**[CWE-78,](http://cwe.mitre.org/data/definitions/78.html) Improper Neutralization of Special Elements Used in an OS Command ("OS Command Injection")(2 flaws)**

External programs are commonly invoked to perform a function required by the overall system. This practice is a form of reuse and might even be considered a crude form of component-based software engineering. Command and argument injection [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-vulnerab) occur when an application fails to [sanitize](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-sa) untrusted input and uses it in the execution of external programs.

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the Runtime.getRuntime() method. The semantics of Runtime.exec() are poorly defined, so it is best not to rely on its behavior any more than necessary, but typically it invokes the command directly without a shell. If you want a shell, you can use /bin/sh -c on POSIX or cmd.exe on Windows. The variants of exec() that take the command line as a single string split it using a StringTokenizer. On Windows, these tokens are concatenated back into a single argument string before being executed.

Consequently, command injection attacks cannot succeed unless a command interpreter is explicitly invoked. However, argument injection attacks can occur when arguments have spaces, double quotes, and so forth, or when they start with a - or / to indicate a switch

Example

class DirList {

  public static void main(String[] args) throws Exception {

    String dir = System.getProperty("dir");

    Runtime rt = Runtime.getRuntime();

    Process proc = rt.exec("cmd.exe /C dir " + dir);

    int result = proc.waitFor();

    if (result != 0) {

      System.out.println("process error: " + result);

    }

    InputStream in = (result == 0) ? proc.getInputStream() :

                                     proc.getErrorStream();

    int c;

    while ((c = in.read()) != -1) {

      System.out.print((char) c);

    }

  }

}

## Compliant Solution (Sanitization)

This compliant solution [sanitizes](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-sa) the untrusted user input by permitting only a small group of whitelisted characters in the argument that will be passed to Runtime.exec(); all other characters are excluded

// ...

if (!Pattern.matches("[0-9A-Za-z@.]+", dir)) {

  // Handle error

}

// ...

Although it is a compliant solution, this sanitization approach rejects valid directories. Also, because the command interpreter invoked is system dependent, it is difficult to establish that this solution prevents command injections on every platform on which a Java program might run.

Reference URL :

### https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487877

### http://cwe.mitre.org/data/definitions/78.html

**Credentials Management(28 flaws)**

**1)Unprotected Storage of Credentials (CWE ID 256)(2)**

Password management issues occur when a password is stored in plaintext in an application's properties or configuration file. Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource.

Noncompliant Code Example:

Impact of file and Lineno : Corbabridge.java 936

conn = DriverManager.getConnection(url, user, **pwd**);

Compliant Solution:

A cryptographic hash function is a special class of hash function that has certain properties which make it suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (a hash function) which is designed to also be a one-way function, that is, a function which is infeasible to invert

hash(‘sha256’, $password), or even md5($password) to “secure” user passwords. Passwords

conn = DriverManager.getConnection(url, user, **pwdhash**);

Reference URL :

https://cwe.mitre.org/data/definitions/256.html

https://www.owasp.org/index.php/Category:Sensitive\_Data\_Protection\_Vulnerability

**2)Use of Hard-coded Password (CWE ID 259)**

A hard-coded password typically leads to a significant authentication failure that can be difficult for the system administrator to detect. Once detected, it can be difficult to fix, so the administrator may be forced into disabling the product entirely. There are two main variations:

Inbound: the software contains an authentication mechanism that checks for a hard-coded password.

Outbound: the software connects to another system or component, and it contains hard-coded password for connecting to that component.

## Noncompliant Code Example (Hard-Coded Database Password)

The user name and password fields in the SQL connection request are hard coded in this noncompliant code example:

public final Connection getConnection() throws SQLException {

  return DriverManager.getConnection(

      "jdbc:mysql://localhost/dbName",

      "username", "password");

}

## Compliant Solution

This compliant solution reads the user name and password from a configuration file located in a secure directory:

public final Connection getConnection() throws SQLException {

  String username;

  String password;

  // Username and password are read at runtime from a secure config file

  return DriverManager.getConnection(

      "jdbc:mysql://localhost/dbName", username, password);

}

It is also permissible to prompt the user for the user name and password at runtime.

When possible, sensitive information such as passwords should be stored in character arrays rather than strings because the Java Virtual Machine may retain strings long after they are no longer needed. However, this example uses strings because DriverManager.getConnection() requires them.

Reference URL :

<https://cwe.mitre.org/data/definitions/259.html>

https://wiki.sei.cmu.edu/confluence/display/java/MSC03-J.+Never+hard+code+sensitive+information

**Cryptographic Issues(3 flaws)**

**1)Insufficient Entropy (CWE ID 331)(2 flaw)**

Standard pseudo-random number generators cannot withstand cryptographic attacks.

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudo-Random Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts.

## Examples

The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase (DO NOT DO THIS).

String GenerateReceiptURL(String baseUrl) {

Random ranGen = new Random();

ranGen.setSeed((new Date()).getTime());

return(baseUrl + Gen.nextInt(400000000) + ".html");

}

This code uses the Random.nextInt() function to generate "unique" identifiers for the receipt pages it generates. Because Random.nextInt() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

The following code uses Java's SecureRandom class to generate a cryptographically strong pseudo-random number

public static int generateRandom(int maximumValue) {

SecureRandom ranGen = new SecureRandom();

return ranGen.nextInt(maximumValue);

}

ReferenceURL:

https://www.owasp.org/index.php/Insecure\_Randomness

https://cwe.mitre.org/data/definitions/331.html

**2)Use of a Broken or Risky Cryptographic Algorithm (CWE ID 327)(1 flaw)**

The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Well-known techniques may exist to break the algorithm

Using insufficiently strong cryptographic algorithms increases the risk of system hacking and compromising confidential information. We also recommend against using nonstandard encryption methods, since they could be broken with standard attacks.

## Noncompliant Code Example

public byte[] encrypt(String password)

{

try

{

Cipher des = Cipher.getInstance( "DES" );

des.init( Cipher.ENCRYPT\_MODE, key );

return des.doFinal( password.getBytes() );

} catch( Throwable ex )

{

throw new IllegalStateException( ex );

}

}

**Compliant Solution**

// GOOD: AES is a strong algorithm

Cipher des = Cipher.getInstance("AES");

ReferenceURL:

http://cwe.mitre.org/data/definitions/327.html

https://help.semmle.com/wiki/pages/viewpage.action?pageId=1607963

**Encapsulation(154 flaws)**

**1)Deserialization of Untrusted Data (CWE ID 502)(1 flaw)**

Classes that require special handling during object serialization and deserialization must implement special methods with exactly the following signatures

## Noncompliant Code Example (readObject(), writeObject())

This noncompliant code example shows a class Ser with a private constructor, indicating that code external to the class should be unable to create instances of it. The class implements java.io.Serializable and defines public readObject() and writeObject() methods. Consequently, [untrusted code](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-unt) can obtain the reconstituted objects by using readObject() and can write to the stream by using writeObject().

|  |
| --- |
| public class Ser implements Serializable {    private final long serialVersionUID = 123456789;    private Ser() {    // Initialize  }    public static void writeObject(final ObjectOutputStream stream)      throws IOException {      stream.defaultWriteObject();    }    public static void readObject(final ObjectInputStream stream)        throws IOException, ClassNotFoundException {      stream.defaultReadObject();    }  } |

## Compliant Solution (readObject(), writeObject())

This compliant solution declares the readObject() and writeObject() methods private and nonstatic to limit their accessibility:

|  |
| --- |
| private void writeObject(final ObjectOutputStream stream)      throws IOException {    stream.defaultWriteObject();  }   private void readObject(final ObjectInputStream stream)      throws IOException, ClassNotFoundException {    stream.defaultReadObject();  } |

Reducing the accessibility also prevents malicious overriding of the two methods.

Reference URL :

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487769>

<http://cwe.mitre.org/data/definitions/502.html>

**Information Leakage(12 flaws)**

**1)Improper Restriction of XML External Entity Reference ('XXE') (CWE ID 611)(12 flaws)**

XML documents optionally contain a Document Type Definition (DTD), which, among other features, enables the definition of XML entities. It is possible to define an entity by providing a substitution string in the form of a URI. The XML parser can access the contents of this URI and embed these contents back into the XML document for further processing

Since a javax.xml.bind.Unmarshaller parses XML and does not support any flags for disabling XXE, it’s imperative to parse the untrusted XML through a configurable secure parser first, generate a source object as a result, and pass the source object to the Unmarshaller. For example:

**Fix below:**

1. Add below maven dependency

<!-- <https://mvnrepository.com/artifact/org.springframework/spring-oxm> -->

<dependency>

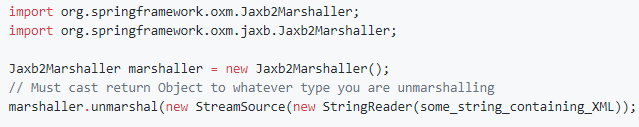
    <groupId>org.springframework</groupId>

    <artifactId>spring-oxm</artifactId>

    <version>5.3.3</version>

</dependency>

1. Use the below code instead of above code



And use StreamSource instead of Source conversion for byteData as below.

Use:  importjavax.xml.transform.stream.StreamSource;

//mResponse = (AttachmentRequestResponse)marshaller.unmarshal((Source) byteData );

mResponse = (AttachmentRequestResponse)marshaller.unmarshal((new StreamSource (byteData) );

**Reference URL :**

https://www.owasp.org/index.php/XML\_External\_Entity\_(XXE)\_Prevention\_Cheat\_Sheet#JAXB\_Unmarshaller

https://cwe.mitre.org/data/definitions/611.html

**2)Trust Boundary Violation (CWE ID 501)(153 flaws)**

A trust boundary can be thought of as line drawn through a program. On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy. The purpose of validation logic is to allow data to safely cross the trust boundary - to move from untrusted to trusted. A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted. By combining trusted and untrusted data in the same data structure, it becomes easier for programmers to mistakenly trust unvalidated data.

Noncompliant Code Example:

Example exsisting code EventPageEx.jsp 754

session.putValue("MRDStatus",MRDStatus);

Compliant Solution:

Fix Solution (valitaion the Parameter value):

if(MRDStatus.matches("[0-9a-zA-Z\_]+") {

session.putValue("MRDStatus",MRDStatus);

}

**Reference URL :**

http://cwe.mitre.org/data/definitions/501.html

**Insufficient Input Validation(703 flaws)**

**1)URL Redirection to Untrusted Site ('OpenRedirect') (CWE ID 601)(695 flaws)**

An http parameter may contain a URL value and could cause the web application to redirect the request to the specified URL. By modifying the URL value to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance.

**Noncompliant Code Example:**

Example exsisting code AccountQuickView.jsp 122

response.sendRedirect("AccountQuickView.jsp?policyID="+search.getPolicyID((context.getPolicyHeaderData().policyNumber).toUpperCase())+"&policyNumber="+context.getPolicyHeaderData().policyNumber);

**Compliant Solution**:

Fix Solution:

ESAPI.httpUtilities().sendRedirect(response,AccountQuickView.jsp?policyID="+search.getPolicyID((context.getPolicyHeaderData().policyNumber).toUpperCase())+"&policyNumber="+context.getPolicyHeaderData().policyNumber);

**Reference URL :**

http://cwe.mitre.org/data/definitions/601.html

**2)Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection') (CWE ID 470)(8 flaws)**

If the application uses external inputs to determine which class to instantiate or which method to invoke, then an attacker could supply values to select unexpected classes or methods. If this occurs, then the attacker could create control flow paths that were not intended by the developer. These paths could bypass authentication or access control checks, or otherwise cause the application to behave in an unexpected manner. This situation becomes a doomsday scenario if the attacker can upload files into a location that appears on the application's classpath ([CWE-427](http://cwe.mitre.org/data/definitions/427.html)) or add new entries to the application's classpath ([CWE-426](http://cwe.mitre.org/data/definitions/426.html)). Under either of these conditions, the attacker can use reflection to introduce new, malicious behavior into the application.

## Noncompliant Code Example

public static Class<?> findClass(String name)

   throws ClassNotFoundException {

  try {

    ClassLoader loader = Thread.currentThread().getContextClassLoader();

    if (loader == null) {

      loader = ClassLoader.getSystemClassLoader();

    }

    if (loader != null) {

      return Class.forName(name, false, loader);

    }

  } catch (ClassNotFoundException exception) {

    // Use current class loader instead

  } catch (SecurityException exception) {

    // Use current class loader instead

  }

  return Class.forName(name);

}

## Compliant Solution

Oracle mitigated this vulnerability in Java 1.7.0 update 7 by patching the com.sun.beans.finder.ClassFinder.findClass() method. The checkPackageAccess() method checks the entire call stack to ensure that Class.forName(), in this instance only, fetches classes only on behalf of trusted methods.

|  |
| --- |
| public static Class<?> findClass(String name)     throws ClassNotFoundException {    checkPackageAccess(name);    try {      ClassLoader loader = Thread.currentThread().getContextClassLoader();      if (loader == null) {        // Can be null in IE (see 6204697)        loader = ClassLoader.getSystemClassLoader();      }      if (loader != null) {        return Class.forName(name, false, loader);      }      } catch (ClassNotFoundException exception) {      // Use current class loader instead    } catch (SecurityException exception) {      // Use current class loader instead    }    return Class.forName(name);  } |

**Reference URL :**

**https://wiki.sei.cmu.edu/confluence/display/java/SEC52-J.+Do+not+expose+methods+that+use+reduced-security+checks+to+untrusted+code**

https://wiki.sei.cmu.edu/confluence/display/java/SEC52-J.+Do+not+expose+methods+that+use+reduced-security+checks+to+untrusted+code

**Race Conditions(8 flaws)**

**1)Race Condition within a Thread (CWE ID 366)(8 flaws)**

If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.

public classRace {

static int foo = 0;  
public static void main() {

new Threader().start();  
foo = 1;

}  
public static class Threader extends Thread {

public void run() {

System.out.println(foo);

}

}

}

**Reference URL :**

https://cwe.mitre.org/data/definitions/366.html

**Information Leakage(12 flaws)**

**API Abuse(21 flaws)**

**1)J2EE Bad Practices: Direct Management of Connections (CWE ID 245)(21 flaws)**

The J2EE standard forbids the direct management of connections. It requires that applications use the container's resource management facilities to obtain connections to resources. Every major web application container provides pooled database connection management as part of its resource management framework. Duplicating this functionality in an application is difficult and error prone, which is part of the reason it is forbidden under the J2EE standard.

## Noncompliant Code Example

dbConnection = DriverManager.getConnection("jdbc:oracle:thin:@165.30.130.106:1521:hor2","dbo","dbo");

## Compliant Solution

InitialContext ctx = new InitialContext();

DataSource datasource = (DataSource) ctx.lookup("..");

Connection conn = datasource.getConnection();

**Reference URL :**

https://cwe.mitre.org/data/definitions/245.html

**Code Quality(177 flaws)**

**1)Improper Resource Shutdown or Release (CWE ID 404)(1 flaw)**

When a resource is created or allocated, the developer is responsible for properly releasing the resource as well as accounting for all potential paths of expiration or invalidation, such as a set period of time or revocation

## Noncompliant Code Example

This noncompliant code example attempts to address exhaustion of database connections by adding cleanup code in a finally block. However, rs, stmt, or conn could be null, causing the code in the finally block to throw a NullPointerException.

Statement stmt = null;

ResultSet rs = null;

Connection conn = getConnection();

try {

  stmt = conn.createStatement();

  rs = stmt.executeQuery(sqlQuery);

  processResults(rs);

} catch (SQLException e) {

  // Forward to handler

} finally {

  if (rs != null) {

    rs.close();

  }

  if (stmt != null) {

    stmt.close();

  } if (conn !=null) {

       conn.close();

    }

}

## Compliant Solution

This compliant solution ensures that resources are released as required:

Statement stmt = null;

ResultSet rs = null;

Connection conn = getConnection();

try {

  stmt = conn.createStatement();

  rs = stmt.executeQuery(sqlQuery);

  processResults(rs);

} catch (SQLException e) {

  // Forward to handler

} finally {

  try {

    if (rs != null) {rs.close();}

  } catch (SQLException e) {

    // Forward to handler

  } finally {

    try {

      if (stmt != null) {stmt.close();}

    } catch (SQLException e) {

      // Forward to handler

    } finally {

      try {

        if (conn != null) {conn.close();}

      } catch (SQLException e) {

        // Forward to handler

      }

    }

  }

}

Reference URL :

https://cwe.mitre.org/data/definitions/404.html

https://wiki.sei.cmu.edu/confluence/display/java/FIO04-J.+Release+resources+when+they+are+no+longer+needed

**2)Use of Wrong Operator in String Comparison(CWE ID 597)(176 flaws)**

In Java, using == or != to compare two strings for equality actually compares two objects for equality, not their values. Chances are good that the two references will never be equal. While this weakness often only affects program correctness, if the equality is used for a security decision, it could be leveraged to affect program security.

**Noncompliant Code Example:**

Existing code TransactionValidate.jsp (line 513)

if (("SURR".equals(TT) && txnAmount != ""))

**Compliant Solution:**

if (("SURR".equals(TT) && !txnAmount.equals("")))

**Reference URL :**

https://cwe.mitre.org/data/definitions/597.html

**Cryptographic Issues(30 flaws)**

1**)Sensitive Cookie in HTTPS Session Without'Secure' Attribute (CWE ID 614)(30 flaws)**

The Secure attribute for sensitive cookies in HTTPS sessions is not set, which could cause the user agent to send those cookies in plaintext over an HTTP session.

**Noncompliant Code Example:**

The snippet of code below, taken from a servlet doPost() method, sets an accountID cookie (sensitive) without calling setSecure(true).

Cookie c = new Cookie(ACCOUNT\_ID, acctID);  
response.addCookie(c);

**Compliant Solution:**

Cookie c = new Cookie(ACCOUNT\_ID, acctID);

c.setSecure(true);  
response.addCookie(c);

**Reference URL :**

https://cwe.mitre.org/data/definitions/614.html

**Information Leakage(925 flaws)**

**1)Information Exposure Through Sent Data (CWE ID 201)(190 flaws)**

The accidental exposure of sensitive information through sent data refers to the transmission of data which are either sensitive in and of itself or useful in the further exploitation of the system through standard data channels

**Noncompliant Code** Example: Account72TQValues.jsp 520

<%=sWarning%><span class="EMessage"><%=sErrorMessage%></span>

**Compliant Solution**

<%= ESAPI.encoder().encodeForHTML(sWarning)%><span class="EMessage"><%= **ESAPI.encoder().encodeForHTM**L(sErrorMessage)%></span>

Reference URL :

https://cwe.mitre.org/data/definitions/201.html

https://www.codota.com/code/java/methods/org.owasp.encoder.Encode/forHtml

2)**Information Exposure Through an ErrorMessage (CWE ID 209)(735 flaws)**

The sensitive information may be valuable information on its own (such as a password), or it may be useful for launching other, more deadly attacks. If an attack fails, an attacker may use error information provided by the server to launch another more focused attack. For example, an attempt to exploit a path traversal weakness ([CWE-22](https://cwe.mitre.org/data/definitions/22.html)) might yield the full pathname of the installed application. In turn, this could be used to select the proper number of ".." sequences to navigate to the targeted file. An attack using SQL injection ([CWE-89](https://cwe.mitre.org/data/definitions/89.html)) might not initially succeed, but an error message could reveal the malformed query, which would expose query logic and possibly even passwords or other sensitive information used within the query.

**Noncompliant Code Example:**

public BankAccount getUserBankAccount(String username, String accountNumber) {

BankAccount userAccount = null;  
String query = null;  
try {

if (isAuthorizedUser(username)) {

query = "SELECT \* FROM accounts WHERE owner = "  
+ username + " AND accountID = " + accountNumber;  
DatabaseManager dbManager = new DatabaseManager();  
Connection conn = dbManager.getConnection();  
Statement stmt = conn.createStatement();  
ResultSet queryResult = stmt.executeQuery(query);  
userAccount = (BankAccount)queryResult.getObject(accountNumber);

}

} catch (SQLException ex) {

String logMessage = "Unable to retrieve account information from database,\nquery: " + query;  
Logger.getLogger(BankManager.class.getName()).log(Level.SEVERE, logMessage, ex);

}

**Compliant Solution**

} catch (SQLException ex) {

String logMessage = "Unable to retrieve account information from database,\nquery: " + query;  
Logger.getLogger(BankManager.class.getName()).log(Level.SEVERE, ESAPI.encoder().encodeForHTML( logMessage)), ex);

}

**Reference URL :**

https://cwe.mitre.org/data/definitions/209.html

https://www.codota.com/code/java/methods/org.owasp.encoder.Encode/forHtml

**Time and State(1 flaw)**

1)J2EE Bad Practices: Use of System.exit()(CWE ID 382)(1 flaw)

It is never a good idea for a web application to attempt to shut down the application container. Access to a function that can shut down the application is an avenue for Denial of Service (DoS) attacks.

**Noncompliant Code Example**

This noncompliant code example uses System.exit() to forcefully shut down the JVM and terminate the running process. The program lacks a security manager; consequently, it lacks the capability to check whether the caller is permitted to invoke System.exit().

public class InterceptExit {

public static void main(String[] args) {

// ...

System.exit(1); // Abrupt exit

System.out.println("This never executes");

}

}

**Compliant Solution**

This compliant solution installs a custom security manager PasswordSecurityManager that overrides the checkExit() method defined in the SecurityManager class. This override is required to enable invocation of cleanup code before allowing the exit. The default checkExit() method in the SecurityManager class lacks this facility.

class PasswordSecurityManager extends SecurityManager {

private boolean isExitAllowedFlag;

public PasswordSecurityManager(){

super();

isExitAllowedFlag = false;

}

public boolean isExitAllowed(){

return isExitAllowedFlag;

}

@Override

public void checkExit(int status) {

if (!isExitAllowed()) {

throw new SecurityException();

}

super.checkExit(status);

}

public void setExitAllowed(boolean f) {

isExitAllowedFlag = f;

}

}

public class InterceptExit {

public static void main(String[] args) {

PasswordSecurityManager secManager =

new PasswordSecurityManager();

System.setSecurityManager(secManager);

try {

// ...

System.exit(1); // Abrupt exit call

} catch (Throwable x) {

if (x instanceof SecurityException) {

System.out.println("Intercepted System.exit()");

// Log exception

} else {

// Forward to exception handler

}

}

// ...

secManager.setExitAllowed(true); // Permit exit

// System.exit() will work subsequently

// ...

}

}

This implementation uses an internal flag to track whether the exit is permitted. The method setExitAllowed() sets this flag. The checkExit() method throws a SecurityException when the flag is unset (that is, false). Because this flag is not initially set, normal exception processing bypasses the initial call to System.exit(). The program catches the SecurityException and performs mandatory cleanup operations, including logging the exception. The System.exit() method is enabled only after cleanup is complete.

**Reference URL :**

https://wiki.sei.cmu.edu/confluence/display/java/ERR09-J.+Do+not+allow+untrusted+code+to+terminate+the+JVM

http://cwe.mitre.org/data/definitions/382.html

**Improper Output Neutralization for Logs (CWE ID 117)**

**Compliant Solution**

private final Logger logger = ESAPI.getLogger("SecurityFIlter");

if(!skipHeaderValidation){

String maliciousHeader = validateHeaders(request);

isValidHeader = ( maliciousHeader == null || "".equalsIgnoreCase(maliciousHeader.trim()))?true:false;

if(!isValidHeader){

if(null != request.getSession(false)) request.getSession(false).invalidate();

logger.error(Logger.EVENT\_FAILURE, "header "+ESAPI.encoder().encodeForHTML(maliciousHeader)+" = "+ESAPI.encoder().encodeForHTML(request.getHeader(maliciousHeader))+" is not valid");

printRequest(httpUtilities, request);

response.sendError(HttpServletResponse.SC\_FORBIDDEN,"Parameter "+ESAPI.encoder().encodeForHTML(maliciousHeader)+" or Value "+ESAPI.encoder().encodeForHTML(request.getHeader(maliciousHeader))+" is not valid");

return;

}

}

boolean isValidUrlAndUri = validateURLandURI(request);

if (!isValidUrlAndUri) {

logger.error(Logger.EVENT\_FAILURE, "URL/URI with Value: " + ESAPI.encoder().encodeForHTML(request.getRequestURL().toString())+" is not valid");

printRequest(httpUtilities, request);

response.sendError(HttpServletResponse.SC\_FORBIDDEN, "URL/URI with Value: " + ESAPI.encoder().encodeForHTML(request.getRequestURL().toString())+" is not valid");

return;

Reference: <https://cwe.mitre.org/data/definitions/117.html>

**Solution 2: for CWE ID 117 : Fix for Spring Boot Application.**

1. Add the next code to my pom.xml

<dependency>

<groupId>org.owasp.esapi</groupId>

<artifactId>esapi</artifactId>

<version>2.2.0.0</version>

</dependency>

1. Add ESAPI.properties and validation.properties files to src/main/resources/.
2. Update the property ESAPI.Logger in the file ESAPI.properties with below attributes

ESAPI.Logger=org.owasp.esapi.logging.slf4j.Slf4JLogFactory

Logger.UserInfo=false

Logger.ClientInfo=false

Logger.LogEncodingRequired=false

4. Update the application.properties with the below

logging.file=C:\\app\\Log\\policyadministrationinquiry.log

logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG

#logging.path=C:\\logs

logging.level.org.springframework.web=DEBUG

logging.level.org.springframework.jdbc=TRACE

logging.level.org.springframework.ws=TRACE

#logging.level.org.hibernate.SQL=DEBUG

logging.level.com.zaxxer.hikari.HikariConfig=DEBUG

logging.level.com.aig=DEBUG

**SQL Injection (CWE ID 89)**

**Compliant Solution**

**Cause:**

String sqlLipc = "SELECT \* FROM t\_lipc\_policy\_common WHERE pc\_cont="+pc\_cont+"";

**Normal case**

we use to pass parameter in url as <http://dwsas....:8080/check_policy?pc_cont>= **A24D500000**

**Result:** SELECT \* FROM t\_lipc\_policy\_common WHERE pc\_cont=**A24D500000**

**Vulnerable Case:**

Attacker could change the parameter pc\_cont into [http://dwsas....:8080/check\_policy?pc\_cont=**A24D500000%20OR%201%3D1**](http://dwsas....:8080/check_policy?pc_cont=A24D500000%20OR%201%3D1)

**This results in:** SELECT \* FROM t\_lipc\_policy\_common WHERE pc\_cont=**A24D500000 OR 1=1**

When this query is passed to the database, it will return all the policy details.

Fix :
We can fix this flaw by using a Prepared Statement to create a Parameterized Query as below:

Query change: String sqlLipc =“SELECT * FROM t_lipc_policy_common WHERE pc_cont=?”
-    Statement statement = connection.createStatement();
-    ResultSet rs = statement.executeQuery(sqlLipc);
+    PreparedStatement statement = connection.prepareStatement(sqlLipc);
+    statement.setString(1, request.getParameter("pc_cont"));
If an attacker attempts to supply a value that’s not a simple integer, then statement.setString() will throw a SQLException rather than permitting the query to complete.


**Inclusion of Functionality from Untrusted Control Sphere (CWE ID 829)**

**Compliant Solution**

This can be done by MW team in WAS configuration since they only have access privilege.

**Fix:**

Security can be reinforced by adding these HTTP headers to the responses sent by HIS (IBM HTTP Server).

For that edit the IHS configuration file (httpd.conf), make sure that the mod\_headers is enabled / loaded.

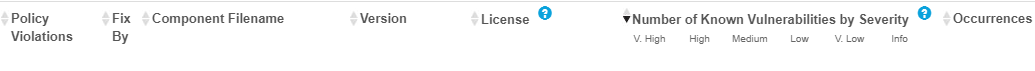
The following line should not be commented (or should be added if not existing)

1. LoadModule headers\_module modules/mod\_headers.so
2. Then use / add directives similar to the ones below :

Header set X-Content-Type-Options "nosniff" Header set X-XSS-Protection "1; mode=block" Header set Content-Security-Policy "default-src 'self';"

**Multiple jars vulnerability fixes below:**

**Jar issue1:**



**Fix:**

**Use the below dependency in pom.xml instead of the old version (1.6)**

|  |
| --- |
| <dependency> |
|  | <groupId>commons-beanutils</groupId> |
|  | <artifactId>commons-beanutils</artifactId> |
|  | <version>1.9.4</version> |
|  | <exclusions> |
|  | <exclusion> |
|  | <groupId>commons-collections</groupId> |
|  | <artifactId>commons-collections</artifactId> |
|  | </exclusion> |
|  | </exclusions> |
|  | </dependency> |