Security Code Review- Identifying Web Vulnerabilities

Kiran Maraju, CISSP, CEH, ITIL, SCJP

Email: Kiran_maraju@yahoo.com

Security Code Review- Identifying Web Vulnerabilities

1.1.1 Abstract

This paper gives an introduction of security code review inspections, and provides details about web application security vulnerabilities identification in the source code. This paper gives the details of the inspections to perform on the Java/J2EE source code. This paper explains the process of identifying vulnerable code and remediation details. This paper illustrates the specific locations of code flows to be checked to identify web application vulnerabilities.

1.1.2 Introduction

Security code review is to do code inspection to identify vulnerabilities in the code. Vulnerabilities in the code exist due to the improper design or implementation in SDLC Process life cycle while developing the application.

JAVA/J2EE Security Code Review Inspection List		
Input Validation Inspection	Unreleased resource-Possible DOS inspection	Memory Reference Inspection
J2EE Configuration	Resource injection	Database access Inspection
Access Control – Authentication Inspection	Logging level inspection	Computation/Numeric/Control Flow Inspection
Access Control-Authorization Inspection	Configuration level inspection	Network communication Inspection
Session Implementation inspection	Sensitive data inspection	Web Container based Inspection
Class level inspection	Exception Management inspection	I/O (Input-Output) Inspection
Object level inspection	Serialization inspection	File Level Inspection
Method level Inspection	Synchronization/Thread level inspection	Native method Inspection
Variable/constant level inspection	Cryptography Inspection	Packaging level Inspection
Inheritance Inspection	EJB(Bean component Inspection)	Inner classes/ reflection inspection

1.1.3 Input Validation Flaws

Input data requested from the client to server is not validated before being used by a web application. Attackers use the unvalidated input flaws to attack web server and backend components. Client-side validation like javascript and vbscript can be easily bypassed. Attackers use the vulnerable unvalidated input calls to be processed by server to execute the malicious content. Security code review involves identifying such unvalidated input calls and prevents those calls to be processed by the server. The following security code review checks are to be performed on tainted calls of servlet to identify unvalidated input from the attacker.

- Checking for Unvalidated sources of input from URL parameters in javax.servlet.HttpServletRequest class getParameter () method

- Checking for Unvalidated sources of input from Form fields in javax.servlet.HttpServletRequest class getQueryString () method
- Checking for Unvalidated sources of input from Cookies javax.servlet.HttpServletRequest class getCookies () method
- Checking for Unvalidated sources of input from HTTP headers javax.servlet.HttpServletRequest class getHeaders () method

Through the application, an attacker enters malicious data through HTML form of the input fields of the application. The following code is vulnerable because the application accepts the values provided by the client in an iterator and responds back to client with the attacker's script which leads to reflected XSS vulnerability when the request.getParameteNames method was called. The following sample code illustrates the unvalidated input vulnerability.

```
Iterator iter = request.getParameterNames ();
While (iter.hasNext ()) {
String paramName = (String) iter.next ();
out.println (paramName +request.getParameter (paramName));
    }
```

Remediation:
Unvalidated input vulnerabilities can be prevented either by accepting the valid data and rejecting every thing else such as creating whitelist of the legitimate input characters or cleaning the input or reject the entire input.

Rejecting the input:
The following code is used as whitelist input validation by accepting specific character patterns and rejecting everything else as a part of servlet request call

```
String input = request.getParameter ("input");
 String characterPattern = "/ [^A-z]/";
If (! input. matches (characterPattern))
{
  out.println ("Invalid Input");
}
```

Cleaning the input:
The following code is used as input validation by accepting the input and cleaning the malicious charcters by replacing with a space.

```
String filterPattern="[<> {}\\ [\\]; \\&]";
String inputStr = s.replaceAll (filterPattern," ");
```

1.1.4 Cross-Site Scripting Vulnerability

Cross-site scripting is an attack against web applications. It occurs when an web application accepts input containing malicious script through the application. This input is then sent as part of the response to the client as a reflected script which can lead to phishing attack, and the malicious script can be injected in the database as stored form so that whenever the page is accessed the injected script is executed like<Script> alert (document. cookie)</script>

The following sample code illustrates the XSS vulnerability.

```
Iterator iter = request.getParameterNames ();
While (iter.hasNext ()) {
String paramName = (String) iter.next ();
out.println (paramName +request.getParameter (paramName));
}
```

To prevent XSS attacks, it is important to focus on the areas where application code returns output to a user (client). It is important to identify calls that require output to be encoded. The code of the application replaces the special characters before displaying the output is also one of the remediation to xss vulnerability.

<	&lt;
>	&gt;

(	&#40;
)	&#41;
#	&#35;
&	&#38;

The approach to preventing XSS attacks is encoding. Server-side encoding is a function in which scripting tags in all dynamic content can be replaced with codes in a chosen character set. For example, < and > becomes &lt; and &gt;. The J2EE , Struts and webworks frameworks support encoding.

1.1.5 Buffer Overflow Flaws

Lack of proper validation of input to the web components leads to buffer overflows. As Java is a Type safety language so overrunning the buffer is avoidable but Java based web applications can invoke native methods, libraries that are written in C and C++ using native calls. These native calls introduce vulnerabilities. To mitigate this vulnerability focus should be given on the areas where the Java Native Interface (JNI) API calls. All input passed to these calls should be validated to prevent an attacker from injecting malicious commands into the application. These JNI API calls should be reviewed to determine memory corruption does not take place which leads to buffer overflow.

The following code illustrates JNI call in JSP which uses native TestDLL.dll using System.loadLibrary method.

```
Class JNICALL
{
public native String test (String name);
static
{System.loadLibrary ("TestDLL") ;}
}
```

1.1.6 SQL Injection Flaws

SQL Queries and SQL Injection Attacks: SQL queries uses unvalidated user input as vulnerable script In an SQL injection attack, an attacker sends malicious input through the application interface like login page and these queries are executed in the backend database and attacker can gain control over a database or perform DOS attacks like Database shutdown. The following example illustrates JSP login form where the attacker can enter malicious commands like "XP_cmdshell" or "*; OR 1=1 —"etc.

```
String Uname = request.getParameter ("User");
String Pword =request.getParameter (Password");
String s = "SELECT * FROM Table WHERE Username = ' " + UName + " " AND Password = ' " + Pword " ' ";
Statement stmt = connection.createStatement ();
ResultSet rs = stmt.executeQuery (s);
If (rs.next ()) {
        UID = rs.getInt (1)
        User = rs.getString (2)
        }
PrintWriter writer= response.getWriter ();
writer.println ("User Name: "+ User);
 }
```

In the above example, the SQL query is created using the username and password transferred to the server without validating the input which is vulnerable to a SQL injection attack. To mitigate the risk of SQL injection is to use only stored procedures or parameterized database calls. Using prepared statement all the queries will be treated as a string but not commands to be executed by the database.
The following code can be used as remediation for the above vulnerable code.

```
String s = "SELECT * FROM Table WHERE Username = ' " + UName + " " AND Password = ' " + Pword " ' ";
PreparedStatement stmt = connnection.prepareStatement(s);
stmt.setString (1, UName);
stmt.setString (2, Pword);
```

```
ResultSet results = stmt.execute ();
```

1.1.7 Command Injection Attacks

Executing System/Operating System Commands through the application are the sources of Command Injection Attacks. Java has the provision to execute system commands with the Runtime. Exec () method. Improper validation of the exec method arguments leads to vulnerability in the source code and attacker can pass malicious commands through the exec argument and gain control of the system.
The following sample code illustrates the Command injection vulnerability.

```
Class Execclass
{
  public static void main (String args [])
  {
        Runtime rt = Runtime.getRuntime ();
        Process proc = rt.exec ("cmd.exe /C") ;}
}
```

1.1.8 Improper Error/ Exception Handling

Improper error messages can reveal information about an application which may helps an attacker in exploiting the application. With the improper error handling detailed internal error messages such as stack traces, database dumps, exception and error details are displayed to the end user. printStackTrace () method reveals the application details and technology usage. The remediation for the improper error handling vulnerability is to redirect all the error messages to log file like using Log4j .The calls should be reviewed to determine whether the appropriate details are displayed to the user. All errors, exceptions, should be logged in log file instead of displaying directly to the end user. If the application shows the attacker a stack trace, it gives technology details. This information will enable the attacker to target known vulnerabilities in these components. A web application must be configured with a default error page. Whenever an exception comes the events should be logged to the log file and default error page should be displayed to the end user.
The following example shows the improper error handling vulnerability in the code

```
try {}
catch (ArrayIndexOutOfBoundsException e) {
System.out.println ("exception: " + e.getMessage ());
e.printStackTrace ();
}
```

The web.xml should include at least the following entries:

```
 <Error-page>
 <exception-type>java.lang.Throwable</exception-type>
 <location>/error.jsp</location>
  </error-page>
<Error-page>
<error-code>500</error-code>
<location>/error.jsp</location>
 </error-page>
```

1.1.9 Improper Access control

Improper assignment of restrictions on the privileges of authenticated users is allowed to leads to improper access control flaws. Attackers can exploit these flaws to access other users' accounts, view sensitive files, or use unauthorized functions. Improper Access Control vulnerabilities are identified by inspecting declarative and programmatic access control of J2EE applications.

- Declarative deployment descriptor specifies restriction of users, beans and methods at the deployment descriptor level.

- Programmatic access control can query identity of the user to verify privileges of user or business logic in the Bean program.

J2EE Access control Security covers both web-tier and EJB-tiers. At Web-tier, the access control is performed on resources which are represented in the form of an URL and at EJB-tier the access control can be applied on business methods of a bean in the application.
Web-tier container enforces security policy specified in web.xml deployment descriptor and EJB container will enforce security policy specified in ejb-jar.xml deployment descriptor.

Web-tier Declarative Access Control:
Web-tier declarative access control applied in web.xml, by defining the security constraint applies to which application resources. The <url-pattern> element inside the <web-resource-collection> element provides restricted access to URL's. The <url-pattern> can refer to a directory, filename, or a <servlet-mapping>. The web-resource-collection element designates the access to which URLs should be restricted the auth-constraint element in web.xml specifies roles that should have access to the given URLs in url-pattern element.

```
<Security-constraint>
        <Web-resource-collection>
        <web-resource-name>admin pages</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
        </web-resource-collection>
        <Auth-constraint>
        <role-name>administrator</role-name>
        <role-name>Normal user</role-name>
        </auth-constraint>
</security-constraint>
```

EJB-tier Declarative Access Control:
EJB components provide declarative access control with the elements defined in the ejb-jar.xml deploy descriptor.
Security Roles are defined in ejb-jar.xml

```
<Enterprise-beans>
<Security-role>
        <role-name>administrator</role-name>
        </security-role>
        <Security-role>
        <role-name> Normal user </role-name>
        </security-role>
 </enterprise-beans>
```

EJB-Tier Programmatic access control:
EJB-Tier Programmatic access control can be implemented in the code of the Bean logic of the application. It can be implemented with the isCallerInRole () and getCallerPrincipal () methods.
GetCallerPrincipal method is used to find out by whom the bean method is called and isCallerInRole method is used to check whether the caller of the application belongs to privileged role.

```
Public interface javax.ejb.EJBContext {
Public java.security.Principal getCallerPrincipal ();
Public Boolean isCallerInRole (String role);
}
```

1.1.10 Improper Authentication Flaws

Improper assignment of account credentials and session tokens leads to Improper authentication vulnerability. Attackers can compromise passwords, session cookies and other tokens to bypass authentication. Vulnerabilities exist in the areas of weak credentials for authentication usage, unencrypted credentials and inefficient session usage. These vulnerabilities leads to privilege escalation attacks. These mechanisms must be reviewed to ensure proper implementation.

Web-Tier Authentication:
Authenticating users in web-tier application can be implemented in the following options.

- Form

- Basic
- DIGEST

Form-Based Authentication:
Form based Authentication can be used in implementation by defining FORM in the <auth-method> element of web.xml. Using this application can use form login page login.jsp as shown in below. <

```
<Login-config>
            <auth-method>FORM</auth-method>
            <Form-login-config>
                        <form-login-page>login.jsp</form-login-page>
            </form-login-config>
</login-config>
```

BASIC Authentication:
Form based Authentication can be used in implementation by defining FORM in the <auth-method> element of web.xml. Using this application can use authentication scheme provided by browser as shown in below.

```
<Login-config>
<auth-method>BASIC</auth-method>
<realm-name>Login</realm-name>
</login-config>
```

EJB-Tier Authentication:
Two Possible Authentication Schemes in J2EE

- User credential information passed to web-tier and forwards them for authentication at EJB- tier
- User credential information passed to web-tier and web-tier authenticates the user at Web-tier. Web-tier then forwards the identity of a user to the EJB-tier.

1.1.11 Improper Session Management

Inspect the proper configurations for web.xml file for improper session management vulnerabilities. Inspect the Session Timeouts in the code. Session timeout can be checked in web.xml file.

- Using session. invalidate () to explicitly deactivate the sessions
- Session timeout can be using session.setMaxInactiveInterval ()

```
<Session-config>
<Session-timeout>
 Time-in-minutes
</session-timeout>
</session-config>
```

1.1.12 Improper Cookie Management

Improper Cookie management leads to cookie poisoning and privilege escalation vulnerabilities in the application. The following vulnerable code set the cookie as persistent and stored in the local system for one hour.

```
Public class SetCookie extends HttpServlet {
Public void doGet () {
Cookie cookie = new Cookie ("Session Cookie" + value);
cookie.setMaxAge (3600);
response.addCookie (cookie);
 }
```

The cookie values life time should be small time period and should be encrypted with efficient algorithms so that it can't be readable by anyone

1.1.13 Denial of Service

Denial of service vulnerability in the code can be identified in the areas where resource consumption or the places where attacker can disrupt the services to the legitimate users. Unreleased Resources causes Denial of Service vulnerability. Denial of service attacks cause a Web application to fail by causing the application to shut down or by consuming the available resources. Identify the calls like System.exit() to prevent from DOS. The exit method forces termination of all threads in the JVM which leads to the DOS for the legitimate user.
The following program fails to release the database connection resource

```
Statement stmt = conn.createStatement ();
 ResultSet rs = stmt.executeQuery ();
 Stmt.close ();
```

The remediation for the above example is to release resources in a finally block. Never rely on finalize () to reclaim resources. If, the statement objects in the above program won't be closed frequently then the database will run out of available resources and not be able to execute any more SQL queries.
The above vulnerable program can be rewritten as follows:

```
 Statement stmt;
 Try {
         stmt = conn.createStatement ();
         stmt.executeQuery ();
 }
 Finally {
         If (stmt! = null) {
 Try {
        stmt.close ();
} catch (SQLException e) {
}
} catch (SQLException e) {
}
```

1.1.14 Improper Configuration Flaws

Proper server configuration is critical to a secure web application. The focus of the code review areas are from most common improper configuration vulnerabilities like Unpatched versions of the products, default files access, access to backup files, access to admin pages, using default accounts and passwords provided by vendor etc.

Disabling the default servlet access:
Url-pattern element in web.xml can disable the default servlet i.e. Invoking servlet.

```
<url-pattern>/servlet/*</url-pattern>
```

Disabling Unnecessary HTTP methods:
Web-resource-collection in web.xml has http-method element that restricts the HTTP methods. The following shows the sample configuration of web.xml to disable PUT and DELETE HTTP methods.

```
<Security-constraint>
<Web-resource-collection>
<web-resource-name>Disallowed Location</web-resource-name>
<url-pattern>/*</url-pattern>
 <http-method>PUT</http-method>
 <http-method>DELETE</http-method>
         </web-resource-collection>
 </security-constraint>
```

Restricting access to specific directory:
Web-resource-collection in web.xml has an element to restrict access of the clients to specific directories. The following web.xml configuration shows data in the sensitive directory of the Web application should be protected.

```
 <Security-constraint>
<Web-resource-collection>
<web-resource-name>Sensitive</web-resource-name>
<url-pattern>/sensitive/*</url-pattern>
</web-resource-collection>
</security-constraint>
```

1.1.15 Transport Level Flaws

One of the element in deployment descriptor web.xml indicates which transport-level protections should be used when accessing the resources. Confidential invokes the use of SSL. For example, the following configuration shows the server to only permit HTTPS connections

```
 <Security-constraint>
 <User-data-constraint>
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
 </security-constraint>
```

Insecure transmission of secrets:
Plain text transmission of secrets is critical vulnerability. Transmission of sensitive data should be done through encryption. By using HTTPS, using SSL encryption is the solution to protect sensitive web traffic.

1.1.16 Code Review Tools

Some of the Code Review Tools for JAVA/J2EE which can be used for security

1) Escjava
   URL: http://research.compaq.com/SRC/esc/download.html
2) Hammurapi
   URL: http://www.hammurapi.org/
3) Jlint
   URL: http://www.willowriver.net/products/jlint.php
4) JavaPathFinder
   URL: http://javapathfinder.sourceforge.net/
5) JavaPureCheck
   URL: http://java.sun.com/products/archive/100percent/4.1.1/index.html
6) Checkstyle
   URL: http://eclipse-cs.sourceforge.net/
7) Pmd
   URL: http://sourceforge.net/projects/pmd
8) Findbugs
   URL: http://findbugs.sourceforge.net/

1.1.17 References

1) http://www.owasp.org/documentation/topten.html (www.owasp.org)
2) http://www.javapassion.com/j2ee/EJBSecurity_speakernoted.pdf
3) http://www.ouncelabs.com/audit/SoftwareSecurityAssuranceFramework.pdf
4) The Dirty Dozen: The Top Web Application Vulnerabilities and How to Hunt Them Down at the Source (www.ouncelabs.com)
5) Controlling Web Application Behavior with web.xml
http://developer.java.sun.com/developer/Books/javaserverpages/servlets_javaserver/servlets_javaserver05.pdf
6) Handling Java Web Application Input
http://today.java.net/pub/a/today/2005/09/20/handling-web-app-input.html
8) http://www.owasp.org/index.php/Cryptography
7) Configuring Security in Web Applications
http://e-docs.bea.com/wls/docs70/webapp/security.html
8) web.xml Deployment Descriptor Elements

http://e-docs.bea.com/wls/docs61/webapp/web_xml.html
9) http://www.javapractices.com/Topic86.cjp