Analysis Results

BenchmarkJava-master.zip

Report Date

2024-09-10 12:56:58

Report Author

admin

Classification Method

CWE/SANS Top 25 2023

Product Version

11.0-SNAPSHOT+f428841a

Rules Version

11-SNAPSHOT.130321

Confidentiality Note

!

This report is intended only for the person(s) or entity to which it is addressed and contains confidential and privileged information. If you are not the intended recipient, you must not view, use, copy, disclose, or otherwise disseminate this report or any part of it. Doing so is strictly prohibited, and may result in legal proceedings. If you received this in error, please notify the sender immediately and destroy any copies of this information.

01 Project Information

Project Name

BenchmarkJava-master.zip

UUID

eb929e8d-12d7-47da-8402-bc205e3187f0

Project in DerScanner

Dynamics by vulnerabilities

Vulnerabilities are divided by severity level: critical, medium, low and info.

CRITICAL LEVEL

Likely to lead to compromise confidential data and violation of the integrity of the system.

MEDIUM LEVEL

May be less likely to lead to compromising confidential data and violating the integrity of the system, or are less serious security

LOW LEVEL

Can become a potential security risk.

INFORMATION

Signal a violation of good programming practice.

First of all, pay attention to vulnerabilities of critical and medium levels.

Dynamics by rating

The app score is calculated on a scale from 0 to 5. Score is calculated based on the number of critical and medium level vulnerabilities. The impact of critical vulnerabilities is greater than that of medium level vulnerabilities, and does not take into account the amount of code. Medium level vulnerabilities are taken into account based on their frequency and total number of source code lines.

Scan History

	Date and Time	Status	Languages	Lines of Code	Number of Vulnerabilities					Score
					Critical	Medium	Low	Info	Total	
1/1	2024-09-04 09:42:42	completed	Config files, Java, Scala, Kotlin, VBScript, HTML5, JavaScript	3 568 828	1 166	36 043	4 332	5 918	47 459	0.0/5.0

02

Scan Information

1/1 2024-09-04 09:42:42
11-SNAPSHOT.130256

Scan Statistics

Status
completed

Score
0.0/5.0

Duration
0:42:31

Lines of Code
3 568 828

Vulnerabilities

1 166	36 043	4 332	5 918
Critical	Medium	Low	Info
47 459			
Total			

Language	Status	Duration	Lines of Code	Number of Vulnerabilities				
				Critical	Medium	Low	Info	Total
Config files	completed	0:02:59	3 049 523	0	29 066	4	0	29 070
JVM languages	completed	0:31:44	283 640	1 134	4 820	4 315	5 915	16 184
VBScript	completed	0:00:11	117 002	0	0	0	0	0
HTML5	completed	0:06:05	112 573	30	2 155	0	0	2 185
JavaScript	completed	0:01:30	6 090	2	2	13	3	20

Language Statistics

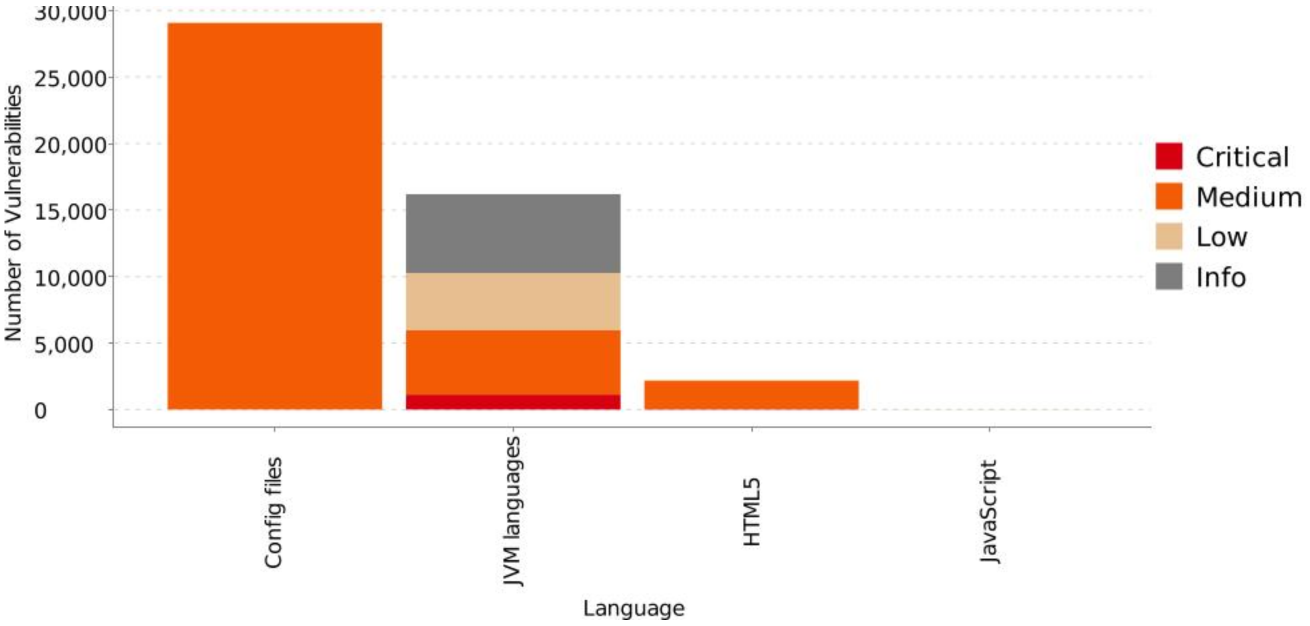
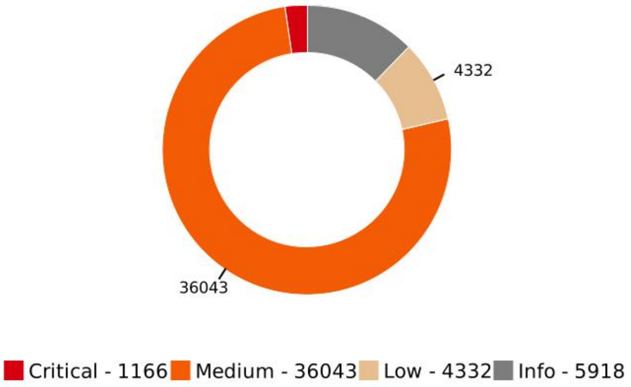
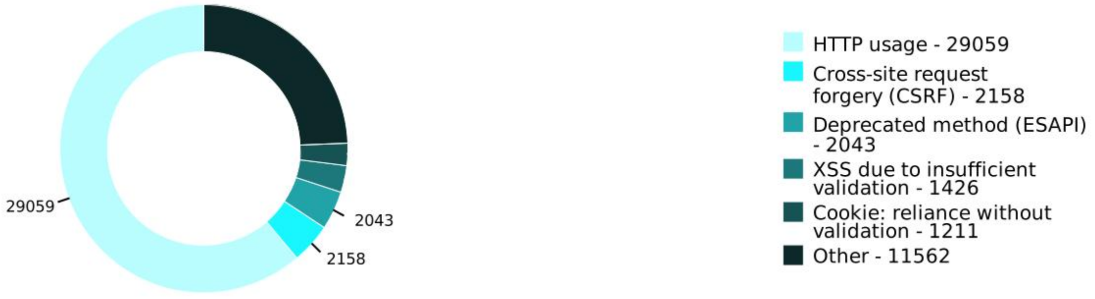


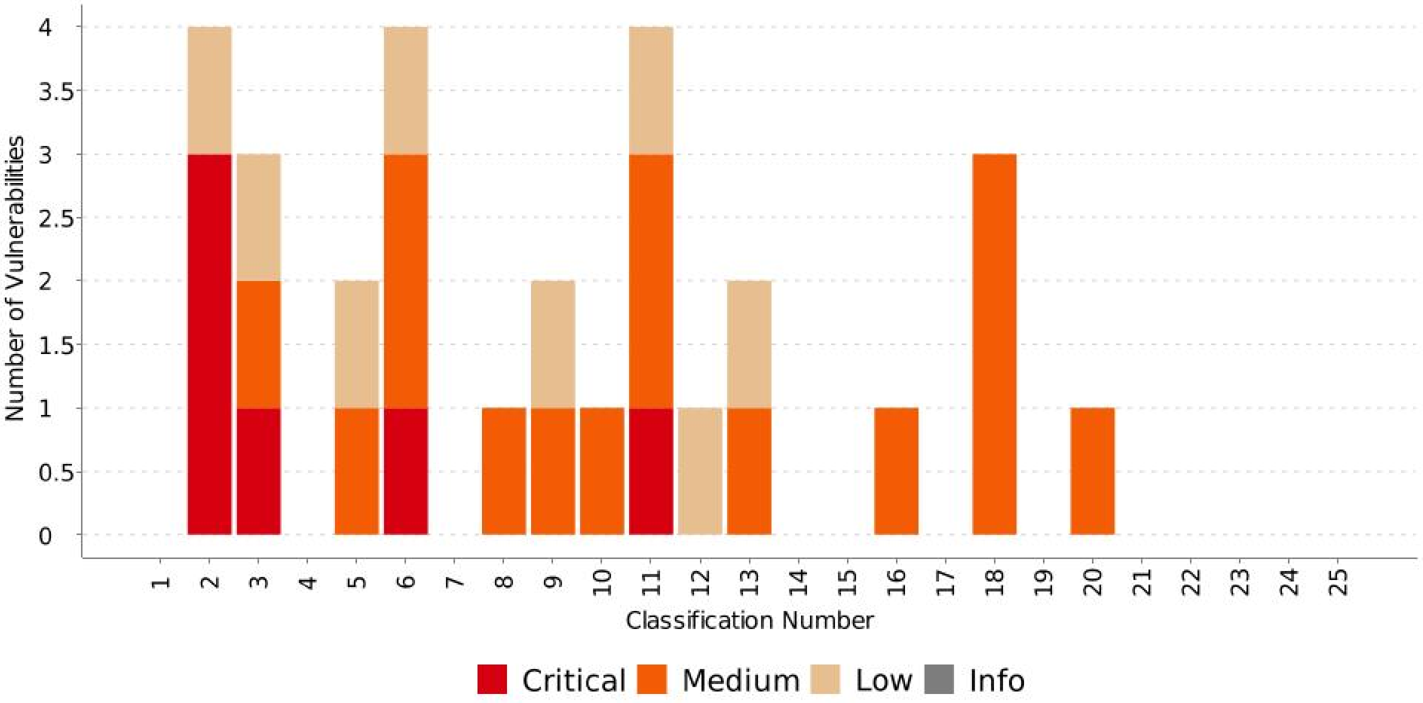
Diagram of identified vulnerabilities



Vulnerability Types



Classification by CWE/SANS Top 25 2023



	Vulnerabilities					Occurrences				
	Critical	Medium	Low	Info	Total	Critical	Medium	Low	Info	Total
1	0	0	0	0	0	0	0	0	0	0
2	3	0	1	0	4	603	0	1426	0	2029
3	1	1	1	0	1	232	158	54	0	444

	Vulnerabilities					Occurrences				
	Critical	Medium	Low	Info	Total	Critical	Medium	Low	Info	Total
4	0	0	0	0	0	0	0	0	0	0
5	0	1	1	0	2	0	236	2	0	238
6	1	2	1	0	2	232	394	54	0	680
7	0	0	0	0	0	0	0	0	0	0
8	0	1	0	0	1	0	648	0	0	648
9	0	1	1	0	2	0	2155	3	0	2158
10	0	1	0	0	1	0	648	0	0	648
11	1	2	1	0	3	1	3	1	0	5
12	0	0	1	0	1	0	0	1	0	1
13	0	1	1	0	1	0	2	1	0	3
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	1	0	0	1	0	236	0	0	236
17	0	0	0	0	0	0	0	0	0	0
18	0	3	0	0	3	0	15	0	0	15
19	0	0	0	0	0	0	0	0	0	0
20	0	1	0	0	1	0	1	0	0	1
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0

Vulnerability List

Vulnerabilities are displayed accordingly to export settings: **16 selected**

Actual: **16 of 47459**

CWE-79

Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)

Critical vulnerabilities		**4***
Reflected XSS	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00013.java:56		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00014.java:56		Confirmed
Persistent XSS	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00012.java:78		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00021.java:69		Confirmed
Medium-level vulnerabilities		**0**
Low-level vulnerabilities		**0**
Info-level vulnerabilities		**0**

CWE-89

Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)

Critical vulnerabilities		**2***
SQL injection	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00027.java:52		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00018.java:59		Confirmed

* Rejected vulnerabilities are not taken into account

CWE-89 Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)

Medium-level vulnerabilities		0
Low-level vulnerabilities		0
Info-level vulnerabilities		0

CWE-78 Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)

Critical vulnerabilities		0
Medium-level vulnerabilities		2*
Command injection	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00006.java:66		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00007.java:61		Confirmed
Low-level vulnerabilities		0
Info-level vulnerabilities		0

CWE-20 Improper Input Validation

Critical vulnerabilities		2*
SQL injection	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00027.java:52		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00018.java:59		Confirmed
Medium-level vulnerabilities		2*
Command injection	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00006.java:66		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00007.java:61		Confirmed
Low-level vulnerabilities		0

* Rejected vulnerabilities are not taken into account

CWE-20 Improper Input Validation

Info-level vulnerabilities		0

CWE-22 Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)

Critical vulnerabilities		0
Medium-level vulnerabilities		2*
Path manipulation	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00001.java:71		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00002.java:72		Confirmed
Low-level vulnerabilities		0
Info-level vulnerabilities		0

CWE-352 Cross-Site Request Forgery (CSRF)

Critical vulnerabilities		0
Medium-level vulnerabilities		2*
Cross-site request forgery (CSRF)	HTML5	2
✓ BenchmarkJava-master/src/main/webapp/cmdi-00/BenchmarkTest00006.html:12		Confirmed
✓ BenchmarkJava-master/src/main/webapp/cmdi-00/BenchmarkTest00007.html:12		Confirmed
Low-level vulnerabilities		0
Info-level vulnerabilities		0

CWE-434 Unrestricted Upload of File with Dangerous Type

Critical vulnerabilities		0
Medium-level vulnerabilities		2*

* Rejected vulnerabilities are not taken into account

CWE-434 Unrestricted Upload of File with Dangerous Type

Medium-level vulnerabilities		
Path manipulation	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00001.java:71		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00002.java:72		Confirmed
Low-level vulnerabilities		0
Info-level vulnerabilities		0

CWE-862 Missing Authorization

Critical vulnerabilities		1*
Empty password	Java	1
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/HibernateUtil.java:74		Confirmed
Medium-level vulnerabilities		1*
Missing authorization	Java	1
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/HibernateUtil.java:74		Confirmed
Low-level vulnerabilities		0
Info-level vulnerabilities		0

CWE-476 NULL Pointer Dereference

Critical vulnerabilities	0
Medium-level vulnerabilities	0
Low-level vulnerabilities	0
Info-level vulnerabilities	0

* Rejected vulnerabilities are not taken into account

CWE-287 Improper Authentication

Critical vulnerabilities		**0**
Medium-level vulnerabilities		**0**
Low-level vulnerabilities		**0**
Info-level vulnerabilities		**0**

CWE-77 Improper Neutralization of Special Elements used in a Command (Command Injection)

Critical vulnerabilities		**0**
Medium-level vulnerabilities		**2***
Command injection	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00006.java:66		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00007.java:61		Confirmed
Low-level vulnerabilities		**0**
Info-level vulnerabilities		**0**

CWE-798 Use of Hard-coded Credentials

Critical vulnerabilities		**0**
Medium-level vulnerabilities		**2***
Hardcoded password	Java	2
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/HibernateUtil.java:154		Confirmed
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/LDAPServer.java:110		Confirmed
Low-level vulnerabilities		**0**
Info-level vulnerabilities		**0**

* Rejected vulnerabilities are not taken into account

CWE-306

Missing Authentication for Critical Function

Critical vulnerabilities		**0**
Medium-level vulnerabilities		**1***
Missing authorization	Java	1
✓ BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/HibernateUtil.java:74		Confirmed
Low-level vulnerabilities		**0**
Info-level vulnerabilities		**0**

* Rejected vulnerabilities are not taken into account

Analysis Results

CWE-250

Empty password (Java)

Description

An empty password may lead to an application compromise.
Eliminating the security risks related to hardcoded empty passwords is extremely difficult. The information that a certain account accepts an empty password is accessible to at least every developer of the application. Moreover, after the application is installed, removing an empty password from its code is possible only via an update. Constant strings are easily extracted from the compiled application by decompilers. Therefore, an attacker does not necessarily need to have an access to the source code to know parameters of the special account. If these parameters become known to an attacker, system administrators will be forced either to neglect the safety, or to restrict the access to the application.

Example

In the following example, the connection to the database is established with an empty password:

```
DriverManager.getConnection(url, "user", "");
```

The account that allows to enter with an empty password threatens the security of the system. If the line given above is included into the final version of the application, correcting the error without updating the code will not be possible.
In the following example, the password variable is initialized to an empty value, which is then replaced by the value derived from the password storage and compared to the value provided by the user:

```
String storedPassword = "";
String temp;

if ((temp = readPassword()) != null) {
   storedPassword = temp;
}

if(storedPassword.equals(userPassword))
   // Access protected resources
}
```

If the readPassword() method is unable to get the stored password (due to a database error or for any other reason), the attacker will get access to protected resources by providing an empty password.
In case of a mobile application, security threat is even higher, considering the risk of the device loss.

Recommendations

- Do not use empty passwords.
- Store not passwords but values of cryptographically secure hash function from the password. Use specialized hash functions designed for this purpose (bcrypt, scrypt). Use salt obtained from cryptographically secure pseudorandom number generator to resist attacks which use rainbow tables.
- If the hardcoded password is used for the initial authorization, provide the special authentication mode for this purpose in which the user is required to provide his/her own unique password.
- Store authentication information in an encrypted form in a separate configuration file or in a database. Secure the encryption key. If encryption is not possible, limit the access to the repository as much as possible.
- For secure password storage on the platforms using the SQLite database (including Android), use the SQLCipher extension.

Links

1. Use of hard-coded password
2. CWE-259: Use of Hard-coded Password
3. OWASP Top 10 2017-A2-Broken Authentication
4. OWASP Top 10 2013-A5-Security Misconfiguration
5. OWASP Top 10 2013-A6-Sensitive Data Exposure
6. Handling passwords used for auth in source code - stackoverflow.com
7. How to securely hash passwords? - security.stackexchange.com
8. CWE CATEGORY: OWASP Top Ten 2017 Category A2 - Broken Authentication
9. CWE CATEGORY: OWASP Top Ten 2017 Category A6 - Security Misconfiguration

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/HibernateUtil.java:74

Level Critical

Status Confirmed

```
Class.forName("org.hsqldb.jdbcDriver");
//                              System.out.println("Driver Loaded.");
String url = "jdbc:hsqldb:benchmarkDataBase;sql.enforce_size=false";

conn = DriverManager.getConnection(url, "sa", "");

//                              System.out.println("Got Connection.");
st = conn.createStatement();
} catch (SQLException | ClassNotFoundException e) {
```

CWE-78

Persistent XSS (Java)

Description

Persistent XSS or server XSS attack is possible.
Cross-site scripting is one of the most common types of attacks on web applications. XSS attacks take seventh place in the “OWASP Top 10 2017” list of ten most significant vulnerabilities in web applications.
The main phase of any XSS attack is an imperceptible for the victim execution of a malicious code in the context of the vulnerable application. For this purpose, the functionality of the client application (browser) is used that allows to automatically execute scripts embedded in web page code. In most cases, these malicious scripts are implemented in JavaScript.
Consequences of an XSS attack vary from violations of application functionality to complete loss of user data confidentiality. The malicious code during the XSS attack can steal user HTTP-cookie, which gives an attacker the ability to make requests to the server on behalf of the user.
OWASP suggests the following classification of XSS attacks:

- Server type XSS occurs when data from an untrusted source is included in the response returned by the server. The source of such data can be both user input and server database (where it had been previously injected by an attacker who exploited vulnerabilities in the server-side application).
- Client type XSS occurs when the raw data from the user input contains code that changes the Document Object Model (DOM) of the web page received from the server. The source of such data can be both the DOM and the data received from the server (e.g., in response to an AJAX request).

Typical server type attack scenario:

1. Unvalidated data, usually from a HTTP request, gets into the server part of the application.
2. The server dynamically generates a web page that contains the unvalidated data.
3. In the process of generating a web page, server does not prevent the inclusion of an executable code that can be executed in the client (browser), such as JavaScript code language, HTML-tags, HTML-attributes, Flash, ActiveX, etc., in the page code.
4. The victim’s client application displays the web page that contains the malicious code injected via data from an untrusted source.
5. Since malicious code is injected in the web page coming from the known server, the client part of the application (browser) executes it with the rights set for the application.

6. This violates the same-origin policy, according to which the code from the one source must not get an access to resources from another source.

Client type attacks are executed in a similar way with the only difference that the malicious code is injected during the phase of the client application work with the document object model received from the server.

Example

In the following example, the JSP code retrieves the information about the employee with a given ID from the database and displays the his/her name:

```
<%...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
if (rs != null) {
  rs.next();
  String name = rs.getString("name");
}
%>

Employee Name: <%= name %>
```

If the name values do not contain special characters, the code behaves correctly. But if the name value is derived from data from an untrusted source (e.g., user input), the attacker can store the malicious code in the database. Such attacks are particularly dangerous because they may affect a large number of users.

Example for Scala application based on the Play framework: the raw method of the play.templates.JavaExtensions class returns a text without template escaping.

Recommendations

- Implement a validation mechanism. Whitelist is more secure but less flexible than the blacklist. The blacklist must at least include the characters “&”, “<”, “>”, and quotation marks.
- Many web application servers provide their own mechanisms of protection against XSS, but they may not be considered sufficient. There is no guarantee that the application will run in conjunction with the server that updates these mechanisms timely and completely.

Links

1. OWASP: Cross-site Scripting (XSS)
2. CWE-79: Improper Neutralization of Input During Web Page Generation
3. Types of Cross-Site Scripting - OWASP

4. OWASP: XSS Prevention Cheat Sheet
5. OWASP Top 10-2017 A7-Cross-Site Scripting (XSS)
6. CWE CATEGORY: OWASP Top Ten 2017 Category A3 - Sensitive Data Exposure
7. CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
8. CWE-81: Improper Neutralization of Script in an Error Message Web Page
9. CWE-83: Improper Neutralization of Script in Attributes in a Web Page
10. Cross-site Scripting (XSS) Affecting jquery-mobile package

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00012.java:78

Level Critical

Status Confirmed

```
75 javax.naming.directory.Attribute attr2 = attrs.get("street");
76 if (attr != null) {
77     response.getWriter()
78             .println(
79                     "LDAP query results:<br>"
80                             + "Record found with name "
81                             + attr.get()
```

Trace

InitialDirContext.search()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00012.java:68

```
65         (javax.naming.directory.InitialDirContext) ctx;
66 boolean found = false;
67 javax.naming.NamingEnumeration<javax.naming.directory.SearchResult> results =
68         idc.search(base, filter, filters, sc);
69 while (results.hasMore()) {
70     javax.naming.directory.SearchResult sr =
71             (javax.naming.directory.SearchResult) results.next();
```

PrintWriter.println()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00012.java:78

```
75 javax.naming.directory.Attribute attr2 = attrs.get("street");
76 if (attr != null) {
77     response.getWriter()
78         .println(
79             "LDAP query results:<br>"
80                 + "Record found with name "
81                 + attr.get()
```

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00021.java:69

Level	Critical
Status	Confirmed

```
66 javax.naming.directory.Attribute attr2 = attrs.get("street");
67 if (attr != null) {
68     response.getWriter()
69         .println(
70             "LDAP query results:<br>"
71                 + "Record found with name "
72                 + attr.get()
```

Trace

DirContext.search()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00021.java:59

```
56 // System.out.println("Filter " + filter);
57 boolean found = false;
58 javax.naming.NamingEnumeration<javax.naming.directory.SearchResult> results =
59      ctx.search(base, filter, filters, sc);
60 while (results.hasMore()) {
61    javax.naming.directory.SearchResult sr =
62          (javax.naming.directory.SearchResult) results.next();
```

PrintWriter.println()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00021.java:69

```
66 javax.naming.directory.Attribute attr2 = attrs.get("street");
67 if (attr != null) {
68    response.getWriter()
69          .println(
70                "LDAP query results:<br>"
71                      + "Record found with name "
72                      + attr.get()
```

CWE-78

Reflected XSS (Java)

Description

The reflected XSS or client type XSS attack is possible.
Cross-site scripting is one of the most common types of attacks on web applications. XSS attacks take seventh place in the “OWASP Top 10 2017” list of ten most significant vulnerabilities in web applications.
The main phase of any XSS attack is an imperceptible for the victim execution of a malicious code in the context of the vulnerable application. For this purpose, the functionality of the client application (browser) is used that allows to automatically execute scripts embedded in web page code. In most cases, these malicious scripts are implemented in JavaScript.
Consequences of an XSS attack vary from violations of application functionality to complete loss of user data confidentiality. The malicious code during the XSS attack can steal user HTTP-cookie, which gives an attacker the ability to make requests to the server on behalf of the user.
OWASP suggests the following classification of XSS attacks:

- Server type XSS occurs when data from an untrusted source is included in the response returned by the server. The source of such data can be both user input and server database (where it had been previously injected by an attacker who exploited vulnerabilities in the server-side application).
- Client type XSS occurs when the raw data from the user input contains code that changes the Document Object Model (DOM) of the web page received from the server. The source of such data can be both the DOM and the data received from the server (e.g., in response to an AJAX request).

Typical server type attack scenario:

1. Unvalidated data, usually from a HTTP request, gets into the server part of the application.
2. The server dynamically generates a web page that contains the unvalidated data.
3. In the process of generating a web page, server does not prevent the inclusion of an executable code that can be executed in the client (browser), such as JavaScript code language, HTML-tags, HTML-attributes, Flash, ActiveX, etc., in the page code.
4. The victim’s client application displays the web page that contains the malicious code injected via data from an untrusted source.
5. Since malicious code is injected in the web page coming from the known server, the client part of the application (browser) executes it with the rights set for the application.
6. This violates the same-origin policy, according to which the code from the one source must not get an access to resources from another source.

Client type attacks are executed in a similar way with the only difference that the malicious code is injected during the phase of the client application work with the document object model received from the server.

Example

In the following example, the JSP code reads the valueof the eid parameter (employee ID) and displays it on the page:

```
<% String eid = request.getParameter("eid"); %>
Employee ID: <%= eid %>
```

If the user follows the attacker’s link containing malicious script as the value of eid, this script will be executed in the browser, in particular, sending cookies, session IDs or other victim’s valuable information to the attacker.

Example for Scala application based on the Play framework: the raw method of the play.templates.JavaExtensions class returns a text without template escaping.

Recommendations

- Implement a validation mechanism. Whitelist is more secure but less flexible than the blacklist. The blacklist must at least include the characters “&”, “<”, “>”, and quotation marks.
- Many web application servers provide their own mechanisms of protection against XSS, but they may not be considered sufficient. There is no guarantee that the application will run in conjunction with the server that updates these mechanisms timely and completely.

Links

1. OWASP: Cross-site Scripting (XSS)
2. CWE-79: Improper Neutralization of Input During Web Page Generation
3. Types of Cross-Site Scripting - OWASP
4. OWASP: XSS Prevention Cheat Sheet
5. OWASP Top 10-2017 A7-Cross-Site Scripting (XSS)

6. CWE CATEGORY: OWASP Top Ten 2017 Category A3 - Sensitive Data Exposure
7. CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)
8. CWE-81: Improper Neutralization of Script in an Error Message Web Page
9. CWE-83: Improper Neutralization of Script in Attributes in a Web Page

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00013.java:56

Level Critical

Status Confirmed

```
53
54      response.setHeader("X-XSS-Protection", "0");
55      Object[] obj = {"a", "b"};

56      response.getWriter().format(java.util.Locale.US, param, obj);

57    }
58 }
```

Trace

HttpServletRequest.getHeaders()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00013.java:45

```
42 response.setContentType("text/html;charset=UTF-8");
43
44 String param = "";

45 java.util.Enumeration<String> headers = request.getHeaders("Referer");

46
47 if (headers != null && headers.hasMoreElements()) {
48     param = headers.nextElement(); // just grab first element
```

PrintWriter.format()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00013.java:56

```
53
54      response.setHeader("X-XSS-Protection", "0");
55      Object[] obj = {"a", "b"};

56      response.getWriter().format(java.util.Locale.US, param, obj);

57    }
58 }
```

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00014.java:56

Level Critical

Status Confirmed

```
53
54      response.setHeader("X-XSS-Protection", "0");
55      Object[] obj = {"a", "b"};

56      response.getWriter().format(param, obj);

57    }
58 }
```

Trace

HttpServletRequest.getHeaders()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00014.java:45

```
42 response.setContentType("text/html;charset=UTF-8");
43
44 String param = "";

45 java.util.Enumeration<String> headers = request.getHeaders("Referer");

46
47 if (headers != null && headers.hasMoreElements()) {
48    param = headers.nextElement(); // just grab first element
```

PrintWriter.format()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00014.java:56

```
53
54      response.setHeader("X-XSS-Protection", "0");
55      Object[] obj = {"a", "b"};

56      response.getWriter().format(param, obj);

57    }
58 }
```

CWE-120

CWE-862

SQL injection (Java)

Description

SQL injection is possible. This can be exploited to bypass the authentication mechanism, access all database entries, or execute malicious code with application rights.
Client side code injection attacks take the first place in the “OWASP Top 10 2017” web application vulnerabilities ranking and the seventh place in the “OWASP Mobile Top 10 2014”. ranking. The level of potential damage from such an attack depends on the user input validation performance and file protection mechanisms.
SQL Injection occurs when a database query is based on data from an untrusted source (e.g., user input). In the absence of proper validation an attacker can modify the query to execute malicious SQL query.
The most common variants of SQL injection:

- Direct addition of malicious code into a string variable, based on which the SQL query is generated.
- Premature termination of the correct SQL command via the “– ” sequence of characters (interpreted as the beginning of a comment). The contents of the string after this sequence will be ignored during the execution of SQL command.
- Addition of the “;” character (interpreted as the end of the command), and other malicious commands (request splitting) to the input string variable.
- Password guessing via the sequential execution of SQL queries.

Example

Let the application substitute the user entered city name to the SQL query. For

example, for the input string “London” we have:
SELECT * FROM Orders WHERE City = 'London'
An attacker can enter the following query:
London'; drop table Orders--
In this case, a query will be built that searches through the table and then deletes the table:
SELECT * FROM Orders WHERE City = 'London';drop table Orders-- '

Recommendations

- Do not make assumptions about the type or amount of data entered by a user.
- Implement a mechanism of validation for data entered by a user.
- Escape special characters (“;”, “–”, “/*”, “*/”, “‘”; the exact list depends on the database type).
- Use stored procedures to validate user input along with the mechanism of parameters filtering.

Links

1. OWASP Top 10 2017-A1-Injection
2. OWASP: SQL Injection
3. WASC-19: SQL Injection
4. CAPEC-66: SQL Injection
5. Understanding SQL Injection – cisco.com
6. CWE CATEGORY: OWASP Top Ten 2017 Category A1 - Injection
7. CWE-89

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00018.java:59

Level	Critical
Status	Confirmed

```
56 try {
57     java.sql.Statement statement =
58         org.owasp.benchmark.helpers.DatabaseHelper.getSqlStatement();
59     int count = statement.executeUpdate(sql);
60     org.owasp.benchmark.helpers.DatabaseHelper.outputUpdateComplete(sql, response);
61 } catch (java.sql.SQLException e) {
62     if (org.owasp.benchmark.helpers.DatabaseHelper.hideSQLErrors) {
```

Trace

HttpServletRequest.getHeaders()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00018.java:45

```
42 response.setContentType("text/html;charset=UTF-8");
43
44 String param = "";
45 java.util.Enumeration<String> headers = request.getHeaders("BenchmarkTest00018");
46
47 if (headers != null && headers.hasMoreElements()) {
48     param = headers.nextElement(); // just grab first element
```

Statement.executeUpdate()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00018.java:59

```
56 try {
57     java.sql.Statement statement =
58         org.owasp.benchmark.helpers.DatabaseHelper.getSqlStatement();
59     int count = statement.executeUpdate(sql);
60     org.owasp.benchmark.helpers.DatabaseHelper.outputUpdateComplete(sql, response);
61 } catch (java.sql.SQLException e) {
62     if (org.owasp.benchmark.helpers.DatabaseHelper.hideSQLErrors) {
```

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00027.java:52

Level Critical

Status Confirmed

```
49 try {
50     java.sql.Statement statement =
51             org.owasp.benchmark.helpers.DatabaseHelper.getSqlStatement();
52     int count = statement.executeUpdate(sql);
53     org.owasp.benchmark.helpers.DatabaseHelper.outputUpdateComplete(sql, response);
54 } catch (java.sql.SQLException e) {
55     if (org.owasp.benchmark.helpers.DatabaseHelper.hideSQLErrors) {
```

Trace

HttpServletRequest.getParameter()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00027.java:44

```
41 // some code
42 response.setContentType("text/html;charset=UTF-8");
43
44 String param = request.getParameter("BenchmarkTest00027");
45 if (param == null) param = "";
46
47 String sql = "INSERT INTO users (username, password) VALUES ('foo','" + param + "')";
```

Statement.executeUpdate()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00027.java:52

```
49 try {
50     java.sql.Statement statement =
51             org.owasp.benchmark.helpers.DatabaseHelper.getSqlStatement();
52     int count = statement.executeUpdate(sql);
```

```
    org.owasp.benchmark.helpers.DatabaseHelper.outputUpdateComplete(sql, response);
} catch (java.sql.SQLException e) {
    if (org.owasp.benchmark.helpers.DatabaseHelper.hideSQLErrors) {
```

CWE-434

Cross-site request forgery (CSRF) (HTML5)

Description

Cross-Site Request Forgery (CSRF) is possible.
Cross-Site Request Forgery (CSRF) attacks rank eighth on the OWASP Top 10 2013. CSRF is a type of attack that occurs when a malicious website, email or blog forces a user's browser to perform an action on another site where the user is logged in.
Possible scenario of an attack:
The victim goes to a site created by the attacker, and a request is secretly sent on his behalf to another server (for example, a payment system server) that performs some kind of malicious operation (e.g., transferring money to the attacker's account). In order to carry out this attack, the victim must be authenticated on the server to which the request is sent and the request must not require any confirmation from the user, which cannot be ignored or forged by the attacking script.

Example

In the following example, the web application allows administrators to create new accounts:

```
<form method="POST" action="/new_user">
 Name of new user: <input type="text" name="username">
 Password for new user: <input type="password" name="user_passwd">
   <input type="submit" name="action" value="Create User">
</form>
```

An attacker can create a malicious web site that contains the following code:

```
<form method="POST" action="http://www.example.com/new_user">
 <input type="hidden" name="username" value="hacker">
 <input type="hidden" name="user_passwd" value="hacked">
</form>
<script>
 document.usr_form.submit();
</script>
```

If the administrator visits a malicious web site during an open session on the vulnerable site, unbeknown to him/her an account will be created, which an attacker will use later.

Most browsers with every HTTP request transfer the referer header containing the address of the web site from which the transition occurs. However, since the attacker can overwrite the referer contents, this header does not help to counteract CSRF-attacks.

Recommendations

- If the application uses cookies, include in each form a secret value that can be validated on the server to verify the legitimacy of the request. The identifier (token) must be unique for each request, and not for the session. The token should not be easy to guess; it needs to be protected as well as the session token, for instance, via TLS.
- Use the framework provided mechanisms of protection against CSRF.
- Use additional mechanisms for verifying the request legitimacy, e.g., CAPTCHA, re-authentication, one-time tokens.
- Send the session ID not only as a cookie, but also as the value of the hidden field. The server must verify that these values match. An attacker will not be able to modify the value of the session identifier due to the same origin policy.
- Limit the session time. CSRF-attacks are successful only if they are carried out while the victim’s session on the vulnerable web site is valid. Reducing the session time reduces the likelihood of CSRF.

The described techniques only protect against CSRF, but not against cross-site scripting (XSS).

Links

1. OWASP Top 10 2013-A8-Cross-Site Request Forgery (CSRF)
2. CWE-352: Cross-Site Request Forgery (CSRF)
3. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet - OWASP
4. CWE-1034

Vulnerability Entries

BenchmarkJava-master/src/main/webapp/cmdi-00/BenchmarkTest00006.html:12

Level	Medium
Status	Confirmed

```
9 <title>BenchmarkTest00006</title>
10 </head>
11 <body>
12    <form action="/benchmark/cmdi-00/BenchmarkTest00006" method="POST" id="
FormBenchmarkTest00006" autocomplete="off">
13        <div>
14            <label>Please explain your answer:</label>
15        </div>
```

BenchmarkJava-master/src/main/webapp/cmdi-00/BenchmarkTest00007.html:12

Level	Medium
Status	Confirmed

```
9 <title>BenchmarkTest00007</title>
10 </head>
11 <body>
12    <form action="/benchmark/cmdi-00/BenchmarkTest00007" method="POST" id="
FormBenchmarkTest00007" autocomplete="off">
13        <div>
14            <label>Please explain your answer:</label>
15        </div>
```

CWE-306

CWE-862

CWE-829

Command injection (Java)

Description

Executing commands obtained from data from an untrusted source is insecure.
Injection vulnerabilities take the first place in the “OWASP Top 10 2017” web-application vulnerabilities ranking. Command injection vulnerabilities are divided into two categories:

1. An attacker modifies the command itself;

2. An attacker replaces the value of the environment variables, which implicitly changes the semantics of the command being executed.

In the given case, the application is prone to the vulnerability of the first type.
A possible attack scenario:

1. The application receives input data from an untrusted source, for example, user input.
2. The data obtained is used as a part of the string that defines the command.
3. Execution of the command gives an attacker the privileges which he did not previously possess.

Example

In the following example, the application executes the script for creating the database backup. The application takes a parameter that determines the type of backup as an argument and runs the script with elevated privileges.

```
String btype = request.getParameter("backuptype");
String cmd = new String("cmd.exe /K
\"c:\\util\\rmanDB.bat "+btype+"&&c:\\utl\\cleanup.bat\"")
System.Runtime.getRuntime().exec(cmd);
```

The problem here is the absence of validation for the backuptype parameter. Typically Runtime.exec() does not carry out several commands, but in this case first cmd.exe is started to execute multiple instructions with a single Runtime.exec() call. As soon as the command line shell is started, it can perform multiple commands separated by the “&&” symbols. If an attacker sets the "&& del c:\\dbms\\*.*" string as a parameter, the command for removing the specified directory will be run with elevated privileges. Similar considerations apply to both web applications and mobile applications.

Recommendations

- Do not allow users to directly control the commands executed by an application. If the behavior of the application should be dependent on the user input, suggest the user to choose from a specific list of legitimate commands.
- If user data is a command argument, the whitelist may be too cumbersome. Blacklist is also inefficient, as it is difficult to maintain it up to date and comprehensive. In this case, it is recommended to use the whitelist of characters allowed in the command parameters.
- An attacker can change the semantics of the command not only by changing it, but also by affecting its environment. Environment must not be considered a trusted source. The values of environment variables must also be validated.

Links

1. OWASP Top 10 2017-A1-Injection

2. OWASP Top 10 2013-A1-Injection
3. CWE-77: Improper Neutralization of Special Elements used in a Command (‘Command Injection’)
4. CWE-78: Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)
5. CWE CATEGORY: OWASP Top Ten 2017 Category A1 - Injection

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00006.java:66

Level Medium

Status Confirmed

```
63
64 ProcessBuilder pb = new ProcessBuilder();
65
66 pb.command(argList);
67
68 try {
69     Process p = pb.start();
```

Trace

HttpServletRequest.getHeader()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00006.java:46

```
43
44 String param = "";
45 if (request.getHeader("BenchmarkTest00006") != null) {
46     param = request.getHeader("BenchmarkTest00006");
47 }
48
49 // URL Decode the header value since req.getHeader() doesn't. Unlike req.getParameter().
```

ProcessBuilder.command()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00006.java:66

```
63
64 ProcessBuilder pb = new ProcessBuilder();
65

66 pb.command(argList);

67
68 try {
69     Process p = pb.start();
```

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00007.java:61

Level	Medium
Status	Confirmed

```
58 Runtime r = Runtime.getRuntime();
59
60 try {

61     Process p = r.exec(args, argsEnv);

62     org.owasp.benchmark.helpers.Utils.printOSCommandResults(p, response);
63 } catch (IOException e) {
64     System.out.println("Problem executing cmdi - TestCase");
```

Trace

HttpServletRequest.getHeader()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00007.java:46

```
43
44 String param = "";
45 if (request.getHeader("BenchmarkTest00007") != null) {
```

```
46     param = request.getHeader("BenchmarkTest00007");
47 }
48
49 // URL Decode the header value since req.getHeader() doesn't. Unlike req.getParameter().
```

Runtime.exec()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00007.java:61

```
58 Runtime r = Runtime.getRuntime();
59
60 try {
61     Process p = r.exec(args, argsEnv);
62     org.owasp.benchmark.helpers.Utils.printOSCommandResults(p, response);
63 } catch (IOException e) {
64     System.out.println("Problem executing cmdi - TestCase");
```

CWE-676

Hardcoded password (Java)

Description

Password is hardcoded. This may lead to an application data compromise.
Eliminating security risks related to hardcoded passwords is extremely difficult. These passwords are at least accessible to every developer of the application. Moreover, after the application is installed, removing password from its code is possible only via an update. Constant strings are easily extracted from the compiled application by decompilers. Therefore, an attacker does not necessarily need to have an access to the source code to know the parameters of the special account. If these parameters become known to an attacker, system administrators will be forced either to neglect the safety, or to restrict the access to the application.

Example

In the following example, the connection to the database is established with a hardcoded password:
DriverManager.getConnection(url, "user", "pass");
In case of a mobile application, security threat is even higher, considering the risk of the device loss.
In the following example, the mobile application uses hardcoded parameters to

display the protected page using WebView:

```
webview.setWebViewClient(new WebViewClient() {
   public void onReceivedHttpAuthRequest(WebView view, HttpAuthHandler handler,
String host, String realm) {
      handler.proceed("user", "pass");
   }
});
```

As in the previous example, the code will work, but anyone who has access to the application source code or bytecode will be able to learn the values of username and password for the special account.

Recommendations

- Store not passwords but values of cryptographically secure hash function from the password. Use specialized hash functions designed for this purpose (bcrypt, scrypt). Use salt obtained from cryptographically secure pseudorandom number generator to resist attacks which use rainbow tables.
- If the hardcoded password is used for the initial authorization, provide the special authentication mode for this purpose in which the user is required to provide his/her own unique password.
- Store authentication information in an encrypted form in a separate configuration file or in a database. Secure the encryption key. If encryption is not possible, limit the access to the repository as much as possible.
- For secure password storage on the platforms using the SQLite database (including Android), use the SQLCipher extension.

Links

1. Use of hard-coded password
2. CWE-259: Use of Hard-coded Password
3. OWASP Top 10 2017-A2-Broken Authentication
4. OWASP Top 10 2017-A3-Sensitive Data Exposure
5. OWASP Top 10 2013-A5-Security Misconfiguration
6. OWASP Top 10 2013-A6-Sensitive Data Exposure
7. Handling passwords used for auth in source code - stackoverflow.com
8. How to securely hash passwords? - security.stackexchange.com
9. CWE-798: Use of Hard-coded Credentials
10. CWE CATEGORY: OWASP Top Ten 2017 Category A2 - Broken Authentication
11. CWE CATEGORY: OWASP Top Ten 2017 Category A6 - Security Misconfiguration

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/HibernateUtil.java:154

Level	Medium
Status	Confirmed

```
151 tx = session.beginTransaction();
152 User user = new User();
153 user.setName("User1");

154 user.setPassword("P455w0rd");

155 user.setHobbyId(2);
156 session.save(user);
157 session.flush();
```

BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/LDAPServer.java:110

Level	Medium
Status	Confirmed

```
107 LDAPManager emd = new LDAPManager();
108 LDAPPerson ldapP = new LDAPPerson();
109 ldapP.setName("foo");

110 ldapP.setPassword("MrFooPa$$word");

111 ldapP.setAddress("AddressForFoo #345");
112
113 emd.insert(ldapP);
```

CWE-250

CWE-131

Missing authorization (Java)

Description

The software does not perform an authorization check when an actor attempts to access a resource or perform an action.
When access control checks are not applied, this can lead to a wide range of problems, including information exposures, denial of service, and arbitrary code execution.
Missing authorization weaknesses may arise when a single-user application is ported to a multi-user environment.

Example

In the following example, the application automatically allows the user to send a request without asking him for a username and password:

```
Connection conn = DriverManager.getConnection(url, "user", "password123");
Statement statement = conn.createStatement();
String sql = "SELECT * FROM developers";
resultSet = statement.executeQuery(sql);
```

Recommendations

- It is recommended to implement an access control mechanism everywhere to guarantee authorized user access to the requested object.

Links

1. CWE-862: Missing Authorization

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/helpers/HibernateUtil.java:74

Level Medium

Status Confirmed

```
71    Class.forName("org.hsqldb.jdbcDriver");
72    //                              System.out.println("Driver Loaded.");
73    String url = "jdbc:hsqldb:benchmarkDataBase;sql.enforce_size=false";

74    conn = DriverManager.getConnection(url, "sa", "");
```

```
//                              System.out.println("Got Connection.");
    st = conn.createStatement();
} catch (SQLException | ClassNotFoundException e) {
```

CWE-311

CWE-807

Path manipulation (Java)

Description

Using data from an untrusted source when working with the file system may give an attacker access to important system files.
By manipulating variables that reference files with “dot-dot-slash (../)” sequences and its variations or by using absolute file paths, it may be possible to access arbitrary files and directories stored on file system including application source code or configuration and critical system files.

Example

In the following example, the application uses the value of the HTTP request parameter to specify the name of the file that is to be deleted. An attacker can set the string ../../tomcat/conf/server.xml as a parameter and thus delete the configuration file.

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" + rName);
rFile.delete()
```

Recommendations

- Create a white list of acceptable names from which the user can choose. Do not use values entered by the user without validation.

Links

1. OWASP Top 10 2017-A1-Injection
2. OWASP Top 10 2017-A5-Broken Access Control
3. OWASP Top 10 2013-A4-Insecure Direct Object References
4. CWE-73: External Control of File Name or Path
5. Path Traversal - OWASP
6. CWE CATEGORY: OWASP Top Ten 2017 Category A1 - Injection
7. CWE-23
8. CWE-36

9. Restrict path access to prevent path traversal
10. A01:2021 - Broken Access Control
11. CWE-35: Path Traversal
12. A03:2021 - Injection

Vulnerability Entries

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00001.java:71

Level Medium

Status Confirmed

```
68
69 try {
70     fileName = org.owasp.benchmark.helpers.Utils.TESTFILES_DIR + param;
71     fis = new java.io.FileInputStream(new java.io.File(fileName));
72     byte[] b = new byte[1000];
73     int size = fis.read(b);
74     response.getWriter()
```

Trace

HttpServletRequest.getCookies()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00001.java:54

```
51 // some code
52 response.setContentType("text/html;charset=UTF-8");
53
54 javax.servlet.http.Cookie[] theCookies = request.getCookies();
55
56 String param = "noCookieValueSupplied";
57 if (theCookies != null) {
```

new File()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00001.java:71

```
68
69 try {
70    fileName = org.owasp.benchmark.helpers.Utils.TESTFILES_DIR + param;

71    fis = new java.io.FileInputStream(new java.io.File(fileName));

72    byte[] b = new byte[1000];
73    int size = fis.read(b);
74    response.getWriter()
```

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00002.java:72

Level	Medium
Status	Confirmed

```
69 try {
70    fileName = org.owasp.benchmark.helpers.Utils.TESTFILES_DIR + param;
71

72    fos = new java.io.FileOutputStream(fileName, false);

73    response.getWriter()
74            .println(
75                    "Now ready to write to file: "
```

Trace

HttpServletRequest.getCookies()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00002.java:54

```
51 // some code
52 response.setContentType("text/html;charset=UTF-8");
53
```

```
54 javax.servlet.http.Cookie[] theCookies = request.getCookies();

55
56 String param = "noCookieValueSupplied";
57 if (theCookies != null) {
```

new FileOutputStream()

BenchmarkJava-master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00002.java:72

```
69 try {
70     fileName = org.owasp.benchmark.helpers.Utils.TESTFILES_DIR + param;
71

72     fos = new java.io.FileOutputStream(fileName, false);

73     response.getWriter()
74         .println(
75             "Now ready to write to file: "
```

03

About CWE/SANS Top 25 2023

The report contains vulnerability classification by **CWE/SANS Top 25 2023**. CWE/SANS Top 25 2023 is a joint project by SANS Institute, MITRE, and security specialists throughout the globe. This framework details 25 major errors leading to serious vulnerabilities in software, alongside remediation advice.

Note that some vulnerabilities may belong to the number of categories or to none at all. To see the full list of vulnerabilities, choose the **By severity** classification method.

CWE-787

Out-of-bounds Write

The software writes data past the boundaries of the intended buffer. This can result in corruption of data, a crash, or malicious code execution.

CWE-79

Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)

The software does not neutralize, or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

CWE-89

Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)

The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

CWE-416

Use After Free

The use of previously-freed memory can have any number of adverse consequences, ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the flaw.

CWE-78

Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)

The software constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.

CWE-125

Out-of-bounds Read

The software reads data past the boundaries of the intended buffer. Typically, this can allow attackers to read sensitive information from other memory locations or cause a crash. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent read operation then produces undefined or unexpected results.

CWE-352

Cross-Site Request Forgery (CSRF)

The web application does not, or cannot sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request. This enables an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc. and can result in exposure of data or unintended code execution.

CWE-20

Improper Input Validation

The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly. When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

CWE-22

Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)

The software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.

CWE-434

Unrestricted Upload of File with Dangerous Type

The software allows to upload or transfer files of dangerous types (e.g., .asp or .php) that can be automatically processed within the product's environment.

CWE-862

Missing Authorization

The software does not perform an authorization check when an actor attempts to access a resource or perform an action. When access control checks are not applied, users are able to access data or perform actions that they should not be allowed to perform. This can lead to a wide range of problems, including information exposure, denial of service, and arbitrary code execution.

CWE-287

Improper Authentication

When an actor claims to have a given identity, the software does not prove or insufficiently proves that the claim is correct. This weakness can lead to the exposure of resources or functionality to unintended actors, possibly providing attackers with sensitive information, or even execute arbitrary code.

CWE-502

Deserialization of Untrusted Data

The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.

CWE-476

NULL Pointer Dereference

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

CWE-190

Integer Overflow or Wraparound

An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs.

CWE-77

Improper Neutralization of Special Elements used in a Command (Command Injection)

The software constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component. If a malicious user injects a character (such as a semi-colon) that delimits the end of one command and the beginning of another, it may be possible to then insert an entirely new and unrelated command that was not intended to be executed.

CWE-119

Improper Restriction of Operations within the Bounds of a Memory Buffer

The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer that may be associated with other variables, data structures, or internal program data. As a result, an attacker may be able to execute arbitrary code, alter the intended control flow, read sensitive information, or cause the system to crash.

CWE-918

Server-Side Request Forgery (SSRF)

The web server receives a URL or similar request from an upstream component and retrieves the contents of this URL, but it does not sufficiently ensure that the request is being sent to the expected destination. By providing URLs to unexpected hosts or ports, attackers can make it appear that the server is sending the request, possibly bypassing access controls such as firewalls that prevent the attackers from accessing the URLs directly.

CWE-362

Concurrent Execution using Shared Resource with Improper Synchronization (Race Condition)

The product contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence. This can have security

CWE-798

Use of Hard-coded Credentials

The software contains hard-coded credentials, such as a password or cryptographic key. Hard-coded credentials typically create a significant hole that allows an attacker to bypass the authentication that has been configured by the software administrator.

CWE-306

Missing Authentication for Critical Function

The software does not perform any authentication for functionality that requires a provable user identity or consumes a significant amount of resources. Exposing critical functionality essentially provides an attacker with the privilege level of that functionality.

CWE-269

Improper Privilege Management

The product does not properly assign, modify, track, or check privileges for an actor, creating an unintended sphere of control for that actor.

implications when the expected synchronization is in security-critical code, such as recording whether a user is authenticated or modifying important state information that should not be influenced by an outsider.

CWE-94

Improper Control of Generation of Code (Code Injection)

The product constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment. When a product allows a user's input to contain code syntax, it might be possible for an attacker to craft the code in such a way that it will alter the intended control flow of the product. Such an alteration could lead to arbitrary code execution.

CWE-276

Incorrect Default Permissions

During installation, installed file permissions are set to allow anyone to modify those files.

CWE-863

Incorrect Authorization

The software performs an authorization check when an actor attempts to access a resource or perform an action, but it does not correctly perform the check. This allows attackers to bypass intended access restrictions.