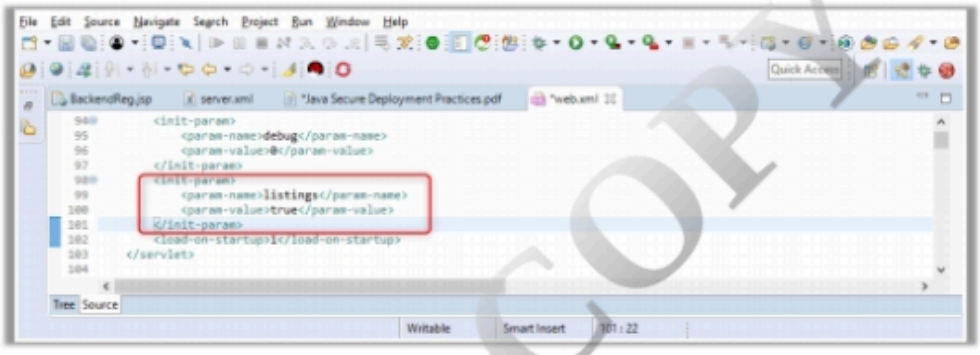
Code Pattern: Look for configuration where the developer is attempting to set the **session-timeout** value to a large value in the **web.xml** file



Recommendation: Always set **session-timeout** value to as minimum as possible

Code Review for Enabling Directory Listing

Code Pattern: Look for configuration where the developer is attempting to set the directory **listing parameter** value to **true** in web.xml file of the tomcat server



Recommendation: Do not enable the directory listing feature. Set **param** value as **"false"** in web.xml

Code Review: Check List Approach

What does a Checklist Contain?

- Data Validation and Encoding Controls
- Encryption Controls
- Authentication and Authorization Controls
- Session Management
- Exception Handling
- Auditing and Logging
- Security Configurations

Sample Checklist: Input Validation

- Ensure that all external inputs are validated
- Ensure that data is validated server-side (security should not rely on client-side validations)
- Ensure that the application validates respective data types and rejects unexpected input
- Ensure that the application efficiently evaluates input length
- Ensure that there is strong separation between data and commands
- Ensure that there is strong separation between data and client side scripts
- Ensure that data is validated for special characters before being passed to SQL, LDAP, OS and third party commands
- Ensure that http headers are validated for each request (e.g. referer)

Sample Checklist: Authentication

Ensure that user credentials are encrypted in the data store
Ensure that security policies are configurable (not hardcoded)
Ensure that standard security frameworks are used (instead of custom code)
Ensure that SSL is used to protect user credentials and authentication tokens
Ensure that authentication cookies are not persisted
Ensure that authentication cookies are encrypted
Ensure that cookie names and paths are used
Ensure that application handles user management events such as authentication failure, password reset, password change, account lockout and cancel account
Ensure that application handles suspicious events such as multiple failed logon attempts, session replay and attempted access to restricted resources
Ensure that strong passwords policies are enforced
Ensure that authentication credentials are not passed by HTTP GET

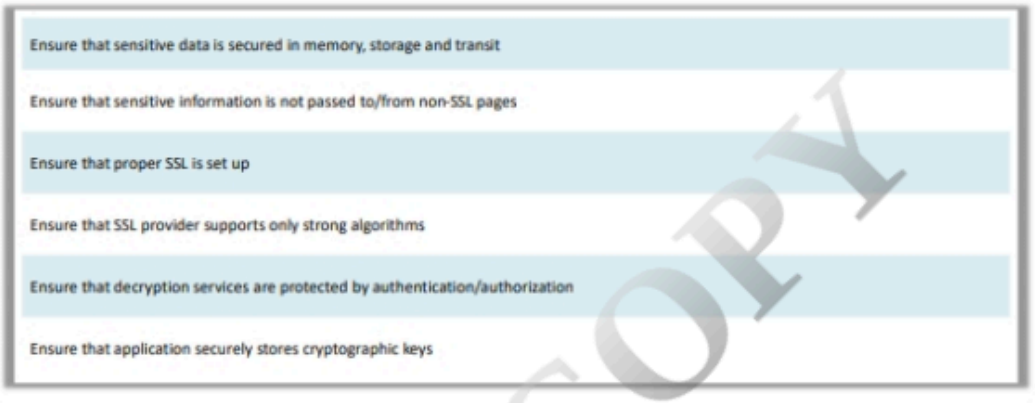
Sample Checklist: Authorization

Ensure that authentication and authorization should be the first logic executed for each request
Ensure that authorization checks are granular (page and directory level)
Ensure that application denies access to pages and data by default
Ensure that application re-authenticates for requests that have side-effects
Ensure that ACLs are configured for all files
Ensure that authorization is performed based on roles defined
Ensure that application implements strong authorization and can not be bypassed through parameter manipulation
Ensure that authorization cannot be bypassed through cookie manipulation

Sample Checklist: Session Management

Ensure that application should not pass session parameters in URLs
Ensure that duration for session cookies expiration should be as minimum as possible
Ensure that cookies are encrypted
Ensure that session data is validated
Ensure that private data in cookies is kept to a minimum
Ensure that application avoids excessive cookie use
Ensure that session id is complex and generated randomly
Ensure that session storage is secure
Ensure that application properly handles invalid session ids
Ensure that session timeout are enforced
Ensure that logout invalids the session
Ensure that session resources are released when session invalidated

Sample Checklist: Cryptography



Sample Checklist: Exception Handling

Sample Checklist: Logging

- Ensure that all user / system actions are logged
- Ensure that sensitive information is not logged (e.g. passwords)
- Ensure that Logging is enabled for user management events (e.g. password reset)
- Ensure that unusual activity such as multiple login attempts are logged
- Ensure that logs have enough detail to reconstruct events for audit purposes
- Ensure that logging is highly configurable (logging levels)

SAST Finding

- The SAST findings keep the development team informed about the findings gathered by the security team during the secure code review
- The findings generally include vulnerabilities, architectural problems, organization issues and any mismatch in the best practices followed
- The security team members may include the following features in the findings report:
  - **Select a meaningful title for the findings report**: The title should be clear and should convey the details of the issue clearly to the development
  - **Determine clearly the location of the issue**: The development team should easily identify the location of the issue from the report
  - **Provide detailed description of the security issues**: The security team should include the following details regarding the security issue:
    - A clear description about the vulnerability
    - Analysis of the possible attack areas
    - Description about the factors leading to the attack
- **Provide a description about the risk ratings**: The risk rating enables the development team regarding the value assigned to each security issue found. The report may also contain the explanation behind assigning any particular value to an identified security issue that may enable the development team to understand the impact of the security issue at a glance. The commonly used risk ratings during code review are as follows:
  - Critical
  - High
  - Moderate
  - Low
- The security team should follow some standards while assigning risk ratings to the security issues during secure code review
- **Provide proper corrective measures to the development team**: The security team after the secure code review may provide certain corrective measures that enable the development team to easily rectify the security issues. The rectification measures may also include:
  - Other course of action
  - Total amount of effort required for rectification

SAST Report

- **The SAST report consists of the following details:**

1. A **list of vulnerabilities** and **its related details** along with the severity level
2. The **rectification measures** taken for controlling the vulnerabilities and errors found in the code
3. **Graphical representation** of the **security measures** required for the application
4. **Control measures** that can be taken for future security as well

SAST Reporting

- Report the findings of the code review in a structured way
- The report should include:
  - **Weakness Metadata**
    - Details of Place to find the issue
  - **Through Description**
    - Detailed description of the issue
  - **Recommendation**
    - Recommendation to mitigate the issue
  - **Assign appropriate Priority**
    - Give priority to the issue for the developer to prepare his to –do-list

SAMPLE REPORT

SQL INJECTION

Location: \myProject\UserModule\User.jsp?username=

Description: The code below builds a dynamic SQL statement without validating the input from request parameter. This may lead to SQL INJECTION

```
String query="select FirstName,LastName from Users
WHERE UserName='" +request.getParameter("username")
+ "'"; PreparedStatement
stmt=con.prepareStatement(query);
ResultSet rs=stmt.executeQuery();
```

Recommendation: Use parameterized SQl query; do not dynamically concatenate the request parameter value to the string query.

Ref:https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

SAST Reporting (Cont'd)

Review /Engagement Reference:			
Package/Component/Class Name/Line Number:			
Finding Title:			
Severity: High			
Finding Description	Location(s)	Risk Description	Recommendation
No input validation of the **HTTPRequest object.getID()** function. Lack of input validation may make the application vulnerable to many types of injection	com.inc.dostuff.java Lines 20, 55,106 com.inc.main.java Lines 34, 99	Discussion of the likelihood and impact to the business if the flaw were to be exploited.	It is critical that this be addressed prior to deployment to production

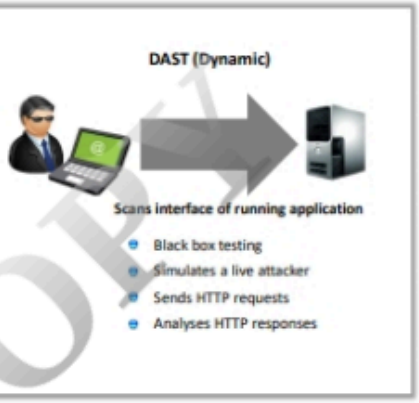
Finding Report Sample

Risk Rating

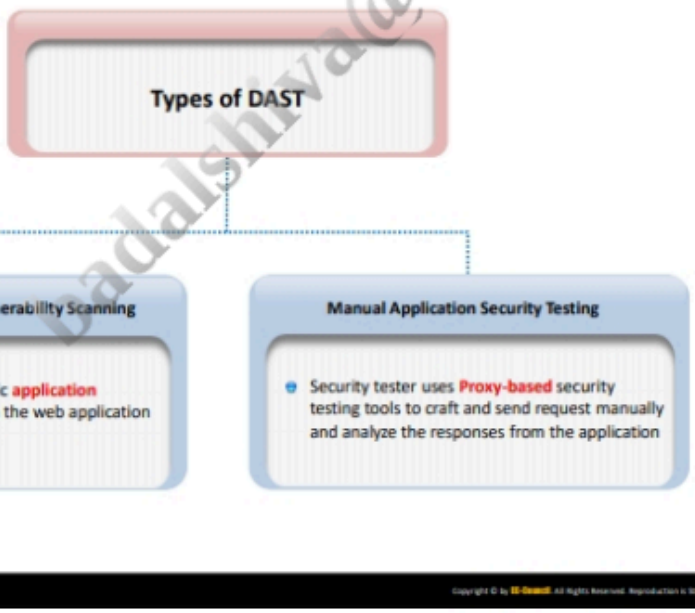
Dynamic Application Security Testing

Dynamic Application Security Testing (DAST)

- DAST is a security testing technique which involves **simulating** attacks against the application and analyzes how the application behaves
- The application is tested **dynamically** from the outside when application is running
- DAST is generally performed by the penetration testers or security practitioners on a working system in pre-production, a test environment, or even in production
- These individuals typically uses automated web application vulnerability scanners to conduct DAST
- These scanners believe in **fuzzing** the application inputs with attack payloads in order to detect security weaknesses
- Typical web application vulnerability scanners are classified into:
  - **Classic application security scanners:** They crawl and automatically scan all detected webpages. However, they have limited coverage within the application as they do not have understanding of business flows
  - **Proxy-based security testing tools:** These are scanners integrated in web application proxies which provide fine-grained control over each request that gets scanned



Types of DAST



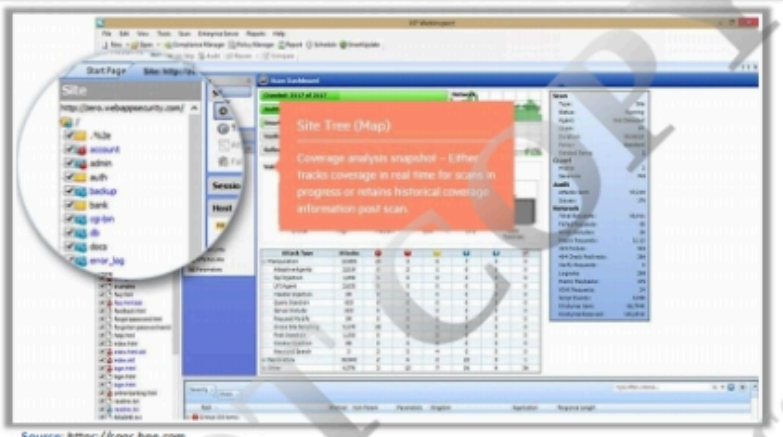
SAST vs DAST

SAST	DAST
White box security testing	Black box security testing
Requires a source code	Requires a running application
Finds vulnerability earlier in SDLC	Finds the vulnerability towards the end of SDLC
Less expensive to fix vulnerability	More expensive to fix vulnerability
Runtime and environment related issues can't be discovered	Runtime and environment related issues can be discovered
Typically supports all kinds of software	Typically scans only apps like web application and web services

Automated Application Vulnerability Scanning Tools

Web Application Security Scanners: WebInspect

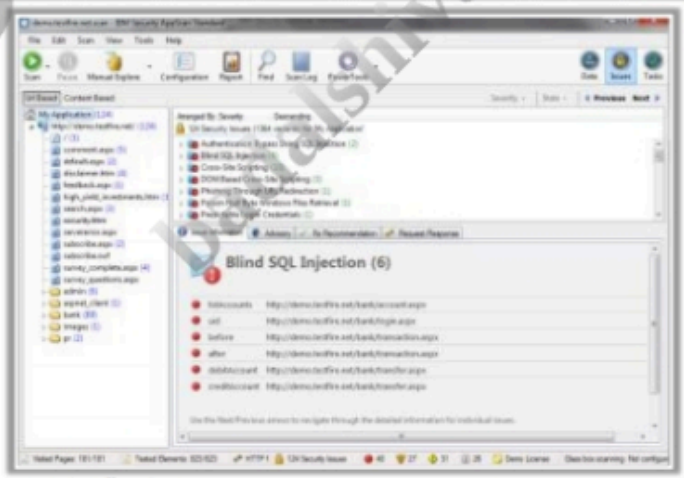
- WebInspect is a web application security assessment solution designed to thoroughly analyze today's complex **web applications** and **web services** for security vulnerabilities
- It provides the broadest dynamic application security testing (DAST) coverage and detects new types of vulnerabilities that often go undetected by **black-box security testing** technologies



Source: https://saas.hpe.com

Web Application Security Scanners: IBM SecurityAppScan

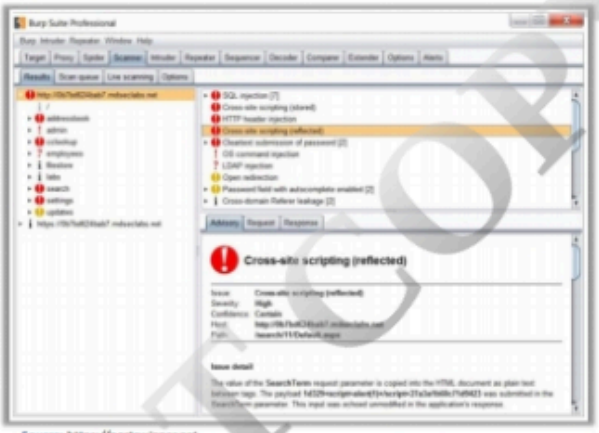
- **IBM** Security **AppScan** helps organizations decrease the likelihood of web application attacks and costly data breaches by automating application security vulnerability testing



Source: https://www.ibm.com

Burp Suite

- Burp Suite is an integrated platform for performing security testing of web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application's attack surface, through to finding and exploiting security vulnerabilities
- Install and configure Burp Suite in the browser of your interest



Source: https://portswigger.net

OWASP Zed Attack Proxy (ZAP)

- OWASP ZAP is an open source, easy to use, integrated penetration testing tool for finding vulnerabilities in web applications

DAST using OWASP ZAP

1. Install and configure ZAP in the browser of your interest
2. Insert the URL of web application that you want to test and start the attack

Source: https://www.owasp.org

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.

Additional Proxy-based Security Testing Tools

Choosing Between SAST and DAST

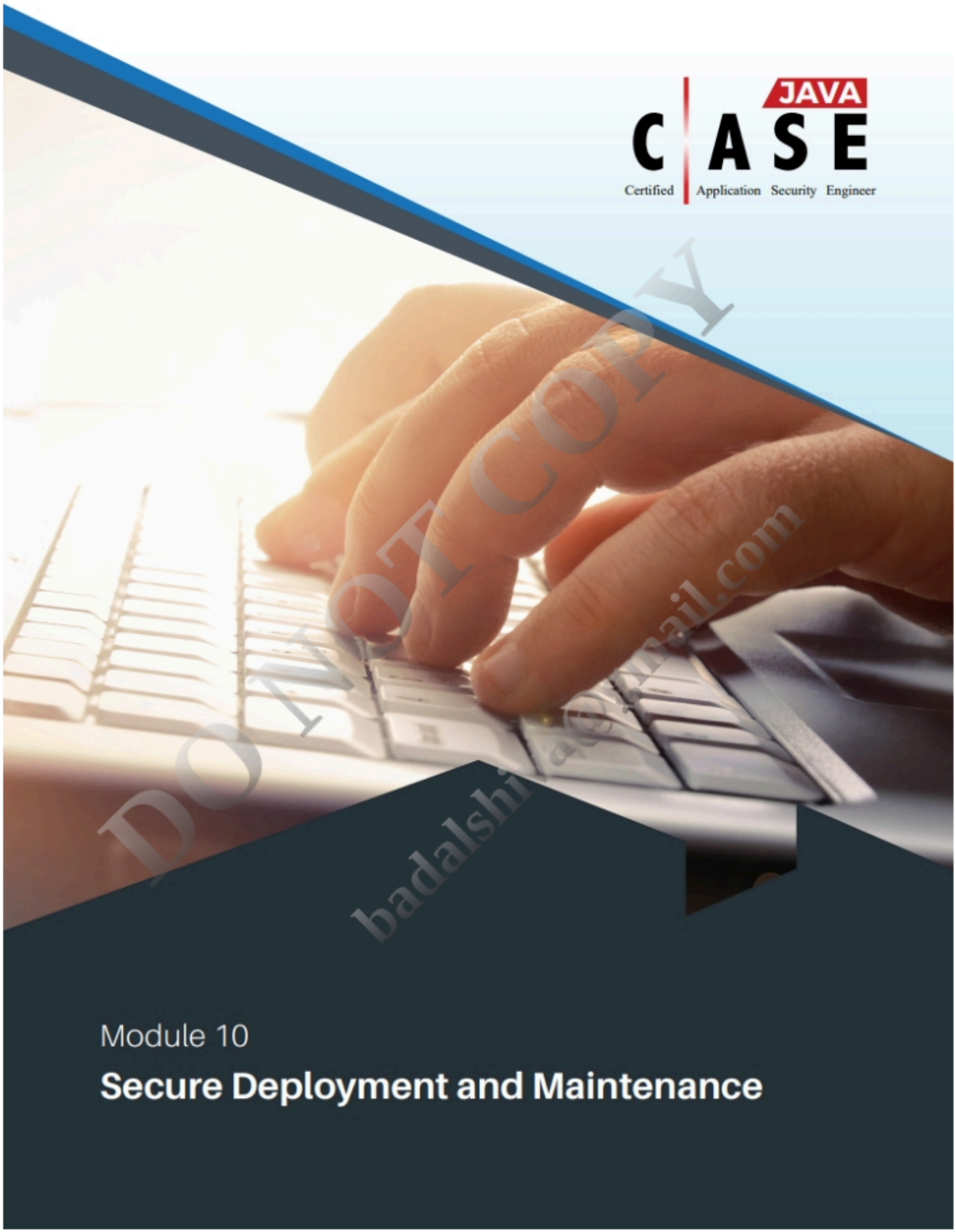
1. SAST is white box security testing approach while DAST is a black box security testing approach
2. These approaches work differently on their part to find security flaws in the application
3. Relying on DAST alone does not ensure that security flaws will exist that you have not tested for
4. Relying on SAST alone does not ensure that security flaw will not exist that may arise when application is running
5. After SAST, security tester may have a better chance of finding different problems when application is running
6. Thus, both SAST and DAST are required in a secure software development program

Module Summary

- SAST is one of the software security assurance approaches executed in an attempt to identify security-related weaknesses in the code
- Manual Source Code Review involves manually inspecting the source code line by line to detect any defects and security related flaws
- The static code analysis tools automate the entire process of secure code review; however, they scan the code superficially
- With manual secure code review, you can analyze and detect many security vulnerabilities especially those missed out during the static code analysis
- DAST is a security testing technique which involves simulating attacks against the application and analyzing how the application behaves
- SAST is a white box security testing approach while DAST is a black box security testing approach

This page is intentionally left blank.

JAVA CASE
Certified Application Security Engineer



Module 10

Secure Deployment and Maintenance

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 the Importance of Secure Deployment
2. Discuss Security Practices at Host Level
3. Discuss Security Practices at Network Level
4. Discuss Security Practices at Application Level
5. Discuss Security Practices at Web Container Level (Tomcat)
6. Discuss Security Practices at Oracle Database Level
7. Discuss Security Maintenance and Monitoring Activities

Secure Deployment

1. Deployment is the **last phase** of software development lifecycle where the application is moved from development environment to production environment
2. It is the first time where the **system administrator** is involved in software development
3. The application can be exposed to different types of attacks, if an administrator commits **configuration mistakes** in the deployment phase
4. **Security** should also be the key consideration while deploying any application

Prior Deployment Activity: Check the Integrity of Application Package before Deployment

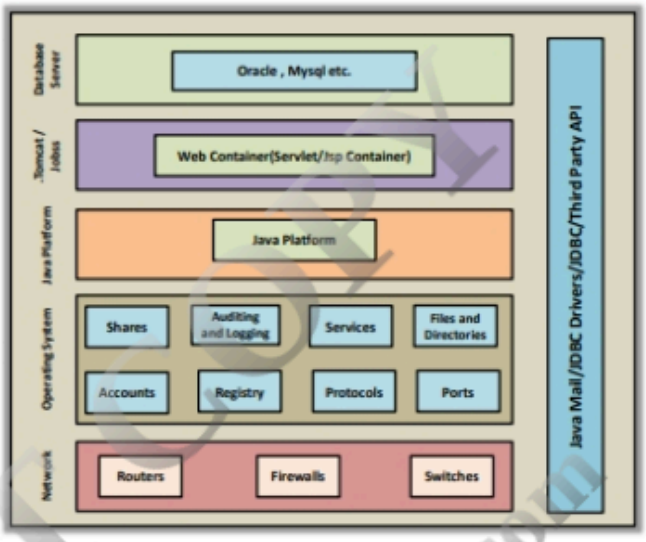
- Software vendors generally provide **checksum** of the application with the application package
- System administrators should again generate a checksum of the application and compare it with the checksum provided with the package to ensure the integrity of the application
- Two **identical checksums** confirm that the application is not tampered during the delivery

Prior Deployment Activity: Review the Deployment Guide Provided by the Software Vendor

- Use **deployment checklists** provided by the software vendor
- **Do not build and use** your own deployment checklist as it may affect the application security

Deployment Activities: Ensuring Security at Various Levels

- Secure deployment involves ensuring security at various levels from bottom to top
- It involves considering security related to the potential customer environments which include:
  - **Host Level** Deployment Security
  - **Container Level (Tomcat)** Deployment Security
  - **Oracle Level** Deployment Security



Ensuring Security at Host Level

Check and Configure the Security of Machine Hosting Web Server, Application Server, Database Server and Network Devices

I. Firstly, the production environment will be secured, if the machine hosting the **web server** is secured

II. The physical security of a host machine, its OS security, and the security of the all other software installed on the machine should be ensured

III. Check the **host server** and all of the services upon which the application security will depend

Physical Security

- Ensure that **server machine is placed** in a secured area
- Ensure that an appropriate physical access control is implemented to restrict unauthorized physical access

Host Level Security

Component	Characteristics
Patches and Updates	Not deploying patches and updates to the web server can allow attackers to run malicious code or intrude the server
Services	Services are easy targets for attackers to access the privilege and exploit the functioning of the server Do not run the service that is not required for the server functioning It is important to monitor running services
Protocols	Do not implement protocols that are insecure and can not provide encrypted authentication Example: Telnet, File Transfer Protocol(FTP), Simple Mail transfer Protocol(SMTP)
Accounts	Audit the user accounts regularly to avoid any malicious activities Provide limited privileges to account to prevent privilege elevation Delete accounts that are no more beneficial for the application Implement strong password policies
Files and Directories	Implement NTFS permissions on files and directories on web server this will allow access to only Windows services
Shares	Secure file shares process with NTFS permissions Limited shares reduces intrusion possibility on web server
Ports	Services running on server listen to specific ports Auditing ports regularly will help to detect any irregular service on the server
Registry	Secure the registry by: Applying restricted Windows ACL Blocking remote registry administration

Ensuring Security at Network Level

Network Level Security: Router

Component	Characteristics
Patches and Updates	Router operating system is patched with up-to-date software
Protocols	Unused protocols and ports are blocked Ingress and egress filtering is implemented ICMP traffic is screened from the internal network TTL expired messages with values of 1 or 0 are blocked (route tracing is disabled) Directed broadcast traffic is not forwarded Large ping packets are screened Routing Information Protocol (RIP) packets, if used, are blocked at the outermost router
Administrative access	Unused management interfaces on the router are disabled A strong administration password policy is enforced Static routing is used Web-facing administration is disabled
Services	Unused services are disabled (for example bootps and finger)
Auditing and logging	Logging is enabled for all denied traffic Logs are centrally stored and secured Auditing against the logs for unusual patterns is in place
Intrusion detection	IDS is in place to identify and notify an active attack

Network Level Security: Firewall

Component	Characteristics
Patches and Updates	Firewall software and OS are patched with latest security updates
Filters	Packet filtering policy blocks all but required traffic in both directions Application-specific filters are in place to restrict unnecessary traffic
Logging and Auditing	All permitted traffic is logged Denied traffic is logged Logs are cycled with a frequency that allows quick data analysis All devices on the network are synchronized to a common time source
Perimeter networks	Perimeter network is in place if multiple networks require access to the server Firewall is placed between untrusted networks

Network Level Security: Switch

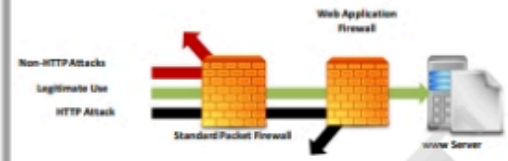
Component	Characteristics
Patches and Updates	Latest security patches are tested and installed, or the threat from known vulnerabilities is mitigated
VLANs	Make sure VLANs are not overused or overly trusted
Insecure defaults	All factory passwords are changed Minimal administrative interfaces are available Access controls are configured to secure SNMP community strings
Services	Unused services are disabled
Encryption	Switched traffic is encrypted

Ensuring Security at Application Level

Web Application Firewall (WAF)

1. A traditional firewall cannot secure webservers from a malicious traffic attack as the attack occurs at layer 7 of the network stack
2. A WAF provides a security layer that protects the web server from the malicious traffic
3. WAF is either appliance based or cloud based and gets deployed through a proxy placed ahead of the web application
4. It uses a rule based filter that monitors and analyzes the traffic before it reaches the web application

Placement of WAF and its Working



Scope of protection in different security products

Benefits of WAF

- WAF implementation secures the existing and productive web applications
- Many WAF's have functionalities that can be used in design process and minimizing the work load
- WAF provides cookies protection with encryption and signature methodology
- It secures applications from cross-site request forgery and negates parameter tampering by URL encryption
- WAF can detect data validation issues by in-depth testing of character, character length, the range of a value, etc.

WAF Limitations

CASE JAVA

- Do not consider WAF as a replacement for a proper **application security** such as user authentication or input filtering
- WAF is not a technology which if once deployed, can be ignored by the administrator
- The working of WAF is different from the **next-generation firewall (NGFW)**. WAF inspects traffic based on a particular protocol unlike NGFW which can make changes in the existing network
- WAF does not provide complete security from all web attacks as it can not read database commands
- Only if WAF manages the session itself, it can partially prevent issues like session fixation and anti-automation
- Deployment of WAF does not ensure security from **false positives**

WAF Vendors

CASE JAVA

Barracuda WAF

Source: http://www.barracuda.com

A10 WAF

Source: https://www.a10networks.com

Citrix NetScaler MPX WAF

Source: https://www.citrix.co.in

Imperva WAF

Source: https://www.imperva.com

MonitorApp WAF

Source: http://www.monitorapp.com

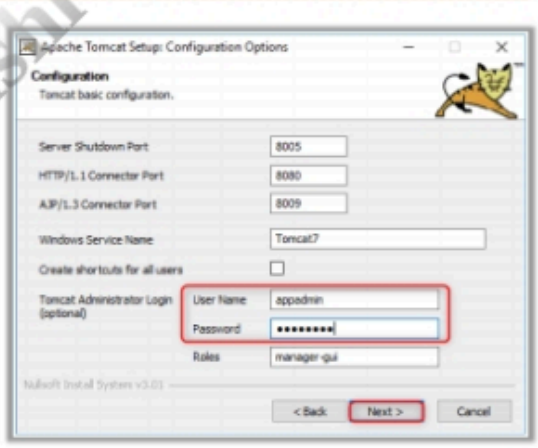
Trustwave WAF

Source: https://www.trustwave.com

Ensuring Security at Web Container Level (Tomcat)

Install and Configure Tomcat Securely

- While installing **Tomcat** uncheck the options **Native, Documentation,** and **Examples** in **Choose Components** window
- Select **User Name** other than **admin**, and give a secure **Password** as per the organization password policy in **Configuration** window



Remove Server Banner

- The **server banner** gives out the product and version details which leads to information disclosure vulnerability
- To prevent Tomcat from displaying product and version details, remove the banner from the **HTTP header**
- To **remove** the **server banner**
  - Open **$Tomcat/conf** folder
  - Edit **server.xml** file
  - Set server ="" in the **Connector port**

  Example: `<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" Server =" " redirectPort="8443" />`

Header Response with Server Banner

▼ General
Request URL: http:/[redacted]/
Request Method: GET
Status Code: 200 OK
Remote Address: [redacted]
Referrer Policy: no-referrer-when-downgrade

▼ Response Headers view source
Content-Type: text/html;charset=ISO-8859-1
Date: Mon, 22 Jan 2018 08:23:06 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

Header Response without Server Banner

▼ General
Request URL: http://[redacted]/
Request Method: GET
Status Code: 200 OK
Remote Address: [redacted]
Referrer Policy: no-referrer-when-downgrade

▼ Response Headers
Content-Type: text/html;charset=ISO-8859-1
Date: Mon, 22 Jan 2018 08:31:44 GMT
Server:
Transfer-Encoding: chunked

Start Tomcat with Security Manager

- Security manager prevents running **untrusted applets**
- It protects browser from executing **servlets, JSPs, JSP beans, and tag libraries** containing trojan
- To run Tomcat with security manager
  - Add the security manager entries to the JAVA_OPTS environment variable
  - Open catalina.policy file located at <Tomcat_Home>/conf/ in a text editor
  - Alter the SecurityManager permission
  - Start Tomcat with security parameter

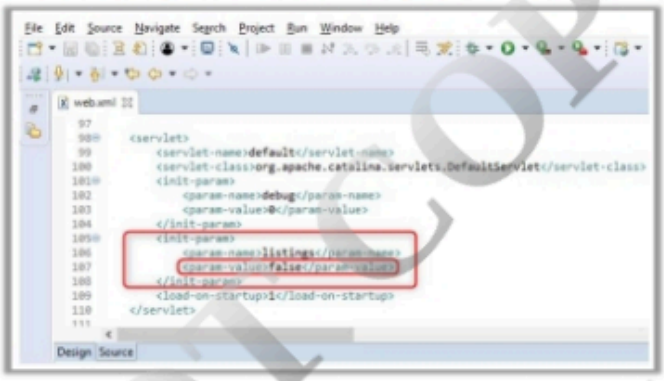
Starting Tomcat with Security Manager

```
$CATALINA_HOME/bin/catalina.sh start -security (Unix)

%CATALINA_HOME%\bin\catalina start -security (Windows)
```

Configure Default Servlet Not to Serve Index Pages

- Ensure that the default servlet configuration settings do not render index pages in the absence of welcome files
- Edit the **listings** to **false** in **CATALINA_HOME/conf/web.xml** file to configure the server not to serve index pages in absence of welcome files

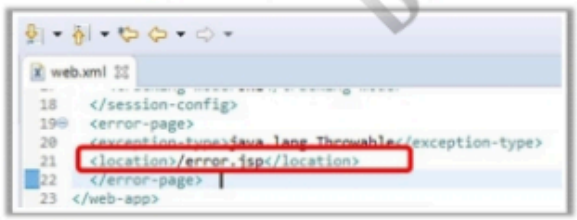
Default Servlet Configuration to not Serve Index Pages in Absence of Welcome Files



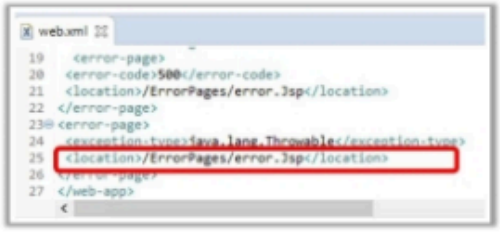
Replace Default Error Page

1. Default error page displays the entire **StackTrace** resulting in sensitive information disclosure
2. Edit **CATALINA_HOME/conf/web.xml** file to replace the default error. This will display a blank page and protects the application from displaying the StackTrace to the user
3. Configure an error page in **CATALINA_HOME/webapps/APP_NAME/WEB-INF/web.xml** to display a custom error page

Configuration to Replace Default Error Page



Configuring Custom Error Page in web.xml



Replace Default server.xml

- Replace the **server.xml** with **server-minimal.xml** which contains the minimum required configuration
- **To replace server.xml**
  - Rename **server.xml** file present in CATALINA_HOME/conf/ to /originalserver.xml
  - Rename **server-minimal.xml** file present in CATALINA_HOME/conf/ to server.xml

Protect Shutdown Port

- Tomcat listens to **8005 port** (default) for **shutdown commands**
- When running a publicly accessible server, it should be ensured that external access to the shutdown port is prevented by using a firewall
- To prevent unintended shutting down of Tomcat services, disable the shutdown port
- To disable the shutdown port
  - Edit the port attribute to -1 in the server.xml file
- Setting **port attribute** to **-1** ensures that Tomcat is shutdown only by the user who owns the Tomcat process
- Alter the shutdown command by changing the **shutdownphrase** in the **CATALINA_HOME/conf/server.xml** file
- Ensure that Tomcat users only have read rights for server.xml file

Disabling Shutdown Port

Altering Shutdown Command

Restrict Access to Tomcat Manager Applications

- Make sure to restrict the IP addresses that can access the Tomcat Manager or administrative web applications
- **To enable connections only from the localhost (127.0.0.1) on the Tomcat server**
  1. Edit context element of ${tomcat_home}/conf/Catalina/localhost/admin.xml
  2. Edit context element of ${tomcat_home}/conf/Catalina/localhost/manager.xml

```
web.xml  server.xml  context.xml
25      -->
26  <Context antiResourceLocking="false" privileged="true">
27  <Valve className="org.apache.catalina.valves.RemoteAddrValve" allow="127\.0\.0\.1"/>
28  </Context>
```

- **To enable connections using host name use RemoteHostValve**

```
context.xml
40 <Context>
41 <Valve className="org.apache.catalina.valves.RemoteHostValve" allow=".*\.ucar\.edu" />
42 </Context>
```

Protecting Resources with Realms

- **Realm** is a database of usernames and passwords for authentication of web applications
- server.xml file has many realms
- Make sure to enable useful realms and others should be disabled
- **LockOutRealm** prevents **brute force attacks** and **denial of service attacks** by locking users, with the number of failed login attempts

Example: server.xml to implement lock out functionality to a UserDatabase Realm

```
server.xml
160 <Realm className="org.apache.catalina.realm.LockOutRealm" >
161 <Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="UserDatabase"/>
162 </Realm>
Design  Source
```