Snyk Code Report

40 high issues **46** medium issues

22 low issues

SCAN COVERAGE

Java files: **347** XML files: **6** HTML files: **69**

JavaScript files: **95**

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeUpdate, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson3.java (line : 63)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson3.java

```
53:33  public AttackResult completed( @RequestParam Strin        SOURCE  0
53:33  public AttackResult completed( @RequestParam String quer          1
54:28  return injectableQuery( query );                                  2
57:42  protected AttackResult injectableQuery( String query ){           3
63:33  statement.executeUpdate( query );                                 4
 63:9  statement.executeUpdate( query);                          SINK    5
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeUpdate, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson4.java (line : 62)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson4.java

```
54:33  public AttackResult completed( @RequestParam Strin          SOURCE  0
54:33  public AttackResult completed( @RequestParam String quer            1
55:28  return injectableQuery( query );                                    2
58:42  protected AttackResult injectableQuery( String query ){             3
62:33  statement.executeUpdate( query );                                   4
 62:9  statement.executeUpdate( query);                            SINK    5
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; – ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeUpdate, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java (line : 158)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java

```
59:33  public AttackResult completed( @RequestParam Strin                    SOURCE  0
59:33  public AttackResult completed( @RequestParam String name                      1
60:43  return injectableQueryConfidentiality( name, auth_tan);                       2
63:57  protected AttackResult injectableQueryConfidentiality( S                      3
66:9   "SELECT * FROM employees WHERE last_name = '"                                 4
65:12  String query =                                                                5
                      "SELECT * FROM employees WHERE
                          + name
                          + "' AND auth_tan = '"
                          + auth_tan
                          + "'";
77:25  log(connection, query);                                                       6
147:49 public static void log(Connection connection, String act                      7
148:14 action = action. replace('\'', '"');                                          8
148:14 action = action.replace( '\'', '"');                                          9
148:5  action = action.replace('\'', '"');                                           10
154:9  "INSERT INTO access_log (time, action) VALUES ('" + tim                       11
154:9  "INSERT INTO access_log (time, action) VALUES ('" + tim                       12
153:12 String logQuery =                                                             13
                      "INSERT INTO access_log (time,
158:31 statement.executeUpdate( logQuery);                                           14
158:7  statement.executeUpdate( logQuery);                                   SINK    15
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeUpdate, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson9.java (line : 75)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson9.java

```
60:33  public AttackResult completed( @RequestParam Strin        SOURCE  0
60:33  public AttackResult completed( @RequestParam String name          1
61:37  return injectableQueryIntegrity( name, auth_tan);                 2
64:51  protected AttackResult injectableQueryIntegrity( String           3
67:9   "SELECT * FROM employees WHERE last_name = '"                     4
```

```
66:12  String query =                                                    5
                       "SELECT * FROM employees WHERE
                           + name
                           + "' AND auth_tan = '"
                           + auth_tan
                           + "'";
75:45  SqlInjectionLesson8.log(connection, query);                       6
```

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java

```
147:49 public static void log(Connection connection, String act          7
148:14 action = action.replace('\'', '"');                               8
148:14 action = action.replace('\'', '"');                               9
148:5  action = action.replace('\'', '"');                               10
154:9  "INSERT INTO access_log (time, action) VALUES ('" + tim           11
154:9  "INSERT INTO access_log (time, action) VALUES ('" + tim           12
153:12 String logQuery =                                                 13
                       "INSERT INTO access_log (time,
158:31 statement.executeUpdate(logQuery);                                14
158:7  statement.executeUpdate(logQuery);                    SINK        15
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.

- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson2.java (line : 65)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson2.java

```
58:33  public AttackResult completed(@RequestParam Strin        SOURCE  0
58:33  public AttackResult completed(@RequestParam String quer          1
59:28  return injectableQuery(query);                                   2
62:42  protected AttackResult injectableQuery(String query){            3
65:50  ResultSet results = statement.executeQuery(query);               4
65:27  ResultSet results =statement.executeQuery(query);        SINK    5
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain

access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java (line : 78)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java

```
59:33  public AttackResult completed( @RequestParam Strin        SOURCE  0
59:33  public AttackResult completed( @RequestParam String name          1
60:43  return injectableQueryConfidentiality( name, auth_tan);           2
63:57  protected AttackResult injectableQueryConfidentiality( S          3
```

```
66:9    "SELECT * FROM employees WHERE last_name = '"                    4
65:12   String query =                                                   5
                        "SELECT * FROM employees WHERE
                            + name
                            + "' AND auth_tan = '"
                            + auth_tan
                            + "'";
77:25   log(connection, query);                                          6
147:49  public static void log(Connection connection, String act         7
78:52   ResultSet results = statement.executeQuery( query);             8
78:29   ResultSet results = statement.executeQuery( query);   SINK      9
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallenge.java (line : 69)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallenge.java

```
57:7   @RequestParam String username_reg,                                  SOURCE  0
57:7   @RequestParam String username_reg,                                          1
61:48  AttackResult attackResult = checkArguments(username_reg                     2
93:39  private AttackResult checkArguments(String username_reg                      3
67:13  "select userid from sql_challenge_users where userid =                      4
67:13  "select userid from sql_challenge_users where userid =                      5
66:16  String checkUserQuery =                                                     6
                          "select userid from sql_cha
69:54  ResultSet resultSet = statement.executeQuery(checkUserQu                   7
69:31  ResultSet resultSet =statement.executeQuery(checkUs                 SINK    8
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5.java (line : 80)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5.java

```
70:33  public AttackResult completed(String query){        SOURCE  0
70:33  public AttackResult completed(String query){                1
72:28  return injectableQuery(query);                              2
75:42  protected AttackResult injectableQuery(String query){       3
80:32  statement.executeQuery(query);                              4
 80:9  statement.executeQuery(query);                      SINK    5
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson10.java (line : 71)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson10.java

```
58:33  public AttackResult completed( @RequestParam Strin      SOURCE  0
58:33  public AttackResult completed( @RequestParam String acti       1
```

```
59:40  return injectableQueryAvailability(action_string);                    2
62:54  protected AttackResult injectableQueryAvailability(Stri               3
64:20  String query = "SELECT * FROM access_log WHERE action LI              4
64:20  String query = "SELECT * FROM access_log WHERE action LI              5
64:12  String query = "SELECT * FROM access_log WHERE action LI              6
71:52  ResultSet results = statement.executeQuery(query);                    7
71:29  ResultSet results = statement.executeQuery(query);          SINK      8
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson9.java (line : 76)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson9.java

```
60:33  public AttackResult completed(@RequestParam Strin                    SOURCE  0
60:33  public AttackResult completed(@RequestParam String name                      1
61:37  return injectableQueryIntegrity(name, auth_tan);                             2
64:51  protected AttackResult injectableQueryIntegrity(String                       3
67:9   "SELECT * FROM employees WHERE last_name = '"                                4
66:12  String query =                                                               5
                    "SELECT * FROM employees WHERE
                        + name
                        + "' AND auth_tan = '"
                        + auth_tan
                        + "'";
76:52  ResultSet results = statement.executeQuery(query);                           6
76:29  ResultSet results = statement.executeQuery(query);                   SINK    7
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into prepareStatement, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5b.java (line : 65)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5b.java

```
56:7   @RequestParam String userid, @RequestParam String          SOURCE  0
56:7   @RequestParam String userid, @RequestParam String login_           1
58:41  return injectableQuery(login_count, userid);                       2
61:62  protected AttackResult injectableQuery(String login_coun           3
62:26  String queryString = "SELECT * From user_data WHERE Logi           4
62:12  String queryString = "SELECT * From user_data WHERE Logi           5
```

```
66:15  queryString, ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSe   6
65:11  connection.prepareStatement(                    SINK      7
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; – ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

> Unsanitized input from an HTTP parameter flows into prepareStatement, where it is used in an SQL query. This may result in an SQL Injection vulnerability.
>
> Found in:
> **src/main/java/org/owasp/webgoat/lessons/challenges/challenge5/Assignment5.java (line : 60)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/challenges/challenge5/Assignment5.java

51:7	`@RequestParam String username_login,` `@RequestPara`	SOURCE	0
51:7	`@RequestParam String username_login,` `@RequestParam Strir`		1
61:15	`"select password from challenge_users where userid = '"`		2
60:11	`connection.prepareStatement(`	SINK	3

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from unverified JWT claims flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpoint.java (line : 104)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpoint.java

```
101:53  final String kid = (String)header.get("kid");                    SOURCE  0
101:53  final String kid = (String)header.get("kid");                            1
101:44  final String kid =(String) header.get("kid");                            2
101:38  final String(kid = (String) header.get("kid");                           3
107:39  "SELECT key FROM jwt_keys WHERE id = '" + kid+ "'");                     4
107:39  "SELECT key FROM jwt_keys WHERE id = '" + kid + "'");                    5
104:31  connection                                                       SINK    6
                                                    .create
                                                    .execut
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.

- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 75)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java

```
75:45  public static final String JWT_PASSWORD = Text
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.

- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 65)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java

```
65:41  public static final String PASSWORD = "bm5nhSk
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.

- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 66)**

Data Flow

```
66:46  private static final String JWT_PASSWORD ="bm
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 57)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java

```
57:7  TextCodec.BASE64.encode( SECRETS[new Random()
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/challenges/SolutionConstants.java (line : 34)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/challenges/SolutionConstants.java

```
34:21   String PASSWORD = "!!webgoat_admin_1234!!";
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/challenges/SolutionConstants.java (line : 36)**

Data Flow

```
36:32  String ADMIN_PASSWORD_LINK = "375afe1104f4a487
```

SOURCE | SINK | 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.

Found in: **src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionAC.java (line : 32)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionAC.java

```
32:53  public static final String PASSWORD_SALT_SIMPI
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

Avoid hardcoding values that are meant to be secret. Found hardcoded secret.

Found in: **src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionAC.java (line : 33)**

Data Flow

```
33:52  public static final String PASSWORD_SALT_ADMIN
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 159)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java

```
159:19  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASS
```

SOURCE SINK 0

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

> The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 184)**

Data Flow

```
184:19  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASS
```

SOURCE SINK 0

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

> The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 207)**

Data Flow

```
207:19  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASS
```

SOURCE SINK 0

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

> The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 113)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java

```
113:17  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASS
```

SOURCE SINK 0

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

> The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 140)**

Data Flow

140:11	`Jwts.parser().setSigningKey(JWT_PASSWORD).par`	SOURCE SINK	0

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6a.java (line : 74)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6a.java

```
56:33  public AttackResult completed(@RequestParam(value                    SOURCE  0
56:33  public AttackResult completed(@RequestParam(value = "us                      1
57:28  return injectableQuery(userId);                                              2
62:39  public AttackResult injectableQuery(String accountName)                      3
66:15  query = "SELECT * FROM user_data WHERE last_name = '" +                      4
66:15  query = "SELECT * FROM user_data WHERE last_name = '" +                      5
 66:7  query = "SELECT * FROM user_data WHERE last_name = '" +                      6
74:52  ResultSet results = statement.executeQuery(query);                           7
74:29  ResultSet results = statement.executeQuery(query);                   SINK    8
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.

- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/SqlOnlyInputValidation.java (line : 51)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/SqlOnlyInputValidation.java

```
47:30  public AttackResult attack(@RequestParam("userid_          SOURCE  0
47:30  public AttackResult attack(@RequestParam("userid_sql_on           1
48:9   if (userId.contains(" ")) {                                      2
51:58  AttackResult attackResult = lesson6a.injectableQuery(us           3
```

src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6a.java

```
62:39  public AttackResult injectableQuery(String accountName)           4
66:15  query = "SELECT * FROM user_data WHERE last_name = '" +           5
```

```
66:15  query = "SELECT * FROM user_data WHERE last_name = '" +
66:7   query = "SELECT * FROM user_data WHERE last_name = '" +
74:52  ResultSet results = statement.executeQuery(query);
74:29  ResultSet results = statement.executeQuery(query);   SINK
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

> Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5a.java (line : 67)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5a.java

```
55:7   @RequestParam String account, @RequestParam Strin        SOURCE  0
55:7   @RequestParam String account, @RequestParam String opera         1
56:28  return injectableQuery( account + " " + operator + " " +         2
56:28  return injectableQuery( account + " " + operator + " " +         3
56:28  return injectableQuery( account + " " + operator + " " +         4
56:28  return injectableQuery( account + " " + operator + " " )+        5
56:28  return injectableQuery( account + " " + operator + " " +         6
59:42  protected AttackResult injectableQuery( String accountNa         7
63:11  "SELECT * FROM user_data WHERE first_name = 'John' and           8
63:11  "SELECT * FROM user_data WHERE first_name = 'John' and           9
62:7   query =                                                          10
67:52  ResultSet results = statement.executeQuery( query );            11
67:29  ResultSet results = statement.executeQuery( query);      SINK    12
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/SqlOnlyInputValidationOnKeywords.java (line : 57)

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/SqlOnlyInputValidationOnKeywords.java

```
52:7   @RequestParam("userid_sql_only_input_validation_c        SOURCE  0
52:7   @RequestParam("userid_sql_only_input_validation_on_keyw          1
53:14  userId = userId. toUpperCase().replace("FROM", "").replac       2
53:14  userId = userId.toUpperCase( ).replace("FROM", "").replac       3
53:14  userId = userId.toUpperCase().replace( "FROM", "").replac       4
53:14  userId = userId.toUpperCase().replace("FROM", "").replac        5
```

```
53:5   userId = userId.toUpperCase().replace("FROM", "").repla   6
54:9   if (userId.contains(" ")) {                               7
57:58  AttackResult attackResult = lesson6a.injectableQuery(us   8
```

src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6a.java

```
62:39  public AttackResult injectableQuery(String accountName)  9
66:15  query = "SELECT * FROM user_data WHERE last_name = '" + a  10
66:15  query = "SELECT * FROM user_data WHERE last_name = '" + a  11
66:7   query = "SELECT * FROM user_data WHERE last_name = '" +   12
74:52  ResultSet results = statement.executeQuery(query);        13
74:29  ResultSet results = statement.executeQuery(query);  SINK  14
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

SQL Injection

SNYK-CODE | CWE-89

Unsanitized input from an HTTP parameter flows into prepareStatement, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Found in: **src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/Servers.java (line : 72)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/Servers.java

```
67:28  public List<Server> sort( @RequestParam String col                SOURCE  0
67:28  public List<Server> sort( @RequestParam String column) th                 1
73:15  "select id, hostname, ip, mac, status, description from                   2
72:11  connection.prepareStatement(                                      SINK    3
```

Fix Analysis

Details

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and possibly gain access to other systems within the network. This is one of the most commonly exploited categories of vulnerability, but can largely be avoided through good coding practices.

Best practices for prevention

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a very strict allowlist of permitted characters, avoiding special characters such as `? & / < > ; - ' " \` and spaces. Use a vendor-supplied escaping routine if possible.

- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for particular tasks.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/lessontemplate/SampleAttack.java (line : 44)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/lessontemplate/SampleAttack.java

```
44:24   String secretValue = "secr37Value";
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or

other unforeseen circumstances (such as new hardware) in the future.

XML External Entity (XXE) Injection

SNYK-CODE | CWE-611

Unsanitized input from the HTTP request body flows into createXMLStreamReader, which allows expansion of external entity references. This may result in a XXE attack leading to the disclosure of confidential data or denial of service.

Found in: **src/main/java/org/owasp/webgoat/lessons/xxe/BlindSendFileAssignment.java (line : 96)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/xxe/BlindSendFileAssignment.java

```
87:34  public AttackResult addComment(@RequestBody Strin        SOURCE  0
87:34  public AttackResult addComment(@RequestBody String comm          1
91:9   if (commentStr.contains(fileContentsForUser)) {                  2
96:43  Comment comment = comments.parseXml(commentStr);                  3
```

src/main/java/org/owasp/webgoat/lessons/xxe/CommentsCache.java

```
96:30  protected Comment parseXml(String xml)throws JAXBExcep           4
105:45 var xsr = xif.createXMLStreamReader(new StringReader(xml         5
105:41 var xsr = xif.createXMLStreamReader(new StringReader(xml         6
105:15 var xsr =xif.createXMLStreamReader(new StringReade       SINK    7
```

Fix Analysis

Details

For convenience, XML documents can use system identifiers to enable access to stored content, whether local or remote. The XML processor then uses the system identifier to access the resource rather than using the URI. When this weakness exists, the application permits user-supplied data, which could include the address of an XML external identity, to be passed directly to the XML parser. The application will then attempt to retrieve documents from outside of secure, controlled areas.

Attackers can exploit this weakness to expose sensitive data, execute port scanning on the server side, or launch a denial-of-service attack (DoS) such as Billion Laughs.

Best practices for prevention

- When possible, disable loading of data from external entities. The method of doing this will vary based on the language and XML parser being used.
- Use a local, static document type definitions (DTDs) and ensure that external DTDs are disallowed entirely.
- If user input cannot be avoided, perform validation against an allowlist of possible data sources. However, as long as external DTDs are allowed, XML code remains inherently vulnerable to attacks exploiting this weakness.

XML External Entity (XXE) Injection

SNYK-CODE | CWE-611

> Unsanitized input from the HTTP request body flows into createXMLStreamReader, which allows expansion of external entity references. This may result in a XXE attack leading to the disclosure of confidential data or denial of service.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/xxe/ContentTypeAssignment.java (line : 75)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/xxe/ContentTypeAssignment.java

```
62:7   @RequestBody String commentStr,                          SOURCE   0
62:7   @RequestBody String commentStr,                                   1
75:45  Comment comment = comments.parseXml(commentStr);                  2
```

src/main/java/org/owasp/webgoat/lessons/xxe/CommentsCache.java

```
96:30   protected Comment parseXml(String xml)throws JAXBExcep    3
105:45  var xsr = xif.createXMLStreamReader(new StringReader(xml  4
105:41  var xsr = xif.createXMLStreamReader(new StringReader(xm   5
105:15  var xsr =xif.createXMLStreamReader(new StringReade  SINK  6
```

Fix Analysis

Details

For convenience, XML documents can use system identifiers to enable access to stored content, whether local or remote. The XML processor then uses the system identifier to access the resource rather than using the URI. When this weakness exists, the application permits user-supplied data, which could include the address of an XML external identity, to be passed directly to the XML parser. The application will then attempt to retrieve documents from outside of secure, controlled areas.

Attackers can exploit this weakness to expose sensitive data, execute port scanning on the server side, or launch a denial-of-service attack (DoS) such as Billion Laughs.

Best practices for prevention

- When possible, disable loading of data from external entities. The method of doing this will vary based on the language and XML parser being used.
- Use a local, static document type definitions (DTDs) and ensure that external DTDs are disallowed entirely.
- If user input cannot be avoided, perform validation against an allowlist of possible data sources. However, as long as external DTDs are allowed, XML code remains inherently vulnerable to attacks exploiting this weakness.

XML External Entity (XXE) Injection

SNYK-CODE | CWE-611

> Unsanitized input from the HTTP request body flows into createXMLStreamReader, which allows expansion of external entity references.

This may result in a XXE attack leading to the disclosure of confidential data or denial of service.

Found in: **src/main/java/org/owasp/webgoat/lessons/xxe/SimpleXXE.java (line : 76)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/xxe/SimpleXXE.java

```
73:68  public AttackResult createNewComment(HttpServletRe          SOURCE  0
73:68  public AttackResult createNewComment(HttpServletRequest            1
76:39  var comment = comments.parseXml(commentStr);                       2
```

src/main/java/org/owasp/webgoat/lessons/xxe/CommentsCache.java

```
96:30   protected Comment parseXml(String xml)throws JAXBExcept           3
105:45  var xsr = xif.createXMLStreamReader(new StringReader(xml          4
105:41  var xsr = xif.createXMLStreamReader(new StringReader(xml          5
105:15  var xsr =xif.createXMLStreamReader(new StringReade        SINK    6
```

Fix Analysis

Details

For convenience, XML documents can use system identifiers to enable access to stored content, whether local or remote. The XML processor then uses the system identifier to access the resource rather than using the URI. When this weakness exists, the application permits user-supplied data, which could include the address of an XML external identity, to be passed directly to the XML parser. The application will then attempt to retrieve documents from outside of secure, controlled areas.

Attackers can exploit this weakness to expose sensitive data, execute port scanning on the server side, or launch a denial-of-service attack (DoS) such as Billion Laughs.

Best practices for prevention

- When possible, disable loading of data from external entities. The method of doing this will vary based on the language and XML parser being used.
- Use a local, static document type definitions (DTDs) and ensure that external DTDs are disallowed entirely.

- If user input cannot be avoided, perform validation against an allowlist of possible data sources. However, as long as external DTDs are allowed, XML code remains inherently vulnerable to attacks exploiting this weakness.

Deserialization of Untrusted Data

SNYK-CODE | CWE-502

Unsanitized input from an HTTP parameter flows into fromXML, where it is used to deserialize an object. This may result in an Unsafe Deserialization vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/vulnerablecomponents/VulnerableComponentsLesson.java (line : 57)

Data Flow

src/main/java/org/owasp/webgoat/lessons/vulnerablecomponents/VulnerableComponentsLesson.java

```
40:47  public @ResponseBody AttackResult completed(@Requ                    SOURCE  0
40:47  public @ResponseBody AttackResult completed(@RequestPar                      1
50:13  payload                                                                      2
50:13  payload                                                "+", "")              3
                                .replace(
50:13  payload                                                                      4
                                .replace("+", "")
                                .replace(                                "\r", "")
50:13  payload                                                                      5
                                .replace("+", "")
                                .replace("\r", "")
                                .replace(                                "\n", "")
50:13  payload                                                                      6
                                .replace("+", "")
                                .replace("\r", "")
                                .replace("\n", "")
                                .replace(                                "> ", ">")
```

```
50:13  payload                                                    " <", "<"   7
                              .replace("+", "")
                              .replace("\r", "")
                              .replace("\n", "")
                              .replace("> ", ">")
                              .replace(
49:9   payload =                                                            8
57:43  contact = (Contact) xstream.fromXML( payload );                      9
57:27  contact = (Contact) xstream.fromXML( payload);          SINK         10
```

Fix Analysis

Details

Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called deserialization. Serialization is commonly used for communication (sharing objects between multiple hosts) and persistence (store the object state in a file or a database). It is an integral part of popular protocols like *Remote Method Invocation (RMI)*, *Java Management Extension (JMX)*, *Java Messaging System (JMS)*, *Action Message Format (AMF)*, *Java Server Faces (JSF) ViewState*, etc.

Deserialization of untrusted data (CWE-502), is when the application deserializes untrusted data without sufficiently verifying that the resulting data will be valid, letting the attacker to control the state or the flow of the execution.

Java deserialization issues have been known for years. However, interest in the issue intensified greatly in 2015, when classes that could be abused to achieve remote code execution were found in a popular library (Apache Commons Collection). These classes were used in zero-days affecting IBM WebSphere, Oracle WebLogic and many other products.

An attacker just needs to identify a piece of software that has both a vulnerable class on its path, and performs deserialization on untrusted data. Then all they need to do is send the payload into the deserializer, getting the command executed.

> Developers put too much trust in Java Object Serialization. Some even de-serialize objects pre-authentication. When deserializing an Object in Java you typically cast it to an expected type, and therefore Java's strict type system will ensure you only get valid object trees. Unfortunately, by the time the type checking happens, platform code has already created and executed significant logic. So, before the final type is checked a lot of code is executed from the readObject() methods of various objects, all of which is out of the developer's control. By combining the readObject() methods of various classes which are available on the classpath of the vulnerable application an attacker can execute functions (including calling Runtime.exec() to execute local OS commands).

Deserialization of Untrusted Data

SNYK-CODE | CWE-502

Unsanitized input from an HTTP parameter flows into java.io.ObjectInputStream, where it is used to deserialize an object. This may result in an Unsafe Deserialization vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/deserialization/InsecureDeserializationTask.java (line : 58)

Data Flow

src/main/java/org/owasp/webgoat/lessons/deserialization/InsecureDeserializationTask.java

```
49:33  public AttackResult completed(@RequestParam Strin                    SOURCE  0
49:33  public AttackResult completed(@RequestParam String toke                      1
55:16  b64token = token.replace('-', '+').replace('_', '/');                       2
55:16  b64token = token.replace('-', '+').replace('_', '/');                       3
55:16  b64token = token.replace('-', '+').replace('_', '/');                       4
 55:5  b64token = token.replace('-', '+').replace('_', '/');                       5
58:56  new ObjectInputStream(new ByteArrayInputStream(Base64.g                     6
58:35  new ObjectInputStream(new ByteArrayInputStream(Base64.ge                    7
58:31  new ObjectInputStream(new ByteArrayInputStream(Base64.g                     8
58:13  new ObjectInputStream(new ByteArrayInputStream(Base(                SINK    9
```

Fix Analysis

Details

Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called deserialization. Serialization is commonly used for communication (sharing objects between multiple hosts) and persistence (store the object state in a file or a

database). It is an integral part of popular protocols like *Remote Method Invocation (RMI)*, *Java Management Extension (JMX)*, *Java Messaging System (JMS)*, *Action Message Format (AMF)*, *Java Server Faces (JSF) ViewState*, etc.

Deserialization of untrusted data (CWE-502), is when the application deserializes untrusted data without sufficiently verifying that the resulting data will be valid, letting the attacker to control the state or the flow of the execution.

Java deserialization issues have been known for years. However, interest in the issue intensified greatly in 2015, when classes that could be abused to achieve remote code execution were found in a popular library (Apache Commons Collection). These classes were used in zero-days affecting IBM WebSphere, Oracle WebLogic and many other products.

An attacker just needs to identify a piece of software that has both a vulnerable class on its path, and performs deserialization on untrusted data. Then all they need to do is send the payload into the deserializer, getting the command executed.

> Developers put too much trust in Java Object Serialization. Some even de-serialize objects pre-authentication. When deserializing an Object in Java you typically cast it to an expected type, and therefore Java's strict type system will ensure you only get valid object trees. Unfortunately, by the time the type checking happens, platform code has already created and executed significant logic. So, before the final type is checked a lot of code is executed from the readObject() methods of various objects, all of which is out of the developer's control. By combining the readObject() methods of various classes which are available on the classpath of the vulnerable application an attacker can execute functions (including calling Runtime.exec() to execute local OS commands).

Cross-Site Request Forgery (CSRF)

SNYK-CODE | CWE-352

> CSRF protection is disabled by disable. This allows the attackers to execute requests on a user's behalf.
>
> Found in: **src/main/java/org/owasp/webgoat/webwolf/WebSecurityConfig.java (line : 59)**

Data Flow

src/main/java/org/owasp/webgoat/webwolf/WebSecurityConfig.java

```
59:5   security.and().csrf().disable()).formLogin().
```

SOURCE SINK 0

Fix Analysis

Details

Cross-site request forgery is an attack in which a malicious third party takes advantage of a user's authenticated credentials (such as a browser cookie) to impersonate that trusted user and perform unauthorized actions. The web application server cannot tell the difference between legitimate and malicious requests. This type of attack generally begins by tricking the user with a social engineering attack, such as a link or popup that the user inadvertently clicks, causing an unauthorized request to be sent to the web server. Consequences vary: At a standard user level, attackers can change passwords, transfer funds, make purchases, or connect with contacts; from an administrator account, attackers can then make changes to or even take down the app itself.

Best practices for prevention

- Use development frameworks that defend against CSRF, using a nonce, hash, or some other security device to the URL and/or to forms.
- Implement secure, unique, hidden tokens that are checked by the server each time to validate state-change requests.
- Never assume that authentication tokens and session identifiers mean a request is legitimate.
- Understand and implement other safe-cookie techniques, such as double submit cookies.
- Terminate user sessions when not in use, including automatic timeout.
- Ensure rigorous coding practices and defenses against other commonly exploited CWEs, since cross-site scripting (XSS), for example, can be used to bypass defenses against CSRF.

Cross-Site Request Forgery (CSRF)

SNYK-CODE | CWE-352

> CSRF protection is disabled by disable. This allows the attackers to execute requests on a user's behalf.
>
> Found in: **src/main/java/org/owasp/webgoat/container/WebSecurityConfig.java (line : 80)**

Data Flow

src/main/java/org/owasp/webgoat/container/WebSecurityConfig.java

```
80:5  security.and().csrf().disable());        SOURCE SINK 0
```

Fix Analysis

Details

Cross-site request forgery is an attack in which a malicious third party takes advantage of a user's authenticated credentials (such as a browser cookie) to impersonate that trusted user and perform unauthorized actions. The web application server cannot tell the difference between legitimate and malicious requests. This type of attack generally begins by tricking the user with a social engineering attack, such as a link or popup that the user inadvertently clicks, causing an unauthorized request to be sent to the web server. Consequences vary: At a standard user level, attackers can change passwords, transfer funds, make purchases, or connect with contacts; from an administrator account, attackers can then make changes to or even take down the app itself.

Best practices for prevention

- Use development frameworks that defend against CSRF, using a nonce, hash, or some other security device to the URL and/or to forms.
- Implement secure, unique, hidden tokens that are checked by the server each time to validate state-change requests.
- Never assume that authentication tokens and session identifiers mean a request is legitimate.
- Understand and implement other safe-cookie techniques, such as double submit cookies.
- Terminate user sessions when not in use, including automatic timeout.
- Ensure rigorous coding practices and defenses against other commonly exploited CWEs, since cross-site scripting (XSS), for example, can be used to bypass defenses against CSRF.

Server-Side Request Forgery (SSRF)

SNYK-CODE | CWE-918

Unsanitized input from an HTTP header flows into exchange, where it is used as an URL to perform a request. This may result in a Server-Side Request Forgery vulnerability.

Found in:
src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignmentForgotPassword.java (line : 104)

Data Flow

src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignmentForgotPassword.java

```
70:19  String host = request.getHeader("host");                          SOURCE  0
70:19  String host = request.getHeader("host");                                  1
70:12  String host = request.getHeader("host");                                  2
72:13  && (host.contains(webWolfPort)                                            3
73:16  ||host.contains(webWolfHost))) { // User indeed changed                   4
75:29  fakeClickingLinkEmail(host, resetLink);                                   5
100:38 private void fakeClickingLinkEmail(String host, String r                  6
106:15 String.format("http://%s/PasswordReset/reset/reset-pass                   7
104:7  new RestTemplate()                                                SINK    8
                    .exchange(
```

Fix Analysis

Details

In a server-side request forgery attack, a malicious user supplies a URL (an external URL or a network IP address such as 127.0.0.1) to the application's back end. The server then accesses the URL and shares its results, which may include sensitive information such as AWS metadata, internal configuration information, or database contents with the attacker. Because the request comes from the back end, it bypasses access controls, potentially exposing information the user does not have sufficient privileges to receive. The attacker can then exploit this information to gain access, modify the web application, or demand a ransom payment.

Best practices for prevention

- Blacklists are problematic and attackers have numerous ways to bypass them; ideally, use a whitelist of all permitted domains and IP addresses.
- Use authentication even within your own network to prevent exploitation of server-side requests.
- Implement zero trust and sanitize and validate all URL and header data returning to the server from the user. Strip invalid or suspect characters, then inspect to be certain it contains a valid and expected value.
- Ideally, avoid sending server requests based on user-provided data altogether.
- Ensure that you are not sending raw response bodies from the server directly to the client. Only deliver expected responses.
- Disable suspect and exploitable URL schemas. Common culprits include obscure and little-used schemas such as `file://`, `dict://`, `ftp://`, and `gopher://`.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into html, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/lessons/challenges/js/challenge8.js (line : 18)**

Data Flow

src/main/resources/lessons/challenges/js/challenge8.js

```
7:43   $.get("challenge/8/votes/", function (votes){        SOURCE  0
7:43   $.get("challenge/8/votes/", function (votes){                1
14:31  var percent = votes[i] * 100 / totalVotes;                    2
18:42  $("#nrOfVotes" + i).html(votes[i]);                           3
18:42  $("#nrOfVotes" + i).html(votes[i]);                           4
18:37  $("#nrOfVotes" + i).html(votes[i]);                   SINK    5
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be

interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.

- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into html, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/lessons/challenges/js/challenge8.js (line : 52)**

Data Flow

```
46:50  $.get("challenge/8/vote/" + stars, function (resu        SOURCE  0
46:50  $.get("challenge/8/vote/" + stars, function (result){            1
47:13  if (result["error"]) {                                           2
52:34  $("#voteResultMsg").html(result["message"]);                     3
52:34  $("#voteResultMsg").html(result["message"]);                     4
52:29  $("#voteResultMsg").html(result["message"]);             SINK    5
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually

JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `<` and `>` can be coded as `>` in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into innerHTML, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/lessons/clientsidefiltering/js/clientSideFiltering.js (line : 38)**

Data Flow

src/main/resources/lessons/clientsidefiltering/js/clientSideFiltering.js

```
17:70  $.get("clientSideFiltering/salaries?userId=" + use      SOURCE  0
17:70  $.get("clientSideFiltering/salaries?userId=" + userId, f       1
26:29  for (var i = 0; i < result.length; i++) {                      2
27:42  html = html + '<tr id = "' + result[i].UserID + '"</tr>        3
```

```
27:42  html = html + '<tr id = "' + result[i].UserID + '"</tr>                4
28:20  html = html + '<td>' + result[i].UserID + '</td>';                     5
29:20  html = html + '<td>' + result[i].FirstName + '</td>';                  6
30:20  html = html + '<td>' + result[i].LastName + '</td>';                   7
31:20  html = html + '<td>' + result[i].SSN + '</td>';                        8
32:20  html = html + '<td>' + result[i].Salary + '</td>';                     9
33:20  html = html + '</tr>';                                                10
35:16  html = html + '</tr></table>';                                        11
38:28  newdiv.innerHTML = html;                                              12
38:28  newdiv.innerHTML = html;                                      SINK    13
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into append, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/lessons/jwt/js/jwt-final.js (line : 6)**

Data Flow

src/main/resources/lessons/jwt/js/jwt-final.js

```
5:23  }).then(function (result){                    SOURCE  0
5:23  }).then(function (result){                            1
6:28  $("#toast").append(result);                           2
6:21  $("#toast").append(result);                    SINK   3
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.

- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into append, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/lessons/csrf/js/csrf-review.js (line : 41)**

Data Flow

src/main/resources/lessons/csrf/js/csrf-review.js

```
35:40  $.get('csrf/review', function ([result,]status) {          SOURCE  0
35:40  $.get('csrf/review', function ([result,]status) {                  1
36:33  for (var i = 0; i <[result.]length; i++) {                         2
37:52  var comment = html.replace('USER',[result[]i].user);               3
38:55  comment = comment.replace('DATETIME',[result[]i].dateTime          4
39:54  comment = comment.replace('COMMENT',[result[]i].text);             5
40:52  comment = comment.replace('STARS',[result[]i].stars)               6
40:52  comment = comment.replace('STARS',[result[i].]stars)               7
40:35  comment = comment.[replace(]'STARS', result[i].stars)              8
41:35  $("#list").append([comment)];                                      9
41:28  $("#list").[append(]comment);                              SINK    10
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers

- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into append, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/lessons/xxe/js/xxe.js (line : 78)**

Data Flow

src/main/resources/lessons/xxe/js/xxe.js

```
72:37  $.get("xxe/comments", function (result, status) {        SOURCE  0
72:37  $.get("xxe/comments", function (result, status) {                1
74:29  for (var i = 0; i < result.length; i++) {                        2
75:48  var comment = html.replace('USER',result[i].user);               3
```

```
76:51  comment = comment.replace('DATETIME',result[i].dateTime                4
77:50  comment = comment.replace('COMMENT',result[i].text);                   5
77:50  comment = comment.replace('COMMENT',result[i].text);                   6
77:31  comment = comment.replace('COMMENT', result[i].text);                  7
78:29  $(field).append(comment);                                              8
78:22  $(field).append(comment);                                      SINK    9
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.

Type	Origin	Description
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

Unsanitized input from data from a remote resource flows into append, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).

Found in: **src/main/resources/lessons/xss/js/stored-xss.js (line : 40)**

Data Flow

src/main/resources/lessons/xss/js/stored-xss.js

```
35:58  $.get('CrossSiteScripting/stored-xss', function ([          SOURCE  0
35:58  $.get('CrossSiteScripting/stored-xss', function ([result            1
36:33  for (var i = 0; i <[result.]length; i++) {                          2
37:52  var comment = html.replace('USER',[result[]i].user);                3
38:55  comment = comment.replace('DATETIME',[result[]i].dateTime           4
39:54  comment = comment.replace('COMMENT',[result[]i].text);              5
39:54  comment = comment.replace('COMMENT',[result[i].]text);              6
39:35  comment = comment.[replace(]'COMMENT', result[i].text);             7
40:35  $("#list").append([comment)];                                       8
40:28  $("#list").[append(]comment);                               SINK    9
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query

parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into append, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/lessons/sqlinjection/js/assignment13.js (line : 57)**

Data Flow

src/main/resources/lessons/sqlinjection/js/assignment13.js

```
43:73  $.get("SqlInjectionMitigations/servers?column=" +          SOURCE  0
43:73  $.get("SqlInjectionMitigations/servers?column=" + column          1
45:29  for (var i = 0; i < result.length; i++) {                         2
46:45  var server = html.replace('ID',result[i].id);                     3
48:17  if (result[i].status === 'offline') {                             4
53:49  server = server.replace('HOSTNAME',result[i].hostname);            5
54:43  server = server.replace('IP',result[i].ip);                       6
55:44  server = server.replace('MAC',result[i].mac);                     7
```

```
56:52  server = server.replace('DESCRIPTION',result[i].descri   8
56:52  server = server.replace('DESCRIPTION',result[i].descri   9
56:29  server = server.replace('DESCRIPTION', result[i].descr   10
57:34  $("#servers").append(server);                            11
57:27  $("#servers").append(server);                    SINK    12
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `<` and `>` can be coded as `>` in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.

Type	Origin	Description
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

Unsanitized input from data from a remote resource flows into append, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).

Found in: **src/main/resources/lessons/jwt/js/jwt-voting.js (line : 63)**

Data Flow

src/main/resources/lessons/jwt/js/jwt-voting.js

```
43:36  $.get("JWT/votings", function ([result,]status) {                    SOURCE  0
43:36  $.get("JWT/votings", function ([result,]status) {                            1
44:29  for (var i = 0; i <[result.]length; i++) {                                   2
45:60  var voteTemplate = html.replace('IMAGE_SMALL',[result[]i                     3
53:59  voteTemplate = voteTemplate.replace(/TITLE/g,[result[]i]                     4
54:64  voteTemplate = voteTemplate.replace('INFORMATION',[resul                     5
55:61  voteTemplate = voteTemplate.replace('NO_VOTES',[result[]                     6
56:60  voteTemplate = voteTemplate.replace('AVERAGE',[result[]i                     7
56:60  voteTemplate = voteTemplate.replace('AVERAGE',[result[i]]                    8
56:41  voteTemplate = voteTemplate.[replace(]'AVERAGE', result[                     9
59:28  voteTemplate =[voteTemplate.]replace(/HIDDEN_VIEW_VOTES/                     10
59:41  voteTemplate = voteTemplate.[replace(]/HIDDEN_VIEW_VOTES/                    11
61:28  voteTemplate =[voteTemplate.]replace(/HIDDEN_VIEW_RATING/                    12
61:41  voteTemplate = voteTemplate.[replace(]/HIDDEN_VIEW_RATIN                     13
63:36  $("#votesList").append([voteTemplate]);                                      14
63:29  $("#votesList").[append(]voteTemplate);                              SINK    15
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked, malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Cross-site Scripting (XSS)

SNYK-CODE | CWE-79

> Unsanitized input from data from a remote resource flows into html, where it is used to dynamically construct the HTML page on client side. This may result in a DOM Based Cross-Site Scripting attack (DOMXSS).
>
> Found in: **src/main/resources/webgoat/static/js/goatApp/support/GoatUtils.js (line : 57)**

Data Flow

src/main/resources/webgoat/static/js/goatApp/support/GoatUtils.js

```
56:69  $.get(goatConstants.cookieService, {}, function(r          SOURCE  0
56:69  $.get(goatConstants.cookieService, {}, function(reply)·           1
57:51  $("#lesson_cookies").html(reply);                                 2
57:46  $("#lesson_cookies").html(reply);                         SINK    3
```

Fix Analysis

Details

A cross-site scripting attack occurs when the attacker tricks a legitimate web-based application or site to accept a request as originating from a trusted source.

This is done by escaping the context of the web application; the web application then delivers that data to its users along with other trusted dynamic content, without validating it. The browser unknowingly executes malicious script on the client side (through client-side languages; usually JavaScript or HTML) in order to perform actions that are otherwise typically blocked by the browser's Same Origin Policy.

Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability.

Escaping means that the application is coded to mark key characters, and particularly key characters included in user input, to prevent those characters from being interpreted in a dangerous context. For example, in HTML, `<` can be coded as `&lt` ; and `>` can be coded as `&gt` ; in order to be interpreted and displayed as themselves in text, while within the code itself, they are used for HTML tags. If malicious content is injected into an application that escapes special characters and that malicious content uses `<` and `>` as HTML tags, those characters are nonetheless not interpreted as HTML tags by the browser if they've been correctly escaped in the application code and in this way the attempted attack is diverted.

The most prominent use of XSS is to steal cookies (source: OWASP HttpOnly) and hijack user sessions, but XSS exploits have been used to expose sensitive information, enable access to privileged services and functionality and deliver malware.

Types of attacks

There are a few methods by which XSS can be manipulated:

Type	Origin	Description
Stored	Server	The malicious code is inserted in the application (usually as a link) by the attacker. The code is activated every time a user clicks the link.
Reflected	Server	The attacker delivers a malicious link externally from the vulnerable web site application to a user. When clicked,

Type	Origin	Description
		malicious code is sent to the vulnerable web site, which reflects the attack back to the user's browser.
DOM-based	Client	The attacker forces the user's browser to render a malicious page. The data in the page itself delivers the cross-site scripting data.
Mutated		The attacker injects code that appears safe, but is then rewritten and modified by the browser, while parsing the markup. An example is rebalancing unclosed quotation marks or even adding quotation marks to unquoted parameters.

Affected environments

The following environments are susceptible to an XSS attack:

- Web servers
- Application servers
- Web application environments

How to prevent

This section describes the top best practices designed to specifically protect your code:

- Sanitize data input in an HTTP request before reflecting it back, ensuring all data is validated, filtered or escaped before echoing anything back to the user, such as the values of query parameters during searches.
- Convert special characters such as `?`, `&`, `/`, `<`, `>` and spaces to their respective HTML or URL encoded equivalents.
- Give users the option to disable client-side scripts.
- Redirect invalid requests.
- Detect simultaneous logins, including those from two separate IP addresses, and invalidate those sessions.
- Use and enforce a Content Security Policy (source: Wikipedia) to disable any features that might be manipulated for an XSS attack.
- Read the documentation for any of the libraries referenced in your code to understand which elements allow for embedded HTML.

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from an HTTP parameter flows into org.springframework.util.FileCopyUtils.copy, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to read arbitrary files.

Found in: **src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java (line : 67)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java

```
51:41  public AttackResult uploadFileHandler( @RequestPar                SOURCE  0
51:41  public AttackResult uploadFileHandler( @RequestParam("up                  1
52:10  if (! file. getOriginalFilename().toLowerCase().endsWith(                 2
55:31  return processZipUpload( file);                                           3
60:41  private AttackResult processZipUpload( MultipartFile file                 4
66:53  var uploadedZipFile = tmpZipDirectory.resolve( file. getC                 5
67:26  FileCopyUtils.copy( file. getBytes(), uploadedZipFile.toF                 6
67:26  FileCopyUtils.copy( file.getBytes( ), uploadedZipFile.toF                7
67:7   FileCopyUtils.copy( file.getBytes(), uploadedZipFile                SINK  8
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI

programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from a command line argument flows into exists, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to bypass the logic of the application in the conditional expression.
>
> Found in: **.mvn/wrapper/MavenWrapperDownloader.java (line : 57)**

Data Flow

.mvn/wrapper/MavenWrapperDownloader.java

```
50:39  File baseDirectory = new File(args[0]);                          SOURCE  0
50:39  File baseDirectory = new File(args[0]);                                  1
50:34  File baseDirectory = new File(args[0]);                                  2
50:14  File baseDirectory = new File(args[0]);                                  3
51:57  System.out.println("- Using base directory: " + baseDire                 4
55:45  File mavenWrapperPropertyFile = new File(baseDirectory,                  5
55:14  File mavenWrapperPropertyFile = new File(baseDirectory, I                6
57:12  if(mavenWrapperPropertyFile.exists()) {                                  7
57:12  if(mavenWrapperPropertyFile.exists())) {                                 8
57:9   if(mavenWrapperPropertyFile.exists()){                          SINK    9
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from a command line argument flows into exists, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to bypass the logic of the application in the conditional expression.

Found in: **.mvn/wrapper/MavenWrapperDownloader.java (line : 79)**

Data Flow

```
50:39  File baseDirectory = new File(args[0]);                          SOURCE  0
50:39  File baseDirectory = new File(args[0]);                                  1
50:34  File baseDirectory = new File(args[0]);                                  2
50:14  File baseDirectory = new File(args[0]);                                  3
51:57  System.out.println("- Using base directory: " + baseDire                 4
78:36  File outputFile = new File(baseDirectory.getAbsolutePat                  5
78:36  File outputFile = new File(baseDirectory.getAbsolutePath                 6
78:31  File outputFile = new File(baseDirectory.getAbsolutePath                 7
78:14  File outputFile = new File(baseDirectory.getAbsolutePath                 8
79:13  if(!outputFile.getParentFile().exists()) {                               9
79:13  if(!outputFile.getParentFile().exists()) {                               10
79:13  if(!outputFile.getParentFile().exists()) {                               11
 79:9  if(!outputFile.getParentFile().exists()){                        SINK    12
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip` .

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from a command line argument flows into java.io.FileInputStream, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to read arbitrary files.
>
> Found in: **.mvn/wrapper/MavenWrapperDownloader.java (line : 60)**

Data Flow

```
50:39  File baseDirectory = new File(args[0]);                          SOURCE  0
50:39  File baseDirectory = new File(args[0]);                                  1
50:34  File baseDirectory = new File(args[0]);                                  2
50:14  File baseDirectory = new File(args[0]);                                  3
51:57  System.out.println("- Using base directory: " + baseDire                 4
55:45  File mavenWrapperPropertyFile = new File(baseDirectory,                  5
55:14  File mavenWrapperPropertyFile = new File(baseDirectory, I                6
57:12  if(mavenWrapperPropertyFile.exists()) {                                  7
60:75  mavenWrapperPropertyFileInputStream = new FileInputStrea                 8
60:59  mavenWrapperPropertyFileInputStream = new FileInputS              SINK    9
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip` .

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from a command line argument flows into mkdirs, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to manipulate arbitrary files.
>
> Found in: **.mvn/wrapper/MavenWrapperDownloader.java (line : 80)**

Data Flow

50:39	File baseDirectory = new File(`args[0]`);	SOURCE	0
50:39	File baseDirectory = new File(`args[0]`);		1
50:34	File baseDirectory = new `File(` args[0]);		2

```
50:14  File baseDirectory = new File(args[0]);                          3
51:57  System.out.println("- Using base directory: " + baseDire          4
78:36  File outputFile = new File( baseDirectory. getAbsolutePat        5
78:36  File outputFile = new File( baseDirectory.getAbsolutePath        6
78:31  File outputFile = new File( baseDirectory.getAbsolutePath        7
78:14  File outputFile = new File(baseDirectory.getAbsolutePath         8
79:13  if(! outputFile. getParentFile().exists()) {                      9
80:17  if(! outputFile. getParentFile().mkdirs()) {                      10
80:17  if(! outputFile.getParentFile( ).mkdirs()) {                      11
80:17  if(! outputFile.getParentFile().mkdirs( )) {          SINK        12
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from a command line argument flows into java.io.FileOutputStream, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to write to arbitrary files.
>
> Found in: **.mvn/wrapper/MavenWrapperDownloader.java (line : 111)**

Data Flow

50:39	File baseDirectory = new File(`args[0]`);	SOURCE	0
50:39	File baseDirectory = new File(`args[0]`);		1
50:34	File baseDirectory = new`File(` args[0]);		2
50:14	File`baseDirectory = new File(args[0]);`		3

```
51:57  System.out.println("- Using base directory: " + baseDire                4
78:36  File outputFile = new File(baseDirectory.getAbsolutePat                 5
78:36  File outputFile = new File(baseDirectory.getAbsolutePath                6
78:31  File outputFile = new File(baseDirectory.getAbsolutePath                7
78:14  File outputFile = new File(baseDirectory.getAbsolutePath                8
79:13  if(!outputFile.getParentFile().exists()) {                              9
85:51  System.out.println("- Downloading to: " + outputFile.get               10
87:38  downloadFileFromURL(url,outputFile);                                    11
97:63  private static void downloadFileFromURL(String urlString                12
111:53 FileOutputStream fos = new FileOutputStream(destination                13
111:36 FileOutputStream fos = new FileOutputStream(destinat    SINK           14
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip` .

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from an HTTP parameter flows into org.springframework.util.FileCopyUtils.copyToByteArray, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to read arbitrary files.
>
> Found in:
> **src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (line : 97)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java

```
90:16  var id = request.getParameter("id");                                  SOURCE  0
90:16  var id = request.getParameter("id");                                          1
90:11  var id = request.getParameter("id");                                          2
92:85  new File(catPicturesDirectory, (id == null ? RandomUtils                      3
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils                      4
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils                      5
92:15  new File(catPicturesDirectory, (id == null ? RandomUtils                      6
91:11  var catPicture =
                        new File(catPicturesDirectory, (                             7
94:11  if (catPicture.getName().toLowerCase().contains("path-t                       8
97:49  .body(FileCopyUtils.copyToByteArray(catPicture));                             9
97:19  .body(FileCopyUtils.copyToByteArray(catPicture));                     SINK    10
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip` .

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from an HTTP parameter flows into org.springframework.util.FileCopyUtils.copyToByteArray, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to read arbitrary files.
>
> Found in:
> **src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (line : 103)**

Data Flow

```
90:16  var id = request.getParameter("id");                                  SOURCE  0
90:16  var id = request.getParameter("id");                                          1
90:11  var id = request.getParameter("id");                                          2
92:85  new File(catPicturesDirectory, (id == null ? RandomUtils                      3
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils                      4
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils                      5
92:15  new File(catPicturesDirectory, (id == null ? RandomUtils                      6
91:11  var catPicture =                                                              7
                        new File(catPicturesDirectory, (
94:11  if (catPicture.getName().toLowerCase().contains("path-t                       8
99:11  if (catPicture.exists()) {                                                    9
102:69 .location(new URI("/PathTraversal/random-picture?id=" +(                      10
103:76 .body(Base64.getEncoder().encode(FileCopyUtils.copyToByt                      11
103:46 .body(Base64.getEncoder().encode(FileCopyUtils.copy'                  SINK    12
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from an HTTP parameter flows into exists, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to bypass the logic of the application in the conditional expression.

Found in:
src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (line : 99)

Data Flow

90:16	`var id = request.getParameter("id");`	SOURCE	0
90:16	`var id = request.getParameter("id");`		1
90:11	`var id = request.getParameter("id");`		2
92:85	`new File(catPicturesDirectory, (id == null ? RandomUtils`		3
92:42	`new File(catPicturesDirectory, (id == null ? RandomUtils`		4
92:42	`new File(catPicturesDirectory, (id == null ? RandomUtils`		5
92:15	`new File(catPicturesDirectory, (id == null ? RandomUtils`		6
91:11	`var catPicture = new File(catPicturesDirectory, (`		7
94:11	`if (catPicture.getName().toLowerCase().contains("path-t`		8
99:11	`if (catPicture.exists()) {`		9
99:11	`if (catPicture.exists()) {`		10
99:7	`if (catPicture.exists()) {`	SINK	11

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from an HTTP parameter flows into listFiles, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to manipulate arbitrary files.

Found in:
src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (line : 108)

Data Flow

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path

traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from an HTTP parameter flows into java.util.zip.ZipFile, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to write to arbitrary files.

Found in: **src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java (line : 69)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java

```
51:41  public AttackResult uploadFileHandler(@RequestPar                    SOURCE  0
51:41  public AttackResult uploadFileHandler(@RequestParam("up                      1
52:10  if (!file.getOriginalFilename().toLowerCase().endsWith(                      2
55:31  return processZipUpload(file);                                               3
60:41  private AttackResult processZipUpload(MultipartFile fil                      4
66:53  var uploadedZipFile = tmpZipDirectory.resolve(file.getC                      5
66:53  var uploadedZipFile = tmpZipDirectory.resolve(file.getO                        6
66:29  var uploadedZipFile =tmpZipDirectory.resolve(file.getOr                        7
66:11  var uploadedZipFile = tmpZipDirectory.resolve(file.getOr                     8
67:43  FileCopyUtils.copy(file.getBytes(),uploadedZipFile.toF                        9
69:33  ZipFile zip = new ZipFile(uploadedZipFile.toFile());                         10
69:33  ZipFile zip = new ZipFile(uploadedZipFile.toFile()));                        11
69:25  ZipFile zip = new ZipFile(uploadedZipFile.toFile());           SINK        12
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from a zip file flows into java.nio.file.Files.copy, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to write to arbitrary files.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java (line : 75)**

Data Flow

73:53	`File f = new File(tmpZipDirectory.toFile(),e.getN`	SOURCE	0
73:53	`File f = new File(tmpZipDirectory.toFile(),e.getName())`		1
73:22	`File f = new File(tmpZipDirectory.toFile(), e.getName())`		2
73:14	`File f = new File(tmpZipDirectory.toFile(), e.getName())`		3
75:24	`Files.copy(is,f.toPath(), StandardCopyOption.REPLACE_EX`		4
75:24	`Files.copy(is,f.toPath(), StandardCopyOption.REPLACE_EX`		5
75:9	`Files.copy(is, f.toPath(), StandardCopyOption.REPLA`	SINK	6

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

SNYK-CODE | CWE-614

> Cookie misses a call to setSecure. Set the Secure flag to true to protect the cookie from man-in-the-middle attacks.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 134)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java

```
134:27  Cookie cookie = new Cookie("access_token", to
```

SOURCE SINK 0

Fix Analysis

Details

In a session hijacking attack, if a cookie containing sensitive data is set without the `secure` attribute, an attacker might be able to intercept that cookie. Once the attacker has this information, they can potentially impersonate a user, accessing confidential data and performing actions that they would not normally be authorized to do. Attackers often gain access to this sensitive cookie data when it is transmitted insecurely in plain text over a standard HTTP session, rather than being encrypted and sent over an HTTPS session. This type of attack is highly preventable by following best practices when setting sensitive session cookies.

Best practices for prevention

- Set the `secure` attribute in the response header when setting cookies on the client side, and use a test tool to verify that secure cookie transmission is in place.
- Always use HTTPS for all login pages and never redirect from HTTP to HTTPS, which leaves secure session data open to interception.
- Follow other best practices when it comes to session cookies, such as setting the HttpOnly flag and maintaining highly time-limited sessions.
- Consider implementing browser checks and providing secure data only within a browser that supports tight cookie security.
- Generate session IDs in a way that is not easily predictable, invalidate sessions upon logout, and never reuse session IDs.
- Educate developers to use built-in secure session-management functionality within the development environment instead of taking a DIY approach.

Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

SNYK-CODE | CWE-614

Cookie misses a call to setSecure. Set the Secure flag to true to protect the cookie from man-in-the-middle attacks.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 139)**

Data Flow

```
139:27  Cookie cookie = new Cookie("access_token", ""
```

SOURCE SINK 0

Fix Analysis

Details

In a session hijacking attack, if a cookie containing sensitive data is set without the `secure` attribute, an attacker might be able to intercept that cookie. Once the attacker has this information, they can potentially impersonate a user, accessing confidential data and performing actions that they would not normally be authorized to do. Attackers often gain access to this sensitive cookie data when it is transmitted insecurely in plain text over a standard HTTP session, rather than being encrypted and sent over an HTTPS session. This type of attack is highly preventable by following best practices when setting sensitive session cookies.

Best practices for prevention

- Set the `secure` attribute in the response header when setting cookies on the client side, and use a test tool to verify that secure cookie transmission is in place.
- Always use HTTPS for all login pages and never redirect from HTTP to HTTPS, which leaves secure session data open to interception.
- Follow other best practices when it comes to session cookies, such as setting the HttpOnly flag and maintaining highly time-limited sessions.
- Consider implementing browser checks and providing secure data only within a browser that supports tight cookie security.
- Generate session IDs in a way that is not easily predictable, invalidate sessions upon logout, and never reuse session IDs.
- Educate developers to use built-in secure session-management functionality within the development environment instead of taking a DIY approach.

Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

SNYK-CODE | CWE-614

Cookie misses a call to setSecure. Set the Secure flag to true to protect the cookie from man-in-the-middle attacks.

Found in:
src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (line : 76)

Data Flow

src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java

```
76:25   Cookie cookie = new Cookie(COOKIE_NAME, "");
```

SOURCE SINK 0

Fix Analysis

Details

In a session hijacking attack, if a cookie containing sensitive data is set without the `secure` attribute, an attacker might be able to intercept that cookie. Once the attacker has this information, they can potentially impersonate a user, accessing confidential data and performing actions that they would not normally be authorized to do. Attackers often gain access to this sensitive cookie data when it is transmitted insecurely in plain text over a standard HTTP session, rather than being encrypted and sent over an HTTPS session. This type of attack is highly preventable by following best practices when setting sensitive session cookies.

Best practices for prevention

- Set the `secure` attribute in the response header when setting cookies on the client side, and use a test tool to verify that secure cookie transmission is in place.
- Always use HTTPS for all login pages and never redirect from HTTP to HTTPS, which leaves secure session data open to interception.
- Follow other best practices when it comes to session cookies, such as setting the HttpOnly flag and maintaining highly time-limited sessions.
- Consider implementing browser checks and providing secure data only within a browser that supports tight cookie security.
- Generate session IDs in a way that is not easily predictable, invalidate sessions upon logout, and never reuse session IDs.
- Educate developers to use built-in secure session-management functionality within the development environment instead of taking a DIY approach.

Improper Neutralization of CRLF Sequences in HTTP Headers

SNYK-CODE | CWE-113

Unsanitized input from an HTTP parameter flows into addCookie and reaches an HTTP header returned to the user. This may allow a malicious input that contain CR/LF to split the http response into two responses and the second response to be controlled by the attacker. This may be used to mount a range of attacks such as cross-site scripting or cache poisoning.

Found in:
src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (line : 95)

Data Flow

src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java

```
62:7   @RequestParam String username,                                  SOURCE  0
62:7   @RequestParam String username,                                          1
68:35  return credentialsLoginFlow(username,password, response                 2
82:7   String username,String password, HttpServletResponse re                 3
83:33  String lowerCasedUsername =username.toLowerCase();                      4
83:33  String lowerCasedUsername =username.toLowerCase());                     5
83:12  String lowerCasedUsername = username.toLowerCase();                     6
91:31  String newCookieValue =EncDec.encode(lowerCasedUsernam                  7
91:14  String newCookieValue = EncDec.encode(lowerCasedUsername                8
92:30  Cookie newCookie = newCookie(COOKIE_NAME, newCookieValu                  9
92:14  Cookie newCookie = new Cookie(COOKIE_NAME, newCookieValu                 10
93:7   newCookie.setPath("/WebGoat");                                         11
94:7   newCookie.setSecure(true);                                            12
95:26  response.addCookie(newCookie);                                        13
95:7   response.addCookie(newCookie);                                  SINK  14
```

Fix Analysis

Details

CRLF is an abbreviation for the terms "carriage return" and "line feed." These two special characters are a legacy of old-fashioned printing terminals used in the early days of computing. However, today both are still often used as delimiters between data. When this weakness exists, CR and LF characters (represented respectively in code as `\r` and `\n`) are permitted to be present in HTTP headers, usually due to poor planning for data handling during development.

CRLF sequences in HTTP headers are known as "response splitting" because these characters effectively split the response from the browser, causing the single line to be accepted as multiple lines by the server (for example, the single line First Line\r\nSecond Line would be accepted by the server as two lines of input).

While response splitting in itself is not an attack, and can be completely harmless unless exploited, its presence could lead to an injection attack (known as CRLF injection) and a variety of unpredictable and potentially dangerous behavior. This weakness can be exploited in a number of ways, such as page hijacking or cross-user defacement, in which an attacker displays false site content and/or captures confidential information such as credentials. It can even lead to cross-site scripting attacks, in which attackers can cause malicious code to execute in the user's browser.

For example, the following code is vulnerable:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) {
            Cookie cookie = new Cookie("name",
request.getParameter("name"));
            response.addCookie(cookie);
          }
```

because the user may provide a name parameter with a value like `XYZ\r\nHTTP/1.1 200 OK\nATTACKER CONTROLLED`. In this case, they will produce a second HTTP response:

```
HTTP/1.1 200 OK
          ATTACKER CONTROLLED
```

A possible fix is to remove all non-alphanumerical characters:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) {
            String name = request.getParameter("name")
                .replaceAll("[^a-zA-Z ]", "");
            Cookie cookie = new Cookie("name", name);
```

```
            response.addCookie(cookie);
        }
```

In this case, the attacker would be unable to produce a second HTTP response.

Best practices for prevention

- Assume all input is potentially malicious. Define acceptable responses wherever possible, and if not possible, encode CR and LF characters to prevent header splitting.
- Replace both `\r` (carriage return) and `\n` (line feed) with "" (empty string)-many platforms handle these characters interchangeably so the weakness may still exist if one of the two is permitted. Follow best practices and strip all other special characters (", /, , ;, etc., as well as spaces) wherever possible. Be sure to sanitize special characters in both directions-from the browser to the server and also in data sent back to the browser. Ideally, adopt current development resources, such as languages and libraries, that block CR and LF injection in headers. Be vigilant with all input types that could potentially be tampered with or modified at the user end (intentionally or unintentionally), which could lead to injection attacks. These include GET, POST, cookies, and other HTTP headers.

Improper Neutralization of CRLF Sequences in HTTP Headers

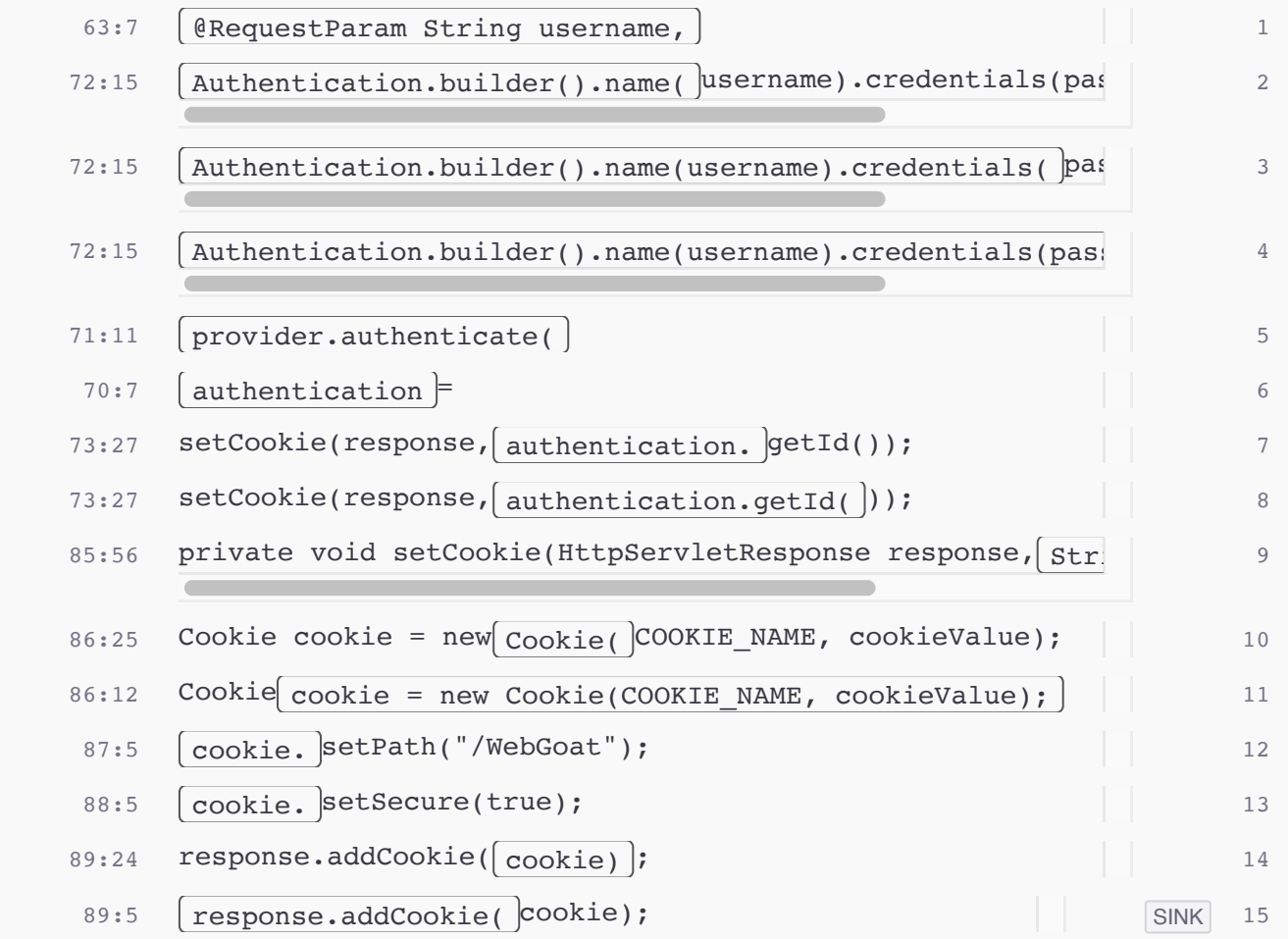
SNYK-CODE | CWE-113

> Unsanitized input from an HTTP parameter flows into addCookie and reaches an HTTP header returned to the user. This may allow a malicious input that contain CR/LF to split the http response into two responses and the second response to be controlled by the attacker. This may be used to mount a range of attacks such as cross-site scripting or cache poisoning.
>
> Found in:
> **src/main/java/org/owasp/webgoat/lessons/hijacksession/HijackSessionAssignment.java (line : 89)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/hijacksession/HijackSessionAssignment.java

63:7	`@RequestParam String username,`	SOURCE	0



Fix Analysis

Details

CRLF is an abbreviation for the terms "carriage return" and "line feed." These two special characters are a legacy of old-fashioned printing terminals used in the early days of computing. However, today both are still often used as delimiters between data. When this weakness exists, CR and LF characters (represented respectively in code as `\r` and `\n`) are permitted to be present in HTTP headers, usually due to poor planning for data handling during development.

CRLF sequences in HTTP headers are known as "response splitting" because these characters effectively split the response from the browser, causing the single line to be accepted as multiple lines by the server (for example, the single line First Line\r\nSecond Line would be accepted by the server as two lines of input).

While response splitting in itself is not an attack, and can be completely harmless unless exploited, its presence could lead to an injection attack (known as CRLF injection) and a variety of unpredictable and potentially dangerous behavior. This weakness can be exploited in a number of ways, such as page hijacking or cross-user defacement, in which an attacker displays false site content and/or captures confidential information such as credentials. It can even lead to cross-site scripting attacks, in which attackers can cause malicious code to execute in the user's browser.

For example, the following code is vulnerable:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) {
            Cookie cookie = new Cookie("name",
request.getParameter("name"));
            response.addCookie(cookie);
          }
```

because the user may provide a name parameter with a value like `XYZ\r\nHTTP/1.1 200 OK\nATTACKER CONTROLLED`. In this case, they will produce a second HTTP response:

```
HTTP/1.1 200 OK
          ATTACKER CONTROLLED
```

A possible fix is to remove all non-alphanumerical characters:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) {
            String name = request.getParameter("name")
                .replaceAll("[^a-zA-Z ]", "");
            Cookie cookie = new Cookie("name", name);
            response.addCookie(cookie);
          }
```

In this case, the attacker would be unable to produce a second HTTP response.

Best practices for prevention

- Assume all input is potentially malicious. Define acceptable responses wherever possible, and if not possible, encode CR and LF characters to prevent header splitting.
- Replace both `\r` (carriage return) and `\n` (line feed) with "" (empty string)-many platforms handle these characters interchangeably so the weakness may still exist if one of the two is permitted. Follow best practices and strip all other special characters (", /, , ;, etc., as well as spaces) wherever possible. Be sure to sanitize special characters in both directions-from the browser to the server and also in data sent back to the browser. Ideally, adopt current development resources, such as languages and libraries, that block CR and LF injection in headers. Be vigilant with all input types that could potentially be tampered with or modified at the user end (intentionally or unintentionally), which could lead to injection attacks. These include GET, POST, cookies, and other HTTP headers.

Sensitive Cookie Without 'HttpOnly' Flag

SNYK-CODE | CWE-1004

> Cookie misses a call to setHttpOnly. Set the HttpOnly flag to true to protect the cookie from possible malicious code on client side.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 134)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java

```
134:27  Cookie cookie = new Cookie("access_token", to
```

SOURCE SINK 0

Fix Analysis

Details

The `HttpOnly` flag is a simple parameter used when setting a user cookie to ensure that cookies with sensitive session data are visible only to the browser rather than to scripts. This helps prevent cross-site scripting attacks, in which an attacker gains access to sensitive session information and uses this information to trick legitimate web-based applications into disclosing confidential information or accepting illegitimate requests. When developers use the `HttpOnly` flag to set the cookie, they ensure that this sensitive session information is not readable or writable except by the browser (read) and server (write), respectively. While most modern browsers and versions now recognize the HttpOnly flag, some legacy and custom browsers still do not.

Best practices for prevention

- Include the `HttpOnly` attribute in the response header when setting cookies on the client side. Be aware, however, that this crucial step provides only partial remediation.
- Integrate client-side scripts to determine browser version; require browser compatibility or avoid transmitting sensitive data to browsers that do not support `HttpOnly`.
- Understand and evaluate risks of third-party components or plugins, which may expose cookies.
- Educate developers in a zero-trust approach, understanding the risks and best practices to prevent cross-site scripting, such as sanitizing all user input for code and special characters.

Sensitive Cookie Without 'HttpOnly' Flag

SNYK-CODE | CWE-1004

> Cookie misses a call to setHttpOnly. Set the HttpOnly flag to true to protect the cookie from possible malicious code on client side.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 139)**

Data Flow

```
139:27  Cookie cookie = new Cookie("access_token", ""
```

SOURCE SINK 0

Fix Analysis

Details

The `HttpOnly` flag is a simple parameter used when setting a user cookie to ensure that cookies with sensitive session data are visible only to the browser rather than to scripts. This helps prevent cross-site scripting attacks, in which an attacker gains access to sensitive session information and uses this information to trick legitimate web-based applications into disclosing confidential information or accepting illegitimate requests. When developers use the `HttpOnly` flag to set the cookie, they ensure that this sensitive session information is not readable or writable except by the browser (read) and server (write), respectively. While most modern browsers and versions now recognize the HttpOnly flag, some legacy and custom browsers still do not.

Best practices for prevention

- Include the `HttpOnly` attribute in the response header when setting cookies on the client side. Be aware, however, that this crucial step provides only partial remediation.
- Integrate client-side scripts to determine browser version; require browser compatibility or avoid transmitting sensitive data to browsers that do not support `HttpOnly` .
- Understand and evaluate risks of third-party components or plugins, which may expose cookies.
- Educate developers in a zero-trust approach, understanding the risks and best practices to prevent cross-site scripting, such as sanitizing all user input for code and special characters.

Sensitive Cookie Without 'HttpOnly' Flag

SNYK-CODE | CWE-1004

Cookie misses a call to setHttpOnly. Set the HttpOnly flag to true to protect the cookie from possible malicious code on client side.

Found in:
src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (line : 76)

Data Flow

src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java

```
76:25   Cookie cookie = new Cookie( COOKIE_NAME, "");
```

SOURCE SINK 0

Fix Analysis

Details

The `HttpOnly` flag is a simple parameter used when setting a user cookie to ensure that cookies with sensitive session data are visible only to the browser rather than to scripts. This helps prevent cross-site scripting attacks, in which an attacker gains access to sensitive session information and uses this information to trick legitimate web-based applications into disclosing confidential information or accepting illegitimate requests. When developers use the `HttpOnly` flag to set the cookie, they ensure that this sensitive session information is not readable or writable except by the browser (read) and server (write), respectively. While most modern browsers and versions now recognize the HttpOnly flag, some legacy and custom browsers still do not.

Best practices for prevention

- Include the `HttpOnly` attribute in the response header when setting cookies on the client side. Be aware, however, that this crucial step provides only partial remediation.
- Integrate client-side scripts to determine browser version; require browser compatibility or avoid transmitting sensitive data to browsers that do not support `HttpOnly`.
- Understand and evaluate risks of third-party components or plugins, which may expose cookies.
- Educate developers in a zero-trust approach, understanding the risks and best practices to prevent cross-site scripting, such as sanitizing all user input for code and special characters.

Sensitive Cookie Without 'HttpOnly' Flag

SNYK-CODE | CWE-1004

Cookie misses a call to setHttpOnly. Set the HttpOnly flag to true to protect the cookie from possible malicious code on client side.

Found in:
src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (line : 92)

Data Flow

```
92:30  Cookie newCookie = new Cookie( COOKIE_NAME, ne
```

SOURCE | SINK | 0

Fix Analysis

Details

The `HttpOnly` flag is a simple parameter used when setting a user cookie to ensure that cookies with sensitive session data are visible only to the browser rather than to scripts. This helps prevent cross-site scripting attacks, in which an attacker gains access to sensitive session information and uses this information to trick legitimate web-based applications into disclosing confidential information or accepting illegitimate requests. When developers use the `HttpOnly` flag to set the cookie, they ensure that this sensitive session information is not readable or writable except by the browser (read) and server (write), respectively. While most modern browsers and versions now recognize the HttpOnly flag, some legacy and custom browsers still do not.

Best practices for prevention

- Include the `HttpOnly` attribute in the response header when setting cookies on the client side. Be aware, however, that this crucial step provides only partial remediation.
- Integrate client-side scripts to determine browser version; require browser compatibility or avoid transmitting sensitive data to browsers that do not support `HttpOnly`.
- Understand and evaluate risks of third-party components or plugins, which may expose cookies.
- Educate developers in a zero-trust approach, understanding the risks and best practices to prevent cross-site scripting, such as sanitizing all user input for code and special characters.

Sensitive Cookie Without 'HttpOnly' Flag

SNYK-CODE | CWE-1004

Cookie misses a call to setHttpOnly. Set the HttpOnly flag to true to protect the cookie from possible malicious code on client side.

Found in:
src/main/java/org/owasp/webgoat/lessons/hijacksession/HijackSessionAssignment.java (line : 86)

Data Flow

src/main/java/org/owasp/webgoat/lessons/hijacksession/HijackSessionAssignment.java

```
86:25  Cookie cookie = new Cookie( COOKIE_NAME, cooki
```

SOURCE SINK 0

Fix Analysis

Details

The `HttpOnly` flag is a simple parameter used when setting a user cookie to ensure that cookies with sensitive session data are visible only to the browser rather than to scripts. This helps prevent cross-site scripting attacks, in which an attacker gains access to sensitive session information and uses this information to trick legitimate web-based applications into disclosing confidential information or accepting illegitimate requests. When developers use the `HttpOnly` flag to set the cookie, they ensure that this sensitive session information is not readable or writable except by the browser (read) and server (write), respectively. While most modern browsers and versions now recognize the HttpOnly flag, some legacy and custom browsers still do not.

Best practices for prevention

- Include the `HttpOnly` attribute in the response header when setting cookies on the client side. Be aware, however, that this crucial step provides only partial remediation.
- Integrate client-side scripts to determine browser version; require browser compatibility or avoid transmitting sensitive data to browsers that do not support `HttpOnly`.
- Understand and evaluate risks of third-party components or plugins, which may expose cookies.
- Educate developers in a zero-trust approach, understanding the risks and best practices to prevent cross-site scripting, such as sanitizing all user input for code and special characters.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 81)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java

```
65:41  public static final String PASSWORD = "bm5nhSk
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Permissive Cross-domain Policy

SNYK-CODE | CWE-942

> Setting targetOrigin to "*" in postMessage may enable malicious parties to intercept the message. Consider using an exact target origin instead.
>
> Found in: **src/main/resources/webgoat/static/js/libs/ace.js (line : 1740)**

Data Flow

src/main/resources/webgoat/static/js/libs/ace.js

```
1740:38  win.postMessage(messageName,"*");        SOURCE  0
1740:13  win.postMessage(messageName, "*");       SINK    1
```

Fix Analysis

Details

As a legacy of early web design and site limitations, most web applications default, for security reasons, to a "same origin policy". This means that browsers can only retrieve data from another site if the two sites share the same domain. In today's complex online environment, however, sites and applications often need to retrieve data from other domains. This is done under fairly limited conditions through an exception to the same origin policy known as "cross-origin resource sharing".

Developers may create definitions of trusted domains that are broader than absolutely necessary, inadvertently opening up wider access than intended. This weakness could result in data exposure or loss, or even allow an attacker to take over the site or application.

Best practices for prevention

- Avoid using wildcards for cross-origin resource sharing. Instead, define intended domains explicitly.
- Ensure that your site or app is well defended against cross-site scripting attacks (XSS), which could lead to takeover via an overly permissive cross-domain policy.
- Do not mix secure and insecure protocols when defining cross-domain policies.
- Consider defining a clear approved list to specify which domains will be given resource-level access; use this approved list to validate all domain access requests.

- Clearly define which methods (view, read, and update) are permitted for each resource and domain to avoid abuse.

Use of Password Hash With Insufficient Computational Effort

SNYK-CODE | CWE-916

> The MD5 hash (used in java.security.MessageDigest.getInstance) is insecure. Consider changing it to a secure hash algorithm
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (line : 55)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java

```
55:52  MessageDigest md = MessageDigest.getInstance("MD5   SOURCE  0
55:26  MessageDigest md = MessageDigest.getInstance("MD5");  SINK  1
```

Fix Analysis

Details

Sensitive information should never be stored in plain text, since this makes it very easy for unauthorized users, whether malicious insiders or outside attackers, to access. Hashing methods are used to make stored passwords and other sensitive data unreadable to users. For example, when a password is defined for the first time, it is hashed and then stored. The next time that user attempts to log on, the password they enter is hashed following the same procedure and compared with the stored value. In this way, the original password never needs to be stored in the system.

Hashing is a one-way scheme, meaning a hashed password cannot be reverse engineered. However, if an outdated or custom programmed hashing scheme is used, it becomes simple for an attacker with powerful modern computing power to gain access to the hashes used. This opens up access to all stored password information, leading to breached security. Therefore, it is essential for developers to understand modern, secure password hashing techniques.

Best practices for prevention

- Use strong standard algorithms for hashing rather than simpler but outdated methods or DIY hashing schemes, which may have inherent weaknesses.
- Use modular design for all code dealing with hashing so it can be swapped out as security standards change over time.
- Use salting in combination with hashing (While this places more demands on resources, it is an essential step for tighter security.).
- Implement zero-trust architecture to ensure that access to password data is granted only for legitimate business purposes.
- Increase developer awareness of current standards in data security and cryptography.

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from an HTTP parameter flows into transferTo, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to copy arbitrary directories.

Found in: **src/main/java/org/owasp/webgoat/webwolf/FileServer.java (line : 78)**

Data Flow

src/main/java/org/owasp/webgoat/webwolf/FileServer.java

```
74:34  public ModelAndView importFile( @RequestParam("fil          SOURCE  0
74:34  public ModelAndView importFile( @RequestParam("file") Mu            1
78:48  myFile.transferTo(new File(destinationDir, myFile. getOr           2
78:48  myFile.transferTo(new File(destinationDir, myFile.getOri           3
78:27  myFile.transferTo(new File( destinationDir, myFile.getOr           4
78:23  myFile.transferTo( new File(destinationDir, myFile.getOr           5
78:5   myFile.transferTo( new File(destinationDir, myFile.g       SINK    6
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
```

```
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from an HTTP parameter flows into org.springframework.util.FileCopyUtils.copy, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to read arbitrary files.

Found in: **src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUpload.java (line : 39)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUpload.java

```
37:7  @RequestParam("uploadedFile") MultipartFile file,        SOURCE  0
37:7  @RequestParam("uploadedFile") MultipartFile file,                1
39:26 return super.execute(file, fullName);                            2
```

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadBase.java

```
31:34 protected AttackResult execute(MultipartFile file, Strir        3
32:9  if (file.isEmpty()) {                                            4
44:26 FileCopyUtils.copy(file.getBytes(), uploadedFile);               5
44:26 FileCopyUtils.copy(file.getBytes()), uploadedFile);              6
44:7  FileCopyUtils.copy(file.getBytes(), uploadedFile);       SINK    7
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from an HTTP parameter flows into createNewFile, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to manipulate arbitrary files.

Found in:
src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRemoveUserInput.java (line : 36)

Data Flow

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRemoveUserInput.java

```
35:7   @RequestParam("uploadedFileRemoveUserInput") Mult        SOURCE   0
35:7   @RequestParam("uploadedFileRemoveUserInput") MultipartF           1
36:32  return super.execute(file, file.getOriginalFilename());           2
36:32  return super.execute(file, file.getOriginalFilename());           3
```

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadBase.java

```
31:54  protected AttackResult execute(MultipartFile file, Strin         4
42:30  var uploadedFile = new File(uploadDirectory, fullName);           5
42:11  var uploadedFile = new File(uploadDirectory, fullName);           6
43:7   uploadedFile.createNewFile();                                     7
43:7   uploadedFile.createNewFile());                          SINK     8
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from an HTTP parameter flows into createNewFile, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to manipulate arbitrary files.

Found in: **src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadFix.java (line : 39)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadFix.java

```
38:7   @RequestParam(value = "fullNameFix", required = f          SOURCE  0
38:7   @RequestParam(value = "fullNameFix", required = false) S          1
39:51  return super.execute(file, fullName != null ? fullName.           2
39:51  return super.execute(file, fullName != null ? fullName.r          3
39:32  return super.execute(file, fullName != null ? fullName.r          4
```

src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadBase.java

```
31:54  protected AttackResult execute(MultipartFile file, Strin          5
42:30  var uploadedFile = new File(uploadDirectory, fullName);           6
42:11  var uploadedFile = new File(uploadDirectory, fullName);           7
43:7   uploadedFile.createNewFile();                                     8
43:7   uploadedFile.createNewFile());                          SINK     9
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Improper Neutralization of CRLF Sequences in HTTP Headers

SNYK-CODE | CWE-113

Unsanitized input from the HTTP request body flows into setHeader and reaches an HTTP header returned to the user. This may allow a malicious input that contain CR/LF to split the http response into two responses and the second response to be controlled by the attacker. This may be used to mount a range of attacks such as cross-site scripting or cache poisoning.

Found in: **src/main/java/org/owasp/webgoat/webwolf/jwt/JWTController.java (line : 39)**

Data Flow

src/main/java/org/owasp/webgoat/webwolf/jwt/JWTController.java

```
35:26  public JWTToken encode( @RequestBody MultiValueMap                SOURCE  0
35:26  public JWTToken encode( @RequestBody MultiValueMap<Strin                  1
36:18  var header = formData. getFirst("header");                                2
36:18  var header = formData.getFirst( "header");                                3
 36:9  var header = formData.getFirst("header");                                 4
39:28  return JWTToken.encode( header, payload, secretKey);                      5
```

src/main/java/org/owasp/webgoat/webwolf/jwt/JWTToken.java

```
66:33  public static JWTToken encode( String header, String payl                 6
67:25  var headers = parse( header);                                             7
45:44  private static Map<String, Object> parse( String header)                  8
48:14  return reader.readValue( header, TreeMap.class);                          9
80:25  headers.forEach((k, v) -> jws.setHeader(k, v));                           10
80:48  headers.forEach((k, v) -> jws.setHeader(k, v));                           11
80:31  headers.forEach((k, v) -> jws.setHeader( k, v));                  SINK    12
```

Fix Analysis

Details

CRLF is an abbreviation for the terms "carriage return" and "line feed." These two special characters are a legacy of old-fashioned printing terminals used in the early days of computing. However, today both are still often used as delimiters between data. When this weakness exists, CR and LF characters (represented respectively in code as `\r` and `\n`) are permitted to be present in HTTP headers, usually due to poor planning for data handling during development.

CRLF sequences in HTTP headers are known as "response splitting" because these characters effectively split the response from the browser, causing the single line to be accepted as multiple lines by the server (for example, the single line First Line\r\nSecond Line would be accepted by the server as two lines of input).

While response splitting in itself is not an attack, and can be completely harmless unless exploited, its presence could lead to an injection attack (known as CRLF injection) and a variety of unpredictable and potentially dangerous behavior. This weakness can be exploited in a number of ways, such as page hijacking or cross-user defacement, in which an attacker displays false site content and/or captures confidential information such as credentials. It can even lead to cross-site scripting attacks, in which attackers can cause malicious code to execute in the user's browser.

For example, the following code is vulnerable:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) {
            Cookie cookie = new Cookie("name",
request.getParameter("name"));
            response.addCookie(cookie);
          }
```

because the user may provide a name parameter with a value like `XYZ\r\nHTTP/1.1 200 OK\nATTACKER CONTROLLED`. In this case, they will produce a second HTTP response:

```
HTTP/1.1 200 OK
          ATTACKER CONTROLLED
```

A possible fix is to remove all non-alphanumerical characters:

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) {
            String name = request.getParameter("name")
                .replaceAll("[^a-zA-Z ]", "");
            Cookie cookie = new Cookie("name", name);
            response.addCookie(cookie);
```

```
        }
```

In this case, the attacker would be unable to produce a second HTTP response.

Best practices for prevention

- Assume all input is potentially malicious. Define acceptable responses wherever possible, and if not possible, encode CR and LF characters to prevent header splitting.
- Replace both `\r` (carriage return) and `\n` (line feed) with "" (empty string)-many platforms handle these characters interchangeably so the weakness may still exist if one of the two is permitted. Follow best practices and strip all other special characters (", /, , ;, etc., as well as spaces) wherever possible. Be sure to sanitize special characters in both directions-from the browser to the server and also in data sent back to the browser. Ideally, adopt current development resources, such as languages and libraries, that block CR and LF injection in headers. Be vigilant with all input types that could potentially be tampered with or modified at the user end (intentionally or unintentionally), which could lead to injection attacks. These include GET, POST, cookies, and other HTTP headers.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/insecurelogin/InsecureLoginTask.java (line : 36)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/insecurelogin/InsecureLoginTask.java

```
36:43   if ("CaptainJack".equals(username) &&"BlackPe
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task

may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in:
> **src/main/java/org/owasp/webgoat/lessons/cryptography/XOREncodingAssignment.java (line : 40)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/cryptography/XOREncodingAssignment.java

```
40:51  if (answer_pwd1 != null && answer_pwd1.equals(
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

Do not hardcode passwords in code. Found password string

Found in:
src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java (line : 84)

Data Flow

src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java

61:7 `"somethingVeryRandomWhichNoOneWillEverTypeInA` SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/idor/IDORLogin.java (line : 51)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/idor/IDORLogin.java

```
51:46  idorUserInfo.get("bill").put("password", "buff   SOURCE SINK  0
```

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found hardcoded password used in password.
>
> Found in: **src/main/resources/lessons/jwt/js/jwt-refresh.js (line : 10)**

Data Flow

src/main/resources/lessons/jwt/js/jwt-refresh.js

```
10:43  data: JSON.stringify({user: user, password: "b
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Unprotected Storage of Credentials

SNYK-CODE | CWE-256

> An attacker might be able to detect the value of the password due to the exposure of comparison timing. When the functions Arrays.equals() or String.equals() are called, they will exit earlier if fewer bytes are matched. Use password encoder such as BCrypt for comparing passwords.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java (line : 55)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java

```
55:64  public AttackResult completed(@RequestParam St
```

SOURCE SINK 0

Fix Analysis

Details

If credentials are not protected or not sufficiently protected through strong encryption, attackers can access this information in a number of ways. Developers may rely on plain-text storage of credentials when they believe the system is completely secure from attack or only accessible to insiders. This confidence is misguided and dangerous. If a malicious insider-such as a former employee-or a hostile attacker using SQL injection, XML injection, or a brute-force attack accesses the system, they can access this credential information to gain unauthorized permissions within the system and to export other confidential and secure information.

Best practices for prevention

- Ensure that passwords are never stored in plain text, even for "purely internal" use.
- Never rely on password encoding, such as base 64 encoding; choose a complex encryption algorithm that includes salting, then hashing.
- Implement zero-trust approaches in which users have access only to information needed for legitimate business purposes.
- To the greatest extent possible, secure the application against injection attacks and other types of weaknesses.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/it/java/org/owasp/webgoat/JWTLessonIntegrationTest.java (line : 197)**

Data Flow

src/it/java/org/owasp/webgoat/JWTLessonIntegrationTest.java

197:41	`.signWith(SignatureAlgorithm.HS256,"deletingT`	SOURCE SINK	0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 44)**

Data Flow

src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java

44:18	`String key ="qwertyqwerty1234";`	SOURCE SINK	0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 55)**

Data Flow

```
55:43  Jwt jwt = Jwts.parser().setSigningKey( "qwerty
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 78)**

Data Flow

```
78:65   .signWith(io.jsonwebtoken.SignatureAlgorithm.H
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-

relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/deserialization/DeserializeTest.java (line : 71)**

Data Flow

src/test/java/org/owasp/webgoat/lessons/deserialization/DeserializeTest.java

71:9 `"rO0ABXNyADFvcmcuZHVtbXkuaW5zZWN1cmUuZnJhbWV3` SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

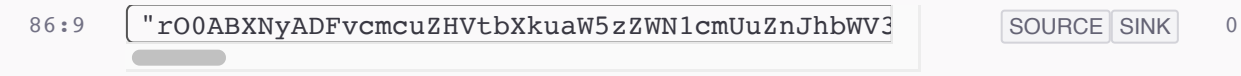
- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/deserialization/DeserializeTest.java (line : 86)**

Data Flow



Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.

- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/deserialization/DeserializeTest.java (line : 100)**

Data Flow

```
100:9  "rO0ABXQAVklmIHlvdSBkZXNlcmlhbGl6ZSBtZSBkb3du
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java (line : 187)**

Data Flow

src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java

```
187:32  params.put("password",("password"));        SOURCE SINK  0
```

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java (line : 108)**

Data Flow

src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java

```
108:28  params.put("password", "BlackPearl");
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java (line : 118)**

Data Flow

```
118:28  params.put("password", "ajnaeliclm^&&@kjn.");
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (line : 25)**

Data Flow

src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java

```
25:26  userForm.setPassword("test1234");
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (line : 37)**

Data Flow

```
37:26  userForm.setPassword("test12345");        SOURCE SINK  0
```

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798,CWE-259

> Do not hardcode passwords in code. Found password string
>
> Found in: **src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (line : 50)**

Data Flow

```
50:26  userForm.setPassword("test12345");   SOURCE SINK  0
```

Fix Analysis

Details

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Best practices for prevention

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard-code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

> The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.
>
> Found in: **src/it/java/org/owasp/webgoat/JWTLessonIntegrationTest.java (line : 70)**

Data Flow

src/it/java/org/owasp/webgoat/JWTLessonIntegrationTest.java

```
70:18  Jwt jwt =Jwts.parser().setSigningKey(TextCode
```

SOURCE SINK 0

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

> The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 55)**

Data Flow

src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java

```
55:15  Jwt jwt = Jwts.parser().setSigningKey("qwertyc
```

SOURCE SINK 0

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347

> The parse method does not validate the JWT signature. Consider using 'parseClaimsJws', 'parsePlaintextJws' instead.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 57)**

Data Flow

```
57:9  Jwts.parser()
                    .setSigningKeyResolver(
                        new SigningKeyResol
                          @Override
```

SOURCE SINK 0

```
                    public byte[] res
                      return TextCode
                    }
                  })
                .parse(
```

Fix Analysis

Details

Some JSON Web Token (JWT) parse methods from the io.jsonwebtoken.jjwt library accept a JWT whose signature is empty although a signing key has been set for the parser. This means that an attacker can create arbitrary JWTs that will be accepted if these methods are used.

Best practices for prevention

- Always enforce JWT signature verification by using parseClaimsJws or parsePlaintextJws methods or by overriding JwtHandlerAdapter's onPlaintextJws or onClaimsJws methods.

References

- Reading a JWS

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from cookies flows into exists, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to bypass the logic of the application in the conditional expression.
>
> Found in: **src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java (line : 89)**

Data Flow

src/it/java/org/owasp/webgoat/IntegrationTest.java

SOURCE 0

```
209:25  String result = RestAssured.given()
                            .when()
                            .relaxed
                            .cookie(
```

```
209:25  String result = RestAssured.given()
                            .when()
                            .relaxedHTTPSVa
                            .cookie(
```

```
209:25  String result = RestAssured.given()
                            .when()
                            .relaxedHTTPSVa
                            .cookie("WEBWOl
                            .get(
```

```
209:25  String result = RestAssured.given()
                            .when()
                            .relaxedHTTPSVa
                            .cookie("WEBWOl
                            .get(webWolfUrl
                            .then(
```

```
209:25  String result = RestAssured.given()
                            .when()
                            .relaxedHTTPSVa
                            .cookie("WEBWOl
                            .get(webWolfUrl
                            .then()
                            .extract(
```

```
209:25  String result = RestAssured.given()
                            .when()
                            .relaxedHTTPSVa
                            .cookie("WEBWOl
                            .get(webWolfUrl
                            .then()
                            .extract().resp
```

```
209:25  String result = RestAssured.given()
                            .when()
                            .relaxedHTTPSVa
                            .cookie("WEBWOl
                            .get(webWolfUrl
```

```
                                    .then()
                                    .extract().resp
209:25  String result = RestAssured.given()                                  7
                                    .when()
                                    .relaxedHTTPSVa
                                    .cookie("WEBWOL
                                    .get(webWolfUrl
                                    .then()
                                    .extract().resp
209:16  String result = RestAssured.given()                                  8
216:18  result = result.replace("%20", " ");                                 9
216:18  result = result.replace("%20", " ");                                 10
216:9   result = result.replace("%20", " ");                                 11
217:16  return result;                                                       12
```

src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java

```
60:26   webwolfFileDir = getWebWolfFileServerLocation();                     13
60:9    webwolfFileDir = getWebWolfFileServerLocation();                     14
88:32   Path webWolfFilePath = Paths.get(webwolfFileDir);                    15
88:14   Path webWolfFilePath = Paths.get(webwolfFileDir);                    16
89:13   if (webWolfFilePath.resolve(Paths.get(this.getUser(), h              17
89:13   if (webWolfFilePath.resolve(Paths.get(this.getUser(), h              18
89:13   if (webWolfFilePath.resolve(Paths.get(this.getUser(), h              19
89:13   if (webWolfFilePath.resolve(Paths.get(this.getUser(), h              20
89:9    if (webWolfFilePath.resolve(Paths.get(this.getUser(                  SINK 21
```

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from cookies flows into exists, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to bypass the logic of the application in the conditional expression.

Found in: **src/it/java/org/owasp/webgoat/XXEIntegrationTest.java (line : 55)**

Data Flow

src/it/java/org/owasp/webgoat/IntegrationTest.java

209:25 `String result =`

```
RestAssured.given()
                    .when()
                    .relaxed
                    .cookie(
```

SOURCE 0

209:25 `String result =`

```
RestAssured.given()
                    .when()
                    .relaxedHTTPSV
                    .cookie(
```

1

209:25 `String result =`

```
RestAssured.given()
                    .when()
                    .relaxedHTTPSV
                    .cookie("WEBWO
                    .get(
```

2

209:25 `String result =`

```
RestAssured.given()
                    .when()
                    .relaxedHTTPSV
                    .cookie("WEBWO
                    .get(webWolfUr
                    .then(
```

3

209:25 `String result =`

```
RestAssured.given()
                    .when()
                    .relaxedHTTPSV
                    .cookie("WEBWO
                    .get(webWolfUr
```

4

```
                                    .then()
                                    .extract(
```

```
209:25  String result =  RestAssured.given()                                    5
                                    .when()
                                    .relaxedHTTPSV
                                    .cookie("WEBWO
                                    .get(webWolfUr
                                    .then()
                                    .extract().res
```

```
209:25  String result =  RestAssured.given()                                    6
                                    .when()
                                    .relaxedHTTPSV
                                    .cookie("WEBWO
                                    .get(webWolfUr
                                    .then()
                                    .extract().res
```

```
209:25  String result =  RestAssured.given()                                    7
                                    .when()
                                    .relaxedHTTPSV
                                    .cookie("WEBWO
                                    .get(webWolfUr
                                    .then()
                                    .extract().res
```

```
209:16  String result = RestAssured.given()                                     8
216:18  result = result. replace("%20", " ");                                   9
216:18  result = result.replace( "%20", " ");                                   10
216:9   result = result.replace("%20", " ");                                    11
217:16  return result;                                                          12
```

src/it/java/org/owasp/webgoat/XXEIntegrationTest.java

```
33:37   webWolfFileServerLocation = getWebWolfFileServerLocation               13
33:9    webWolfFileServerLocation = getWebWolfFileServerLocatio                 14
54:32   Path webWolfFilePath = Paths.get( webWolfFileServerLocat                15
54:14   Path webWolfFilePath = Paths.get(webWolfFileServerLocati                16
```

55:13	`if (webWolfFilePath.resolve(Paths.get(this.getUser(), '`	17
55:13	`if (webWolfFilePath.resolve(Paths.get(this.getUser(), '`	18
55:13	`if (webWolfFilePath.resolve(Paths.get(this.getUser(), "]`	19
55:13	`if (webWolfFilePath.resolve(Paths.get(this.getUser(), "]`	20
55:9	`if (webWolfFilePath.resolve(Paths.get(this.getUser(` SINK	21

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

> Unsanitized input from cookies flows into java.nio.file.Files.delete, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to delete arbitrary files.
>
> Found in: **src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java (line : 90)**

Data Flow

src/it/java/org/owasp/webgoat/IntegrationTest.java

```
209:25  String result = RestAssured.given()                    SOURCE  0
                                .when()
                                .relaxed
                                .cookie(

209:25  String result = RestAssured.given()                            1
                                .when()
                                .relaxedHTTPSV
                                .cookie(
```

209:25 `String result =`

```
RestAssured.given()
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(
```

2

209:25 `String result =`

```
RestAssured.given()
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then(
```

3

209:25 `String result =`

```
RestAssured.given()
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then()
                                .extract(
```

4

209:25 `String result =`

```
RestAssured.given()
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then()
                                .extract().res
```

5

209:25 `String result =`

```
RestAssured.given()
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then()
                                .extract().res
```

6

```
209:25  String result = RestAssured.given()
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then()
                                .extract().res
209:16  String result = RestAssured.given()
216:18  result = result.replace("%20", " ");
216:18  result = result.replace("%20", " ");
216:9   result = result.replace("%20", " ");
217:16  return result;
```

src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java

```
60:26   webwolfFileDir = getWebWolfFileServerLocation();
60:9    webwolfFileDir = getWebWolfFileServerLocation();
88:32   Path webWolfFilePath = Paths.get(webwolfFileDir);
88:14   Path webWolfFilePath = Paths.get(webwolfFileDir);
89:13   if (webWolfFilePath.resolve(Paths.get(this.getUser(), h
90:26   Files.delete(webWolfFilePath.resolve(Paths.get(this.get
90:26   Files.delete(webWolfFilePath.resolve(Paths.get(this.get
90:13   Files.delete(webWolfFilePath.resolve(Paths.get(this
```

SINK

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI

programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Path Traversal

SNYK-CODE | CWE-23

Unsanitized input from cookies flows into java.nio.file.Files.delete, where it is used as a path. This may result in a Path Traversal vulnerability and allow an attacker to delete arbitrary files.

Found in: **src/it/java/org/owasp/webgoat/XXEIntegrationTest.java (line : 56)**

Data Flow

src/it/java/org/owasp/webgoat/IntegrationTest.java

209:25 `String result =` SOURCE 0

```
RestAssured.given()
                        .when()
                        .relaxed
                        .cookie(
```

209:25 `String result =` 1

```
RestAssured.given()
                        .when()
                        .relaxedHTTPSV
                        .cookie(
```

209:25 `String result =` 2

```
RestAssured.given()
                        .when()
                        .relaxedHTTPSV
                        .cookie("WEBWO
                        .get(
```

209:25 `String result =` 3

```
RestAssured.given()
                        .when()
                        .relaxedHTTPSV
                        .cookie("WEBWO
                        .get(webWolfUr
                        .then(
```

209:25 `String result =` 4

```
RestAssured.given()
                        .when()
                        .relaxedHTTPSV
                        .cookie("WEBWO
                        .get(webWolfUr
                        .then()
                        .extract(
```

```
209:25  String result = RestAssured.given()                                  5
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then()
                                .extract().res

209:25  String result = RestAssured.given()                                  6
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then()
                                .extract().res

209:25  String result = RestAssured.given()                                  7
                                .when()
                                .relaxedHTTPSV
                                .cookie("WEBWO
                                .get(webWolfUr
                                .then()
                                .extract().res

209:16  String result = RestAssured.given()                                  8
216:18  result = result.replace("%20", " ");                                 9
216:18  result = result.replace("%20", " ");                                 10
216:9   result = result.replace("%20", " ");                                 11
217:16  return result;                                                       12
```

src/it/java/org/owasp/webgoat/XXEIntegrationTest.java

```
33:37   webWolfFileServerLocation = getWebWolfFileServerLocation            13
33:9    webWolfFileServerLocation = getWebWolfFileServerLocatio             14
54:32   Path webWolfFilePath = Paths.get(webWolfFileServerLocat              15
54:14   Path webWolfFilePath = Paths.get(webWolfFileServerLocati             16
55:13   if (webWolfFilePath.resolve(Paths.get(this.getUser(), '              17
```

56:26	`Files.delete(webWolfFilePath.resolve(Paths.get(this.get`		18
56:26	`Files.delete(webWolfFilePath.resolve(Paths.get(this.get`		19
56:13	`Files.delete(webWolfFilePath.resolve(Paths.get(this`	SINK	20

Fix Analysis

Details

A Directory Traversal attack (also known as path traversal) aims to access files and directories that are stored outside the intended folder. By manipulating 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, configuration, and other critical system files.

Being able to access and manipulate an arbitrary path leads to vulnerabilities when a program is being run with privileges that the user providing the path should not have. A website with a path traversal vulnerability would allow users access to sensitive files on the server hosting it. CLI programs may also be vulnerable to path traversal if they are being ran with elevated privileges (such as with the setuid or setgid flags in Unix systems).

Directory Traversal vulnerabilities can be generally divided into two types:

- **Information Disclosure**: Allows the attacker to gain information about the folder structure or read the contents of sensitive files on the system.

`st` is a module for serving static files on web pages, and contains a vulnerability of this type. In our example, we will serve files from the `public` route.

If an attacker requests the following URL from our server, it will in turn leak the sensitive private key of the root user.

```
curl
http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_
```

Note `%2e` is the URL encoded version of `.` (dot).

- **Writing arbitrary files**: Allows the attacker to create or replace existing files. This type of vulnerability is also known as `Zip-Slip`.

One way to achieve this is by using a malicious `zip` archive that holds path traversal filenames. When each filename in the zip archive gets concatenated to the target extraction folder, without validation, the final path ends up outside of the target folder. If an executable or a configuration file is overwritten with a file containing malicious code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a `zip` archive with one benign file and one malicious file. Extracting the malicious file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwriting the `authorized_keys` file:

```
2018-04-15 22:04:29 .....           19           19  good.txt
          2018-04-15 22:04:42 .....           20           20
../../../../../../root/.ssh/authorized_keys
```

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (line : 122)**

Data Flow

src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java

```
122:69  String token = Jwts.builder().setClaims(claims
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.
- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.

Use of Hardcoded, Security-relevant Constants

SNYK-CODE | CWE-547

> Avoid hardcoding values that are meant to be secret. Found hardcoded secret.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpointTest.java (line : 33)**

Data Flow

src/test/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpointTest.java

```
33:18   String key = "deletingTom";
```

SOURCE SINK 0

Fix Analysis

Details

When constants are hardcoded into applications, this information could easily be reverse-engineered and become known to attackers. For example, if a breached authentication token is hardcoded in multiple places in the application, it may lead to components of the application remaining vulnerable if not all instances are changed. Another negative effect of hard-coding constants is potential unpredictability in the application's performance if the development team fails to update every single instance of the hardcoded constant throughout the code. For these reasons, hard-coding security-relevant constants is considered bad coding practice and should be remedied if present and avoided in future.

Best practices for prevention

- Never hard code security-related constants; use symbolic names or configuration lookup files.
- As hard coding is often done by coders working alone on a small scale, examine all legacy code components and test carefully when scaling.

- Adopt a "future-proof code" mindset: While use of constants may save a little time now and make development simpler in the short term, it could cost time and money adapting to scale or other unforeseen circumstances (such as new hardware) in the future.