Snyk Code Report

69 high issues **51** medium issues **57** low issues

SCAN COVERAGE

.html files: **69** .java files: **348** .js files: **95** .xml files: **6**

SQL Injection

SNYK-CODE | CWE-89 | Sqli

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 String name, @RequestParam String auth_tan) {                SOURCE  0
59:33 public AttackResult completed(@RequestParam String name, @RequestParam String auth_tan) {                        1
60:43 return injectableQueryConfidentiality(name, auth_tan);                                                            2
63:57 protected AttackResult injectableQueryConfidentiality(String name, String auth_tan) {                             3
67:15 +name                                                                                                             4
66:9  "SELECT * FROM employees WHERE last_name = '"                                                                     5
                    + name
66:9  "SELECT * FROM employees WHERE last_name = '"                                                                     6
                    + name
                    + "' AND auth_tan = '"
66:9  "SELECT * FROM employees WHERE last_name = '"                                                                     7
                    + name
                    + "' AND auth_tan = '"
                    + auth_tan
66:9  "SELECT * FROM employees WHERE last_name = '"                                                                     8
                    + name
                    + "' AND auth_tan = '"
                    + auth_tan
                    + "'";
65:12 String query =                                                                                                    9
                  "SELECT * FROM employees WHERE last_name = '"
                      + name
                      + "' AND auth_tan = '"
                      + auth_tan
                      + "'";
77:25 log(connection, query);                                                                                           10
147:49 public static void log(Connection connection, String action) {                                                   11
78:52 ResultSet results = statement.executeQuery(query);                                                                12
78:29 ResultSet results = statement.executeQuery(query);                                                        SINK    13
```

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 | Sqli

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 String name, @RequestParam String auth_tan) {   SOURCE  0
61:37  return injectableQueryIntegrity(name, auth_tan);                                            1
64:51  protected AttackResult injectableQueryIntegrity(String name, String auth_tan) {            2
68:15  +name                                                                                       3
67:9   "SELECT * FROM employees WHERE last_name = '"                                               4
                     + name
67:9   "SELECT * FROM employees WHERE last_name = '"                                               5
                     + name
                     + "' AND auth_tan = '"
67:9   "SELECT * FROM employees WHERE last_name = '"                                               6
                     + name
                     + "' AND auth_tan = '"
                     + auth_tan
67:9   "SELECT * FROM employees WHERE last_name = '"                                               7
                     + name
                     + "' AND auth_tan = '"
                     + auth_tan
                     + "'";
66:12  String query =                                                                              8
                   "SELECT * FROM employees WHERE last_name = '"
                       + name
                       + "' AND auth_tan = '"
                       + auth_tan
                       + "'";
76:52  ResultSet results = statement.executeQuery(query);                                          9
76:29  ResultSet results = statement.executeQuery(query);   SINK                                   10
```

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 | Sqli

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 String name, @RequestParam String auth_tan) {                SOURCE  0
59:33  public AttackResult completed( @RequestParam String name, @RequestParam String auth_tan) {                        1
60:43  return injectableQueryConfidentiality( name, auth_tan);                                                           2
63:57  protected AttackResult injectableQueryConfidentiality( String name, String auth_tan) {                            3
67:15  + name                                                                                                            4
66:9   "SELECT * FROM employees WHERE last_name = '"                                                                     5
                     + name
66:9   "SELECT * FROM employees WHERE last_name = '"                                                                     6
                     + name
                     + "' AND auth_tan = '"
66:9   "SELECT * FROM employees WHERE last_name = '"                                                                     7
                     + name
                     + "' AND auth_tan = '"
                     + auth_tan
66:9   "SELECT * FROM employees WHERE last_name = '"                                                                     8
                     + name
                     + "' AND auth_tan = '"
                     + auth_tan
                     + "'";
65:12  String query =                                                                                                    9
                   "SELECT * FROM employees WHERE last_name = '"
                       + name
                       + "' AND auth_tan = '"
                       + auth_tan
                       + "'";
77:25  log(connection, query);                                                                                           10
147:49 public static void log(Connection connection, String action) {                                                    11
148:14 action = action. replace('¥'', '"');                                                                              12
148:14 action = action.replace( '¥'', '"');                                                                              13
148:5  action = action.replace('¥'', '"');                                                                               14
154:77 "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";                                15
154:9  "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";                                16
154:9  "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";                                17
153:12 String logQuery =                                                                                                 18
                  "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";
158:31 statement.executeUpdate( logQuery);                                                                               19
158:7  statement.executeUpdate( logQuery);                                                                       SINK    20
```

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 | Sqli

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 String name, @RequestParam String auth_tan) {        SOURCE  0
60:33  public AttackResult completed(@RequestParam String name, @RequestParam String auth_tan) {                1
61:37  return injectableQueryIntegrity(name, auth_tan);                                                         2
64:51  protected AttackResult injectableQueryIntegrity(String name, String auth_tan) {                          3
68:15  +name                                                                                                    4
67:9   "SELECT * FROM employees WHERE last_name = '"                                                            5
                     + name
67:9   "SELECT * FROM employees WHERE last_name = '"                                                            6
                     + name
                     + "' AND auth_tan = '"
67:9   "SELECT * FROM employees WHERE last_name = '"                                                            7
                     + name
                     + "' AND auth_tan = '"
                     + auth_tan
67:9   "SELECT * FROM employees WHERE last_name = '"                                                            8
                     + name
                     + "' AND auth_tan = '"
                     + auth_tan
                     + "'";
66:12  String query =                                                                                           9
                   "SELECT * FROM employees WHERE last_name = '"
                       + name
                       + "' AND auth_tan = '"
                       + auth_tan
                       + "'";
75:45  SqlInjectionLesson8.log(connection, query);                                                              10
```

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

```
147:49 public static void log(Connection connection, String action){                                            11
148:14 action = action.replace('¥'', '"');                                                                      12
148:14 action = action.replace('¥'', '"');                                                                      13
148:5  action = action.replace('¥'', '"');                                                                      14
154:77 "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";                       15
154:9  "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";                       16
154:9  "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";                       17
```

153:12	`String logQuery = "INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "')";`		18
158:31	`statement.executeUpdate( logQuery);`		19
158:7	`statement.executeUpdate( logQuery);`	SINK	20

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.

Path Traversal

SNYK-CODE | CWE-23 | PT

> 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
92:42	`new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`		3
92:42	`new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`		4
92:15	`new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`		5
91:11	`var catPicture = new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`		6
97:49	`.body(FileCopyUtils.copyToByteArray(catPicture));`		7
97:19	`.body(FileCopyUtils.copyToByteArray(catPicture));`	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_rsa
```

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 | PT

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
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");                                      3
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");                                      4
92:15  new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");                                      5
91:11  var catPicture =
                       new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");                      6
103:76 .body(Base64.getEncoder().encode(FileCopyUtils.copyToByteArray(catPicture)));                                                 7
103:46 .body(Base64.getEncoder().encode(FileCopyUtils.copyToByteArray(catPicture)));                                         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_rsa
```

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 | PT

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
92:42	`new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`	3
92:42	`new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`	4
92:15	`new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`	5
91:11	`var catPicture = new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");`	6
99:11	`if (catPicture.exists()) {`	7
99:11	`if (catPicture.exists()) {`	8
99:7	`if (catPicture.exists()) { return ResponseEntity.ok() .contentType(MediaType.parseMediaType(MediaType.IMAGE_JPEG_VALUE)) .location(new URI("/PathTraversal/random-picture?id=" + catPicture.getName())) .body(Base64.getEncoder().encode(FileCopyUtils.copyToByteArray(catPicture))); }`	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_rsa
```

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 | PT

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

```
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
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");                                                                      3
92:42  new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");                                                                      4
92:15  new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");                                                                      5
91:11  var catPicture =                                                                                                                                              6
                    new File(catPicturesDirectory, (id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
108:55 StringUtils.arrayToCommaDelimitedString(catPicture.getParentFile().listFiles())                                                                               7
108:55 StringUtils.arrayToCommaDelimitedString(catPicture.getParentFile().listFiles())                                                                               8
108:55 StringUtils.arrayToCommaDelimitedString(catPicture.getParentFile().listFiles())                                                                       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_rsa
```

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 | PT

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 write to 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(@RequestParam("uploadedFileZipSlip") MultipartFile file){`	SOURCE	0
51:41	`public AttackResult uploadFileHandler(@RequestParam("uploadedFileZipSlip") MultipartFile file){`		1
55:31	`return processZipUpload(file);`		2
60:41	`private AttackResult processZipUpload(MultipartFile file){`		3
66:53	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		4
66:53	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename()));`		5
66:29	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		6
66:11	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		7
67:43	`FileCopyUtils.copy(file.getBytes(),uploadedZipFile.toFile());`		8
67:43	`FileCopyUtils.copy(file.getBytes(),uploadedZipFile.toFile()));`		9
67:7	`FileCopyUtils.copy(file.getBytes(), uploadedZipFile.toFile());`	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_rsa
```

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 | PT

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

51:41	`public AttackResult uploadFileHandler(@RequestParam("uploadedFileZipSlip") MultipartFile file){`	SOURCE	0
51:41	`public AttackResult uploadFileHandler(@RequestParam("uploadedFileZipSlip") MultipartFile file){`		1
55:31	`return processZipUpload(file);`		2
60:41	`private AttackResult processZipUpload(MultipartFile file){`		3
66:53	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		4
66:53	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		5
66:29	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		6
66:11	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		7
69:33	`ZipFile zip = new ZipFile(uploadedZipFile.toFile());`		8
69:33	`ZipFile zip = new ZipFile(uploadedZipFile.toFile());`		9
69:25	`ZipFile zip = new ZipFile(uploadedZipFile.toFile());`	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_rsa
```

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 | PT

Unsanitized input from an HTTP parameter 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

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

51:41	`public AttackResult uploadFileHandler(@RequestParam("uploadedFileZipSlip") MultipartFile file){`	SOURCE	0
51:41	`public AttackResult uploadFileHandler(@RequestParam("uploadedFileZipSlip") MultipartFile file){`		1
55:31	`return processZipUpload(file);`		2
60:41	`private AttackResult processZipUpload(MultipartFile file){`		3
66:53	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		4
66:53	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		5
66:29	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		6
66:11	`var uploadedZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());`		7
69:33	`ZipFile zip = new ZipFile(uploadedZipFile.toFile());`		8
69:33	`ZipFile zip = new ZipFile(uploadedZipFile.toFile());`		9
69:25	`ZipFile zip = new ZipFile(uploadedZipFile.toFile());`		10
69:15	`ZipFile zip = new ZipFile(uploadedZipFile.toFile());`		11
70:49	`Enumeration<? extends ZipEntry> entries = zip.entries();`		12
70:49	`Enumeration<? extends ZipEntry> entries = zip.entries();`		13
70:39	`Enumeration<? extends ZipEntry> entries = zip.entries();`		14
72:22	`ZipEntry e = entries.nextElement();`		15
72:22	`ZipEntry e = entries.nextElement();`		16
72:18	`ZipEntry e = entries.nextElement();`		17
73:53	`File f = new File(tmpZipDirectory.toFile(),e.getName());`		18
73:53	`File f = new File(tmpZipDirectory.toFile(),e.getName());`		19
73:22	`File f = new File(tmpZipDirectory.toFile(), e.getName());`		20
73:14	`File f = new File(tmpZipDirectory.toFile(), e.getName());`		21
75:24	`Files.copy(is,f.toPath(), StandardCopyOption.REPLACE_EXISTING);`		22
75:24	`Files.copy(is,f.toPath(), StandardCopyOption.REPLACE_EXISTING);`		23
75:9	`Files.copy(is, f.toPath(), StandardCopyOption.REPLACE_EXISTING);`	SINK	24

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_rsa
```

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 | NonCryptoHardcodedSecret

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

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

```
36:32  String ADMIN_PASSWORD_LINK = "375afe1104f4a487a73823c50a9292a2";
```

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 | NonCryptoHardcodedSecret

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

Found in: **src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (line : 44)**

Data Flow

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

```
44:53  public static final String[] SECRETS = {"secret", "admin", "password", "123456", "passw0rd"};
```

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 | NonCryptoHardcodedSecret

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

Found in: **src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (line : 44)**

Data Flow

```
44:74   public static final String[] SECRETS = {"secret", "admin", "password","123456", "passw0rd"};
```

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 | NonCryptoHardcodedSecret

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

Found in: **src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (line : 44)**

Data Flow

```
44:84   public static final String[] SECRETS = {"secret", "admin", "password", "123456","passw0rd"};
```

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 | NonCryptoHardcodedSecret

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

Found in: **src/main/java/org/owasp/webgoat/lessons/csrf/ForgedReviews.java (line : 56)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/csrf/ForgedReviews.java

```
56:46  private static final String weakAntiCSRF =”2aa14227b9a13d0bede0388a7fba9aa9”;
```

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 | NonCryptoHardcodedSecret

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 : 62)**

Data Flow

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

```
62:46  private static final String JWT_PASSWORD =”bm5n3SkxCX4kKRy4”;
```

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 | NonCryptoHardcodedSecret

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 : 50)**

Data Flow

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

```
50:5   "victory", "business", "available", "shipping", "washington"   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 | NonCryptoHardcodedSecret

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 : 50)**

Data Flow

```
50:16   "victory", "business", "available", "shipping", "washington"   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 | NonCryptoHardcodedSecret

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 : 50)**

Data Flow

```
50:28   "victory", "business",["available", ]"shipping", "washington"
```

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 | NonCryptoHardcodedSecret

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 : 50)**

Data Flow

```
50:41   "victory", "business", "available",["shipping", ]"washington"
```

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 | NonCryptoHardcodedSecret

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 : 50)**

Data Flow

```
50:53  "victory", "business", "available", "shipping",("washington")
```

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 86)**

Data Flow

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

```
62:46  private static final String JWT_PASSWORD =("bm5n3SkxCX4kKRy4";
```

SOURCE 0

```
86:9   Jwts.builder()                                                    io.jsonwebtoken.
                    .setIssuedAt(new Date(System.currentTimeMillis() + TimeUnit.DAYS.toDays(10)))
                    .setClaims(claims)
                    .signWith(
```

SINK 1

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

> Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 107)**

Data Flow

```
62:46   private static final String JWT_PASSWORD = "bm5n3SkxCX4kKRy4";                          SOURCE  0
107:17  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASSWORD).parse(token.replace("Bearer ", ""));  SINK    1
```

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

> Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 137)**

Data Flow

```
62:46   private static final String JWT_PASSWORD = "bm5n3SkxCX4kKRy4";                          SOURCE  0
137:11  Jwts.parser().setSigningKey(JWT_PASSWORD).parse(token.replace("Bearer ", ""));          SINK    1
```

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 126)**

Data Flow

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

```
71:69   public static final String JWT_PASSWORD = TextCodec.BASE64.encode("victory");   SOURCE 0
126:11  Jwts.builder()                                                                  SINK   1
                        .setClaims(claims)
                        .signWith(
```
io.jsonwebtoken.SignatureAlgorithm.HS512, JWT_PASSWORD)

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 155)**

Data Flow

```
71:69   public static final String JWT_PASSWORD = TextCodec.BASE64.encode("victory");   SOURCE 0
155:19  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASSWORD).parse(accessToken);         SINK   1
```

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 180)**

Data Flow

```
71:69   public static final String JWT_PASSWORD = TextCodec.BASE64.encode("victory");   SOURCE  0
180:19  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASSWORD).parse(accessToken);         SINK    1
```

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 203)**

Data Flow

```
71:69   public static final String JWT_PASSWORD = TextCodec.BASE64.encode("victory");   SOURCE  0
203:19  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASSWORD).parse(accessToken);         SINK    1
```

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

Hardcoded value array {...} is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 61)**

Data Flow

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

```
49:42  public static final String[] SECRETS = {                                                                    SOURCE  0
                                                "victory", "business", "available", "shipping", "washington"
                                              };
61:12  return Jwts.builder()                                                                                       SINK    1
                  .setIssuer("WebGoat Token Builder")
                  .setAudience("webgoat.org")
                  .setIssuedAt(Calendar.getInstance().getTime())
                  .setExpiration(Date.from(Instant.now().plusSeconds(60)))
                  .setSubject("tom@webgoat.org")
                  .claim("username", "Tom")
                  .claim("Email", "tom@webgoat.org")
                  .claim("Role", new String[] {"Manager", "Project Administrator"})
                  .signWith(                                                                                                SignatureAlgorithm.HS256,
```

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret

Hardcoded value array {...} is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 78)**

Data Flow

```
49:42  public static final String[] SECRETS = {                                                                    SOURCE  0
                                                "victory", "business", "available", "shipping", "washington"
                                              };
```

78:17	Jwt jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(token);	SINK	1

✓ 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 Insufficiently Random Values - Secrets

SNYK-CODE | CWE-330 | InsecureSecret

Insecure random data flows from nextInt and is used as a secret data. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (line : 53)**

↓ Data Flow

53:14	String secret = SECRETS[new Random().nextInt(SECRETS.length)];	SOURCE	0
53:23	String secret = SECRETS[new Random().nextInt(SECRETS.length)];		1
53:31	String secret = SECRETS[new Random().nextInt(SECRETS.length)];		2
53:31	String secret = SECRETS[new Random().nextInt(SECRETS.length)];		3
53:14	String secret = SECRETS[new Random().nextInt(SECRETS.length)];	SINK	4

✓ Fix Analysis

Details

Computer security relies on random numbers for many things: generating secure, confidential session keys; hashing password data; encryption for transmitting sensitive data, and more. It's easy to understand why. If session keys, for example, were generated sequentially, attackers would be able to guess these easily and then hijack legitimate user sessions. Similarly, if encryption techniques used easy-to-guess numbers, attackers could use brute-force attacks to gain unauthorized access.

In reality, since computers cannot generate truly random numbers, they use "pseudorandom" numbers instead, generated using an algorithm that is "seeded" in a variety of ways to produce highly variable values in a random-seeming order, making them very hard-in theory-for attackers to guess. However, if developers inadvertently make use of a weak random algorithm, attackers may be able to discover the algorithm, seed, or pattern, ultimately unlocking access to commands or sensitive data, which can then be held for ransom or sold.

Best practices for prevention

- Avoid using weak pseudorandom number generators (PRNGs), such as statistical PRNGs. Instead, choose a cryptographically secure PRNG.
- Avoid using predictable seed values, such as user ID or server start time. Instead, use a seed that is itself pseudorandom, such as one taken from an external hardware source.
- Use standard, accepted security algorithms and libraries rather than taking a DIY approach and creating custom code that may contain inherent weaknesses or overlook critical flaws.
- Use static analysis tools to identify potential instances of this weakness in code and then ensure good test coverage with appropriate white-box testing.
- Educate developers about the importance of entropy in security systems development, and consider adopting tools that are FIPS 140-2 compliant.

Use of Insufficiently Random Values - Secrets

SNYK-CODE | CWE-330 | InsecureSecret

Insecure random data flows from nextInt and is used as a secret data. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (line : 71)**

Data Flow

71:14	`String secret = SECRETS[new Random().nextInt(SECRETS.length)];`	SOURCE	0
71:23	`String secret = SECRETS[new Random().nextInt(SECRETS.length)];`		1
71:31	`String secret = SECRETS[new Random().nextInt(SECRETS.length)];`		2
71:31	`String secret = SECRETS[new Random().nextInt(SECRETS.length)];`		3
71:14	`String secret = SECRETS[new Random().nextInt(SECRETS.length)];`	SINK	4

Fix Analysis

Details

Computer security relies on random numbers for many things: generating secure, confidential session keys; hashing password data; encryption for transmitting sensitive data, and more. It's easy to understand why. If session keys, for example, were generated sequentially, attackers would be able to guess these easily and then hijack legitimate user sessions. Similarly, if encryption techniques used easy-to-guess numbers, attackers could use brute-force attacks to gain unauthorized access.

In reality, since computers cannot generate truly random numbers, they use "pseudorandom" numbers instead, generated using an algorithm that is "seeded" in a variety of ways to produce highly variable values in a random-seeming order, making them very hard-in theory-for attackers to guess. However, if developers inadvertently make use of a weak random algorithm, attackers may be able to discover the algorithm, seed, or pattern, ultimately unlocking access to commands or sensitive data, which can then be held for ransom or sold.

Best practices for prevention

- Avoid using weak pseudorandom number generators (PRNGs), such as statistical PRNGs. Instead, choose a cryptographically secure PRNG.
- Avoid using predictable seed values, such as user ID or server start time. Instead, use a seed that is itself pseudorandom, such as one taken from an external hardware source.
- Use standard, accepted security algorithms and libraries rather than taking a DIY approach and creating custom code that may contain inherent weaknesses or overlook critical flaws.
- Use static analysis tools to identify potential instances of this weakness in code and then ensure good test coverage with appropriate white-box testing.
- Educate developers about the importance of entropy in security systems development, and consider adopting tools that are FIPS 140-2 compliant.

Use of Insufficiently Random Values - Secrets

SNYK-CODE | CWE-330 | InsecureSecret

Insecure random data flows from nextInt and is used as a secret data. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 52)**

Data Flow

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

52:30	`public static final String JWT_SECRET = TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);`	SOURCE	0
53:7	`TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);`		1
53:31	`TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);`		2
53:39	`TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);`		3
53:39	`TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);`		4
52:30	`public static final String JWT_SECRET = TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);`	SINK	5

Fix Analysis

Details

Computer security relies on random numbers for many things: generating secure, confidential session keys; hashing password data; encryption for transmitting sensitive data, and more. It's easy to understand why. If session keys, for example, were generated sequentially, attackers would be able to guess these easily and then hijack legitimate user sessions. Similarly, if encryption techniques used easy-to-guess numbers, attackers could use brute-force attacks to gain unauthorized access.

In reality, since computers cannot generate truly random numbers, they use "pseudorandom" numbers instead, generated using an algorithm that is "seeded" in a variety of ways to produce highly variable values in a random-seeming order, making them very hard-in theory-for attackers to guess. However, if developers inadvertently make use of a weak random algorithm, attackers may be able to discover the algorithm, seed, or pattern, ultimately unlocking access to commands or sensitive data, which can then be held for ransom or sold.

Best practices for prevention

- Avoid using weak pseudorandom number generators (PRNGs), such as statistical PRNGs. Instead, choose a cryptographically secure PRNG.
- Avoid using predictable seed values, such as user ID or server start time. Instead, use a seed that is itself pseudorandom, such as one taken from an external hardware source.
- Use standard, accepted security algorithms and libraries rather than taking a DIY approach and creating custom code that may contain inherent weaknesses or overlook critical flaws.
- Use static analysis tools to identify potential instances of this weakness in code and then ensure good test coverage with appropriate white-box testing.
- Educate developers about the importance of entropy in security systems development, and consider adopting tools that are FIPS 140-2 compliant.

Use of Insufficiently Random Values - Secrets

SNYK-CODE | CWE-330 | InsecureSecret

Insecure random data flows from nextInt and is used as a secret data. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 78)**

Data Flow

78:11	Jwt `jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(token);`	SOURCE	0
78:17	Jwt jwt = `Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(` token);		1
78:17	Jwt jwt = `Jwts.parser().setSigningKey(` JWT_SECRET).parseClaimsJws(token);		2
78:45	Jwt jwt = Jwts.parser().setSigningKey(`JWT_SECRET)` .parseClaimsJws(token);		3
52:30	public static final String `JWT_SECRET = TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);`		4
53:7	`TextCodec.BASE64.encode(` SECRETS[new Random().nextInt(SECRETS.length)]);		5
53:31	TextCodec.BASE64.encode(`SECRETS[new Random().nextInt(SECRETS.length)])` ;		6
53:39	TextCodec.BASE64.encode(SECRETS[`new Random().nextInt(` SECRETS.length)]);		7
53:39	TextCodec.BASE64.encode(SECRETS[`new Random().nextInt(` SECRETS.length)]);		8
78:11	Jwt `jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(token);`	SINK	9

Fix Analysis

Details

Computer security relies on random numbers for many things: generating secure, confidential session keys; hashing password data; encryption for transmitting sensitive data, and more. It's easy to understand why. If session keys, for example, were generated sequentially, attackers would be able to guess these easily and then hijack legitimate user sessions. Similarly, if encryption techniques used easy-to-guess numbers, attackers could use brute-force attacks to gain unauthorized access.

In reality, since computers cannot generate truly random numbers, they use "pseudorandom" numbers instead, generated using an algorithm that is "seeded" in a variety of ways to produce highly variable values in a random-seeming order, making them very hard-in theory-for attackers to guess. However, if developers inadvertently make use of a weak random algorithm, attackers may be able to discover the algorithm, seed, or pattern, ultimately unlocking access to commands or sensitive data, which can then be held for ransom or sold.

Best practices for prevention

- Avoid using weak pseudorandom number generators (PRNGs), such as statistical PRNGs. Instead, choose a cryptographically secure PRNG.
- Avoid using predictable seed values, such as user ID or server start time. Instead, use a seed that is itself pseudorandom, such as one taken from an external hardware source.
- Use standard, accepted security algorithms and libraries rather than taking a DIY approach and creating custom code that may contain inherent weaknesses or overlook critical flaws.
- Use static analysis tools to identify potential instances of this weakness in code and then ensure good test coverage with appropriate white-box testing.
- Educate developers about the importance of entropy in security systems development, and consider adopting tools that are FIPS 140-2 compliant.

Use of Insufficiently Random Values - Secrets

SNYK-CODE | CWE-330 | InsecureSecret

Insecure random data flows from nextInt and is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 61)**

Data Flow

```
53:7   TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);        SOURCE  0
53:31  TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);                1
53:39  TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);                2
53:39  TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);                3
61:12  return Jwts.builder()                                                          SINK    4
                   .setIssuer("WebGoat Token Builder")
                   .setAudience("webgoat.org")
                   .setIssuedAt(Calendar.getInstance().getTime())
                   .setExpiration(Date.from(Instant.now().plusSeconds(60)))
                   .setSubject("tom@webgoat.org")
                   .claim("username", "Tom")
                   .claim("Email", "tom@webgoat.org")
                   .claim("Role", new String[] {"Manager", "Project Administrator"})
                   .signWith(
```

Fix Analysis

Details

Computer security relies on random numbers for many things: generating secure, confidential session keys; hashing password data; encryption for transmitting sensitive data, and more. It's easy to understand why. If session keys, for example, were generated sequentially, attackers would be able to guess these easily and then hijack legitimate user sessions. Similarly, if encryption techniques used easy-to-guess numbers, attackers could use brute-force attacks to gain unauthorized access.

In reality, since computers cannot generate truly random numbers, they use "pseudorandom" numbers instead, generated using an algorithm that is "seeded" in a variety of ways to produce highly variable values in a random-seeming order, making them very hard-in theory-for attackers to guess. However, if developers inadvertently make use of a weak random algorithm, attackers may be able to discover the algorithm, seed, or pattern, ultimately unlocking access to commands or sensitive data, which can then be held for ransom or sold.

Best practices for prevention

- Avoid using weak pseudorandom number generators (PRNGs), such as statistical PRNGs. Instead, choose a cryptographically secure PRNG.
- Avoid using predictable seed values, such as user ID or server start time. Instead, use a seed that is itself pseudorandom, such as one taken from an external hardware source.
- Use standard, accepted security algorithms and libraries rather than taking a DIY approach and creating custom code that may contain inherent weaknesses or overlook critical flaws.
- Use static analysis tools to identify potential instances of this weakness in code and then ensure good test coverage with appropriate white-box testing.
- Educate developers about the importance of entropy in security systems development, and consider adopting tools that are FIPS 140-2 compliant.

Use of Insufficiently Random Values - Secrets

SNYK-CODE | CWE-330 | InsecureSecret

Insecure random data flows from nextInt and is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 78)**

Data Flow

```
53:7   TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);        SOURCE  0
53:31  TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);                1
53:39  TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);                2
```

53:39 TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]); 3

78:17 Jwt jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(token); SINK 4

✓ Fix Analysis

Details

Computer security relies on random numbers for many things: generating secure, confidential session keys; hashing password data; encryption for transmitting sensitive data, and more. It's easy to understand why. If session keys, for example, were generated sequentially, attackers would be able to guess these easily and then hijack legitimate user sessions. Similarly, if encryption techniques used easy-to-guess numbers, attackers could use brute-force attacks to gain unauthorized access.

In reality, since computers cannot generate truly random numbers, they use "pseudorandom" numbers instead, generated using an algorithm that is "seeded" in a variety of ways to produce highly variable values in a random-seeming order, making them very hard-in theory-for attackers to guess. However, if developers inadvertently make use of a weak random algorithm, attackers may be able to discover the algorithm, seed, or pattern, ultimately unlocking access to commands or sensitive data, which can then be held for ransom or sold.

Best practices for prevention

- Avoid using weak pseudorandom number generators (PRNGs), such as statistical PRNGs. Instead, choose a cryptographically secure PRNG.
- Avoid using predictable seed values, such as user ID or server start time. Instead, use a seed that is itself pseudorandom, such as one taken from an external hardware source.
- Use standard, accepted security algorithms and libraries rather than taking a DIY approach and creating custom code that may contain inherent weaknesses or overlook critical flaws.
- Use static analysis tools to identify potential instances of this weakness in code and then ensure good test coverage with appropriate white-box testing.
- Educate developers about the importance of entropy in security systems development, and consider adopting tools that are FIPS 140-2 compliant.

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass

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 : 107)**

⇣ Data Flow

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

107:17 Jwt jwt = Jwts.parser().setSigningKey(JWT_PASSWORD).parse(token.replace("Bearer ", "")); 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.

Best practices for prevention

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass

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 : 137)**

Data Flow

137:11 `Jwts.parser().setSigningKey(JWT_PASSWORD).parse(`token.replace("Bearer ", ""));

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.

Best practices for prevention

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass

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 : 155)**

Data Flow

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

155:19 Jwt jwt = `Jwts.parser().setSigningKey(JWT_PASSWORD).parse(`accessToken);

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.

Best practices for prevention

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass

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 : 180)**

Data Flow

180:19 Jwt jwt = `Jwts.parser().setSigningKey(JWT_PASSWORD).parse(`accessToken);

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.

Best practices for prevention

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass

> 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 : 203)**

↓ Data Flow

```
203:19  Jwt jwt = Jwts.parser().setSigningKey(JWT_PASSWORD).parse( accessToken);   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.

Best practices for prevention

- Reading a JWS

SQL Injection

SNYK-CODE | CWE-89 | Sqli

> 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

```
51:44  @RequestParam String username_login, @RequestParam String password_login) throws Exception {   SOURCE  0
51:44  @RequestParam String username_login, @RequestParam String password_login) throws Exception {   1
64:21  + password_login                                                                               2
61:15  "select password from challenge_users where userid = '"                                        3
                          + username_login
```

```
                        + "' and password = '"
                        + password_login
```

```
61:15  "select password from challenge_users where userid = '" + username_login + "' and password = '" + password_login + "'");   4
```

```
60:11  connection.prepareStatement(   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 | Sqli

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 login_count, HttpServletRequest request)   SOURCE   0
58:41  return injectableQuery(login_count, userid);   1
61:62  protected AttackResult injectableQuery(String login_count, String accountName) {   2
62:89  String queryString = "SELECT * From user_data WHERE Login_Count = ? and userid= " + accountName;   3
62:26  String queryString = "SELECT * From user_data WHERE Login_Count = ? and userid= " + accountName;   4
62:12  String queryString = "SELECT * From user_data WHERE Login_Count = ? and userid= " + accountName;   5
66:15  queryString, ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);   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 | Sqli

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 column)throws Exception {`	SOURCE	0
67:28	`public List<Server> sort(@RequestParam String column)throws Exception {`		1
75:21	`+column)) {`		2
73:15	`"select id, hostname, ip, mac, status, description from SERVERS where status <> 'out" + " of order' order by " + column)) {`		3
72:11	`connection.prepareStatement(`	SINK	4

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 | Sqli

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,` email_reg, password_reg);		2
93:39	private AttackResult checkArguments(`String username_reg,` String email_reg, String password_reg) {		3
67:73	"select userid from sql_challenge_users where userid = '" +`username_reg` + "'";		4
67:13	`"select userid from sql_challenge_users where userid = '" + username_reg` + "'";		5
67:13	`"select userid from sql_challenge_users where userid = '" + username_reg + "'";`		6
66:16	String `checkUserQuery = "select userid from sql_challenge_users where userid = '" + username_reg + "'";`		7
69:54	ResultSet resultSet = statement.executeQuery(`checkUserQuery)`;		8
69:31	ResultSet resultSet =`statement.executeQuery(` checkUserQuery);	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 | Sqli

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 = "userid_6a") String userId)`{	SOURCE	0
56:33	public AttackResult completed(`@RequestParam(value = "userid_6a") String userId)`{		1
57:28	return injectableQuery(`userId)`;		2
62:39	public AttackResult injectableQuery(`String accountName)`{		3
66:63	query = "SELECT * FROM user_data WHERE last_name = '" +`accountName` + "'";		4
66:15	query =`"SELECT * FROM user_data WHERE last_name = '" + accountName` + "'";		5
66:15	query =`"SELECT * FROM user_data WHERE last_name = '" + accountName + "'";`		6
66:7	`query = "SELECT * FROM user_data WHERE last_name = '" + accountName + "'";`		7
74:52	ResultSet results = statement.executeQuery(`query)`;		8
74:29	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 | Sqli

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 String action_string){`	SOURCE	0
58:33	`public AttackResult completed(@RequestParam String action_string){`		1
59:40	`return injectableQueryAvailability(action_string);`		2
62:54	`protected AttackResult injectableQueryAvailability(String action){`		3
64:70	`String query = "SELECT * FROM access_log WHERE action LIKE '%" +action+ "%'";`		4
64:20	`String query =("SELECT * FROM access_log WHERE action LIKE '%" + action+ "%'";`		5
64:20	`String query =("SELECT * FROM access_log WHERE action LIKE '%" + action + "%'";`		6
64:12	`String(query = "SELECT * FROM access_log WHERE action LIKE '%" + action + "%'";`		7
71:52	`ResultSet results = statement.executeQuery(query);`		8
71:29	`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 | Sqli

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 String query){`	SOURCE	0
58:33	`public AttackResult completed(@RequestParam String query){`		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 | Sqli

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 | Sqli

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 String operator, @RequestParam String injection) {   SOURCE  0
55:7   @RequestParam String account, @RequestParam String operator, @RequestParam String injection) {           1
56:28  return injectableQuery(account + " " + operator + " " + injection);                                      2
56:28  return injectableQuery(account + " " + operator + " " + injection);                                      3
56:28  return injectableQuery(account + " " + operator + " " + injection);                                      4
56:28  return injectableQuery(account + " " + operator + " " + injection);                                      5
56:28  return injectableQuery(account + " " + operator + " " + injection);                                      6
59:42  protected AttackResult injectableQuery(String accountName){                                              7
63:83  "SELECT * FROM user_data WHERE first_name = 'John' and last_name = '" + accountName + "'";               8
63:11  "SELECT * FROM user_data WHERE first_name = 'John' and last_name = '" + accountName + "'";               9
63:11  "SELECT * FROM user_data WHERE first_name = 'John' and last_name = '" + accountName + "'";               10
62:7   query =                                                                                                  11
                  "SELECT * FROM user_data WHERE first_name = 'John' and last_name = '" + accountName + "'";
67:52  ResultSet results = statement.executeQuery(query);                                                       12
67:29  ResultSet results = statement.executeQuery(query);                                          SINK         13
```

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 | Sqli

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_sql_only_input_validation") String userId){`	SOURCE	0
47:30	`public AttackResult attack(@RequestParam("userid_sql_only_input_validation") String userId){`		1
51:58	`AttackResult attackResult = lesson6a.injectableQuery(userId);`		2

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

62:39	`public AttackResult injectableQuery(String accountName){`		3
66:63	`query = "SELECT * FROM user_data WHERE last_name = '" +accountName+ "'";`		4
66:15	`query =("SELECT * FROM user_data WHERE last_name = '" + accountName+ "'";`		5
66:15	`query =("SELECT * FROM user_data WHERE last_name = '" + accountName + "'";)`		6
66:7	`(query = "SELECT * FROM user_data WHERE last_name = '" + accountName + "'";)`		7
74:52	`ResultSet results = statement.executeQuery(query);`		8
74:29	`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 | Sqli

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_on_keywords") String userId){`	SOURCE	0

```
52:7   @RequestParam("userid_sql_only_input_validation_on_keywords") String userId){                1
53:14  userId = userId.toUpperCase().replace("FROM", "").replace("SELECT", "");                     2
53:14  userId = userId.toUpperCase()).replace("FROM", "").replace("SELECT", "");                    3
53:14  userId = userId.toUpperCase().replace("FROM", "").replace("SELECT", "");                     4
53:14  userId = userId.toUpperCase().replace("FROM", "").replace("SELECT", "");                     5
53:5   userId = userId.toUpperCase().replace("FROM", "").replace("SELECT", "");                     6
57:58  AttackResult attackResult = lesson6a.injectableQuery(userId);                                7
```

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

```
62:39  public AttackResult injectableQuery(String accountName){                                    8
66:63  query = "SELECT * FROM user_data WHERE last_name = '" +accountName+ "'";                    9
66:15  query = "SELECT * FROM user_data WHERE last_name = '" + accountName+ "'";                   10
66:15  query = "SELECT * FROM user_data WHERE last_name = '" + accountName + "'";                  11
66:7   query = "SELECT * FROM user_data WHERE last_name = '" + accountName + "'";                  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 | Sqli

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 String query){                  SOURCE          0
53:33  public AttackResult completed(@RequestParam String query){                                  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 | Sqli

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 String query){`	SOURCE	0
54:33	`public AttackResult completed(@RequestParam String query){`		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.

Path Traversal

SNYK-CODE | CWE-23 | PT

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/ProfileUpload.java (line : 39)**

Data Flow

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

38:7	`@RequestParam(value = "fullName", required = false) String fullName){`	SOURCE	0
38:7	`@RequestParam(value = "fullName", required = false) String fullName){`		1
39:32	`return super.execute(file,fullName);`		2

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

31:54	`protected AttackResult execute(MultipartFile file,String fullName){`		3
42:52	`var uploadedFile = new File(uploadDirectory,fullName);`		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_rsa
```

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 | PT

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

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

```
31:54 protected AttackResult execute(MultipartFile file,(String fullName){          5
42:52 var uploadedFile = new File(uploadDirectory,(fullName);                       6
42:30 var uploadedFile = new(File(uploadDirectory, fullName);                       7
42:11 var(uploadedFile = new File(uploadDirectory, fullName);                       8
43:7  (uploadedFile.createNewFile();                                                9
43:7  (uploadedFile.createNewFile());                                     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_rsa
```

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 | PT

> 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") MultipartFile file)`{	SOURCE	0
35:7	`@RequestParam("uploadedFileRemoveUserInput") MultipartFile file)`{		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,`String fullName)`{		4
42:52	var uploadedFile = new File(uploadDirectory,`fullName)`;		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_rsa
```

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 | PT

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("file") MultipartFile myFile) throws IOException {	SOURCE	0
74:34	public ModelAndView importFile(@RequestParam("file") MultipartFile myFile) throws IOException {		1
78:48	myFile.transferTo(new File(destinationDir, myFile. getOriginalFilename()));		2
78:48	myFile.transferTo(new File(destinationDir, myFile.getOriginalFilename( )));		3
78:27	myFile.transferTo(new File(destinationDir, myFile.getOriginalFilename()));		4
78:23	myFile.transferTo(new File(destinationDir, myFile.getOriginalFilename())) ;		5
78:5	myFile.transferTo(new File(destinationDir, myFile.getOriginalFilename()));	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_rsa
```

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 | NonCryptoHardcodedSecret

> 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.

Use of Insufficiently Random Values - Secrets

SNYK-CODE | CWE-330 | InsecureSecret

Insecure random data flows from org.apache.commons.lang3.RandomStringUtils.randomAlphabetic and is used as a secret data. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/main/java/org/owasp/webgoat/lessons/spoofcookie/encoders/EncDec.java (line : 40)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/spoofcookie/encoders/EncDec.java

40:31	`private static final String SALT = RandomStringUtils.randomAlphabetic(10);`	SOURCE	0
40:38	`private static final String SALT = RandomStringUtils.randomAlphabetic(10);`		1
40:38	`private static final String SALT = RandomStringUtils.randomAlphabetic(10);`		2
40:31	`private static final String SALT = RandomStringUtils.randomAlphabetic(10);`	SINK	3

Fix Analysis

Details

Computer security relies on random numbers for many things: generating secure, confidential session keys; hashing password data; encryption for transmitting sensitive data, and more. It's easy to understand why. If session keys, for example, were generated sequentially, attackers would be able to guess these easily and then hijack legitimate user sessions. Similarly, if encryption techniques used easy-to-guess numbers, attackers could use brute-force attacks to gain unauthorized access.

In reality, since computers cannot generate truly random numbers, they use "pseudorandom" numbers instead, generated using an algorithm that is "seeded" in a variety of ways to produce highly variable values in a random-seeming order, making them very hard-in theory-for attackers to guess. However, if developers inadvertently make use of a weak random algorithm, attackers may be able to discover the algorithm, seed, or pattern, ultimately unlocking access to commands or sensitive data, which can then be held for ransom or sold.

Best practices for prevention

- Avoid using weak pseudorandom number generators (PRNGs), such as statistical PRNGs. Instead, choose a cryptographically secure PRNG.
- Avoid using predictable seed values, such as user ID or server start time. Instead, use a seed that is itself pseudorandom, such as one taken from an external hardware source.
- Use standard, accepted security algorithms and libraries rather than taking a DIY approach and creating custom code that may contain inherent weaknesses or overlook critical flaws.
- Use static analysis tools to identify potential instances of this weakness in code and then ensure good test coverage with appropriate white-box testing.
- Educate developers about the importance of entropy in security systems development, and consider adopting tools that are FIPS 140-2 compliant.

Deserialization of Untrusted Data

SNYK-CODE | CWE-502 | Deserialization

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 String token) throws IOException {`	SOURCE	0
49:33	`public AttackResult completed(@RequestParam String token) throws IOException {`		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
58:83	`new ObjectInputStream(new ByteArrayInputStream(Base64.getDecoder().decode(b64token)))) {`		6
58:56	`new ObjectInputStream(new ByteArrayInputStream(Base64.getDecoder().decode(b64token)))) {`		7
58:35	`new ObjectInputStream(new ByteArrayInputStream(Base64.getDecoder().decode(b64token)))) {`		8
58:31	`new ObjectInputStream(new ByteArrayInputStream(Base64.getDecoder().decode(b64token)))) {`		9
58:13	`new ObjectInputStream(new ByteArrayInputStream(Base64.getDecoder().decode(b64token)))) {`	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 | Deserialization

> 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(@RequestParam String payload) {`	SOURCE	0
40:47	`public @ResponseBody AttackResult completed(@RequestParam String payload) {`		1
50:13	`payload`		2
50:13	`payload .replace("+", "")`		3
50:13	`payload .replace("+", "") .replace("¥r", "")`		4
50:13	`payload .replace("+", "") .replace("¥r", "") .replace("¥n", "")`		5

50:13 `payload .replace("+", "") .replace("¥r", "") .replace("¥n", "") .replace(`"> ", ">") 6

50:13 `payload .replace("+", "") .replace("¥r", "") .replace("¥n", "") .replace("> ", ">") .replace(`" <", "<"); 7

```
payload =
                  payload
                      .replace("+", "")
                      .replace("¥r", "")
                      .replace("¥n", "")
                      .replace("> ", ">")
                      .replace(" <", "<");
```
49:9 — 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).

Cross-Site Request Forgery (CSRF)

SNYK-CODE | CWE-352 | DisablesCSRFProtection

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.

Cross-Site Request Forgery (CSRF)

SNYK-CODE | CWE-352 | DisablesCSRFProtection

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 : 60)**

Data Flow

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

```
60:5  security.and().csrf().disable()).formLogin().loginPage("/login").failureUrl("/login?error=true");   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.

XML External Entity (XXE) Injection

SNYK-CODE | CWE-611 | XXE

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 String commentStr){   SOURCE 0
87:34  public AttackResult addComment(@RequestBody String commentStr){   1
96:43  Comment comment = comments.parseXml(commentStr);                   2
```

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

```
96:30   protected Comment parseXml(String xml)throws JAXBException, XMLStreamException {          3
105:58  var xsr = xif.createXMLStreamReader(new StringReader(xml));                               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 StringReader(xml));          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 | XXE

> 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 JAXBException, XMLStreamException {          3
105:58  var xsr = xif.createXMLStreamReader(new StringReader(xml));                               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 StringReader(xml));          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 | XXE

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(HttpServletRequest request, @RequestBody String commentStr) {   SOURCE  0
73:68  public AttackResult createNewComment(HttpServletRequest request, @RequestBody String commentStr) {           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 JAXBException, XMLStreamException {                3
105:58 var xsr = xif.createXMLStreamReader(new StringReader(xml));                                       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 StringReader(xml));                                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.

Server-Side Request Forgery (SSRF)

SNYK-CODE | CWE-918 | Ssrf

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
75:29  fakeClickingLinkEmail(host, resetLink);                                   3
100:38 private void fakeClickingLinkEmail(String host, String resetLink) {       4
106:80 String.format("http://%s/PasswordReset/reset/reset-password/%s",host,resetLink),  5
106:15 String.format("http://%s/PasswordReset/reset/reset-password/%s", host, resetLink),  6
104:7  new RestTemplate()                                                SINK    7
                    .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://`.

SQL Injection

SNYK-CODE | CWE-89 | Sqli

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 : 87)**

⇣ Data Flow

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

```
84:53  final String kid = (String)header.get("kid");                            SOURCE  0
84:53  final String kid = (String)header.get("kid");                                    1
84:44  final String kid = (String) header.get("kid");                                   2
84:38  final String kid = (String) header.get("kid");                                   3
90:81  "SELECT key FROM jwt_keys WHERE id = '" + kid + "'");                            4
90:39  "SELECT key FROM jwt_keys WHERE id = '" + kid + "'");                            5
90:39  "SELECT key FROM jwt_keys WHERE id = '" + kid + "'");                            6
87:31  connection                                                        SINK    7
                                    .createStatement()
                                    .executeQuery(
```

✓ 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.

Path Traversal

SNYK-CODE | CWE-23 | PT

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
55:50	File mavenWrapperPropertyFile = new File(baseDirectory, MAVEN_WRAPPER_PROPERTIES_PATH);	4
55:45	File mavenWrapperPropertyFile = new File(baseDirectory, MAVEN_WRAPPER_PROPERTIES_PATH);	5
55:14	File mavenWrapperPropertyFile = new File(baseDirectory, MAVEN_WRAPPER_PROPERTIES_PATH);	6
57:12	if(mavenWrapperPropertyFile.exists()) {	7
57:12	if(mavenWrapperPropertyFile.exists()) {	8
57:9	(code block)	SINK 9

```
if(mavenWrapperPropertyFile.exists()) {
                    FileInputStream mavenWrapperPropertyFileInputStream = null;
                    try {
                        mavenWrapperPropertyFileInputStream = new FileInputStream(mavenWrapperPropertyFile);
                        Properties mavenWrapperProperties = new Properties();
                        mavenWrapperProperties.load(mavenWrapperPropertyFileInputStream);
                        url = mavenWrapperProperties.getProperty(PROPERTY_NAME_WRAPPER_URL, url);
                    } catch (IOException e) {
                        System.out.println("- ERROR loading '" + MAVEN_WRAPPER_PROPERTIES_PATH + "'");
                    } finally {
                        try {
                            if(mavenWrapperPropertyFileInputStream != null) {
                                mavenWrapperPropertyFileInputStream.close();
                            }
                        } catch (IOException e) {
                            // Ignore ...
                        }
                    }
                }
```

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_rsa
```

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 | PT

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
78:36	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		4
78:36	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		5
78:31	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		6
78:14	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		7
79:13	`if(!outputFile.getParentFile().exists()) {`		8
79:13	`if(!outputFile.getParentFile().exists()) {`		9
79:13	`if(!outputFile.getParentFile().exists()) {`		10
79:9	`if(!outputFile.getParentFile().exists()) { if(!outputFile.getParentFile().mkdirs()) { System.out.println( "- ERROR creating output directory '" + outputFile.getParentFile().getAbsolutePath } }`	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_rsa
```

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 | PT

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
55:50	File mavenWrapperPropertyFile = new File(`baseDirectory,` MAVEN_WRAPPER_PROPERTIES_PATH);		4
55:45	File mavenWrapperPropertyFile = new`File(` baseDirectory, MAVEN_WRAPPER_PROPERTIES_PATH);		5
55:14	File`mavenWrapperPropertyFile = new File(baseDirectory, MAVEN_WRAPPER_PROPERTIES_PATH);`		6
60:75	mavenWrapperPropertyFileInputStream = new FileInputStream(`mavenWrapperPropertyFile)`;		7
60:59	mavenWrapperPropertyFileInputStream = new`FileInputStream(` mavenWrapperPropertyFile);	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_rsa
```

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 | PT

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
78:36	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		4
78:36	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		5
78:31	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		6
78:14	`File outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		7
80:17	`if(!outputFile.getParentFile().mkdirs()) {`		8
80:17	`if(!outputFile.getParentFile().mkdirs()) {`		9
80:17	`if(!outputFile.getParentFile().mkdirs()) {`	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_rsa
```

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 | PT

> 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
78:36	File outputFile = new File(`baseDirectory.` getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);		4
78:36	File outputFile = new File(`baseDirectory.getAbsolutePath()`), MAVEN_WRAPPER_JAR_PATH);		5
78:31	File outputFile = new`File(` baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);		6
78:14	File`outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);`		7
87:38	downloadFileFromURL(url,`outputFile)`;		8
97:63	private static void downloadFileFromURL(String urlString,`File destination)` throws Exception {		9
111:53	FileOutputStream fos = new FileOutputStream(`destination)`;		10
111:36	FileOutputStream fos = new`FileOutputStream(` destination);	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_rsa
```

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
```

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/challenges/challenge1/Assignment1.java (line : 55)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/challenges/challenge1/Assignment1.java

```
51:38  @RequestParam String username, @RequestParam String password, HttpServletRequest request) {   SOURCE 0
55:16  && PASSWORD                                                                                    SINK   1
                        .replace("1234", String.format("%04d", ImageServlet.PINCODE))
                        .equals(
                                                                                                      password);
```

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/idor/IDORLogin.java (line : 64)**

Data Flow

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

```
59:64  public AttackResult completed(@RequestParam String username, @RequestParam String password) {   SOURCE 0
64:37  if ("tom".equals(username) && idorUserInfo.get("tom").get("password").equals( password)) {       SINK   1
```

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in

the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

> An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.
>
> 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

35:64	`public AttackResult completed(@RequestParam String username,` `@RequestParam String password)` `{`	SOURCE	0
36:43	`if (”CaptainJack”.equals(username) &&` `”BlackPearl”.equals(` `password)) {`	SINK	1

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

> An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java (line : 60)**

Data Flow

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

55:64	`public AttackResult completed(@RequestParam String username,` `@RequestParam String password)` `{`	SOURCE	0

60:37	if (username.equals("Admin") && password.equals(this.password)) {	SINK	1

✓ Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java (line : 90)**

⇣ Data Flow

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

84:29	public AttackResult login(@RequestParam String password, @RequestParam String email) {	SOURCE	0
90:18	} else if (passwordTom.equals(password)) {	SINK	1

✓ Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (line : 85)**

Data Flow

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

63:7	`@RequestParam String password,`	SOURCE	0
85:12	`&& users.get(lowerCasedUsername).equals( password)) {`	SINK	1

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (line : 90)**

Data Flow

63:7	`@RequestParam String password,`	SOURCE	0
90:36	`if (!authPassword.isBlank() && authPassword.equals( password)) {`	SINK	1

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.

- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of weakAntiCSRF because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/csrf/ForgedReviews.java (line : 106)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/csrf/ForgedReviews.java

56:31	`private static final String weakAntiCSRF = "2aa14227b9a13d0bede0388a7fba9aa9";`	SOURCE	0
106:33	`if (validateReq == null || !validateReq.equals(weakAntiCSRF)) {`	SINK	1

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

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
18:42	`$("#nrOfVotes" + i).html(votes[i]);`		2
18:42	`$("#nrOfVotes" + i).html(votes[i]);`		3
18:37	`$("#nrOfVotes" + i).html(votes[i]);`	SINK	4

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.

- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

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	`$.get("challenge/8/vote/" + stars, function (result){`	SOURCE	0
46:50	`$.get("challenge/8/vote/" + stars, function (result){`		1
52:34	`$("#voteResultMsg").html(result["message"]);`		2
52:41	`$("#voteResultMsg").html(result["message"]);`		3
52:34	`$("#voteResultMsg").html(result["message"]);`		4
52:29	`$("#voteResultMsg").html(result["message"]);`	SINK	5

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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/view/LessonContentView.js (line : 178)**

Data Flow

src/main/resources/webgoat/static/js/goatApp/view/LessonContentView.js

117:35	}).then(function ([data){	SOURCE	0
117:35	}).then(function ([data){		1
118:44	self.onSuccessResponse([data,failureCallbackFunctionName, successCallBackFunctionName, informationalCallbackFunctionName)		2
123:42	onSuccessResponse: function ([data,failureCallbackFunctionName, successCallBackFunctionName, informationalCallbackFunctio		3
124:37	this.renderFeedback([data.feedback);		4
124:42	this.renderFeedback(data.[feedback);		5
176:39	renderFeedback: function ([feedback){		6
177:52	var s = this.removeSlashesFromJSON([feedback);		7
169:46	removeSlashesFromJSON: function ([str){		8
173:24	return[str.replace(/¥¥(.)/g, "$1");		9
173:28	return str.[replace(/¥¥(.)/g, "$1");		10
177:21	var[s= this.removeSlashesFromJSON(feedback);		11
178:51	this.$curFeedback.html(polyglot.t([s)|| "");		12
178:49	this.$curFeedback.html(polyglot.[t(s) || "");		13
178:40	this.$curFeedback.html([polyglot.t(s) || "")		14
178:35	this.$curFeedback.[html(polyglot.t(s) || "");	SINK	15

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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/view/LessonContentView.js (line : 185)**

Data Flow

117:35	}).then(function (data){	SOURCE 0
117:35	}).then(function (data){	1
118:44	self.onSuccessResponse(data, failureCallbackFunctionName, successCallBackFunctionName, informationalCallbackFunctionName)	2
123:42	onSuccessResponse: function (data, failureCallbackFunctionName, successCallBackFunctionName, informationalCallbackFunctic	3
125:35	this.renderOutput(data.output || "");	4
125:40	this.renderOutput(data.output || "");	5
125:35	this.renderOutput(data.output || "");	6
125:35	this.renderOutput(data.output || "");	7
183:37	renderOutput: function (output){	8
184:52	var s = this.removeSlashesFromJSON(output);	9
169:46	removeSlashesFromJSON: function (str){	10
173:24	return str.replace(/¥¥(.)/g, "$1");	11
173:28	return str.replace(/¥¥(.)/g, "$1");	12
184:21	var s = this.removeSlashesFromJSON(output);	13
185:49	this.$curOutput.html(polyglot.t(s) || "");	14
185:47	this.$curOutput.html(polyglot.t(s) || "");	15
185:38	this.$curOutput.html(polyglot.t(s) || "");	16
185:33	this.$curOutput.html(polyglot.t(s) || "");	SINK 17

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

Use of Hardcoded Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in here.

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

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in here.

Found in: **src/main/java/org/owasp/webgoat/lessons/challenges/SolutionConstants.java (line : 35)**

Data Flow

35:25	`String PASSWORD_TOM = "thisisasecretfortomonly";`		SOURCE SINK	0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in here.

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", "buffalo");`	SOURCE SINK	0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in here.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 61)**

Data Flow

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

61:41	`public static final String PASSWORD = "bm5nhSkxCXZkKRy4";`	SOURCE SINK	0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in here.

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_SIMPLE = "DeliberatelyInsecure1234";
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in here.

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 = "DeliberatelyInsecure1235";
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in here.

Found in: **src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java (line : 62)**

Data Flow

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

62:7 `"somethingVeryRandomWhichNoOneWillEverTypeInAsPasswordForTom";` SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in equals.

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) && "BlackPearl".equals(password)) {` SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated

privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in equals.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 77)**

Data Flow

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

```
77:43   if ("Jerry".equalsIgnoreCase(user) && PASSWORD.equals(password)) {
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in equals.

Found in: **src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java (line : 88)**

Data Flow

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

```
88:11   if (passwordTom.equals(PASSWORD_TOM_9)) {
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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.

Improper Neutralization of CRLF Sequences in HTTP Headers

SNYK-CODE | CWE-113 | HttpResponseSplitting

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
63:7   @RequestParam String username,                                                                          1
72:45  Authentication.builder().name(username).credentials(password).build());                                 2
72:15  Authentication.builder().name(username).credentials(password).build());                                 3
72:15  Authentication.builder().name(username).credentials(password).build());                                 4
72:15  Authentication.builder().name(username).credentials(password).build()));                                5
71:11  provider.authenticate(                                                                                  6
70:7   authentication =                                                                                        7
                   provider.authenticate(
                       Authentication.builder().name(username).credentials(password).build());
73:27  setCookie(response,authentication.getId());                                                             8
73:27  setCookie(response,authentication.getId()));                                                            9
85:56  private void setCookie(HttpServletResponse response,String cookieValue){                                10
86:45  Cookie cookie = new Cookie(COOKIE_NAME,cookieValue);                                                     11
86:25  Cookie cookie = new Cookie(COOKIE_NAME, cookieValue);                                                   12
86:12  Cookie cookie = new Cookie(COOKIE_NAME, cookieValue);                                                   13
89:24  response.addCookie(cookie);                                                                            14
89:5   response.addCookie(cookie);                                                                    SINK    15
```

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 | HttpResponseSplitting

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 response) {		3
83:33	String lowerCasedUsername =`username.` toLowerCase();		4
83:33	String lowerCasedUsername =`username.toLowerCase()`;		5
83:12	String`lowerCasedUsername = username.toLowerCase();`		6
91:45	String newCookieValue = EncDec.encode(`lowerCasedUsername)`;		7
91:31	String newCookieValue =`EncDec.encode(` lowerCasedUsername);		8
91:14	String`newCookieValue = EncDec.encode(lowerCasedUsername);`		9
92:50	Cookie newCookie = new Cookie(COOKIE_NAME,`newCookieValue)`;		10
92:30	Cookie newCookie = new`Cookie(` COOKIE_NAME, newCookieValue);		11
92:14	Cookie`newCookie = new Cookie(COOKIE_NAME, newCookieValue);`		12
95:26	response.addCookie(`newCookie)`;		13
95:7	`response.addCookie(` newCookie);		14

SINK

✓ 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.

Unprotected Storage of Credentials

SNYK-CODE | CWE-256 | ReturnsPassword

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 String username, @RequestParam String password){ 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.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/passwordreset/SimpleMailAssignment.java (line : 65)**

↓ Data Flow

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

60:57 public AttackResult login(@RequestParam String email, @RequestParam String password){ SOURCE 0

65:12 && StringUtils.reverse(username).equals(password)) { SINK 1

✓ Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of password because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6b.java (line : 50)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6b.java

```
58:12  String password = "dave";                          SOURCE  0
50:9   if (userid_6b.equals(getPassword())) {             SINK    1
```

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

> An attacker can guess the secret value of getUserHash because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionACYourHash.java (line : 55)**

Data Flow

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

```
55:25  if (userHash.equals(displayUser.getUserHash())) {  SOURCE  0
55:9   if (userHash.equals(displayUser.getUserHash())) {  SINK    1
```

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of getUserHash because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionACYourHashAdmin.java (line : 62)**

Data Flow

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

62:25	`if (userHash.equals(displayUser.getUserHash())) {`	SOURCE	0
62:9	`if (userHash.equals(displayUser.getUserHash())) {`	SINK	1

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of uniqueCode because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/webwolfintroduction/LandingAssignment.java (line : 51)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/webwolfintroduction/LandingAssignment.java

50:29	`public AttackResult click(String uniqueCode){`	SOURCE	0
51:9	`if (StringUtils.reverse(getWebSession().getUserName()).equals(uniqueCode)) {`	SINK	1

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

Observable Timing Discrepancy (Timing Attack)

SNYK-CODE | CWE-208 | TimingAttack

An attacker can guess the secret value of uniqueCode because it is compared using equals, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Found in: **src/main/java/org/owasp/webgoat/lessons/webwolfintroduction/MailAssignment.java (line : 86)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/webwolfintroduction/MailAssignment.java

85:33	`public AttackResult completed(@RequestParam String uniqueCode){`	SOURCE	0
86:9	`if (uniqueCode.equals(StringUtils.reverse(getWebSession().getUserName()))) {`	SINK	1

Fix Analysis

Details

A timing attack is a form of side-channel attack, meaning it does not take advantage of the way the code is structured, but rather exploits external clues that let an attacker infer the program's state. In a timing attack, program state is inferred from the time it takes to execute a particular operation. For example, an app might use a lookup table of valid session IDs to speed up access. While this is convenient for validated users, an invalid session ID will take far longer to be rejected (since it's not in the lookup table), giving the attackers a valuable way to engineer a brute-force attack. All they need to do is test a large number of randomly generated session IDs in the hope of discovering a valid one. Once access is granted, through the session ID, the malicious actor may impersonate a legitimate user, executing actions or freely accessing secure data. Through brute force, such as a massive bot attack, this weakness can be successfully exploited to bypass even very strong encryption algorithms.

Best practices for prevention

- Implement a constant-time algorithm to ensure that timing is identical regardless of input validity and returned outcome.
- If a constant-time algorithm is impractical due to a desire to optimize performance, choose another technique such as blinding.
- Promote an awareness among developers that encryption is not enough if your app is giving off clues to help attackers.
- Timing attacks often depend on the production environment and are thus difficult to test during development; be sure to use a staging environment as close as possible to production.
- Choose hardened, reliable libraries for encryption and authentication with side-channel attack protection strategies rather than implementing your own methods.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

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/idor/js/idor.js (line : 4)**

Data Flow

src/main/resources/lessons/idor/js/idor.js

3:45	`webgoat.customjs.idorViewProfile = function(data){`	SOURCE	0
3:45	`webgoat.customjs.idorViewProfile = function(data){`		1
5:19	`'name:' +data.name + ' ' +`		2
5:24	`'name:' + data.name+ ' ' +`		3
5:19	`'name:' +data.name+ ' ' +`		4

```
5:9   'name:' + data.name + '<br/>' +                                                                  5
5:9   'name:' + data.name + '<br/>' +                                                                  6
5:9   'name:' + data.name + '<br/>' +
                      'color:' + data.color + '<br/>' +                                                7
5:9   'name:' + data.name + '<br/>' +
                      'color:' + data.color + '<br/>' +                                                8
5:9   'name:' + data.name + '<br/>' +
                      'color:' + data.color + '<br/>' +                                                9
5:9   'name:' + data.name + '<br/>' +
                      'color:' + data.color + '<br/>' +
                      'size:' + data.size + '<br/>'                                                    10
5:9   'name:' + data.name + '<br/>' +
                      'color:' + data.color + '<br/>' +
                      'size:' + data.size + '<br/>'                                                    11
5:9   'name:' + data.name + '<br/>' +
                      'color:' + data.color + '<br/>' +
                      'size:' + data.size + '<br/>'                                                    12
4:46  webgoat.customjs.jquery('#idor-profile').html(                                             SINK  13
```

✓ Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms

- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout() / setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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/lessontemplate/js/idor.js (line : 4)**

Data Flow

src/main/resources/lessons/lessontemplate/js/idor.js

3:45	`webgoat.customjs.idorViewProfile = function(data){`	SOURCE	0
3:45	`webgoat.customjs.idorViewProfile = function(data){`		1

```
5:19  'name:' +data.name + '<br/>'+                                                    2
5:24  'name:' + data.name + '<br/>'+                                                   3
5:19  'name:' +data.name + '<br/>'+                                                    4
5:9   'name:' + data.name + '<br/>'+                                                   5
5:9   'name:' + data.name + '<br/>'+                                                   6
5:9   'name:' + data.name + '<br/>'+ data.color + '<br/>'+                             7
                'color:'
5:9   'name:' + data.name + '<br/>'+ 'color:' + data.color + '<br/>'+                  8
5:9   'name:' + data.name + '<br/>'+                                                   9
                'color:' + data.color + '<br/>'+
5:9   'name:' + data.name + '<br/>'+ data.size + '<br/>'                               10
                'color:' + data.color + '<br/>'+
                'size:'
5:9   'name:' + data.name + '<br/>'+ '<br/>'                                           11
                'color:' + data.color + '<br/>'+
                'size:' + data.size
5:9   'name:' + data.name + '<br/>'+                                                   12
                'color:' + data.color + '<br/>'+
                'size:' + data.size + '<br/>'
4:46  webgoat.customjs.jquery('#idor-profile').html(   SINK                            13
```

✓ Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout() / setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

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

✓ Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.

- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout() / setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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	`$.get("clientSideFiltering/salaries?userId=" + userId, function (result, status) {`	SOURCE	0
17:70	`$.get("clientSideFiltering/salaries?userId=" + userId, function (result, status) {`		1
27:42	`html = html + '<tr id = "' + result[i].UserID + '"</tr>';`		2
27:42	`html = html + '<tr id = "' + result[i].UserID + '"</tr>';`		3
27:52	`html = html + '<tr id = "' + result[i].UserID + '"</tr>';`		4
27:42	`html = html + '<tr id = "' + result[i].UserID + '"</tr>';`		5
27:20	`html = html + '<tr id = "' + result[i].UserID + '"</tr>';`		6
27:20	`html = html + '<tr id = "' + result[i].UserID + '"</tr>';`		7
27:13	`html = html + '<tr id = "' + result[i].UserID + '"</tr>';`		8
28:20	`html = html + '<td>' + result[i].UserID + '</td>';`		9
28:20	`html = html + '<td>' + result[i].UserID + '</td>';`		10
28:20	`html = html + '<td>' + result[i].UserID + '</td>';`		11
28:20	`html = html + '<td>' + result[i].UserID + '</td>';`		12
28:13	`html = html + '<td>' + result[i].UserID + '</td>';`		13

29:20	`html = html + '<td>' + result[i].FirstName + '</td>';`			14
29:20	`html = html + '<td>' + result[i].FirstName + '</td>';`			15
29:20	`html = html + '<td>' + result[i].FirstName + '</td>';`			16
29:20	`html = html + '<td>' + result[i].FirstName + '</td>';`			17
29:13	`html = html + '<td>' + result[i].FirstName + '</td>';`			18
30:20	`html = html + '<td>' + result[i].LastName + '</td>';`			19
30:20	`html = html + '<td>' + result[i].LastName + '</td>';`			20
30:20	`html = html + '<td>' + result[i].LastName + '</td>';`			21
30:20	`html = html + '<td>' + result[i].LastName + '</td>';`			22
30:13	`html = html + '<td>' + result[i].LastName + '</td>';`			23
31:20	`html = html + '<td>' + result[i].SSN + '</td>';`			24
31:20	`html = html + '<td>' + result[i].SSN + '</td>';`			25
31:20	`html = html + '<td>' + result[i].SSN + '</td>';`			26
31:20	`html = html + '<td>' + result[i].SSN + '</td>';`			27
31:13	`html = html + '<td>' + result[i].SSN + '</td>';`			28
32:20	`html = html + '<td>' + result[i].Salary + '</td>';`			29
32:20	`html = html + '<td>' + result[i].Salary + '</td>';`			30
32:20	`html = html + '<td>' + result[i].Salary + '</td>';`			31
32:20	`html = html + '<td>' + result[i].Salary + '</td>';`			32
32:13	`html = html + '<td>' + result[i].Salary + '</td>';`			33
33:20	`html = html + '</tr>';`			34
33:20	`html = html + '</tr>';`			35
33:13	`html = html + '</tr>';`			36
35:16	`html = html + '</tr></table>';`			37
35:16	`html = html + '</tr></table>';`			38
35:9	`html = html + '</tr></table>';`			39
38:28	`newdiv.innerHTML = html;`			40
38:28	`newdiv.innerHTML = html;`		SINK	41

✓ Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be

Type	Source Category	Description	Technical Details
		and processed unsafely by application JavaScript.	combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

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
40:52	`comment = comment.replace('STARS',[result[]i].stars)`		2
40:52	`comment = comment.replace('STARS',[result[i].]stars)`		3
40:62	`comment = comment.replace('STARS', result[i].[stars)]`		4
40:52	`comment = comment.replace('STARS',[result[i].stars)]`		5
40:35	`comment = comment.[replace(]'STARS', result[i].stars)`		6
40:17	`[comment = comment.replace('STARS', result[i].stars)]`		7
41:35	`$("#list").append([comment)];`		8
41:28	`$("#list").[append(]comment);`	SINK	9

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout() / setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.

- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout() / setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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
56:60	`voteTemplate = voteTemplate.replace('AVERAGE',[result[]i].average || '');`		2
56:60	`voteTemplate = voteTemplate.replace('AVERAGE',[result[i].]average || '');`		3
56:70	`voteTemplate = voteTemplate.replace('AVERAGE', result[i].[average]|| '');`		4
56:60	`voteTemplate = voteTemplate.replace('AVERAGE',[result[i].average || '')];`		5
56:41	`voteTemplate = voteTemplate.[replace(]'AVERAGE', result[i].average || '');`		6
56:13	`[voteTemplate = voteTemplate.replace('AVERAGE', result[i].average || '');]`		7
59:28	`voteTemplate =[voteTemplate.]replace(/HIDDEN_VIEW_VOTES/g, hidden);`		8
59:41	`voteTemplate = voteTemplate.[replace(]/HIDDEN_VIEW_VOTES/g, hidden);`		9
59:13	`[voteTemplate = voteTemplate.replace(/HIDDEN_VIEW_VOTES/g, hidden);]`		10
61:28	`voteTemplate =[voteTemplate.]replace(/HIDDEN_VIEW_RATING/g, hidden);`		11
61:41	`voteTemplate = voteTemplate.[replace(]/HIDDEN_VIEW_RATING/g, hidden);`		12
61:13	`[voteTemplate = voteTemplate.replace(/HIDDEN_VIEW_RATING/g, hidden);]`		13

63:36 $("#votesList").append(voteTemplate);

63:29 $("#votesList").append(voteTemplate);

SINK

14

15

✓ Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout() / setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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	`$.get("SqlInjectionMitigations/servers?column=" + column, function ([result,]status) {`	SOURCE	0
43:73	`$.get("SqlInjectionMitigations/servers?column=" + column, function ([result,]status) {`		1
56:52	`server = server.replace('DESCRIPTION',[result[]i].description);`		2
56:52	`server = server.replace('DESCRIPTION',[result[i].]description);`		3
56:62	`server = server.replace('DESCRIPTION', result[i].[description)];`		4
56:52	`server = server.replace('DESCRIPTION',[result[i].description)];`		5
56:29	`server = server.[replace(]'DESCRIPTION', result[i].description);`		6
56:13	`[server = server.replace('DESCRIPTION', result[i].description);]`		7
57:34	`$("#servers").append([server)];`		8
57:27	`$("#servers").[append(]server);`	SINK	9

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

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	`$.get('CrossSiteScripting/stored-xss', function (result, status) {`	SOURCE	0
35:58	`$.get('CrossSiteScripting/stored-xss', function (result, status) {`		1
39:54	`comment = comment.replace('COMMENT',result[i].text);`		2
39:54	`comment = comment.replace('COMMENT',result[i].text);`		3
39:64	`comment = comment.replace('COMMENT', result[i].text);`		4
39:54	`comment = comment.replace('COMMENT',result[i].text);`		5
39:35	`comment = comment.replace('COMMENT', result[i].text);`		6
39:17	`comment = comment.replace('COMMENT', result[i].text);`		7
40:35	`$("#list").append(comment);`		8
40:28	`$("#list").append(comment);`	SINK	9

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from attacker-controlled websites linking to the vulnerable application.	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

DOM-based Cross-site Scripting (XSS)

SNYK-CODE | CWE-79 | DOMXSS

> 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
77:50  comment = comment.replace('COMMENT',[result[]i].text);                         2
77:50  comment = comment.replace('COMMENT',[result[i].]text);                         3
77:60  comment = comment.replace('COMMENT', result[i].[text)];                        4
77:50  comment = comment.replace('COMMENT',[result[i].text)];                         5
77:31  comment = comment.[replace('COMMENT', result[i].text);                         6
77:13  [comment = comment.replace('COMMENT', result[i].text);]                        7
78:29  $(field).append([comment);                                                     8
78:22  $(field).[append(]comment);                                            SINK    9
```

Fix Analysis

Details

DOM-based Cross-Site Scripting (DOM XSS) is a client-side vulnerability in which attacker-controlled data (e.g., from `location.search`, `location.hash`, `document.referrer`, browser storage, `postMessage`, WebSockets) is read by JavaScript and written to dangerous DOM/JS sinks without proper validation or encoding, causing code to execute entirely in the browser. The payload may be delivered by the server or other channels, but—unlike reflected or stored XSS—the server does not perform the injection/echo that triggers execution; the critical issue is a tainted data flow from source to sink within client-side logic, which traditional server-side defenses and many WAFs often miss.

Effective mitigation of DOM XSS vulnerabilities requires implementing comprehensive input validation and output encoding strategies specifically designed for client-side execution contexts. This includes establishing strict allowlisting mechanisms for user-controllable data, implementing context-aware output encoding based on the specific DOM operation being performed, and adopting secure coding practices that minimize the attack surface exposed through dangerous DOM/JavaScript APIs. Additionally, modern web applications should leverage Content Security Policy (CSP) directives and other browser security features to establish defense-in-depth protections against DOM-based attacks.

DOM XSS attacks have been extensively documented in security research and are frequently exploited to achieve various malicious objectives, including session hijacking through cookie theft, credential harvesting via phishing attacks, unauthorized actions performed on behalf of authenticated users, client-side malware distribution, and sensitive data exfiltration. The client-side nature of these attacks makes them effective at bypassing traditional perimeter security controls and targeting users directly within their browser environment.

Types of attacks

Type	Source Category	Description	Technical Details
URL Parameter Injection	Navigation	Malicious payloads embedded within URL query parameters that are processed by vulnerable JavaScript code accessing `location.search` or similar properties.	Commonly exploits `URLSearchParams`, direct string parsing, or framework routing mechanisms that unsafely process query parameters (including History API updates like `pushState`).
Fragment Identifier Exploitation	Navigation	Attack vectors utilizing URL hash fragments (`#`) that are accessible through `location.hash` and processed by client-side routing or content loading mechanisms.	Particularly prevalent in Single Page Applications (SPAs) with client-side routing frameworks that process hash-based navigation.
HTTP Referrer Manipulation	Request Context	Exploitation of `document.referrer` values that contain malicious payloads, typically originating from	Requires social engineering to direct users from attacker-controlled domains with crafted referrer values.

Type	Source Category	Description	Technical Details
		attacker-controlled websites linking to the vulnerable application.	
Browser Storage Attacks	Persistent Storage	Malicious data injection into `localStorage`, `sessionStorage`, or IndexedDB that is subsequently read and processed unsafely by application JavaScript.	Can persist across sessions if the attacker can get data stored (via another bug or social engineering) and may be combined with other vectors for staged exploitation.
Cross-Frame Communication	Inter-Frame Messaging	Exploitation of `postMessage` APIs where malicious data is transmitted between frames or windows and processed without proper origin validation and input sanitization.	Common in applications with embedded iframes or popup windows that implement cross-origin communication mechanisms.
WebSocket Message Injection	Real-time Communication	Malicious payloads delivered through WebSocket connections that are processed by client-side message handlers without adequate input validation.	Particularly effective against real-time applications like chat systems, collaborative tools, or live data feeds.

Affected environments

DOM XSS vulnerabilities can impact various categories of web applications and client-side environments:

- **Single Page Applications (SPAs)** utilizing frameworks such as React, Angular, Vue.js, or custom JavaScript implementations
- **Progressive Web Applications (PWAs)** with extensive client-side functionality and offline capabilities
- **Client-side routing implementations** including hash-based and HTML5 History API routing mechanisms
- **Browser extensions and add-ons** with content script injection and web page interaction capabilities
- **Hybrid mobile applications** utilizing WebView components and JavaScript bridge implementations
- **Rich Internet Applications (RIAs)** with complex client-side business logic and DOM manipulation
- **Content Management Systems (CMS)** with client-side editing interfaces and dynamic content rendering
- **Social media platforms and widgets** implementing client-side content embedding and sharing functionality

Best practices for prevention

This section outlines comprehensive security measures designed to prevent DOM XSS vulnerabilities through secure development practices and technical controls.

Input Validation and Sanitization

- **Implement strict input validation** for all client-side data sources including URL parameters, fragment identifiers, referrer values, storage mechanisms, and cross-frame communications using allowlisting approaches rather than blacklisting methodologies.
- **Establish contextual output encoding** appropriate for the specific DOM context where data will be utilized, including HTML entity encoding, JavaScript string escaping, CSS value encoding, and URL parameter encoding.
- **Validate data origin and integrity** for cross-frame `postMessage` communications by implementing strict origin checking and message format validation to prevent malicious data injection from untrusted sources.
- **Sanitize persistent storage data** by validating all data retrieved from `localStorage`, `sessionStorage`, or other client-side storage mechanisms before processing or rendering within the application.

Secure DOM Manipulation Practices

- **Avoid high-risk DOM sinks** including `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write()`, `document.writeln()`, `eval()`, `Function()`, `setTimeout()` / `setInterval()` with string arguments when processing user-controllable data.
- **Utilize safer DOM manipulation methods** such as `textContent`, `createTextNode()`, and DOM element creation APIs that do not interpret content as executable code. Use `setAttribute()` only for non-URL/non-event attributes and never to set `on*` handlers or dangerous URL attributes from untrusted data.
- **Implement secure templating mechanisms** using frameworks or libraries that provide automatic output encoding and XSS protection rather than manual string concatenation for dynamic content generation.
- **Enforce separation of data and code** by avoiding the construction of JavaScript code through string concatenation with user-controllable data and utilizing structured data formats like JSON for data transmission.

Browser Security Feature Implementation

- **Deploy comprehensive Content Security Policy (CSP)** directives (e.g., `script-src`, `object-src`, `base-uri`, `frame-ancestors`) to limit script execution and reduce impact if a DOM XSS exists.
- **Implement Subresource Integrity (SRI)** for all external JavaScript resources to prevent tampering with third-party libraries that could introduce DOM XSS vulnerabilities.
- **Utilize HTTP security headers** including `X-Content-Type-Options: nosniff`, `Referrer-Policy`, and (if CSP is not available) `X-Frame-Options`, while preferring CSP's `frame-ancestors` directive for clickjacking protection.
- **Adopt modern browser defenses** such as Trusted Types (where supported) to prevent assignment to DOM sinks unless passed through an approved sanitizer. Treat CORS as an access-control mechanism—not a DOM XSS control—and configure it appropriately to limit unnecessary cross-origin data exposure.

Development and Testing Practices

- **Conduct regular security code reviews** focusing specifically on client-side data flow analysis to identify potential DOM XSS vulnerabilities during the development process.
- **Implement automated static analysis** using specialized tools capable of detecting DOM XSS vulnerabilities in JavaScript code and identifying unsafe data flows from sources to sinks.
- **Perform dynamic application security testing (DAST)** with specialized DOM XSS detection capabilities and manual penetration testing to validate the effectiveness of implemented security controls.
- **Establish secure coding guidelines** for development teams including training on DOM XSS attack vectors, secure JavaScript programming practices, and proper use of security-focused libraries and frameworks.

Use of Hardcoded Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword

Do not hardcode passwords in code. Found hardcoded password used in equals.

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

40:32	`if (answer_pwd1 != null && answer_pwd1.equals("databasepassword")) {`	SOURCE SINK	0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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.

Improper Neutralization of CRLF Sequences in HTTP Headers

SNYK-CODE | CWE-113 | HttpResponseSplitting

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<String, String> formData){`	SOURCE	0
35:26	`public JWTToken encode(@RequestBody MultiValueMap<String, String> formData){`		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 payloadAsString, String secretKey) {`		6
67:25	`var headers = parse(header);`		7
45:44	`private static Map<String, Object> parse(String header){`		8
48:31	`return reader.readValue(header, TreeMap.class);`		9
48:14	`return reader.readValue(header, TreeMap.class);`		10

80:25	`headers.forEach((k,(v))-> jws.setHeader(k, v));`		11
80:48	`headers.forEach((k, v) -> jws.setHeader(k,(v)));`		12
80:31	`headers.forEach((k, v) ->(jws.setHeader(k, v));`	SINK	13

✓ 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 Passwords

SNYK-CODE | CWE-798,CWE-259 | NoHardcodedPasswords

> 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: "bm5nhSkxCXZkKRy4"}) SOURCE SINK 0

✓ Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 | TooPermissiveCorsPostMessage

> 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

1740:13 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 Hardcoded Credentials

SNYK-CODE | CWE-798 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> 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("username", "CaptainJack");` 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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (line : 24)**

Data Flow

src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java

24:26 `userForm.setUsername("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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (line : 36)**

Data Flow

36:26 `userForm.setUsername("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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (line : 49)**

Data Flow

49:26 `userForm.setUsername("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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpointTest.java (line : 198)**

Data Flow

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

198:27 `loginJson.put("user", "Jerry");` 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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpointTest.java (line : 227)**

⇣ Data Flow

227:27 `loginJson.put("user", "Jerry");` 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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (line : 70)**

⇣ Data Flow

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

70:34 `Claims claims = createClaims("WebGoat");` 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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (line : 81)**

⇣ Data Flow

```
81:34  Claims claims = createClaims("webgoat");
```

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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (line : 92)**

⇣ Data Flow

```
92:34  Claims claims = createClaims("WebGoat");
```

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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (line : 133)**

Data Flow

```
133:34  Claims claims = createClaims("WebGoat");        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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpointTest.java (line : 226)**

Data Flow

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

```
226:24  claims.put("user","Intruder");        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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpointTest.java (line : 244)**

Data Flow

```
244:24  claims.put("user", "Intruder");                    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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignmentTest.java (line : 90)**

Data Flow

src/test/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignmentTest.java

```
90:23  String username = "webgoat";                        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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/webwolf/user/UserServiceTest.java (line : 48)**

Data Flow

src/test/java/org/owasp/webgoat/webwolf/user/UserServiceTest.java

```
48:20   var username = "guest";
```

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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/webwolf/user/UserServiceTest.java (line : 61)**

Data Flow

```
61:20   var username = "guest";
```

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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/test/java/org/owasp/webgoat/webwolf/user/UserServiceTest.java (line : 70)**

Data Flow

```
70:20   var username = "guest";
```

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 | NoHardcodedCredentials

> Do not hardcode credentials in code.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/challenges/challenge1/Assignment1.java (line : 54)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/challenges/challenge1/Assignment1.java

```
54:9   "admin".equals(username)
```

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 | NoHardcodedCredentials

Do not hardcode credentials in code.

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

```
36:9   if ("CaptainJack".equals(username) && "BlackPearl".equals(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 | NoHardcodedCredentials

Do not hardcode credentials in code.

Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpoint.java (line : 64)**

Data Flow

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

```
64:9   if ("Jerry".equals(user)) {
```

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 | NoHardcodedCredentials

> Do not hardcode credentials in code.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpoint.java (line : 106)**

Data Flow

```
106:13  if ("Jerry".equals(username)) {
```

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 | NoHardcodedCredentials

> Do not hardcode credentials in code.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (line : 77)**

Data Flow

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

```
77:9  if ("Jerry".equalsIgnoreCase(user) && PASSWORD.equals(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 | NoHardcodedCredentials

> Do not hardcode credentials in code.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (line : 54)**

Data Flow

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

```
54:46  private static final String WEBGOAT_USER = "WebGoat";
```

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 | NoHardcodedCredentials

> Do not hardcode credentials in code.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (line : 158)**

Data Flow

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

```
158:13  if ("Guest".equals(user) || !validUsers.contains(user)) {
```

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 | NoHardcodedCredentials

> Do not hardcode credentials in code.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java (line : 60)**

Data Flow

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

```
60:25  if (username.equals("Admin") && password.equals(this.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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword/test

> Do not hardcode passwords in code. Found hardcoded password used in here.
>
> Found in: **src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java (line : 109)**

Data Flow

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

```
109:28  params.put("password","BlackPearl");
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword/test

> Do not hardcode passwords in code. Found hardcoded password used in here.
>
> Found in: **src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java (line : 119)**

Data Flow

```
119:28  params.put("password",("ajnaeliclm^&&@kjn.");
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword/test

> Do not hardcode passwords in code. Found hardcoded password used in here.
>
> 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 passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword/test

> Do not hardcode passwords in code. Found hardcoded password used in here.
>
> 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 passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword/test

> Do not hardcode passwords in code. Found hardcoded password used in here.
>
> 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 passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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 Passwords

SNYK-CODE | CWE-798,CWE-259 | HardcodedPassword/test

Do not hardcode passwords in code. Found hardcoded password used in here.

Found in: **src/test/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignmentTest.java (line : 91)**

Data Flow

src/test/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignmentTest.java

```
91:23  String password = "webgoat";
```

SOURCE SINK 0

Fix Analysis

Details

Developers may use hardcoded passwords during development to streamline setup or simplify authentication while testing. Although these passwords are intended to be removed before deployment, they are sometimes inadvertently left in the code. This introduces serious security risks, especially if the password grants elevated privileges or is reused across multiple systems.

An attacker who discovers a hardcoded password can potentially gain unauthorized access, escalate privileges, exfiltrate sensitive data, or disrupt service availability. If the password is reused across different environments or applications, the compromise can spread quickly and broadly.

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret/test

Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/it/java/org/owasp/webgoat/JWTLessonIntegrationTest.java (line : 217)**

Data Flow

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

```
227:49  .signWith(SignatureAlgorithm.HS256, "deletingTom")
```

SOURCE 0

```
217:9   Jwts.builder()                                                    SignatureAlgorithm.HS256, "d
                        .setHeader(header)
                        .setIssuer("WebGoat Token Builder")
                        .setAudience("webgoat.org")
                        .setIssuedAt(Calendar.getInstance().getTime())
                        .setExpiration(Date.from(Instant.now().plusSeconds(60)))
                        .setSubject("tom@webgoat.org")
                        .claim("username", "Tom")
                        .claim("Email", "tom@webgoat.org")
                        .claim("Role", new String[] {"Manager", "Project Administrator"})
                        .signWith(
```

SINK 1

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret/test

> Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.
>
> 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).signWith(HS512,"wrong_password").compact();`	SOURCE	0
122:20	`String token =Jwts.builder().setClaims(claims).signWith(HS512, "wrong_password").compact();`	SINK	1

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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret/test

> Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 48)**

Data Flow

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

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

48:9	`Jwts.builder()`
	`.setHeaderParam("kid", "webgoat_key")`
	`.setIssuedAt(new Date(System.currentTimeMillis() + TimeUnit.DAYS.toDays(10)))`
	`.setClaims(claims)`
	`.signWith(`
		`io.jsonwebtoken.`	SINK 1

✓ 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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret/test

> Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 55)**

⇣ Data Flow

55:43	`Jwt jwt = Jwts.parser().setSigningKey(`"qwertyqwerty1234"`).parse(token);`		SOURCE 0
55:15	`Jwt jwt =`Jwts.parser().setSigningKey(`"qwertyqwerty1234").parse(token);`		SINK 1

✓ 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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret/test

> Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.
>
> Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (line : 76)**

Data Flow

78:65 `.signWith(io.jsonwebtoken.SignatureAlgorithm.HS512,("bm5n3SkxCX4kKRy4")` SOURCE 0

76:9 `Jwts.builder() .setClaims(claims) .signWith(` `io.jsonwebtoken.SignatureAlgorithm.HS512, "bm5n3SkxCX4kKRy4")` SINK 1

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.

Path Traversal

SNYK-CODE | CWE-23 | PT/test

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 : 96)**

Data Flow

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

229:9 `RestAssured.given() .when() .relaxedHTTPSValidation() .cookie(` `"WEBWOLFSESSION", getWebWolfCookie())` SOURCE 0

229:9 `RestAssured.given() .when() .relaxedHTTPSValidation() .cookie(` `"WEBWOLFSESSION", getWebWolfCookie())` 1

229:9 `RestAssured.given() .when() .relaxedHTTPSValidation() .cookie("WEBWOLFSESSION", getWebWolfCookie()) .get(` `webWolfUrl("/file-server-location"))` 2

229:9 `RestAssured.given() .when() .relaxedHTTPSValidation() .cookie("WEBWOLFSESSION", getWebWolfCookie()) .get(webWolfUrl("/file-server-location")) .then(` `)` 3

229:9 `RestAssured.given() .when() .relaxedHTTPSValidation() .cookie("WEBWOLFSESSION", getWebWolfCookie()) .get(webWolfUrl("/file-server-location")) .then() .extract(` `)` 4

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
```
5

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
```
6

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
                    .asString());
```
7

228:12
```
String result =
                RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
                    .asString();
```
8

239:14 `result = result.replace("%20", " ");` 9

239:14 `result = result.replace("%20", " ");` 10

239:5 `result = result.replace("%20", " ");` 11

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

66:22 `webwolfFileDir = getWebWolfFileServerLocation();` 12

66:5 `webwolfFileDir = getWebWolfFileServerLocation();` 13

95:38 `Path webWolfFilePath = Paths.get(webwolfFileDir);` 14

95:28 `Path webWolfFilePath = Paths.get(webwolfFileDir);` 15

95:10 `Path webWolfFilePath = Paths.get(webwolfFileDir);` 16

96:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists()) {` 17

96:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists()) {` 18

96:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists()) {` 19

96:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists()) {` 20

96:5
```
if (webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists()) {
                Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)));
            }
```
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/%2e%2e/root/.ssh/id_rsa
```

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 | PT/test

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 : 66)**

Data Flow

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

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(
```
”WEBWOLFSESSION”, getWebWolfCookie()) SOURCE 0

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(
```
”WEBWOLFSESSION”, getWebWolfCookie()) 1

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(”WEBWOLFSESSION”, getWebWolfCookie())
                    .get(
```
webWolfUrl(”/file-server-location”)) 2

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(”WEBWOLFSESSION”, getWebWolfCookie())
                    .get(webWolfUrl(”/file-server-location”))
                    .then(
```
) 3

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(”WEBWOLFSESSION”, getWebWolfCookie())
                    .get(webWolfUrl(”/file-server-location”))
                    .then()
                    .extract(
```
) 4

229:9
```
RestAssured.given()
                    .when()
```
) 5

```
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response(
```

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody(
```
)

6

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
                    .asString(
```
);

7

228:12 String
```
result =
                RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
                    .asString();
```

8

239:14 `result = result.replace("%20", " ");`

9

239:14 `result = result.replace("%20", " ");`

10

239:5 `result = result.replace("%20", " ");`

11

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

40:33 `webWolfFileServerLocation = getWebWolfFileServerLocation();`

12

40:5 `webWolfFileServerLocation = getWebWolfFileServerLocation();`

13

65:38 `Path webWolfFilePath = Paths.get(webWolfFileServerLocation);`

14

65:28 `Path webWolfFilePath = Paths.get(webWolfFileServerLocation);`

15

65:10 `Path webWolfFilePath = Paths.get(webWolfFileServerLocation);`

16

66:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists()) {`

17

66:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists()) {`

18

66:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists()) {`

19

66:9 `if (webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists()) {`

20

66:5 SINK

21

```
if (webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists()) {
                Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")));
            }
```

✓ 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_rsa
```

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 | PT/test

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 : 97)**

Data Flow

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

229:9 `RestAssured.given()` `.when()` `.relaxedHTTPSValidation()` `.cookie(` `"WEBWOLFSESSION", getWebWolfCookie())` SOURCE 0

229:9 `RestAssured.given()` `.when()` `.relaxedHTTPSValidation()` `.cookie(` `"WEBWOLFSESSION", getWebWolfCookie())` 1

229:9 `RestAssured.given()` `.when()` `.relaxedHTTPSValidation()` `.cookie("WEBWOLFSESSION", getWebWolfCookie())` `.get(` `webWolfUrl("/file-server-location"))` 2

229:9 `RestAssured.given()` `.when()` `.relaxedHTTPSValidation()` `.cookie("WEBWOLFSESSION", getWebWolfCookie())` `.get(webWolfUrl("/file-server-location"))` `.then(` `)` 3

229:9 `RestAssured.given()` `.when()` `.relaxedHTTPSValidation()` `.cookie("WEBWOLFSESSION", getWebWolfCookie())` `.get(webWolfUrl("/file-server-location"))` `.then()` `.extract(` `)` 4

229:9 `RestAssured.given()` `.when()` `.relaxedHTTPSValidation()` 5

```
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response(
```

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody(
```
)

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
                    .asString(
```
);

228:12 String
```
result =
                RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
                    .asString();
```

239:14 `result = result.replace("%20", " ");`

239:14 `result = result.replace("%20", " ");`

239:5 `result = result.replace("%20", " ");`

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

66:22 `webwolfFileDir = getWebWolfFileServerLocation();`

66:5 `webwolfFileDir = getWebWolfFileServerLocation();`

95:38 `Path webWolfFilePath = Paths.get(webwolfFileDir);`

95:28 `Path webWolfFilePath = Paths.get(webwolfFileDir);`

95:10 `Path webWolfFilePath = Paths.get(webwolfFileDir);`

97:20 `Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)));`

97:20 `Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)));`

97:7 `Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)));` 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_rsa
```

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 | PT/test

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 : 67)**

Data Flow

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

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(
```
”WEBWOLFSESSION”, getWebWolfCookie()) SOURCE 0

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(
```
”WEBWOLFSESSION”, getWebWolfCookie()) 1

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(”WEBWOLFSESSION”, getWebWolfCookie())
                    .get(
```
webWolfUrl(”/file-server-location”)) 2

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(”WEBWOLFSESSION”, getWebWolfCookie())
                    .get(webWolfUrl(”/file-server-location”))
                    .then(
```
) 3

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(”WEBWOLFSESSION”, getWebWolfCookie())
                    .get(webWolfUrl(”/file-server-location”))
                    .then()
                    .extract(
```
) 4

229:9
```
RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie(”WEBWOLFSESSION”, getWebWolfCookie())
                    .get(webWolfUrl(”/file-server-location”))
                    .then()
                    .extract()
                    .response(
```
) 5

```
229:9  RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody(                                                    )   6
229:9  RestAssured.given()
                    .when()
                    .relaxedHTTPSValidation()
                    .cookie("WEBWOLFSESSION", getWebWolfCookie())
                    .get(webWolfUrl("/file-server-location"))
                    .then()
                    .extract()
                    .response()
                    .getBody()
                    .asString(                                                   );  7
228:12 String result =
                      RestAssured.given()
                          .when()
                          .relaxedHTTPSValidation()
                          .cookie("WEBWOLFSESSION", getWebWolfCookie())
                          .get(webWolfUrl("/file-server-location"))
                          .then()
                          .extract()
                          .response()
                          .getBody()
                          .asString();                                           8
239:14 result = result.replace("%20", " ");                                      9
239:14 result = result.replace("%20", " ");                                      10
239:5  result = result.replace("%20", " ");                                      11
```

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

```
40:33  webWolfFileServerLocation = getWebWolfFileServerLocation());              12
40:5   webWolfFileServerLocation = getWebWolfFileServerLocation();               13
65:38  Path webWolfFilePath = Paths.get(webWolfFileServerLocation);              14
65:28  Path webWolfFilePath = Paths.get(webWolfFileServerLocation);              15
65:10  Path webWolfFilePath = Paths.get(webWolfFileServerLocation);              16
67:20  Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")));   17
67:20  Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")));   18
67:7   Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")));   SINK 19
```

✓ 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_rsa
```

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 Without 'HttpOnly' Flag

SNYK-CODE | CWE-1004 | WebCookieMissesCallToSetHttpOnly

> 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, cookieValue);   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 | WebCookieMissesCallToSetHttpOnly

> 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 : 130)**

Data Flow

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

```
130:27  Cookie cookie = new Cookie("access_token", 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 | WebCookieMissesCallToSetHttpOnly

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 : 135)**

Data Flow

```
135:27  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 | WebCookieMissesCallToSetHttpOnly

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 | WebCookieMissesCallToSetHttpOnly

> 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, newCookieValue);        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.

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass/test

> 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 : 68)**

Data Flow

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

```
68:19   Jwt jwt = Jwts.parser().setSigningKey(TextCodec.BASE64.encode(key)).parse( token);        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.

Best practices for prevention

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass/test

> 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("qwertyqwerty1234").parse(token);   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.

Best practices for prevention

- Reading a JWS

JWT Signature Verification Bypass

SNYK-CODE | CWE-347 | JwtVerificationBypass/test

> 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 SigningKeyResolverAdapter() {
                         @Override
                         public byte[] resolveSigningKeyBytes(JwsHeader header, Claims claims) {
                           return TextCodec.BASE64.decode(key);
                         }
                                                                                  token);   SOURCE SINK  0
```

```
                    })
                .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.

Best practices for prevention

- Reading a JWS

Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

SNYK-CODE | CWE-614 | WebCookieMissesCallToSetSecure

> 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 : 130)**

⇣ Data Flow

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

```
130:27  Cookie cookie = new Cookie("access_token", 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 | WebCookieMissesCallToSetSecure

> 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 : 135)**

⇣ Data Flow

```
135:27  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 | WebCookieMissesCallToSetSecure

> 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.

Use of Password Hash With Insufficient Computational Effort

SNYK-CODE | CWE-916 | InsecureHash

> 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.

Use of Hardcoded Credentials

SNYK-CODE | CWE-798 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/it/java/org/owasp/webgoat/ChallengeIntegrationTest.java (line : 32)**

⇣ Data Flow

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

32:28 `params.put("username","admin");` 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 | NoHardcodedCredentials/test

> Do not hardcode credentials in code.
>
> Found in: **src/it/java/org/owasp/webgoat/IntegrationTest.java (line : 29)**

Data Flow

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

29:39 `@Getter private final String user = "webgoat";` 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 | NoHardcodedCredentials

> Do not hardcode credentials in code.
>
> Found in: **src/main/java/org/owasp/webgoat/lessons/challenges/challenge7/PasswordResetLink.java (line : 15)**

Data Flow

src/main/java/org/owasp/webgoat/lessons/challenges/challenge7/PasswordResetLink.java

15:35 `if (username.equalsIgnoreCase("admin")) {` 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.

Hardcoded Secret

SNYK-CODE | CWE-547 | HardcodedSecret/test

> Hardcoded value string is used as a cipher key. Generate the value with a cryptographically strong random number generator such as java.security.SecureRandom instead.

Found in: **src/test/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpointTest.java (line : 37)**

Data Flow

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

33:18 `String key = "deletingTom";` — SOURCE 0

37:9 SINK 1

```
Jwts.builder()                                                                                                io.js
                    .setHeaderParam(
                        "kid", "hacked' UNION select '" + key + "' from INFORMATION_SCHEMA.SYSTEM_USERS --")
                    .setIssuedAt(new Date(System.currentTimeMillis() + TimeUnit.DAYS.toDays(10)))
                    .setClaims(claims)
                    .signWith(
```

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.

Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

SNYK-CODE | CWE-614 | WebCookieSecureDisabledByDefault

Cookie misses the Secure attribute (it is false by default). Set it to true to protect the cookie from man-in-the-middle attacks.

Found in: **src/main/resources/lessons/spoofcookie/js/handler.js (line : 9)**

Data Flow

src/main/resources/lessons/spoofcookie/js/handler.js

9:11 `document.cookie= 'spoof_auth=;Max-Age=0;secure=true';` — 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.

Unsafe JQuery Plugin

SNYK-CODE | CWE-79,CWE-116 | UnsafeJqueryPlugin

> Unsanitized input to JQuery plugin from options flows into $, where it may be evaluated as HTML. If the input is intended to be used as a selector, then a JQuery method that only accepts selectors should be used. If the input is intended to be used as HTML, the developer needs to ensure that it's well documented that the user is responsible for sanitizing the input.
>
> Found in: **src/main/resources/webgoat/static/js/libs/jquery-ui-1.10.4.js (line : 12144)**

Data Flow

src/main/resources/webgoat/static/js/libs/jquery-ui-1.10.4.js

12135:27	`$.fn.position = function(options) {`	SOURCE	0
12135:27	`$.fn.position = function(options) {`		1
12141:26	`options = $.extend( {},options);`		2
12141:14	`options = $.extend(){}, options );`		3
12141:2	`options = $.extend( {}, options );`		4
12144:15	`target = $(options.of ),`		5
12144:23	`target = $( options.of ),`		6
12144:15	`target = $(options.of ),`		7
12144:12	`target =$(options.of ),`	SINK	8

Fix Analysis

Details

Popular frameworks, such as JQuery, may be extended by with third party plugins. A third party plug-in might, for example, add an additional function to the JQuery framework. Such a function might accept parameters from the client (the developer using the plugin).

Clients of these plugins do not know the details of the plugin's implementation, so the inputs and outputs should be documented by the plugin developer.

Of particular importance, the plugin developer should document whether a particular input should be sanitized by the client.

The JQuery framework exposes an API that consists of the following:

- Methods that use some input to select an element within the DOM
- Methods that use some input to mutate the DOM

As a result, a plugin might, for example, take user input and write it to the DOM without sanitizing it internally, expecting the client to have performed the necessary sanitization. This case may lead to cross-site scripting vulnerabilties.

Best practices for prevention

- Document which inputs should be sanitized
  - It's recommended that JQuery plugin developers thoroughly document any inputs that may lead to cross-side scripting attacks.
- Avoid ambiguous JQuery APIs In addition to the onus of documentation, JQuery plugin developers should take particular care to ensure that user input is not handled in unexpected ways.
  - JQuery has a very flexible API, and there are some methods that either perform a selection OR mutation, depending on the type of input given.
  - As a result, a plugin developer may intend their plugin to accept user input to perform a selection, but certain user inputs may instead perform dynamic HTML construction.
  - If user input is intended to only be used to perform selection, a method that only performs selection should be used