**OWASP Vulnerability Report on Selective Issues**

**Background: System Diagram**

A diagram of a cloud

Description automatically generated

Two versions of FileUpload Java Servlet are simultaneously deployed under different context paths. The original version is vulnerable to vulnerabilities mentioned in this report, whereas the updated version contains remediation for these vulnerabilities.

**Background: Application Setup**

A diagram of a diagram

Description automatically generated

**Authorization Testing – Testing for Insecure Direct Object References**

**Vulnerable Point**  
When user is accessing fileContent.jsp through the request path “/FileContentServlet/{fileId}”, the value of the request parameter “fileId” is used directly to retrieve a database record.

The FileContentServlet takes the value of this parameter and uses it in a query to the database and the application returns the file content information to the user.

By modifying the value of the “fileId”, it is possible to retrieve any invoice object regardless of user’s access right of the file.

**Testing**  
User can easily navigate and view different file contents by changing the “fileId” parameter from the sample request below.

/tc/hss-servlet/FileContentServlet/2 -> /tc/hss-servlet/FileContentServlet/3

**Remediation**  
Given the context of current FileUploadServlet, this vulnerability is non-applicable. This is because there is no authorization layer in the application and therefore anyone is already able to access all the files within the database.

However, once the scope of the application extends and introduces file access rights within different users, the use of the value of the “fileId” request parameter directly to retrieve a database record will introduce mentioned vulnerability.

There are several ways to remediate this potential vulnerability once authentication and authorization layer has been introduced to the application.

1. Enforcing Access Control  
   For every file read request, verify that authenticated user has permission to view the file content associated with the “fileId” request parameter. This can be achieved by implementing role-based or resource-based access checks to ensure only users with correct access rights can access the file content.
2. Using Indirect References for Sensitive Identifiers  
   Instead of exposing database primary key, using indirect references such as UUID or encrypted tokens can ensure that manipulation of file references are difficult and cannot be done easily.

**Input Validation Testing – Stored Cross Site Scripting**

**Vulnerable Point**  
When user is uploading a file, it is possible to upload HTML content. This means that any XSS payload can be injected into the file uploaded, opening opportunity of various browser-based attacks to any other users that might open the malicious file.

**Testing**  
A malicious file containing HTML content such as:

<script>window.location.href=<https://www.google.com>;</script>

Can be uploaded through the FileUploadServlet. When a user tries to view the content of the file through FileContentServlet, the stored Javascript runs and redirects the user to Google website.

**Remediation**

1. Input Sanitization & Escaping Specific HTML Characters  
   Various utilities provide out of the box expression language to escape HTML characters to prevent tags like “<script>” from executing.  
     
   For instance, the StringEscapeUtils from Apache Commons Library provides functionality to sanitize input or escape HTML characters and can be easily applied by modifying the input or the fetched file content from the database:

// sanitization on the input or on fetched fileContent from the database and convert to String

fileContent = StringEscapeUtils.*escapeHtml4*(fileContent);

HTML characters such as “<”, “>”, “&” are converted to “&lt;”, “&gt;”, and “&amp;”, rendering the content as plaintext.

The escape of characters can be performed on other fronts as well. For example, JSTL (JSP Standard Tag Library) also provides escaping utility within its tag libraries.

1. Setting CSP (Content Security Policy)  
   To prevent stored XSS, we can specify the Content Security Policy to specify which content sources are allowed to be loaded on our web page.  
     
   Directives such as “default-src ‘self’” can be configured to only allow content from the same origin as the page to be allowed to be loaded. This prevents malicious content from external sources. However, in our case, the malicious content is stored and served from our database.  
     
   To counter this, we can configure CSP with a single use, randomized nonce value to only allow inline script that has matching nonce value to be allowed to run. Since nonce is randomized and not reused, the stored XSS will be rejected by CSP.  
     
   Configuration of CSP with nonce value can be achieved as follows:

SecureRandom secureRandom = new SecureRandom();

byte[] nonceBytes = new byte[16];

secureRandom.nextBytes(nonceBytes);

String nonce = Base64.getEncoder().encodeToString(nonceBytes);

response.setHeader("Content-Security-Policy", "default-src 'self'; script-src 'self' 'nonce-" + nonce + "'; style-src 'self';");

The CSP in effect will look as follows:  


However, it is important to realize that CSP is dependent on the browser and its version. Because of this, it is important to pair CSP with another security feature such as escaping or input sanitization.

**Input Validation Testing – SQL Injection**

**Vulnerable Point**  
When user is uploading the file, the backend application constructs SQL query of String data type through raw concatenation to provide SQL query for the JDBC MySQL DB driver. Because user supplied data is directly used in construction of SQL query, this opens up vulnerability of SQL injection attack, where the user can place malicious SQL commands within the file.

**Testing**  
SQL command can be injected through fileName or fileContent. An example of crafting SQL injection through fileContent is as follows:

'); CREATE TABLE MALICIOUS (id INT PRIMARY KEY); --

When this fileContent is read by the server and SQL query is constructed, the final outcome will look as follows:

INSERT INTO file\_uploads (fileName, fileContent) VALUES ('sql injection.txt', ''); CREATE TABLE MALICIOUS (id INT PRIMARY KEY); --')

This will insert empty files and create a new table named MALICIOUS. If more detrimental SQL commands like DROP were used, this will be fatal to the entire application.

Certain configuration or different types of SQL database may or may not allow multiple queries. Allowing multiple queries allows piggybacking of SQL queries as demonstrated above. However, even when multiple queries are not allowed, certain SQL injections such as “’OR 1=1” can bypass existing conditional logics and are highly impactful.

**Remediation**

1. Prevent Multiple Queries  
   While preventing multiple queries do not stop SQL injections, it limits the attack surface by preventing chained queries.
2. Use of Parameterized Queries  
   Use of parameterized queries such as PreparedStatements in Java ensures that user input is passed as parameters, not concatenated into query string. This separates data from query and prevents any SQL from being injected.  
     
   To prevent user input from being concatenated into query directly, the existing SQL query construction:

String query = "INSERT INTO file\_uploads (fileName, fileContent) VALUES ('" + fileName + "', '" + content + "')";

Can be changed to use PreparedStatement as follows:

String query = "INSERT INTO file\_uploads (fileName, fileContent) VALUES (?, ?)";

try (PreparedStatement preparedStatement = connection.prepareStatement(query)) {

preparedStatement.setString(1, fileName);

preparedStatement.setString(2, content);

**Testing for Error Handling and Security Misconfiguration**

**Vulnerable Point of Insufficient Error Handling**  
Currently entire stack trace of the exception is thrown and displayed on the error page. This could disclose any sensitive information, expose potential security weaknesses, or reveal application logic insights.

A screenshot of a computer error

Description automatically generated

**Vulnerable Point of Security Misconfiguration**Currently database connection string, user, and password are hardcoded into application. This poses potential vulnerabilities when source code is put on public repository. The hardcoded credentials can also be leaked as well.

**Remediation**

1. To prevent entire stack trace from being displayed and exposing information, a generic error message can be displayed to the user while the stack trace is only logged into a place where the application owner can access. In addition, logger can be implemented if further organization of logs are needed based on levels.
2. To prevent hardcoded credentials, the credentials can be put into environments variables. This can be done by placing “context.xml” containing environment variable information onto respective web application’s “META-INF” folder within tomcat.

The environment variables from context.xml can be referenced as follows:  
 Context env = (Context) new InitialContext().lookup("java:comp/env");

String jdbcURL = (String) env.lookup("DB\_URL");

String dbUser = (String) env.lookup("DB\_USER");

String dbPassword = (String) env.lookup("DB\_PW");

**Business Logic Testing – Preventing DoS through File Upload**

**Vulnerable Point**  
The application temporarily stores user uploaded file into a temporary directory and then stores the content of the temporary file onto the database. This means that if user provided data is large enough to consume the server’s computational power, there is possibility of denial-of-service vulnerability introduced through file upload.

**Testing**  
A large file (about 4GB) that exceeds the memory of the server was attempted to be uploaded and the server experienced severe choke, and the instance had to be stopped and restarted.

**Remediation**  
There are several fronts where file size limitation can be imposed.

1. Application Layer  
   A MultipartConfig can be specified on the servlet. This configuration includes fields such as maxFileSize for maximum allowed size for each file, fileSizeThreshold for the threshold value where file larger than the value will be stored on the disk instead of the memory, and maxRequestSize which specifies maximum allowed size for a request.

A screenshot of a error message

Description automatically generated

However, the user’s file is already available at the application layer and could still pose DoS threat to the application.

1. System Layer  
   Instead of specifying file limit on the application layer, the limit can be imposed on the first entry point of our system, which is the Apache HTTP Server. By imposing a certain limit on the value of “Content-Length” header, a DoS attempt of large file can be avoided without reading the entire file. Any request that exceeds limit value of “Content-Length” can be rejected.  
     
   This can be set up on Apache HTTP Server by enabling mod\_rewrite module and configuring the rewrite engine as follows:

RewriteEngine On

RewriteCond %{HTTP:Content-Length} >1000

RewriteRule .\* - [R=413,L]

And the subsequent error page will be displayed for any request with “Content-Length” > 1000:

A close up of a sign

Description automatically generated

Combination of “Content-Length” check and file size check at the application layer can reject unrealistic request at the system layer by only reading the file’s header. For any file that exceeds the file size limit but was under Content-Length limit will be rejected at the application layer.