

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java (行: 78)

## データフロー

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

```
59:33 パブリックAttackResult完了(@RequestParam String name,@RequestParam String auth_tan) { ソース 0
59:33 パブリックAttackResult完了(@RequestParam String name,@RequestParam String auth_tan) { return 1
60:43 injectableQueryConfidentiality( name, auth_tan); 保護されたAttackResult 2
63:57 injectableQueryConfidentiality( String name, String auth_tan) { 3
67:15 +名前 4
66:9 「SELECT * FROM 従業員 WHERE 姓 = 5
      +名前
66:9 「SELECT * FROM 従業員 WHERE 姓 = 6
      +名前
      +「かつ auth_tan =
66:9 「SELECT * FROM 従業員 WHERE 姓 = 7
      +名前
      +「AND auth_tan = +
      auth_tan
66:9 「SELECT * FROM 従業員 WHERE 姓 = 8
      +名前
      +「AND auth_tan = +
      auth_tan
      +"""; 
65:12 弦 クエリ = 9
      「SELECT * FROM 従業員 WHERE 姓 =
      +名前
      +「AND auth_tan =
      auth_tan
      +"""; 
77:25 log(connection, query); public 10
147:49 static void log(Connection connection, String action) { ResultSet 結果 = 11
78:52 statement.executeQuery( query); ResultSet 結果 = 12
78:29 statement.executeQuery( query); 13
シングル
```

## ✓修正分析

## 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。

● ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。

● ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。

● コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。

● すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。

● パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。

● ? & / < > ; -などの文字 \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。

● SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson9.java (行: 76)

## データフロー

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

```

60:33    パブリックAttackResult完了(@RequestParam String name,@RequestParam String auth_tan){          ソース 0
60:33    パブリックAttackResult完了( @RequestParam String name, @RequestParam String auth_tan) { 戻り値
61:37    injectableQueryIntegrity( name, auth_tan); 保護されたAttackResult
64:51    injectableQueryIntegrity( String name, String auth_tan) {          1
68:15    +名前          2
67:9      「SELECT * FROM 従業員 WHERE 姓 =          3
          +名前          4
67:9      「SELECT * FROM 従業員 WHERE 姓 =          5
          +名前          6
          +「かつ auth_tan =          7
67:9      「SELECT * FROM 従業員 WHERE 姓 =          8
          +名前          9
          +「AND auth_tan = +          10
          auth_tan          11
67:9      「SELECT * FROM 従業員 WHERE 姓 =          12
          +名前          13
          +「AND auth_tan = +          14
          auth_tan          15
          +""";          16
66:12    弦          クエリ =
          「SELECT * FROM 従業員 WHERE 姓 =          17
          +名前          18
          +「AND auth_tan = +          19
          auth_tan          20
          +""";          21
76:52    ResultSet 結果 = statement.executeQuery( クエリ);          22
76:29    ResultSet 結果 = statement.executeQuery( クエリ);          23

```

## 修正分析

### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、

機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

### 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。
- ? & / < > ; -などの文字 \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteUpdateに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java (行: 158)

### データフロー

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

```
59:33 パブリックAttackResult完了(@RequestParam String name,@RequestParam String auth_tan) {  
59:33 パブリックAttackResult完了(@RequestParam String name, @RequestParam String auth_tan) { return  
60:43 injectableQueryConfidentiality( name, auth_tan); 保護されたAttackResult  
63:57 injectableQueryConfidentiality( String name, String auth_tan) {  
67:15 +名前  
66:9 「SELECT * FROM 従業員 WHERE 姓 =  
+名前  
66:9 「SELECT * FROM 従業員 WHERE 姓 =  
+名前  
+かつ auth_tan =  
66:9 「SELECT * FROM 従業員 WHERE 姓 =  
+名前  
+AND auth_tan =+  
auth_tan  
66:9 「SELECT * FROM 従業員 WHERE 姓 =  
+名前  
+AND auth_tan =+  
auth_tan  
+""";  
65:12 弦 クエリ =  
「SELECT * FROM 従業員 WHERE 姓 =  
+名前  
+AND auth_tan =+  
auth_tan  
+""";  
77:25 log(接続、クエリ);  
147:49 パブリック静的voidログ(接続接続、文字列アクション){  
148:14 アクション ≠ action.replace('\"', '\"');  
148:14 アクション ≠ action.replace( '\"', '\"');  
148:5 アクション = action.replace('\"', '\"');  
154:77 「access_log (time, action) に VALUES ('" + time + "', '" + action + "') を挿入します」;  
154:9 「INSERT INTO access_log (time, action) VALUES ('" + time + "', '" + action + "') +'"');  
154:9 「access_log (time, action) に VALUES ('" + time + "', '" + action + "') を挿入します」;  
153:12 文字列 logQuery =  
「access_log (time, action) に VALUES ('" + time + "', '" + action + "') を挿入します」;  
158:31 ステートメント.executeUpdate(logQuery);  
158:7 [ステートメント.executeUpdate( ログクエリ);
```

シンク20

### 修正分析

### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。`? & /<>;`などの文字<sup>”</sup>およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteUpdateに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson9.java (行: 75)

## データフロー

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

```

60:33 パブリックAttackResult完了(@RequestParam String name,@RequestParam String auth_tan) {
60:33 パブリックAttackResult完了( @RequestParam String name, @RequestParam String auth_tan) { 戻り値
61:37 injectableQueryIntegrity( name, auth_tan);保護されないAttackResult
64:51 injectableQueryIntegrity( String name, String auth_tan) { [REDACTED]
68:15 +[名前] [REDACTED]
67:9 「SELECT * FROM 従業員 WHERE 姓 =
+名前 [REDACTED]
67:9 「SELECT * FROM 従業員 WHERE 姓 =
+名前 [REDACTED]
+「かつ auth_tan =
67:9 「SELECT * FROM 従業員 WHERE 姓 =
+名前 [REDACTED]
+「AND auth_tan = +
auth_tan
67:9 「SELECT * FROM 従業員 WHERE 姓 =
+名前 [REDACTED]
+「AND auth_tan = +
auth_tan
+"';;
66:12 弦 クエリ =
「SELECT * FROM 従業員 WHERE 姓 =
+名前 [REDACTED]
+「AND auth_tan = +
auth_tan
+"';;
75:45 SqlInjectionLesson8.log(接続、クエリ); [REDACTED]

```

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

```

147:49 パブリック静的voidログ(接続接続、文字列アクション){ [REDACTED]
148:14 アクション{action.replace("\", "");}
148:14 アクション{action.replace( "\\", "");}
148:5 アクション=action.replace("\", "");
154:77 「access_log (time, action) に VALUES (" + time + ", " + action + ") を挿入します」; [REDACTED]
154:9 "INSERT INTO access_log (time, action) VALUES (" + time + ", " + action + ")" + ")";
154:9 「access_log (time, action) に VALUES (" + time + ", " + action + ") を挿入します」;

```

「access\_log (time, action) に VALUES (" + time + ", " + action + ") を挿入します」;

158:31 ステートメント.executeUpdate(logQuery);  
158:7 ステートメント.executeUpdate(logQuery);

19

シンク20

### ✓修正分析

### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

### 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NET の SqlCommand() や PHP の bindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字  
＼およびスペース。可能であれば、ベンダー 提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## パストラバーサル

SNYK コード CWE-23 PT

HTTP パラメータからのサニタイズされていない入力は org.springframework.util.FileCopyUtils.copyToByteArray に流れ込み、そこでパスとして使用されます。

これにより、パストラバーサルの脆弱性が生じ、攻撃者が任意のファイルを読み取ることができる可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (行: 97)

### データフロー

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

```
90:16 var id = request.getParameter("id");
90:16 var id = request.getParameter("id");
90:11 var d = request.getParameter("id");
92:42 新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");新しいファイル
92:42 (catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");新しいファイル(catPicturesDirectory,(id ==
92:15 null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
91:11 var catPicture =
91:11     新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
97:49
97:19 .body(FileCopyUtils.copyToByteArray(catPicture));.body(FileCopyUtils.copyToByteArray(catPicture));
```

ソース 0

1

2

3

4

5

6

7

シンク 8

### ✓修正分析

### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、その他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLI プログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

st は、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 .(ドット)。

- 任意のファイルの書き込み 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

```
2018-04-15 22:04:29 ..... 19 19 良い.txt  
2018-04-15 22:04:42 ..... 20 20 ../../../../../../root/.ssh/承認キー
```

## パストラバーサル

SNYKコード CWE-23 PT

HTTP パラメータからのサニタイズされていない入力は org.springframework.util.FileCopyUtils.copyToByteArray に流れ込み、そこでパスとして使用されます。

これにより、パス トラバーサルの脆弱性が生じ、攻撃者が任意のファイルを読み取ることができる可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (行: 103)

### データフロー

```
90:16 var id = request.getParameter("id"); 0  
90:16 var id = request.getParameter("id"); 1  
90:11 var d = request.getParameter("id"); 2  
92:42 新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");新しいファイル 3  
92:42 (catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");新しいファイル(catPicturesDirectory,(id == 4  
92:15 null ? RandomUtils.nextInt(1, 11) : id) + ".jpg"); 5  
91:11 var catPicture = 6  
         新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");  
103:76 [ ] 7  
103:46 .body(Base64.getEncoder().encode(FileCopyUtils.copyToByteArray(catPicture)));.body(Base64.getEncoder().encode(FileCopyUtils.copyToByteArray(catPicture))); 8 シンク
```

### 修正分析

#### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、その他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩:攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 .(ドット)。

- 任意のファイルの書き込み 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの

抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、

悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/authorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 .....

19

19 良い.txt

2018-04-15 22:04:42 .....

20

20 ..../..../..../..../root/.ssh/承認キー

## パストラバーサル

SNYKコード CWE-23 PT

HTTPパラメータからのサニタイズされていない入力がexistsに流れ込み、パスとして使用されます。これによりパストラバーサルの脆弱性が発生する可能性があります。

攻撃者が条件式でアプリケーションのロジックをバイパスできるようにします。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (行: 99)

### データフロー

```
90:16 var id = request.getParameter("id");
90:16 var id = request.getParameter("id");
90:11 var d = request.getParameter("id");
92:42 新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");新しいファイル
92:42 (catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
92:15 新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
91:11 var catPicture =
91:11     新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
99:11 catPicture が存在する場合 () {
99:11 猫の写真が存在する場合 ()
99:7 猫の写真が存在する場合 () {
99:7     ResponseEntity.ok() を返す
99:7         .contentType(MediaType.parseMediaType(MediaType.IMAGE_JPEG_VALUE))
99:7         .location(新しいURI("/PathTraversal/random-picture?id=" + catPicture.getName()))
99:7         .body(Base64.getEncoder().encode(FileCopyUtils.copyToByteArray(catPicture)));
99:7 }
```

ソース 0

1

2

3

4

5

6

7

8

9

シンク

### 修正分析

### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 .(ドット)。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの

抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 .....	19	19 良い.txt
2018-04-15 22:04:42 .....	20	20 ../../../../../../root/.ssh/承認キー

## パストラバーサル

SNYKコード CWE-23 PT

HTTPパラメータからのサニタイズされていない入力がlistFilesに流れ込み、パスとして使用されます。これにより、パストラバーサルの脆弱性が発生する可能性があります。

攻撃者が任意のファイルを操作できるようになります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java (行: 108)

### データフロー

```

90:16 var id = request.getParameter("id");
90:16 var id = request.getParameter("id");
90:11 var d = request.getParameter("id");
92:42 新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
92:42 新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
92:15 新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
91:11 var catPicture =
    新しいファイル(catPicturesDirectory,(id == null ? RandomUtils.nextInt(1, 11) : id) + ".jpg");
108:55 StringUtils.arrayToCommaDelimitedString(catPicture.getParentFile().listFiles())
108:55 StringUtils.arrayToCommaDelimitedString(catPicture.getParentFile().listFiles())
108:55 StringUtils.arrayToCommaDelimitedString(catPicture.getParentFile().listFiles())

```

ソース 0  
1  
2  
3  
4  
5  
6  
7  
8  
シンク 9

### ✓修正分析

#### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カーレ http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 . (ドット)。

- 任意のファイルの書き込み 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることもあります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 .....	19	19 良い.txt
2018-04-15 22:04:42 .....	20	20 ../../../../../../root/.ssh/承認キー

## パストラバーサル

SNYKコード CWE-23 PT

HTTPパラメータからのサニタイズされていない入力はorg.springframework.util.FileCopyUtils.copyに流れ込み、パスとして使用されます。これはパス トラバーサルの脆弱性が生じ、攻撃者が任意のファイルに書き込むことができるようになります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java (行: 67)

## データフロー

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

```

51:41 パブリックAttackResult uploadFileHandler( @RequestParam("uploadedFileZipSlip") マルチパートファイルファイル) { ソース 0
51:41 パブリックAttackResult uploadFileHandler( @RequestParam("uploadedFileZipSlip") マルチパートファイルファイル) { 1
55:31 processZipUpload(ファイル) を返します。 2
60:41 プライベートAttackResult processZipUpload(MultipartFileファイル) { 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 [アップロードされた ZipFile = 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 [file.getBytes(),uploadedZipFile.toFile()]; FileCopyUtils.copy( シンク 10

```

## ✓修正分析

## 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、その他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります。

カーネル [http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\\_rsa](http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_rsa)

注: %2eは。(ドット)の URL エンコード バージョンです。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に /root/.ssh/authorized\_keys ファイルが上書きされます。

2018-04-15 22:04:29 .....

19

19 良い.txt

2018-04-15 22:04:42 .....

20

20 ../../../../../../root/.ssh/承認キー

HTTPパラメータからのサニタイズされていない入力がjava.util.zip.ZipFileに流れ込み、パスとして使用されます。これによりパストラバーサルが発生する可能性があります。

脆弱性を悪用し、攻撃者が任意のファイルに書き込むことが可能になります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java (行: 69)

## データフロー

51:41	パブリックAttackResult uploadFileHandler( @RequestParam("uploadedFileZipSlip") マルチパートファイルファイル) {	ソース	0
51:41	パブリックAttackResult uploadFileHandler( @RequestParam("uploadedFileZipSlip") MultipartFile file) { return		1
55:31	processZipUpload( file); プライベート		2
60:41	AttackResult processZipUpload( MultipartFile file){		3
66:53	var uploadedZipFile = tmpZipDirectory.resolve( file.getOriginalFilename());		4
66:53	var uploadedZipFile = tmpZipDirectory.resolve( file.getOriginalFilename()); var uploadedZipFile =		5
66:29	tmpZipDirectory.resolve( file.getOriginalFilename());		6
66:11	var アップロードされた ZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());		7
69:33	ZipFile zip = 新しい ZipFile( uploadedZipFile.toFile()); ZipFile zip = 新しい		8
69:33	ZipFile( uploadedZipFile.toFile()); ZipFile zip = 新しい		9
69:25	ZipFile( uploadedZipFile.toFile());	シンク	10

## 修正分析

### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 . (ドット)。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることもあります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/authorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 .....

19

19 良い.txt

2018-04-15 22:04:42 .....

20

20 ../../../../../../root/.ssh/承認キー

HTTPパラメータからのサニタイズされていない入力がjava.nio.file.Files.copyに流れ込み、パスとして使用されます。これによりバストラバーサルが発生する可能性があります。

脆弱性を悪用し、攻撃者が任意のファイルに書き込むことが可能になります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileZipSlip.java (行: 75)

## データフロー

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

```

51:41 パブリックAttackResult uploadFileHandler( @RequestParam("uploadedFileZipSlip") マルチパートファイルファイル) {
51:41 パブリックAttackResult uploadFileHandler( @RequestParam("uploadedFileZipSlip") MultipartFile file) { return
55:31 processZipUpload( file); プライベート
60:41 AttackResult processZipUpload( MultipartFile file){ }
66:53 var uploadedZipFile = tmpZipDirectory.resolve( file.getOriginalFilename());
66:53 var uploadedZipFile = tmpZipDirectory.resolve( file.getOriginalFilename()); var uploadedZipFile =
66:29 tmpZipDirectory.resolve( file.getOriginalFilename());
66:11 var アップロードされた ZipFile = tmpZipDirectory.resolve(file.getOriginalFilename());
69:33 ZipFile zip = 新しい ZipFile( uploadedZipFile.toFile()); ZipFile zip = 新しい
69:33 ZipFile( uploadedZipFile.toFile()); ZipFile zip = 新しい
69:25 ZipFile( uploadedZipFile.toFile());
69:15 ZipFile zip = 新しい ZipFile(uploadedZipFile.toFile());
70:49 列挙<? extends ZipEntry> entries = zip.entries(); 列挙<? extends ZipEntry>
70:49 entries = zip.entries();
70:39 列挙<? extends ZipEntry> entries = zip.entries();
72:22 ZipEntry e = entries.nextElement();
72:22 ZipEntry e = entries.nextElement();
72:18 ZipEntry e = entries.nextElement();
73:53 ファイル f = new File(tmpZipDirectory.toFile(), e.getName());
73:53 ファイル f = new File(tmpZipDirectory.toFile(), e.getName()); ファイル f = new
73:22 File(tmpZipDirectory.toFile(), e.getName());
73:14 ファイル f = new File(tmpZipDirectory.toFile(), e.getName());
75:24 Files.copy(is, f.toPath(), StandardCopyOption.REPLACE_EXISTING);
75:24 Files.copy(is, f.toPath(), StandardCopyOption.REPLACE_EXISTING);
75:9 ファイル.コピー( つまり,f.toPath(),StandardCopyOption.REPLACE_EXISTING);

```

ソース

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

シンク24

## 修正分析

### 詳細

ディレクトリトラバーサル攻撃（バストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、その他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

バストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもバストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩:攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは.(ドット)の URL エンコード バージョンです。

- 任意のファイルの書き込み:攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、バストラバーサルファイル名を含む恶意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、恶意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることになります。

2018-04-15 22:04:29 .....	19	19 良い.txt
2018-04-15 22:04:42 .....	20	20 ../../../../../../root/.ssh/承認キー

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/challenges/SolutionConstants.java (行: 36)

データフロー

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

36:32 文字列 ADMIN\_PASSWORD\_LINK = "375afe1104f4a487a73823c50a9292a2";

ソースシンク

0

修正分析

詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシーコード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。

将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (行: 44)

データフロー

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

44:53 パブリック静的最終String[] SECRETS = {"secret", "admin", "passw0rd", "123456", "passw0rd"};

ソースシンク

0

修正分析

詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

- Machine Translated by Google
- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
  - ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
  - 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (行: 44)

### データフロー

44:74 パブリック静的最終String[] SECRETS = {"secret", "admin", "password", "123456", "passw0rd"};



ソースシンク

0

### ✓修正分析

#### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (行: 44)

### データフロー

44:84 パブリック静的最終String[] SECRETS = {"secret", "admin", "password", "123456", "passw0rd"};



ソースシンク

0

### ✓修正分析

#### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/csrf/ForgedReviews.java (行: 56)

データフロー

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

56:46 プライベート静的最終文字列 weakAntiCSRF = "2aa1427b9a13d0bede0388a7fba9aa9";

ソースシンク

0

✓修正分析

詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシーコードコンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (行: 62)

データフロー

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

62:46 プライベート静的最終文字列 JWT\_PASSWORD = "bm5h3SkxCX4kKRy4";

ソースシンク

0

✓修正分析

詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシーコードコンポーネントを調べて慎重にテストしてください。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 50)

データフロー

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

50:5 ["勝利"]、「ビジネス」、「在庫あり」、「配送」、「ワシントン」

ソースシンク

0

✓修正分析

詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシーコードコンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 50)

データフロー

50:16 「勝利」、「ビジネス」、「入手可能」、「配送」、「ワシントン」

ソースシンク

0

✓修正分析

詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシーコードコンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 50)

### データフロー

50:28 「勝利」、「ビジネス」、「入手可能」、「配送」、「ワシントン」

ソースシンク

0

### 修正分析

### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 50)

### データフロー

50:41 「勝利」、「ビジネス」、「入手可能」、「配送」、「ワシントン」

ソースシンク

0

### 修正分析

### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 50)

### データフロー

50:53 「勝利」、「ビジネス」、「入手可能」、「配送」、「ワシントン」   ソースシンク 0

### 修正分析

#### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

#### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

### ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされた秘密

ハードコードされた文字列値が暗号鍵として使用されています。代わりに、java.security.SecureRandomなどの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JTRefreshEndpoint.java (行: 86)

### データフロー

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

62:46 プライベート静的最終文字列 JWT\_PASSWORD = "bm5h3SkxCX4kKRy4"; ソース 0

86:9 Jwts.ビルダー() .setIssuedAt(新しい日付(System.currentTimeMillis() + TimeUnit.DAYS.toDays(10))) .setClaims(クレーム) .signWith( io.jsonwebtoken. シンク 1 )

### 修正分析

#### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関連する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

#### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/main/java/org/owasp/webgoat/lessons/jwt/JTRefreshEndpoint.java` (行: 107)

#### データフロー

62:46 プライベート静的最終文字列 `JWT_PASSWORD = "bm5n3Skx0X4kKRy4";`

ソース

0

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

シンク

1

#### 修正分析

#### 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

#### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。  
将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

#### ハードコードされた秘密

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/main/java/org/owasp/webgoat/lessons/jwt/JTRefreshEndpoint.java` (行: 137)

#### データフロー

62:46 プライベート静的最終文字列 `JWT_PASSWORD = "bm5n3Skx0X4kKRy4";`

ソース

0

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

シンク

1

#### 修正分析

#### 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

#### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。  
将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java` (行: 126)

## データフロー

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

```
71:69  パブリック静的最終文字列 JWT_PASSWORD = TextCodec.BASE64.encode( "victory"); [ ] ソース 0
126:11  Jwts.ビルダー()
          .setClaims(クレーム)
          .signWith() [ ] シンク 1
```

## 修正分析

### 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。  
将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## ハードコードされた秘密

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java` (行: 155)

## データフロー

```
71:69  パブリック静的最終文字列 JWT_PASSWORD = TextCodec.BASE64.encode( "victory"); [ ] ソース 0
155:19  Jwt jwt = Jwts.parser().setSigningKey( JWT_PASSWORD).parse(accessToken); [ ] シンク 1
```

## 修正分析

### 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされた秘密

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java` (行: 180)

### データフロー

71:69 パブリック静的最終文字列 `JWT_PASSWORD = TextCodec.BASE64.encode( "victory" );` [ ] ソース 0

180:19 `Jwt jwt = Jwts.parser().setSigningKey( JWT_PASSWORD ).parse(accessToken);` シンク 1

### 修正分析

### 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。

将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされた秘密

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java` (行: 203)

### データフロー

71:69 パブリック静的最終文字列 `JWT_PASSWORD = TextCodec.BASE64.encode( "victory" );` [ ] ソース 0

203:19 `Jwt jwt = Jwts.parser().setSigningKey( JWT_PASSWORD ).parse(accessToken);` シンク 1

### 修正分析

### 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされた秘密

ハードコードされた値配列 [...] が暗号鍵として使用されています。代わりに、java.security.SecureRandomなどの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 61)

### データフロー

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

```
49:42 パブリック静的最終String[] SECRETS = {  
    「勝利」、「ビジネス」、「利用可能」、「配送」、「ワシントン」 };  
}  
61:12 Jwts.builder() を返す  
    .setIssuer("WebGoat トークンビルダ  
    署名アルゴリズム.HS256、  
    —") .setAudience("webgoat.org") .setIssuedAt(Calendar.getInstance().getTime()) .setExpiration(Date.from(Instant.now().plusSeconds(60))) .setS  
ソース 0  
シンク 1
```

### 修正分析

#### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

#### 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされた秘密

ハードコードされた値配列 [...] が暗号鍵として使用されています。代わりに、java.security.SecureRandomなどの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 78)

### データフロー

```
49:42 パブリック静的最終String[] SECRETS = {  
    「勝利」、「ビジネス」、「利用可能」、「配送」、「ワシントン」 };  
}  
ソース 0
```



## 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることも一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
  - ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
  - 「将来を見据えたコード」の考え方を採用する：定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。
- 将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

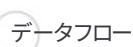
## 不十分なランダム値の使用 - 秘密

SNYK-CODE CWE-330 安全でない秘密

nextIntから安全でないランダムデータが流れ出し、秘密データとして使用されます。暗号的に強力な乱数で値を生成します。

代わりに java.security.SecureRandom などのジェネレーターを使用してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (行: 53)



53:14	文字列 secret = SECRETS[new Random().nextInt(SECRETS.length)];	ソース	0
53:23	文字列 secret = SECRETS[new Random().nextInt(SECRETS.length)];		1
53:31	文字列 secret = SECRETS[ new Random().nextInt( SECRETS.length)]; 文字列 secret =		2
53:31	SECRETS[ new Random().nextInt( SECRETS.length)];		3
53:14	文字列 secret = SECRETS[new Random().nextInt(SECRETS.length)];	シンク	4



## 詳細

コンピュータセキュリティは、安全で機密性の高いセッションキーの生成、パスワードデータのハッシュ化、送信時の暗号化など、多くの点で乱数に依存しています。

機密データなど、他にも多くのデータがあります。理由は簡単に理解できます。例えば、セッションキーが連続的に生成されると、攻撃者は簡単に推測し、

正当なユーザーセッションを乗っ取る可能性があります。同様に、暗号化技術に推測しやすい数字が使用されていた場合、攻撃者はブルートフォース攻撃を用いて不正アクセスを行う可能性があります。

実際には、コンピュータは真の乱数を生成できないため、代わりに「疑似乱数」を使用します。これは、

非常に多様な値をランダムな順序で生成する様々な方法があり、理論上は攻撃者が推測するのは非常に困難です。しかし、開発者が

弱いランダムアルゴリズムを不注意に利用した場合、攻撃者はアルゴリズム、シード、またはパターンを発見し、最終的にコマンドへのアクセスをロック解除したり、

機密データが盗まれ、身代金目的で保持されたり、売却されたりする可能性があります。

## 予防のためのベストプラクティス

- 統計的PRNGなどの弱い疑似乱数生成器（PRNG）の使用は避けてください。代わりに、暗号的に安全なPRNGを選択してください。
- ユーザーIDやサーバーの起動時間など、予測可能なシード値の使用は避けてください。代わりに、外部から取得したシード値など、擬似乱数であるシード値を使用してください。
- ハードウェア ソース。
- 独自のアプローチを採用して、固有の弱点や脆弱性を含む可能性のあるカスタムコードを作成するのではなく、標準的に承認されたセキュリティアルゴリズムとライブラリを使用してください。
- 重大な欠陥を見過ごす。
- 静的解析ツールを使用して、コード内のこの弱点の潜在的なインスタンスを特定し、適切なホワイト ボックス テストで適切なテスト カバレッジを確保します。
- セキュリティシステム開発におけるエントロピーの重要性について開発者に教育し、FIPS 140-2 に準拠したツールの採用を検討します。

## 不十分なランダム値の使用 - 秘密

SNYK-CODE CWE-330 安全でない秘密

nextIntから安全でないランダムデータが流れ出し、秘密データとして使用されます。暗号的に強力な乱数で値を生成します。

代わりに java.security.SecureRandomなどのジェネレーターを使用してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (行: 71)

### データフロー

71:14 文字列 secret = SECRETS[new Random().nextInt(SECRETS.length)];	ソース 0
71:23 文字列 secret = SECRETS[new Random().nextInt(SECRETS.length)];	1
71:31 文字列 secret = SECRETS[ new Random().nextInt( SECRETS.length)]; 文字列	2
71:31 secret = SECRETS[ new Random().nextInt( SECRETS.length)];	3
71:14 文字列 secret = SECRETS[new Random().nextInt(SECRETS.length)];	シンク 4

### 修正分析

#### 詳細

コンピュータセキュリティは、安全で機密性の高いセッションキーの生成、パスワードデータのハッシュ化、送信時の暗号化など、多くの点で乱数に依存しています。

機密データなど、他にも多くのデータがあります。理由は簡単に理解できます。例えば、セッションキーが連続的に生成されると、攻撃者は簡単に推測し、

正当なユーザーセッションを乗っ取る可能性があります。同様に、暗号化技術に推測しやすい数字が使用されていた場合、攻撃者はブルートフォース攻撃を用いて不正アクセスを行う可能性があります。

実際には、コンピュータは真の乱数を生成できないため、代わりに「疑似乱数」を使用します。これは、

非常に多様な値をランダムな順序で生成する様々な方法があり、理論上は攻撃者が推測するのは非常に困難です。しかし、開発者が

弱いランダムアルゴリズムを不注意に利用した場合、攻撃者はアルゴリズム、シード、またはパターンを見出し、最終的にコマンドへのアクセスをロック解除したり、

機密データが盗まれ、身代金目的で保持されたり、売却されたりする可能性があります。

### 予防のためのベストプラクティス

- 統計的PRNGなどの弱い疑似乱数生成器（PRNG）の使用は避けてください。代わりに、暗号的に安全なPRNGを選択してください。
- ユーザーIDやサーバーの起動時間など、予測可能なシード値の使用は避けてください。代わりに、外部から取得したシード値など、擬似乱数であるシード値を使用してください。ハードウェア ソース。
- 独自のアプローチを採用して、固有の弱点や脆弱性を含む可能性のあるカスタムコードを作成するのではなく、標準的に承認されたセキュリティアルゴリズムとライブラリを使用してください。重大な欠陥を見過ごす。
- 静的解析ツールを使用して、コード内のこの弱点の潜在的なインスタンスを特定し、適切なホワイト ボックス テストで適切なテスト カバレッジを確保します。
- セキュリティ システム開発におけるエントロピーの重要性について開発者に教育し、FIPS 140-2 に準拠したツールの採用を検討します。

## 不十分なランダム値の使用 - 秘密

SNYK-CODE CWE-330 安全でない秘密

nextIntから安全でないランダムデータが流れ出し、秘密データとして使用されます。暗号的に強力な乱数で値を生成します。

代わりに java.security.SecureRandomなどのジェネレーターを使用してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 52)

### データフロー

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

52:30/パブリック静的最終文字列 JWT_SECRET =	ソース 0
TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);	
53:7    TextCodec.BASE64.encode( SECRET[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   パブリック静的最終文字列 JWT_SECRET =	シンク 5
TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);	

### 修正分析

#### 詳細

機密データなど、他にも多くのデータがあります。理由は簡単に理解できます。例えば、セッションキーが連続的に生成されると、攻撃者は簡単に推測し、

正当なユーザーセッションを乗っ取る可能性があります。同様に、暗号化技術に推測しやすい数字が使用されていた場合、攻撃者はブルートフォース攻撃を用いて不正アクセスを行う可能性があります。

実際には、コンピュータは真の乱数を生成できないため、代わりに「疑似乱数」を使用します。これは、

非常に多様な値をランダムな順序で生成する様々な方法があり、理論上は攻撃者が推測するのは非常に困難です。しかし、開発者が

弱いランダムアルゴリズムを不注意に利用した場合、攻撃者はアルゴリズム、シード、またはパターンを発見し、最終的にコマンドへのアクセスをロック解除したり、

機密データが盗まれ、身代金目的で保持されたり、売却されたりする可能性があります。

## 予防のためのベストプラクティス

- 統計的PRNGなどの弱い疑似乱数生成器（PRNG）の使用は避けてください。代わりに、暗号的に安全なPRNGを選択してください。
- ユーザーIDやサーバーの起動時間など、予測可能なシード値の使用は避けてください。代わりに、外部から取得したシード値など、擬似乱数であるシード値を使用してください。
- ハードウェアソース。
- 独自のアプローチを採用して、固有の弱点や脆弱性を含む可能性のあるカスタムコードを作成するのではなく、標準的に承認されたセキュリティアルゴリズムとライブラリを使用してください。
- 重大な欠陥を見過ごす。
- 静的解析ツールを使用して、コード内のこの弱点の潜在的なインスタンスを特定し、適切なホワイトボックステストで適切なテストカバレッジを確保します。
- セキュリティシステム開発におけるエントロピーの重要性について開発者に教育し、FIPS 140-2に準拠したツールの採用を検討します。

## 不十分なランダム値の使用 - 秘密

SNYK-CODE CWE-330 安全でない秘密

nextIntから安全でないランダムデータが流れ出し、秘密データとして使用されます。暗号的に強力な乱数で値を生成します。

代わりに java.security.SecureRandomなどのジェネレーターを使用してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 78)

### データフロー

78:11	Jwt jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(トークン);	ソース	0
78:17	Jwt jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(トークン);		1
78:17	Jwt jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(トークン);		2
78:45	Jwt jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(トークン);		3
52:30	パブリック静的最終文字列 JWT_SECRET =		4
	TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);		
53:7	TextCodec.BASE64.encode(SECRETS[新しいRandom().nextInt(SECRETS.length)]);		5
53:31	TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);		6
53:39	TextCodec.BASE64.encode(SECRETS[新しいRandom().nextInt(SECRETS.length)]);		7
53:39	TextCodec.BASE64.encode(SECRETS[新しいRandom().nextInt(SECRETS.length)]);		8
78:11	Jwt jwt = Jwts.parser().setSigningKey(JWT_SECRET).parseClaimsJws(トークン);	シンク	9

### 修正分析

#### 詳細

コンピュータセキュリティは、安全で機密性の高いセッションキーの生成、パスワードデータのハッシュ化、送信時の暗号化など、多くの点で乱数に依存しています。

機密データなど、他にも多くのデータがあります。理由は簡単に理解できます。例えば、セッションキーが連続的に生成されると、攻撃者は簡単に推測し、

正当なユーザーセッションを乗っ取る可能性があります。同様に、暗号化技術に推測しやすい数字が使用されていた場合、攻撃者はブルートフォース攻撃を用いて不正アクセスを行う可能性があります。

実際には、コンピュータは真の乱数を生成できないため、代わりに「疑似乱数」を使用します。これは、

非常に多様な値をランダムな順序で生成する様々な方法があり、理論上は攻撃者が推測するのは非常に困難です。しかし、開発者が

弱いランダムアルゴリズムを不注意に利用した場合、攻撃者はアルゴリズム、シード、またはパターンを発見し、最終的にコマンドへのアクセスをロック解除したり、

機密データが盗まれ、身代金目的で保持されたり、売却されたりする可能性があります。

## 予防のためのベストプラクティス

- 統計的PRNGなどの弱い疑似乱数生成器（PRNG）の使用は避けてください。代わりに、暗号的に安全なPRNGを選択してください。
- ユーザーIDやサーバーの起動時間など、予測可能なシード値の使用は避けてください。代わりに、外部から取得したシード値など、擬似乱数であるシード値を使用してください。
- ハードウェアソース。
- 独自のアプローチを採用して、固有の弱点や脆弱性を含む可能性のあるカスタムコードを作成するのではなく、標準的に承認されたセキュリティアルゴリズムとライブラリを使用してください。
- 重大な欠陥を見過ごす。
- 静的解析ツールを使用して、コード内のこの弱点の潜在的なインスタンスを特定し、適切なホワイトボックステストで適切なテストカバレッジを確保します。
- セキュリティシステム開発におけるエントロピーの重要性について開発者に教育し、FIPS 140-2に準拠したツールの採用を検討します。

## 不十分なランダム値の使用 - 秘密

SNYK-CODE CWE-330 安全でない秘密

nextIntから安全でないランダムデータが流れ出し、暗号鍵として使用されます。暗号的に強力な乱数で値を生成します。

代わりに java.security.SecureRandom などのジェネレーターを使用してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 61)

## データフロー

```

53:7  [TextCodec.BASE64.encode( )]SECRETS[新しいRandom().nextInt(SECRETS.length)];          ソース 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 Jwts.builder() を返す
        .setIssuer("WebGoat トークン ビルダー")
        .setAudience("webgoat.org")
        .setIssuedAt(Calendar.getInstance().getTime())
        .setExpiration(Date.from(Instant.now().plusSeconds(60)))
        .setSubject("tom@webgoat.org")
        .claim("ユーザー名", "トム")
        .claim("メールアドレス", "tom@webgoat.org")
        .claim("役割", 新しい文字列[] {"マネージャー", "プロジェクト管理者"})
        .signWith()                                         署名アルゴリズム.HS256
                                                    シンク 4

```

## 修正分析

## 詳細

コンピュータセキュリティは、安全で機密性の高いセッションキーの生成、パスワードデータのハッシュ化、送信時の暗号化など、多くの点で乱数に依存しています。

機密データなど、他にも多くのデータがあります。理由は簡単に理解できます。例えば、セッションキーが連続的に生成されると、攻撃者は簡単に推測し、

正当なユーザーセッションを乗っ取る可能性があります。同様に、暗号化技術に推測しやすい数字が使用されていた場合、攻撃者はブルートフォース攻撃を用いて不正アクセスを行う可能性があります。

実際には、コンピュータは真の乱数を生成できないため、代わりに「疑似乱数」を使用します。これは、

非常に多様な値をランダムな順序で生成する様々な方法があり、理論上は攻撃者が推測するのは非常に困難です。しかし、開発者が

弱いランダムアルゴリズムを不注意に利用した場合、攻撃者はアルゴリズム、シード、またはパターンを見出し、最終的にコマンドへのアクセスをロック解除したり、

機密データが盗まれ、身代金目的で保持されたり、売却されたりする可能性があります。

## 予防のためのベストプラクティス

- 統計的PRNGなどの弱い疑似乱数生成器（PRNG）の使用は避けてください。代わりに、暗号的に安全なPRNGを選択してください。
- ユーザーIDやサーバーの起動時間など、予測可能なシード値の使用は避けてください。代わりに、外部から取得したシード値など、擬似乱数であるシード値を使用してください。
- ハードウェアソース。
- 独自のアプローチを採用して、固有の弱点や脆弱性を含む可能性のあるカスタムコードを作成するのではなく、標準的に承認されたセキュリティアルゴリズムとライブラリを使用してください。
- 重大な欠陥を見過ごす。
- 静的解析ツールを使用して、コード内のこの弱点の潜在的なインスタンスを特定し、適切なホワイトボックス テストで適切なテストカバレッジを確保します。
- セキュリティシステム開発におけるエントロピーの重要性について開発者に教育し、FIPS 140-2に準拠したツールの採用を検討します。

## 不十分なランダム値の使用 - 秘密

SNYK-CODE CWE-330 安全でない秘密

nextIntから安全でないランダムデータが流れ出し、暗号鍵として使用されます。暗号的に強力な乱数で値を生成します。

代わりに java.security.SecureRandom などのジェネレーターを使用してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 78)

## データフロー

```

53:7  [TextCodec.BASE64.encode( )]SECRETS[新しいRandom().nextInt(SECRETS.length)];          ソース 0
53:31 TextCodec.BASE64.encode(SECRETS[new Random().nextInt(SECRETS.length)]);           1
53:39 TextCodec.BASE64.encode(SECRETS[新しいRandom().nextInt(SECRETS.length)]);           2

```



詳細

コンピュータセキュリティは、安全で機密性の高いセッションキーの生成、パスワードデータのハッシュ化、送信時の暗号化など、多くの点で乱数に依存しています。機密データなど、他にも多くのデータがあります。理由は簡単に理解できます。例えば、セッションキーが連続的に生成されると、攻撃者は簡単に推測し、正当なユーザーセッションを乗っ取る可能性があります。同様に、暗号化技術に推測しやすい数字が使用されていた場合、攻撃者はブルートフォース攻撃を用いて不正アクセスを行う可能性があります。

実際には、コンピュータは真の乱数を生成できないため、代わりに「疑似乱数」を使用します。これは、非常に多様な値をランダムな順序で生成する様々な方法があり、理論上は攻撃者が推測するのは非常に困難です。しかし、開発者が弱いランダムアルゴリズムを不注意に利用した場合、攻撃者はアルゴリズム、シード、またはパターンを見出し、最終的にコマンドへのアクセスをロック解除したり、機密データが盗まれ、身代金目的で保持されたり、売却されたりする可能性があります。

## 予防のためのベストプラクティス

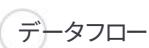
- 統計的PRNGなどの弱い疑似乱数生成器（PRNG）の使用は避けてください。代わりに、暗号的に安全なPRNGを選択してください。
- ユーザーIDやサーバーの起動時間など、予測可能なシード値の使用は避けてください。代わりに、外部から取得したシード値など、擬似乱数であるシード値を使用してください。
- ハードウェア ソース。
- 独自のアプローチを採用して、固有の弱点や脆弱性を含む可能性のあるカスタムコードを作成するのではなく、標準的に承認されたセキュリティアルゴリズムとライブラリを使用してください。
- 重大な欠陥を見過ごす。
- 静的解析ツールを使用して、コード内のこの弱点の潜在的なインスタンスを特定し、適切なホワイト ボックス テストで適切なテスト カバレッジを確保します。
- セキュリティ システム開発におけるエントロピーの重要性について開発者に教育し、FIPS 140-2 に準拠したツールの採用を検討します。

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (行: 107)



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

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

ソースシンク

0



詳細

io.jsonwebtoken.jwtライブラリのJSON Web Token (JWT) 解析メソッドの中には、署名鍵が設定されているにもかかわらず、署名が空のJWTを受け入れるものがあります。パーサー。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

## 予防のためのベストプラクティス

- parseClaimsJwsまたはparsePlaintextJwsメソッドを使用するか、JwtHandlerAdapterのonPlaintextJwsまたはonClaimsJws メソッド。

## 予防のためのベストプラクティス

- [JWSを読む](#)

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (行: 137)

137:11 Jwts.parser().setSigningKey(JWT\_PASSWORD).parse( accessToken ) token.replace("ペアラー ", ""));

ソースシンク

0



詳細

io.jsonwebtoken.jwt ライブライリの JSON Web Token (JWT) 解析メソッドの中には、署名鍵が設定されているにもかかわらず、署名が空の JWT を受け入れるものがあります。パーサー。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

#### 予防のためのベストプラクティス

- parseClaimsJws または parsePlaintextJws メソッドを使用するか、JwtHandlerAdapter の onPlaintextJws または onClaimsJws メソッド。

#### 予防のためのベストプラクティス

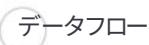
- [JWSを読む](#)

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (行: 155)



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

155:19 Jwt jwt = Jwts.parser().setSigningKey(JWT\_PASSWORD).parse( accessToken );

ソースシンク

0



詳細

io.jsonwebtoken.jwt ライブライリの JSON Web Token (JWT) 解析メソッドの中には、署名鍵が設定されているにもかかわらず、署名が空の JWT を受け入れるものがあります。パーサー。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

#### 予防のためのベストプラクティス

- parseClaimsJws または parsePlaintextJws メソッドを使用するか、JwtHandlerAdapter の onPlaintextJws または onClaimsJws メソッド。

#### 予防のためのベストプラクティス

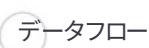
- [JWSを読む](#)

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (行: 180)



180:19 Jwt jwt = Jwts.parser().setSigningKey(JWT\_PASSWORD).parse( accessToken );

ソースシンク

0

## 詳細

io.jsonwebtoken.jwt ライブラリの JSON Web Token (JWT) 解析メソッドの中には、署名鍵が設定されているにもかかわらず、署名が空の JWT を受け入れるものがあります。パーサー。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

## 予防のためのベストプラクティス

- parseClaimsJws または parsePlaintextJws メソッドを使用するか、JwtHandlerAdapter の onPlaintextJws または onClaimsJws メソッド。

## 予防のためのベストプラクティス

- [JWSを読む](#)

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (行: 203)

## データフロー

203:19 Jwt jwt = Jwts.parser().setSigningKey(JWT\_PASSWORD).parse(accessToken);

ソースシンク

0

## 修正分析

## 詳細

io.jsonwebtoken.jwt ライブラリの JSON Web Token (JWT) 解析メソッドの中には、署名鍵が設定されているにもかかわらず、署名が空の JWT を受け入れるものがあります。パーサー。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

## 予防のためのベストプラクティス

- parseClaimsJws または parsePlaintextJws メソッドを使用するか、JwtHandlerAdapter の onPlaintextJws または onClaimsJws メソッド。

## 予防のためのベストプラクティス

- [JWSを読む](#)

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTP パラメータからのサニタイズされていない入力が prepareStatement に流れ込み、SQL クエリで使用されます。これにより、SQL エラーが発生する可能性があります。インジェクションの脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/challenges/challenge5/Assignment5.java (行: 60)

## データフロー

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

51:44 @RequestParam String username\_login,@RequestParam String password\_login) は例外をスローします {

ソース

0

51:44 @RequestParam String username\_login,@RequestParam String password\_login) は例外をスローします {

ソース

1

64:21 + [パスワードログイン]

ソース

2

61:15 「ユーザーID = + username\_login の challenge\_users からパスワードを選択」

ソース

3

+ " パスワード = +  
password\_login

「ユーザーID = + username\_login の challenge\_users からパスワードを選択」  
+ " " パスワード = +  
password\_login  
+ " ")

60:11 接続.prepareStatement(

シンク

5

### ✓修正分析

詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
  - ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
  - コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
  - すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
  - パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字 \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
  - SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
  - 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sql

HTTPパラメータからのサニタイズされていない入力がprepareStatementに流れ込み、SQLクエリで使用されます。これにより、SQLエラーが発生する可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5b.java (行: 65)

データフロー

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

```
56:7 [ @RequestParam 文字列 ユーザーID、 ]@RequestParam 文字列 login_count、HttpServletRequest リクエスト) 0  
ソース  
56:7 [ @RequestParam 文字列 ユーザーID、 ]@RequestParam 文字列 login_count、HttpServletRequest リクエスト) 1  
58:41 injectableQuery(login_count, userid) を返します。保護された 2  
61:62 AttackResult injectableQuery(String login_count, String accountName) { [ ] 3  
62:89 文字列 queryString = "SELECT * From user_data WHERE Login_Count = ? and userid= [アカウント名; ] 4  
62:26 文字列 queryString = "SELECT * From user_data WHERE Login_Count = ? and userid= [ + アカウント名; ] 5  
62:12 文字列 queryString = "SELECT * From user_data WHERE Login_Count = ? and userid= [ + アカウント名; ] 6  
66:15 [ クエリ文字列、 ]ResultSet.TYPE_SCROLL_INSENSITIVE、ResultSet.CONCUR_READ_ONLY); 7  
65:11 [ 接続.prepareStatement( 8  
シンク
```

### ✓修正分析

詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバに直接渡さないようにしてください。
  - ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。

- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ;などの文字- SQL インジェクション \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- シンに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がprepareStatementに流れ込み、SQLクエリで使用されます。これにより、SQLエラーが発生する可能性があります。  
インジェクションの脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/Servers.java (行: 72)

### データフロー

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

```
67:28    パブリックリスト<Server> sort( @RequestParam String column) 例外をスローします {  
67:28    パブリックリスト<Server> sort( @RequestParam String column) 例外をスローします {  
75:21    +列) {  
73:15    { "SERVERSからID、ホスト名、IP、MAC、ステータス、説明を選択します。ステータスは「out」です"  
         + '順序の順序' +列による順  
         序)  
72:11    接続.prepareStatement(  
ソース 0  
1  
2  
3  
4  
シンク
```

### 修正分析

### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

### 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字 \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallenge.java (行: 69)

### データフロー

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

Machine Translated by Google

```
57:7 @RequestParam 文字列 username_reg,
57:7 @RequestParam 文字列 username_reg,
61:48 AttackResult attackResult = checkArguments( username_reg, email_reg, password_reg); プライベート
93:39 AttackResult checkArguments( String username_reg, String email_reg, String password_reg) {
    「sql_challenge_usersからuseridを選択します。userid =
        "" +username_reg + "'";
    「sql_challenge_usersからuseridを選択します。userid =
        "" +ユーザー名_reg + """; +
    「sql_challenge_usersからuseridを選択します。userid =
        "" + username_reg + """;
66:16 文字列 checkUserQuery =
    「sql_challenge_usersからuseridを選択します。userid =
        "" + username_reg + """;
69:54 ResultSet resultSet = statement.executeQuery( checkUserQuery ); ResultSet
69:31 resultSet = statement.executeQuery( checkUserQuery);
```

ソース 0  
1  
2  
3  
4  
5  
6  
7  
8  
シンク 9



## 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

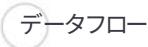
- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字  
\およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTP/パラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6a.java (行: 74)



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

```
56:33 パブリックAttackResult完了(@RequestParam(値 = "userid_6a")文字列userId){
56:33 パブリックAttackResult完了(@RequestParam(値 = "userid_6a")文字列userId){
57:28 injectableQuery(userId) を返します。
62:39 パブリックAttackResult注入可能なクエリ(文字列[アカウント名]{
66:63 クエリ = "SELECT * FROM user_data WHERE last_name = " +アカウント名 + "";
66:15 クエリ = [SELECT * FROM user_data WHERE last_name = " +アカウント名 + ];
66:15 クエリ = [SELECT * FROM user_data WHERE last_name = " +アカウント名 + ""];
66:7 クエリ = "SELECT * FROM user_data WHERE last_name = " +アカウント名 + "";
74:52 ResultSet 結果 = statement.executeQuery( query); ResultSet 結果
74:29 = statement.executeQuery( query);
```

ソース 0  
1  
2  
3  
4  
5  
6  
7  
8  
9



## 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字  
\およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。

脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson10.java (行: 71)

### データフロー

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

```
58:33 パブリックAttackResult完了(@RequestParamStringaction_string){ }  
58:33 パブリックAttackResult完了(@RequestParam String action_string) { 戻り値 }  
59:40 injectableQueryAvailability( action_string); 保護された  
62:54 AttackResult injectableQueryAvailability( String action) { }  
64:70 文字列クエリ = "SELECT * FROM access_log WHERE action LIKE '%" + action + "%'";  
64:20 文字列クエリ = "SELECT * FROM access_log WHERE action LIKE '%" + action + "%'";  
64:20 文字列クエリ = "SELECT * FROM access_log WHERE action LIKE '%" + action + "%'";  
64:12 文字列クエリ = "SELECT * FROM access_log WHERE action LIKE '%" + action + "%'";  
71:52 ResultSet 結果 = statement.executeQuery( query); ResultSet  
71:29 結果 = statement.executeQuery( query);  
ソース 0  
シング 9
```

### ✓修正分析

#### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字  
\およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTP/パラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson2.java (行: 65)

## データフロー

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

```

58:33 パブリックAttackResult完了(@RequestParamStringクエリ){ } 0
58:33 public AttackResult 完了しました( @RequestParam String query) 1
59:28 { return injectableQuery( query); } 2
62:42 protected AttackResult injectableQuery( String query) { ResultSet 3
65:50 結果 = statement.executeQuery( query); ResultSet 結果 = 4
65:27 statement.executeQuery( query); } 5

```

ソース

0

1

2

3

4

シンク

5

## 修正分析

### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

### 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字 \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTP/パラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5.java (行: 80)

## データフロー

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

```

70:33 パブリックAttackResult完了(文字列クエリ){ } 0
70:33 public AttackResult 完了しました( 文字列クエリ ) { return 1
72:28 injectableQuery(クエリ); protected 2
75:42 AttackResult injectableQuery(文字列クエリ) { 3
80:32 { statement.executeQuery(クエリ); } 4
80:9 [ ステートメント.executeQuery( ]クエリ); } 5

```

ソース

0

1

2

3

4

シンク

5

## 修正分析

### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字 \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson5a.java (行: 67)

### データフロー

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

```

55:7 [ @RequestParam 文字列アカウント、 ]@RequestParam 文字列演算子,@RequestParam 文字列注入){
55:7 [ @RequestParam 文字列アカウント、 ]@RequestParam 文字列演算子,@RequestParam 文字列注入{
56:28 return injectableQuery( アカウント + rij + 演算子 + rij + 注射);
56:28 return injectableQuery( アカウント + rij + 演算子 + rij + 注射);
56:28 return injectableQuery( アカウント + rij + 演算子 + rij + 注射);
56:28 return injectableQuery( アカウント + rij + 演算子 + rij + 注射);
56:28 return injectableQuery( アカウント + rij + 演算子 + rij + 注射) ;
59:42 保護されたAttackResult injectableQuery(文字列アカウント名) {
63:83 「SELECT * FROM user_data WHERE first_name = 'John' and last_name = " + [アカウント名 + ""]」
63:11 「SELECT * FROM user_data WHERE first_name = 'John' and last_name = " + アカウント名 + ""; +
63:11 「SELECT * FROM user_data WHERE first_name = 'John' and last_name = " + アカウント名 + "";」
62:7 クエリ =
62:7     「SELECT * FROM user_data WHERE first_name = 'John' and last_name = " + アカウント名 + "";」
67:52 ResultSet 結果 = statement.executeQuery( query); ResultSet結果 = 
67:29 statement.executeQuery( query);
```

ソース 0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
シンク 13

### 修正分析

#### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字 \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

HTTPパラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。

脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/SqlOnlyInputValidation.java (行: 51)

## データフロー

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

```
47:30 パブリックAttackResult攻撃( @RequestParam("userid_sql_only_input_validation") 文字列ユーザーID) {  
47:30 パブリックAttackResult攻撃( @RequestParam("userid_sql_only_input_validation") 文字列ユーザーID) {  
51:58 攻撃結果 attackResult = lessons6a.injectableQuery( userId);
```

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

```
62:39 パブリックAttackResult注入可能なクエリ(文字列[アカウント名] { })  
66:63 クエリ = "SELECT * FROM user_data WHERE last_name = " + [アカウント名 + ""];  
66:15 クエリ = "SELECT * FROM user_data WHERE last_name = " + アカウント名 + "";  
66:15 クエリ = "SELECT * FROM user_data WHERE last_name = " + アカウント名 + "";  
66:7 [ クエリ = "SELECT * FROM user_data WHERE last_name = " + アカウント名 + "";]  
74:52 ResultSet 結果 = statement.executeQuery( query ); ResultSet 結  
74:29 果 = statement.executeQuery( query );
```

### ✓修正分析

詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
  - ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
  - コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
  - すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
  - パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < >;などの文字- SQL インジェクション \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
  - ブランクに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
  - 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

HTTPパラメータからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。

脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/mitigation/SqlOnlyInputValidationOnKeywords.java (行: 5)

データフロー

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

52:7 @RequestParam("userid\_sql\_only\_input\_validation\_on\_keywords") 文字列ユーザーID)

Machine Translated by Google

```
52:7     @RequestParam("userid_sql_only_input_validation_on_keywords") 文字列ユーザーID){  
53:14     userId = userId.toUpperCase().replace("FROM", "").replace("SELECT", "");  
53:14     userId = userId.toUpperCase().replace("FROM", "").replace("SELECT", ""); userId =  
53:14     userId.toUpperCase().replace("FROM", "").replace("SELECT", ""); userId =  
53:14     userId.toUpperCase().replace("FROM", "").replace("SELECT", "");  
53:5     userId = userId.toUpperCase().replace("FROM", "").replace("SELECT", "");  
57:58 攻撃結果 attackResult = lessons6a.injectableQuery( userId );
```

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

```
62:39 パブリックAttackResult注入可能なクエリ(文字列アカウント名) {  
66:63 クエリ = "SELECT * FROM user_data WHERE last_name = " +アカウント名 + "";  
66:15 クエリ = "SELECT * FROM user_data WHERE last_name = " +アカウント名 + "";  
66:15 クエリ = "SELECT * FROM user_data WHERE last_name = " +アカウント名 + ""; }  
66:7 クエリ = "SELECT * FROM user_data WHERE last_name = " +アカウント名 + "";  
74:52 ResultSet 結果 = statement.executeQuery( query ); ResultSet 結  
74:29 果 = statement.executeQuery( query );
```

シンク14



## 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

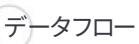
- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。`? & / < > ;`などの文字- SQL インジェクション \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- ンに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTP/パラメータからのサニタイズされていない入力がexecuteUpdateに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson3.java (行: 63)



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

```
53:33 パブリックAttackResult完了(@RequestParam String クエリ){  
53:33 public AttackResult 完了しました( @RequestParam String query ) { return  
54:28 injectableQuery( query ); protected  
57:42 AttackResult injectableQuery( String query )  
63:33 { statement.executeUpdate( query );  
63:9 [ ステートメント.executeUpdate( クエリ);
```

ソース

0

1

1

2

2

3

3

4

4

5

5



## 詳細

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字  
＼およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

HTTPパラメータからのサニタイズされていない入力がexecuteUpdateに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson4.java (行: 62)

### データフロー

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

54:33	パブリックAttackResult完了(@RequestParamStringクエリ){	ソース	0
54:33	パブリックAttackResult完了(@RequestParamStringクエリ){		1
55:28	injectableQuery(クエリ) を返します。		2
58:42	保護されたAttackResult injectableQuery(文字列クエリ) {		3
62:33	ステートメント.executeUpdate(クエリ);		4
62:9	ステートメント.executeUpdate(クエリ);	シンク	5

### ✓修正分析

### 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。

ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。
- コーディングする際は、まずSQLコードを定義し、次にパラメータを渡します。パラメータ化されたクエリでは、準備されたステートメントを使用します。例としては、.NETのSqlCommand()や PHP のbindParam()。
- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
- パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < > ; -などの文字  
＼およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
- SQL インジェクションに対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
- 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

## パストラバーサル

SNYKコード CWE-23 PT

HTTPパラメータからのサニタイズされていない入力がcreateNewFileに流れ込み、パスとして使用されます。これによりパストラバーサルが発生する可能性があります。  
脆弱性を悪用し、攻撃者が任意のファイルを操作できるようになります。

## データフロー

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

```

38:7  @RequestParam(値 = "fullName", 必須 = false) 文字列 fullName){  

38:7  @RequestParam(値 = "fullName", 必須 = false) 文字列 fullName){  

39:32 super.execute(file, fullName) を返します。

```

ソース 0  
1  
2

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

```

31:54 保護された AttackResult を実行 (MultipartFile ファイル、文字列 fullName) { var uploadedFile =  

42:52 new File(uploadDirectory.fullName); var uploadedFile = new  

42:30 File(uploadDirectory.fullName);  

42:11 var uploadedFile = 新しいファイル(uploadDirectory.fullName);  

43:7  アップロードされたファイル。新しいファイルを作成します();  

43:7  アップロードされたファイル.createNewFile();

```

3  
4  
5  
6  
7  
8  
シンク

## 修正分析

## 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 . (ドット)。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に /root/.ssh/authorized\_keys ファイルが上書きされます。

2018-04-15 22:04:29 .....

19

19 良い.txt

2018-04-15 22:04:42 .....

20

20 ../../../../../../root/.ssh/承認キー

## パストラバーサル

SNYK コード CWE-23 PT

HTTP パラメータからのサニタイズされていない入力が createNewFile に流れ込み、パスとして使用されます。これによりパストラバーサルが発生する可能性があります。脆弱性を悪用し、攻撃者が任意のファイルを操作できるようになります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadFix.java (行: 39)

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

```

38:7  [ @RequestParam(値 = "fullNameFix", 必須 = false) 文字列 fullName ]{
38:7  [ @RequestParam(値 = "fullNameFix", 必須 = false) 文字列 fullName ]{
39:51  super.execute(file, fullName != null ? fullName.replace("../", "") : ""); を返します。
39:51  super.execute(file, fullName != null ? fullName.replace("../", "") : "" を返します。
39:32  super.execute(file, fullName != null ? fullName.replace("../", "") : "") を返します。

```

ソース 0  
1  
2  
3  
4

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

```

31:54  保護されたAttackResult実行(MultipartFileファイル、文字列fullName) { [ ]
42:52  var uploadedFile = 新しいファイル(uploadDirectory,fullName); [ ]
42:30  var uploadedFile = 新しいファイル( uploadDirectory, fullName );
42:11  var [ uploadedFile = 新しいファイル(uploadDirectory,fullName); ]
43:7   [ アップロードされたファイル。 ] 新しいファイルを作成します();
43:7   [ アップロードされたファイル.createNewFile( )];

```

5  
6  
7  
8  
9  
シンク 10

### ✓修正分析

### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは。(ドット)の URL エンコード バージョンです。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることもあります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に /root/.ssh/authorized\_keys ファイルが上書きされます。

2018-04-15 22:04:29 .....	19	19 良い.txt
2018-04-15 22:04:42 .....	20	20 ../../../../../../root/.ssh/承認キー

## パストラバーサル

SNYK コード CWE-23 PT

HTTP パラメータからのサニタイズされていない入力が createNewFile に流れ込み、パスとして使用されます。これによりパストラバーサルが発生する可能性があります。脆弱性を悪用し、攻撃者が任意のファイルを操作できるようになります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRemoveUserInput.java (行: 36)

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

```

35:7  @RequestParam("uploadedFileRemoveUserInput") マルチパートファイルファイル) {
35:7  @RequestParam("uploadedFileRemoveUserInput") マルチパートファイルファイル) {
36:32 super.execute(file, file.getOriginalFilename()) を返します。
36:32 super.execute(file, file.getOriginalFilename()) を返します。

```

ソース 0  
1  
2  
3

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

```

31:54  保護されたAttackResult実行(MultipartFileファイル、文字列fullName) {
42:52  var uploadedFile = 新しいファイル(uploadDirectory,fullName);
42:30  var uploadedFile = 新しいファイル(uploadDirectory, fullName);
42:11  var uploadedFile = 新しいファイル(uploadDirectory,fullName);
43:7   アップロードされたファイル。新しいファイルを作成します();
43:7   アップロードされたファイル.createNewFile();

```

4  
5  
6  
7  
8  
9  
シンク

✓修正分析

## 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは、(ドット)の URL エンコード バージョンです。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 .....	19	19 良い.txt
2018-04-15 22:04:42 .....	20	20 ../../../../../../root/.ssh/承認キー

## パストラバーサル

SNYK コード CWE-23 PT

HTTP パラメータからのサンライズされていない入力が transferTo に流れ込み、パスとして使用されます。これにより、パストラバーサルの脆弱性が発生する可能性があります。

攻撃者が任意のディレクトリをコピーできるようになります。

見つかった場所: src/main/java/org/owasp/webgoat/webwolf/FileServer.java (行: 78)

ソース 0  
1  
2  
3  
4  
5  
シンク 6

```
74:34 パブリックModelAndViewインポートファイル @RequestParam("file") マルチパートファイルmyFile)はIOExceptionをスローします{
74:34 public ModelAndView importFile( @RequestParam("file") MultipartFile myFile) throws IOException { myFile.transferTo(new
78:48 File(destinationDir, myFile.getOriginalFilename())); myFile.transferTo(new File(destinationDir,
78:48 myFile.getOriginalFilename()));
78:27 myFile.transferTo(新しいファイル( destinationDir, myFile.getOriginalFilename()));
78:23 myFile.transferTo(新しいファイル(destinationDir, myFile.getOriginalFilename()));
78:5 myFile.transferTo(新しいファイル(destinationDir, myFile.getOriginalFilename()));
```

✓修正分析

## 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unixシステムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリトラバーサルの脆弱性は、一般的に次の2つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダ構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Webページ上で静的ファイルを提供するためのモジュールであり、[このタイプの脆弱性が含まれています](#)。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次のURLを要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 . (ドット)。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることになります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 .....	19	19 良い.txt
2018-04-15 22:04:42 .....	20	20 ../../../../../../root/.ssh/承認キー

## ハードコードされたセキュリティ関連定数の使用

SNYK-CODE CWE-547 非暗号化ハードコードシークレット

秘密にすべき値をハードコードしないでください。ハードコードされた秘密が見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/lessontemplate/SampleAttack.java (行: 44)

データフロー

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

44:24 文字列 secretValue = "secre37Value"; ソースシンク 0

✓修正分析

## 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する: 定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。

将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## 不十分なランダム値の使用 - 秘密

SNYK-CODE CWE-330 安全でない秘密

安全でないランダムデータがorg.apache.commons.lang3.RandomStringUtils.randomAlphabeticから流出し、秘密データとして使用されます。生成代わりに、java.security.SecureRandomなどの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/spoofcookie/encoders/EncDec.java (行: 40)

### データフロー

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

40:31	プライベート静的最終文字列SALT = RandomStringUtils.randomAlphabetic(10);	ソース	0
40:38	プライベート静的最終文字列SALT = RandomStringUtils.randomAlphabetic( 10);		1
40:38	プライベート静的最終文字列SALT = RandomStringUtils.randomAlphabetic( 10);		2
40:31	プライベート静的最終文字列SALT = RandomStringUtils.randomAlphabetic(10);	シンク	3

### 修正分析

### 詳細

コンピュータセキュリティは、安全で機密性の高いセッションキーの生成、パスワードデータのハッシュ化、送信時の暗号化など、多くの点で乱数に依存しています。

機密データなど、他にも多くのデータがあります。理由は簡単に理解できます。例えば、セッションキーが連続的に生成されると、攻撃者は簡単に推測し、

正当なユーザーセッションを乗っ取る可能性があります。同様に、暗号化技術に推測しやすい数字が使用されていた場合、攻撃者はブルートフォース攻撃を用いて不正アクセスを行う可能性があります。

実際には、コンピュータは真の乱数を生成できないため、代わりに「疑似乱数」を使用します。これは、

非常に多様な値をランダムな順序で生成する様々な方法があり、理論上は攻撃者が推測するのは非常に困難です。しかし、開発者が

弱いランダムアルゴリズムを不注意に利用した場合、攻撃者はアルゴリズム、シード、またはパターンを発見し、最終的にコマンドへのアクセスをロック解除したり、

機密データが盗まれ、身代金目的で保持されたり、売却されたりする可能性があります。

## 予防のためのベストプラクティス

- 統計的PRNGなどの弱い疑似乱数生成器（PRNG）の使用は避けてください。代わりに、暗号的に安全なPRNGを選択してください。
- ユーザーIDやサーバーの起動時間など、予測可能なシード値の使用は避けてください。代わりに、外部から取得したシード値など、擬似乱数であるシード値を使用してください。
- ハードウェア ソース。
- 独自のアプローチを採用して、固有の弱点や脆弱性を含む可能性のあるカスタムコードを作成するのではなく、標準的で承認されたセキュリティアルゴリズムとライブラリを使用してください。
- 重大な欠陥を見過ごす。
- 静的解析ツールを使用して、コード内のこの弱点の潜在的なインスタンスを特定し、適切なホワイト ボックス テストで適切なテスト カバレッジを確保します。
- セキュリティ システム開発におけるエンタロピーの重要性について開発者に教育し、FIPS 140-2 に準拠したツールの採用を検討します。

## 信頼できないデータのデシリアライズ

SNYK-CODE CWE-502 デシリアライゼーション

HTTPパラメータからのサニタイズされていない入力がjava.io.ObjectInputStreamに流れ込み、そこでオブジェクトのデシリアライズに使用されます。これにより、次のような結果が生じる可能性があります。  
安全でないデシリアライゼーションの脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/deserialization/InsecureDeserializationTask.java (行: 58)

### データフロー

```
49:33 パブリックAttackResult完了(@RequestParamString トークン)はIOExceptionをスローします{  
49:33 パブリックAttackResult完了(@RequestParamString トークン)はIOExceptionをスローします{  
55:16 b64token = token.replace('-', '+').replace('_', '/');  
55:16 b64token = token.replace( '-', '+').replace( '_', '/') ; b64token =  
55:16 token.replace( '-' , '+').replace( '_' , '/' );  
55:5 b64token = token.replace( '-' , '+').replace( '_' , '/' );  
58:83 新しいObjectInputStream(新しいByteArrayInputStream(Base64.getDecoder().decode(b64token))) {  
58:56 新しい ObjectInputStream(新しい ByteArrayInputStream( Base64.getDecoder().decode( b64token )))) { 新しい  
58:35 ObjectInputStream(新しい ByteArrayInputStream( Base64.getDecoder().decode( b64token ))) { 新しい  
58:31 ObjectInputStream( 新しい ByteArrayInputStream( Base64.getDecoder().decode( b64token ))) { 新しい  
58:13 ObjectInputStream( 新しい ByteArrayInputStream( Base64.getDecoder().decode( b64token ))) {
```



詳細

シリアル化とは、オブジェクトをディスクやデータベースに保存したり、ストリームを通じて送信したりできるバイト列に変換するプロセスです。逆シリアル化は、バイト列からオブジェクトを作成するプロセスはデシリアライゼーションと呼ばれます。シリアル化は、通信（複数のホスト間でオブジェクトを共有すること）によく使用されます。永続性（オブジェクトの状態をファイルやデータベースに保存する）は、リモートメソッド呼び出し（RMI）、Java Management拡張機能（JMX）、Java メッセージング システム（JMS）、アクション メッセージ フォーマット（AMF）、Java Server Faces（JSF） ViewStateなど。

信頼できないデータのデシリアライゼーション ([CWE-502](#)) アプリケーションが信頼できないデータをデシリアライズする際に、結果のデータが有効かどうかを十分に検証せずに、攻撃者が実行の状態やフローを制御できるようになります。

Javaのデシリアライゼーション問題は長年知られていましたが、2015年にリモートアクセスを実現するために悪用される可能性のあるクラスが発見されたことで、この問題への関心が高まりました。人気のあるライブラリ ([Apache Commons Collection](#))でコード実行の脆弱性が見つかりました。これらのクラスは、IBM WebSphere、Oracle WebLogic、および他にもたくさんの製品があります。

攻撃者は、脆弱なクラスが経路上に存在し、信頼できないデータに対してデシリアライズを実行するソフトウェアを特定するだけよい。ペイロードをデシリアライザに送信し、コマンドを実行します。

開発者はJavaのオブジェクトシリализーションに過度の信頼を置いています。認証前にオブジェクトをデシリアライズする開発者もいます。Javaでオブジェクトをデシリアライズする場合、通常、期待される型にキャストするため、Javaの厳密な型システムにより、有効なオブジェクトツリーのみが取得されます。残念ながら、型がチェックが行われる前に、プラットフォームコードはすでに重要なロジックを作成し実行しています。そのため、最終的な型がチェックされる前に、多くのコードが実行されます。様々なオブジェクトのreadObject()メソッドは、開発者のコントロール外にあります。様々なクラスのreadObject()メソッドを組み合わせることで、脆弱なアプリケーションのクラスパスで利用可能な場合、攻撃者は関数を実行できます (Runtime.exec() を呼び出してローカル OS コマンドを実行することを含む)。

### 信頼できないデータのデシリアル化

SNYK-CODE CWE-502 デシリアライゼーション

HTTPパラメータからのサニタイズされていない入力がfromXMLに流れ込み、オブジェクトのデシリアライズに使用されます。これにより、Unsafeエラーが発生する可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/vulnerablecomponents/VulnerableComponentsLesson.java (行: 57)



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

40:47	パブリック @ResponseBody AttackResult 完了( @RequestParam String ペイロード) {	ソース	0
40:47	パブリック @ResponseBody AttackResult 完了( @RequestParam String ペイロード) {		1
50:13	ペイロード		2
50:13	ペイロード 「+」、「」) 。交換する (		3
50:13	ペイロード "\r", """) .replace("+", "") .replace("\r", "") 。交換する (		4
50:13	ペイロード "\n", """) .replace("+", "") .replace("\r", "") 。交換する (		5

```
50:13 ペイロード
        .replace("<", "<")
        .replace(">", ">")
        .replace("\r", "")
        .replace("\n", "")
```

```
50:13 ペイロード
        " <", "<");
        .replace("<", "<")
        .replace(">", ">")
        .replace("\r", "")
        .replace("\n", "")
```

```
49:9 ペイロード =
    ペイロード
        .replace("<", "<")
        .replace(">", ">")
        .replace("\r", "")
        .replace("\n", "")
```

```
57:43 連絡先 = (連絡先) xstream.fromXML( ペイロード ) { 連絡先 = (連
```

```
57:27 絡先) xstream.fromXML( ペイロード );
```

シンク10

### ✓修正分析

### 詳細

シリアル化とは、オブジェクトをディスクやデータベースに保存したり、ストリームを通じて送信したりできるバイト列に変換するプロセスです。逆シリアル化は、バイト列からオブジェクトを作成するプロセスはデシリアライゼーションと呼ばれます。シリアル化は、通信（複数のホスト間でオブジェクトを共有すること）によく使用されます。永続性（オブジェクトの状態をファイルやデータベースに保存する）は、リモートメソッド呼び出し（RMI）、Java Management拡張機能（JMX）、Java メッセージング システム（JMS）、アクション メッセージ フォーマット（AMF）、Java Server Faces（JSF）ViewStateなど。

信頼できないデータのデシリアライゼーション ([CWE-502](#)) アプリケーションが信頼できないデータをデシリアライズする際に、結果のデータが有効かどうかを十分に検証せずに、攻撃者が実行の状態やフローを制御できるようになります。

Javaのデシリアライゼーション問題は長年知られていましたが、2015年にリモートアクセスを実現するために悪用される可能性のあるクラスが発見されたことで、この問題への関心が高まりました。

[人気のあるライブラリ \(Apache Commons Collection\)](#)でコード実行の脆弱性が見つかりました。これらのクラスは、IBM WebSphere、Oracle WebLogic、および

他にも多くの製品があります。

攻撃者は、脆弱なクラスが経路上に存在し、信頼できないデータに対してデシリアライズを実行するソフトウェアを特定するだけよい。

ペイロードをデシリアライザーに送信し、コマンドを実行します。

開発者はJavaのオブジェクトシリアル化に過度の信頼を置いています。認証前にオブジェクトをデシリアライズする開発者もいます。Javaでオブジェクトをデシリアライズする場合、通常、期待される型にキャストするため、Javaの厳密な型システムにより、有効なオブジェクトツリーのみが取得されます。残念ながら、型がチェックが行われる前に、プラットフォームコードはすでに重要なロジックを作成し実行しています。そのため、最終的な型がチェックされる前に、多くのコードが実行されます。様々なオブジェクトのreadObject()メソッドは、開発者のコントロール外にあります。様々なクラスのreadObject()メソッドを組み合わせることで、脆弱なアプリケーションのクラスパスで利用可能な場合、攻撃者は関数を実行できます（Runtime.exec() を呼び出してローカル OS コマンドを実行することを含む）。

## クロスサイトリクエストフォージェリ (CSRF)

SNYK-CODE CWE-352 CSRF保護を無効にする

CSRF保護は無効化によって無効化されます。これにより、攻撃者はユーザーに代わってリクエストを実行できるようになります。

見つかった場所: src/main/java/org/owasp/webgoat/container/WebSecurityConfig.java (行: 80)

### データフロー

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

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

ソースシンク

### ✓修正分析

クロスサイトリクエストフォージェリとは、悪意のある第三者がユーザーの認証情報（ブラウザのCookieなど）を悪用して、ユーザーになります攻撃です。

信頼されたユーザーを騙して不正なアクションを実行させてしまう可能性があります。ウェブアプリケーションサーバーは正当なリクエストと悪意のあるリクエストを区別できません。このタイプの攻撃は通常、ソーシャルエンジニアリング攻撃でユーザーを騙すことから始まります。例えば、ユーザーが誤ってクリックしたリンクやポップアップで不正なリクエストが発生します。

## 予防のためのベストプラクティス

- URL やフォームに対して nonce、ハッシュ、またはその他のセキュリティ デバイスを使用して CSRF を防御する開発フレームワークを使用します。
- 状態変更要求を検証するために毎回サーバーによってチェックされる、安全で一意の隠しトークンを実装します。
- 認証トークンとセッション識別子がリクエストの正当性を意味すると決して想定しないでください。
- ダブルサブミット Cookie などの他のセーフ Cookie テクニックを理解して実装します。
- 自動タイムアウトを含め、使用されていないユーザー セッションを終了します。
- クロスサイトスクリプティング (XSS)などをを利用して、他のよく悪用される CWEに対する厳格なコーディング プラクティスと防御策を確実に実施してください。

CSRF に対する防御。

## クロスサイトリクエストフォージェリ (CSRF)

SNYK-CODE CWE-352 CSRF保護を無効にする

CSRF保護は無効化によって無効化されます。これにより、攻撃者はユーザーに代わってリクエストを実行できるようになります。

見つかった場所: src/main/java/org/owasp/webgoat/webwolf/WebSecurityConfig.java (行: 60)

データフロー

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

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

ソースシンク

0

✓修正分析

## 詳細

クロスサイトリクエストフォージェリとは、悪意のある第三者がユーザーの認証情報（ブラウザのCookieなど）を悪用して、ユーザーになります攻撃です。

信頼されたユーザーを騙して不正なアクションを実行させてしまう可能性があります。ウェブアプリケーションサーバーは正当なリクエストと悪意のあるリクエストを区別できません。このタイプの攻撃は通常、ソーシャルエンジニアリング攻撃でユーザーを騙すことから始まります。例えば、ユーザーが誤ってクリックしたリンクやポップアップで不正なリクエストが発生します。

ウェブサーバーに送信されます。結果は様々です。標準ユーザー レベルでは、攻撃者はパスワードを変更したり、資金を移動したり、購入したり、連絡先に接続したりすることができます。

管理者アカウントからアクセスすると、攻撃者はアプリに変更を加えたり、アプリ自体を削除したりすることさえ可能になります。

## 予防のためのベストプラクティス

- URL やフォームに対して nonce、ハッシュ、またはその他のセキュリティ デバイスを使用して CSRF を防御する開発フレームワークを使用します。
- 状態変更要求を検証するために毎回サーバーによってチェックされる、安全で一意の隠しトークンを実装します。
- 認証トークンとセッション識別子がリクエストの正当性を意味すると決して想定しないでください。
- ダブルサブミット Cookie などの他のセーフ Cookie テクニックを理解して実装します。
- 自動タイムアウトを含め、使用されていないユーザー セッションを終了します。
- クロスサイトスクリプティング (XSS)などをを利用して、他のよく悪用される CWEに対する厳格なコーディング プラクティスと防御策を確実に実施してください。

CSRF に対する防御。

## XML外部エンティティ (XXE)インジェクション

SNYKコード CWE-611 XXE

HTTP リクエスト本体からのサニタイズされていない入力は createXMLStreamReader に流れ込み、外部エンティティ参照の拡張が可能になります。

これにより、XXE 攻撃が発生し、機密データの漏洩やサービス拒否が発生する可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/xxe/BlindSendFileAssignment.java (行: 96)

データフロー

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

87:34 パブリックAttackResult addComment( @RequestBody String commentStr) { [ ] }

ソース

0

87:34 public AttackResult addComment( @RequestBody String commentStr) { コメント [ ] }

1

96:43 comment = comments.parseXml( commentStr); [ ]

2

```

96:30 保護されたコメント parseXml( String xml ) は JAXBException.XMLStreamException をスローします { var
105:58 xsr = xif.createXMLStreamReader(new StringReader( xml )); var xsr =
105:45 xif.createXMLStreamReader(new StringReader([xml])); var xsr =
105:41 xif.createXMLStreamReader( new StringReader(xml) ); var xsr =
105:15 xif.createXMLStreamReader( new StringReader(xml));

```

3  
4  
5  
6  
7  
シンク

## 詳細

利便性のため、XML文書はシステム識別子を使用して、ローカルまたはリモートの保存コンテンツへのアクセスを可能にします。XMLプロセッサはシステム識別子を使用して、URIではなく識別子を使用してリソースにアクセスします。この脆弱性が存在する場合、アプリケーションはユーザーが提供するデータ（例えば、XML外部ID）はXMLパーサーに直接渡されます。アプリケーションは、安全かつ管理された領域外からドキュメントを取得しようとします。

攻撃者はこの脆弱性を悪用して機密データを公開したり、サーバー側でポートスキャンを実行したり、Billion Laughsなどのサービス拒否攻撃 (DoS) を開始したりすることができます。

## 予防のためのベストプラクティス

- 可能であれば、外部エンティティからのデータの読み込みを無効にしてください。その方法は、使用している言語とXMLパーサーによって異なります。
- ローカルの静的ドキュメント タイプ定義 (DTD) を使用し、外部 DTD が完全に禁止されていることを確認します。
- ユーザー入力を避けられない場合は、可能なデータソースのホワイトリストに対して検証を実行します。ただし、外部DTDが許可されている限り、XMLコードはそのまま残ります。この弱点を悪用した攻撃に対して本質的に脆弱です。

## XML外部エンティティ (XXE)インジェクション

SNYKコード CWE-611 XXE

HTTP リクエスト本体からのサンライズされていない入力は createXMLStreamReader に流れ込み、外部エンティティ参照の拡張が可能になります。これにより、XXE 攻撃が発生し、機密データの漏洩やサービス拒否が発生する可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/xxe/ContentTypeAssignment.java (行: 75)



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

```

62:7 [ @RequestBody 文字列コメントStr, ]
62:7 [ @RequestBody 文字列コメントStr, ]
75:45 コメント comment = comments.parseXml( commentStr );

```

ソース  
0  
1  
2

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

```

96:30 保護されたコメント parseXml( String xml ) は JAXBException.XMLStreamException をスローします { var
105:58 xsr = xif.createXMLStreamReader(new StringReader( xml )); var xsr =
105:45 xif.createXMLStreamReader( new StringReader([xml])); var xsr =
105:41 xif.createXMLStreamReader( new StringReader(xml) ); var xsr =
105:15 xif.createXMLStreamReader( new StringReader(xml));

```

3  
4  
5  
6  
7  
シンク

## 詳細

利便性のため、XML文書はシステム識別子を使用して、ローカルまたはリモートの保存コンテンツへのアクセスを可能にします。XMLプロセッサはシステム識別子を使用して、URIではなく識別子を使用してリソースにアクセスします。この脆弱性が存在する場合、アプリケーションはユーザーが提供するデータ（例えば、XML外部ID）はXMLパーサーに直接渡されます。アプリケーションは、安全かつ管理された領域外からドキュメントを取得しようとします。

攻撃者はこの脆弱性を悪用して機密データを公開したり、サーバー側でポートスキャンを実行したり、Billion Laughsなどのサービス拒否攻撃 (DoS) を開始したりすることができます。

## 予防のためのベストプラクティス

- 可能であれば、外部エンティティからのデータの読み込みを無効にしてください。その方法は、使用している言語とXMLパーサーによって異なります。
- ローカルの静的ドキュメント タイプ定義 (DTD) を使用し、外部 DTD が完全に禁止されていることを確認します。

## XML外部エンティティ (XXE)インジェクション

SNYKコード CWE-611 XXE

HTTPリクエスト本体からのサニタイズされていない入力はcreateXMLStreamReaderに流れ込み、外部エンティティ参照の拡張が可能になります。これにより、XXE攻撃が発生し、機密データの漏洩やサービス拒否が発生する可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/xxe/SimpleXXE.java (行: 76)

### データフロー

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

```
73:68    パブリックAttackResult createNewComment(HttpServletRequest リクエスト、@RequestBody String commentStr) {  
73:68    パブリックAttackResult createNewComment(HttpServletRequest リクエスト、@RequestBody String commentStr) {  
76:39    var comment = comments.parseXml( commentStr );
```

ソース 0  
1  
2

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

```
96:30    保護されたコメント parseXml( String xml ) は JAXBException、XMLStreamException をスローします { var  
105:58    xsr = xif.createXMLStreamReader(new StringReader( xml )); var xsr =  
105:45    xif.createXMLStreamReader(new StringReader( xml )); var xsr =  
105:41    xif.createXMLStreamReader( new StringReader(xml) ); var xsr =  
105:15    xif.createXMLStreamReader( new StringReader(xml));
```

3  
4  
5  
6  
7  
シンク

### 修正分析

#### 詳細

利便性のため、XML文書はシステム識別子を使用して、ローカルまたはリモートの保存コンテンツへのアクセスを可能にします。XMLプロセッサはシステム識別子を使用して、URIではなく識別子を使用してリソースにアクセスします。この脆弱性が存在する場合、アプリケーションはユーザーが提供するデータ（例えば、XML外部ID）はXMLパーサーに直接渡されます。アプリケーションは、安全かつ管理された領域外からドキュメントを取得しようとします。

攻撃者はこの脆弱性を悪用して機密データを公開したり、サーバー側でポートスキャニングを実行したり、Billion Laughsなどのサービス拒否攻撃 (DoS) を開始したりすることができます。

#### 予防のためのベストプラクティス

- 可能であれば、外部エンティティからのデータの読み込みを無効にしてください。その方法は、使用している言語とXMLパーサーによって異なります。
- ローカルの静的ドキュメントタイプ定義 (DTD) を使用し、外部 DTD が完全に禁止されていることを確認します。
- ユーザー入力を避けられない場合は、可能なデータソースのホワイトリストに対して検証を実行します。ただし、外部DTDが許可されている限り、XMLコードはそのまま残ります。

この弱点を悪用した攻撃に対して本質的に脆弱です。

## サーバーサイドリクエストフォージェリ (SSRF)

SNYK-CODE CWE-918 Ssrf

HTTPヘッダーからのサニタイズされていない入力がExchangeに流れ込み、リクエストを実行するためのURLとして使用されます。これにより、サーバーサイドリクエストフォージェリの脆弱性が発生する可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignmentForgotPassword.java (行: 104)

### データフロー

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

```
70:19    文字列ホスト = request.getHeader( "host" );  
70:19    文字列ホスト = request.getHeader( "host" );
```

ソース 0  
1

Machine Translated by Google

```
70:12 文字列ホスト = request.getHeader("host");
75:29 fakeClickingLinkEmail( ホスト, resetLink); private
100:38 void fakeClickingLinkEmail( 文字列 ホスト, 文字列 resetLink) { String.format("http://%s/
106:80 PasswordReset/reset/reset-password/%s", ホスト, resetLink),
106:15 "http://%s/PasswordReset/reset/reset-password/%s", ホスト, resetLink), 文字列.format(
104:7 新しい RestTemplate()
    .exchange (
```

シンク

2  
3  
4  
5  
6  
7



## 詳細

サーバーサイドリクエストフォージェリ攻撃では、悪意のあるユーザーがアプリケーションのバックエンドにURL（外部URLまたは127.0.0.1などのネットワークIPアドレス）を提供します。サーバーはURLにアクセスし、その結果を共有します。これにはAWSメタデータ、内部構成情報、データベースなどの機密情報が含まれる場合があります。攻撃者とコンテンツを共有する。リクエストはバックエンドから送信されるため、アクセス制御を回避し、ユーザーが持っていない情報を公開する可能性があります。受信に必要な権限が与えられていないと、攻撃者はこの情報を悪用してアクセス権限を取得し、Webアプリケーションを改変したり、身代金を要求したりする可能性があります。

## 予防のためのベストプラクティス

- ・ ブラックリストには問題があり、攻撃者はそれを回避する方法が多数あります。理想的には、許可されたすべてのドメインと IP アドレスのホワイトリストを使用します。
- ・ サーバー側のリクエストの悪用を防ぐために、独自のネットワーク内でも認証を使用します。
- ・ ゼロトラストを実装し、ユーザーからサーバーに返されるすべてのURLとヘッダーデータをサニタイズして検証します。無効または疑わしい文字を削除し、検査して検証します。有効かつ期待される値が含まれていることを確認します。
- ・ 理想的には、ユーザーが提供するデータに基づいてサーバー要求を送信することは避けてください。
- ・ サーバーからクライアントへ生のレスポンス本文を直接送信していないことを確認してください。期待されるレスポンスのみを配信してください。
- ・ 疑わしい、あるいは悪用される可能性のあるURLスキーマを無効にします。よくある原因としては、file:///gopher://のような、あまり知られていない、あまり使われていないスキーマが挙げられます。、辞書://、ftp://、そして

## SQLインジェクション

SNYK-CODE CWE-89 Sqli

検証されていないJWTクレームからのサニタイズされていない入力がexecuteQueryに流れ込み、SQLクエリで使用されます。これにより、SQLインジェクションが発生する可能性があります。  
脆弱性。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpoint.java (行: 87)



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

```
84:53 最終的な文字列 kid = (String) header.get( "kid");
84:53 最終的な文字列 kid = (String) header.get( "kid");
84:44 最終的な文字列 kid = (String) header.get("kid");
84:38 最終的な文字列 kid = (String) header.get("kid");
90:81 「jwt_keysからキーを選択」 WHERE id =      " + [子供 + """);
90:39 「jwt_keysからキーを選択」 WHERE id =      子供 ]+ """); +
90:39 「jwt_keysからキーを選択」 WHERE id =      + 子供 + """) );
87:31 繋がり
    .createStatement()
    .executeQuery()
```

ソース 0  
1  
2  
3  
4  
5  
6  
7

シンク

2  
3  
4  
5  
6  
7



## 詳細

SQLインジェクション攻撃では、ユーザーは適切な認証情報を提供することなく、データベースに直接SQLクエリを送信してアクセスすることができます。攻撃者は、機密情報をエクスポート、変更、削除したり、パスワードやその他の認証情報を変更したり、社内の他のシステムにアクセスしたりする可能性がある。ネットワーク。これは最も頻繁に悪用される脆弱性の1つですが、適切なコーディングプラクティスによって大部分は回避できます。

## 予防のためのベストプラクティス

- ・ ユーザーが入力したパラメータを SQL サーバーに直接渡さないようにしてください。
- ・ ユーザーが入力したパラメータから SQL クエリを構築するときに文字列連結を使用しないでください。

- すべてのパラメータに強力な型指定を使用して、予期しないユーザー データが拒否されるようにします。
  - パフォーマンス上の理由からユーザーによる直接入力を避けられない場合は、特別な文字の使用を避け、許可された文字の非常に厳格なホワイトリストに基づいて入力を検証します。  
? & / < >;などの文字- SQL インジェクション \およびスペース。可能であれば、ベンダー提供のエスケープルーチンを使用してください。
  - に対する保護を提供する環境および/またはライブラリを使用してアプリケーションを開発します。
  - 理想的には、特定のタスクのみの権限を持つ分離されたアカウントを使用して、最小権限モデルを中心に環境全体を強化します。

パストラバーサル

SNYKコード CWE-23 PT

コマンドライン引数からのサニタイズされていない入力がexistsに流れ込み、パスとして使用されます。これによりパストラバーサルが発生する可能性があります。

脆弱性があり、攻撃者が条件式でアプリケーションのロジックをバイパスできるようになります

見つかった場所: .mvn/wrapper/MavenWrapperDownloader.java (行: 57)



.mvn/wrapper/MavenWrapperDownloader.java

```
50:39 ファイル baseDirectory = new File( args[0] );
50:39 ファイル baseDirectory = new File( args[0] );
50:34 ファイル baseDirectory = new File( args[0] );
50:14 ファイル baseDirectory = new File(args[0]);
55:50 ファイル mavenWrapperPropertyFile = 新しいファイル ( baseDirectory、 MAVEN_WRAPPER_PROPERTIES_PATH );
55:45 ファイル mavenWrapperPropertyFile = 新しいファイル ( baseDirectory、 MAVEN_WRAPPER_PROPERTIES_PATH );
55:14 ファイル mavenWrapperPropertyFile = new File(baseDirectory, MAVEN_WRAPPER_PROPERTIES_PATH);
57:12 if( mavenWrapperPropertyFile.exists() )
57:12 { if( mavenWrapperPropertyFile.exists() {
57:9 if(mavenWrapperPropertyFile.exists() {
    FileInputStream mavenWrapperPropertyFileInputStream = null;
    試す {
        mavenWrapperPropertyFileInputStream = 新しい FileInputStream(mavenWrapperPropertyFile);
        プロパティ mavenWrapperProperties = new Properties();
        mavenWrapperProperties.load(mavenWrapperPropertyFileInputStream);
        url = mavenWrapperProperties.getProperty(PROPERTY_NAME_WRAPPER_URL、 url);
    } キャッチ (IOException e) {
        System.out.println("- 読み込みエラー } 最終的に " + MAVEN_WRAPPER_PROPERTIES_PATH + ""
    }
    試す {
        if(mavenWrapperPropertyFileInputStream != null) {
            mavenWrapperPropertyFileInputStream.close();
        }
    } キャッチ (IOException e) {
        // 無視する ...
    }
}
}
```



詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、章図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドットドットフルッシュ」（「...」）バーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード構成 その他の重要なシステムファイルなどが含まれます

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

バストラバーサリの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CI-Programもバストラバーサリの脆弱性を持つ可能性があります。

昇格された権限 (Unix システムの setuid または setgid フラグなど) で実行されている場合は、トラバーサルが発生します。

ディレクトリトラバーサルの脆弱性は、一般的に次の2つのタイプに分けられます。

- 情報源の改変や削除などが、構造に問題を導き取扱いが不適切となる場合等、他の文書を読み取る上で重要な要素

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 .(ドット)。

- 任意のファイルの書き込み 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

```
2018-04-15 22:04:29 ....          19          19 良い.txt  
2018-04-15 22:04:42 ....          20          20 ../../../../../../root/.ssh/承認キー
```

## パストラバーサル

SNYKコード CWE-23 PT

コマンドライン引数からのサニタイズされていない入力がexistsに流れ込み、パスとして使用されます。これによりパストラバーサルが発生する可能性があります。

脆弱性があり、攻撃者が条件式でアプリケーションのロジックをバイパスできるようになります。

見つかった場所: .mvn/wrapper/MavenWrapperDownloader.java (行: 79)

### データフロー

```
50:39 ファイル baseDirectory = new File( args[0]);          ソース 0  
50:39 ファイル baseDirectory = new File( args[0]); ファイル 1  
50:34 baseDirectory = new File( args[0]); ファイル 2  
50:14 baseDirectory = new File( args[0]); ファイル 3  
78:36 outputFile = new File( baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH); ファイル outputFile = 4  
78:36 new File( baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH); ファイル outputFile = new 5  
78:31 File( baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH); ファイル outputFile = new 6  
78:14 File( baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH); if(! outputFile.getParentFile().exists()) 7  
79:13 { if(! outputFile.getParentFile().exists()) { if(! 8  
79:13 outputFile.getParentFile().exists()) { 9  
79:13 } 10  
79:9 if(!outputFile.getParentFile().exists()) {  
    if(!outputFile.getParentFile().mkdirs()) {  
        システム.out.println(  
            「- 出力ディレクトリの作成中にエラーが発生しました '' + outputFile.getParentFile().getAbsolutePath  
        }  
    }  
}
```

シンク11

### 修正分析

#### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、その他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩:攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

カール [http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\\_rsa](http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_rsa)

注: %2eは URL エンコードされたバージョンです。 .(ドット)。

- 任意のファイルの書き込み 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

```
2018-04-15 22:04:29 ....          19          19 良い.txt  
2018-04-15 22:04:42 ....          20          20 ../../../../../../root/.ssh/承認キー
```

## パストラバーサル

SNYKコード CWE-23 PT

コマンドライン引数からのサニタイズされていない入力がjava.io.FileInputStreamに流れ込み、パスとして使用されます。これにより、Pathエラーが発生する可能性があります。

トラバーサルの脆弱性があり、攻撃者が任意のファイルを読み取ることができます。これにより、Pathエラーが発生する可能性があります。

見つかった場所: .mvn/wrapper/MavenWrapperDownloader.java (行: 60)

### データフロー

50:39	ファイル baseDirectory = new File( args[0] );	ソース	0
50:39	ファイル baseDirectory = new File( args[0] );		1
50:34	ファイル baseDirectory = new File( args[0] );		2
50:14	ファイル baseDirectory = new File(args[0]);		3
55:50	ファイル mavenWrapperPropertyFile = 新しいファイル ( baseDirectory、 MAVEN_WRAPPER_PROPERTIES_PATH );		4
55:45	ファイル mavenWrapperPropertyFile = 新しいファイル ( baseDirectory、 MAVEN_WRAPPER_PROPERTIES_PATH );		5
55:14	ファイル mavenWrapperPropertyFile = 新しい File(baseDirectory,MAVEN_WRAPPER_PROPERTIES_PATH);		6
60:75	mavenWrapperPropertyFileInputStream = 新しい FileInputStream( mavenWrapperPropertyFile );		7
60:59	mavenWrapperPropertyFileInputStream = 新しい FileInputStream( mavenWrapperPropertyFile );	シンク	8

### 修正分析

#### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、その他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります。

カール [http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\\_rsa](http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id_rsa)

注: %2eは URL エンコードされたバージョンです。 .(ドット)。

- 任意のファイルの書き込み 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

Machine Translated by Google  
これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、

悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることがあります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 .... 19 19 良い.txt  
2018-04-15 22:04:42 .... 20 ..../..../..../..../root/.ssh/承認キー

## パストラバーサル

SNYKコード CWE-23 PT

コマンドライン引数からのサニタイズされていない入力がmkdirsに流れ込み、パスとして使用されます。これによりパストラバーサルが発生する可能性があります。

脆弱性を悪用し、攻撃者が任意のファイルを操作できるようになります。

見つかった場所: .mvn/wrapper/MavenWrapperDownloader.java (行: 80)

### データフロー

ソース 0  
50:39 ファイル baseDirectory = new File( args[0] );  
50:39 ファイル baseDirectory = new File( args[0] );  
50:34 ファイル baseDirectory = new File( args[0] );  
50:14 ファイル baseDirectory = new File(args[0]); ファイル  
78:36 outputFile = new File( baseDirectory.getAbsolutePath(), MAVEN\_WRAPPER\_JAR\_PATH ); ファイル outputFile = new  
78:36 File( baseDirectory.getAbsolutePath(), MAVEN\_WRAPPER\_JAR\_PATH );  
78:31 ファイル出力ファイル = 新しいファイル( baseDirectory.getAbsolutePath(), MAVEN\_WRAPPER\_JAR\_PATH ).  
78:14 ファイル outputFile = new File(baseDirectory.getAbsolutePath(), MAVEN\_WRAPPER\_JAR\_PATH); if(!  
80:17 outputFile.getParentFile().mkdirs() { if(!  
80:17 outputFile.getParentFile().mkdirs() { if(!  
80:17 outputFile.getParentFile().mkdirs() {  
シンク 10

### 修正分析

#### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 . (ドット)。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、

悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることがあります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

## パストラバーサル

SNYKコード CWE-23 PT

コマンドライン引数からのサニタイズされていない入力がjava.io.FileOutputStreamに流れ込み、パスとして使用されます。これにより、Pathエラーが発生する可能性があります。トラバーサルの脆弱性があり、攻撃者が任意のファイルに書き込むことができます。

見つかった場所: .mvn/wrapper/MavenWrapperDownloader.java (行: 111)

### データフロー

```

50:39 ファイル baseDirectory = new File( args[0] );
50:39 ファイル baseDirectory = new File( args[0] );
50:34 ファイル baseDirectory = new File( args[0] );
50:14 ファイル baseDirectory = new File(args[0]);
78:36 ファイル出力ファイル = 新しいファイル ( baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH); ファイル出力
78:36 ファイル = 新しいファイル ( baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH); ファイル出力ファイル =
78:31 新しいファイル ( baseDirectory.getAbsolutePath(), MAVEN_WRAPPER_JAR_PATH);
78:14 ファイル出力ファイル = 新しいファイル (baseDirectory.getAbsolutePath(),MAVEN_WRAPPER_JAR_PATH);
87:38 downloadFileFromURL(url, outputFile); private
97:63 static void downloadFileFromURL(String urlString, File destination) throws Exception {
111:53 FileOutputStream fos = new FileOutputStream( destination );
111:36 FileOutputStream fos = new FileOutputStream( destination );

```

ソース 0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
シンク 11

### 修正分析

#### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルートユーザーの機密性の高い秘密鍵が漏洩することになります。

カール http://localhost:8080/public/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/root/.ssh/id\_rsa

注: %2eは URL エンコードされたバージョンです。 . (ドット)。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わることもあります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/authorized\_keysファイルが上書きされます。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/challenges/challenge1/Assignment1.java (行: 55)

## データフロー

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

```
51:38  @RequestParam String ユーザー名、@RequestParam String パスワード、HttpServletRequest リクエスト) {  
55:16  && パスワード  
          .replace("1234", String.format("%04d", ImageServlet.PINCODE))  
          .equals(
```

ソース

0

シンク

1

## 修正分析

## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのロックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは

攻撃者は、ロックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけよい。

有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に

安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
  - パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
  - アプリが攻撃者を助ける手がかりを与える場合、暗号化だけでは不十分であることを開発者に認識させましょう。
  - タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。
- 生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/idor/IDORLogin.java (行: 64)

## データフロー

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

```
59:64  パブリックAttackResult完了(@RequestParam String ユーザー名、@RequestParam String パスワード) {  
64:37  if ("tom".equals(ユーザー名) && idor.userInfo.get("tom").get("password").equals(パスワード)) {
```

ソース

0

シンク

1

## 修正分析

## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのロックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは

攻撃者は、ロックアップテーブルを破壊し、ブルートフォース攻撃を仕掛けたる貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけよい。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与えている場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/insecurelogin/InsecureLoginTask.java (行: 36)

### データフロー

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

```
35:64    パブリックAttackResult完了(@RequestParam String ユーザー名,@RequestParam String パスワード) {  
36:43        if ("CaptainJack".equals(ユーザー名) && "BlackPearl".equals(パスワード)) {
```

ソース

0

シンク

1

### 修正分析

### 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのレックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは

攻撃者は、レックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。

有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に

安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与えている場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java (行: 60)

### データフロー

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

```
55:64    パブリックAttackResult完了(@RequestParam String ユーザー名,@RequestParam String パスワード) {
```

ソース

0



## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリはアクセスを高速化するために、有効なセッションIDのルックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは攻撃者は、ルックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与える場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。  
生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

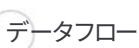
## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest.isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java (行: 90)



## データフロー



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

```
84:29    パブリックAttackResultログイン(@RequestParamStringパスワード,@RequestParamStringメール){  
90:18    } そうでなければ、(passwordTom.equals(password)){
```

ソース

0

シンク

1



## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリはアクセスを高速化するために、有効なセッションIDのルックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは攻撃者は、ルックアップテーブルを破壊し、ブルートフォース攻撃を仕掛けたる貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与える場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。  
生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest.isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (行: 85)

## データフロー

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

```
63:7  [ @RequestParam 文字列パスワード、 ]
85:12  && users.get(lowerCasedUsername).equals(パスワード) {
```

ソース

0

シンク

1

## 修正分析

### 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのルックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは

攻撃者は、ルックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけです。

有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に

安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

### 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与えていている場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。
- 生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest.isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (行: 90)

## データフロー

```
63:7  [ @RequestParam 文字列パスワード、 ]
90:36  if (!authPassword.isBlank() && authPassword.equals(パスワード)) {
```

ソース

0

シンク

1

## 修正分析

### 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのルックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは

攻撃者は、ルックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけです。

有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に

安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

### 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与えていている場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。
- 生産可能です。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

weakAntiCSRFの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。  
値を安全に比較するには java.security.MessageDigest.isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/csrf/ForgedReviews.java (行: 106)

### データフロー

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

56:31 プライベート静的最終文字列 weakAntiCSRF = "2aa14227b9a13d0bede0388a7fba9aa9";  
106:33 検証要求 == null の場合 || !検証要求.equals(weakAntiCSRF) {

ソース

0

シンク

1

### 修正分析

#### 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。  
攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは  
アクセスを高速化するために、有効なセッションIDのルックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは  
攻撃者は、ルックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。  
有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に  
安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

#### 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与えていた場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。  
生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はHTMLに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。  
側。これにより、DOMベースのクロスサイトスクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/lessons/challenges/js/challenge8.js (行: 18)

### データフロー

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

7時43分 \$.get("challenge/8/votes/", 関数 ( votes) { [ ] )  
7時43分 \$.get("challenge/8/votes/", 関数 ( votes) { [ ] )  
18時42分 \$("#nrOfVotes" + i).html( 投票数[ i]); [ ] )  
18時42分 \$("#nrOfVotes" + i).html( 投票[i]); [ ] )  
18時37分 \$("#nrOfVotes" + i).html([ 投票数]i); [ ] )

ソース

0

シンク

1

2

3

4

### 修正分析

#### 詳細

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型 XSS とは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くの WAF は、しばしば失敗します。

DOM XSS 脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定の DOM 操作に基づいて、危険な DOM/JavaScript を通じて露出される攻撃面を最小限に抑える安全なコーディング プラクティスを採用する

API。さらに、最新のウェブ アプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOM ベースの攻撃に対する保護。

DOM XSS 攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

## 攻撃の種類

タイプ	ソースカテゴリの説明	技術的な詳細	
URLパラメータ 注入	ナビゲーション 脆弱な JavaScript によって処理されるパラメータ location.search または同様のプロパティにアクセスするコード。	一般的に URLSearchParams の直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History API の更新を含む) pushState()。	
フラグメント識別子 擷取	ナビゲーション URL ハッシュフラグメント (#) を利用した攻撃ベクトルは、 location.hash を通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA) では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。	
HTTPリファラー 操作	リクエストコンテキスト document.referrer 値に含まれる 悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトが脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。	
ブラウザストレージ 攻撃	永続ストレージローカルストレージへの悪意のあるデータ注入 sessionStorage 、またはその後読み取られる IndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して) そして、 段階的な擷取のために他のベクトルと組み合わせます。	
クロスフレーム コミュニケーション	インターフレーム メッセージング	埋め込み iframe やポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウ メカニズム。	
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	WebSocket 経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History API ルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウェブ

## 予防のためのベストプラクティス

このセクションでは、安全な開発 プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定の DOM コンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間の POST メッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ 、 セッションストレージ 、 または他のクライアント側ストレージメカニズムを使用する前に  
データをサニタイズします。

## 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際には、文字列引数を持つ innerHTML 、 outerHTML 、 adjacentHTML 、 setTimeout() / setInterval() などのリスクの高い DOM シンクを避けてください。
- textContent 実行可能コードなど、より安全な DOM 操作方法を使用してください。setAttribute 、 テキストノードを作成() 、 DOM 要素作成 API はコンテンツを解釈しない  
() は URL/イベント属性以外の属性にのみ使用し、信頼できないデータから on\* ハンドラーや危険な URL 属性を設定することは避けてください。

- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、データ転送用のJSONなどの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー(CSP)ディレクティブ(例: script-src、object-src)を展開し、DOM XSSが存在する場所(ベースURI、frame-ancestors)を使用してスクリプトの実行を制限し、合の影響を軽減します。
- すべての外部JavaScriptリソースにサブリソース整合性(SRI)を実装し、DOM XSSを引き起こす可能性のあるサードパーティライブラリの改ざんを防止します。
- X-Content-Type-Options: nosniff、Referrer-Policy、(CSPが利用できない場合) X-Frame-OptionsなどのHTTPセキュリティヘッダーを活用し、クリックジャッキングの保護にはCSPのframe-ancestorsディレクティブを優先します。
- 承認されたサニタイザーを通過しない限りDOMシンクへの割り当てを防止するために、Trusted Types(サポートされている場合)などの最新のブラウザ防御を採用します。CORSをDOM XSS制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。
- 開発プロセス。
- JavaScriptコードのDOM XSS脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。
- 発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト(DAST)を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立するセキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング(XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はHTMLに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。側。これにより、DOMベースのクロスサイトスクリプティング攻撃(DOMXSS)が発生する可能性があります。

見つかった場所: src/main/resources/lessons/challenges/js/challenge8.js (行: 52)

### データフロー

46:50	\$get("challenge/8/vote/" + 星, 関数(結果){	[ ]	ソース	0
46:50	\$get("challenge/8/vote/" + 星, 関数(結果){	[ ]		1
52:34	\$("#voteResultMsg").html(結果["メッセージ"]);	[ ]		2
52:41	\$("#voteResultMsg").html(結果["メッセージ"]); \$	[ ]		3
52:34	("#voteResultMsg").html(結果["メッセージ"]);	[ ]		4
52:29	(\$("#voteResultMsg").html(result["メッセージ"]));	[ ]	シンク	5

### 修正分析

### 詳細

DOMベースのクロスサイトスクリプティング(DOM XSS)は、攻撃者が制御するデータ(例: location.search document.referrer、ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来のサーバー側の防御や多くのWAFは、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用するAPI。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー(CSP)ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。DOMベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、ブラウザ環境内でユーザーを直接ターゲットにします。

### 攻撃の種類

URLパラメータ 注入	ナビゲーション	された悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
フラグメント識別子 擷取	ナビゲーション	URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる	悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ 攻撃	永続ストレージローカルストレージへの悪意のあるデータ注入	sessionStorage、またはその後読み取られる IndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。
クロスフレーム コミュニケーション	インターフレーム メッセージング	悪意のあるデータが投稿されたpostMessage APIの悪用 フレームまたはウィンドウ間で送信され、処理される 適切なオリジン検証と入力サニタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	WebSocket経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ ディレクトリ、セッションストレージ、または他のクライアント側ストレージメカニズムを使用する前にデータをサニタイズします。

### 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際に、文字列引数を持つinnerHTML、外側のHTML、insertAdjacentHTMLユーティリティ、ドキュメント.write()、ドキュメント.writeln()、評価()、関数()、setTimeout() / setInterval()などのリスクの高いDOMシンクを回避します。
- textContent実行可能コードなど、より安全なDOM操作方法を使用してください。setAttribute、テキストノードを作成()、DOM要素作成APIはコンテンツを解釈しない()はURL/イベント属性以外の属性にのみ使用し、信頼できないデータからon\*ハンドラーや危険なURL属性を設定することは避けてください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、データ転送用の JSON などの構造化データ形式。

### ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、合の影響が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティ ライブリリースの改ざんを防止します。
- クリックジャッキング保護のために、CSP のframe-ancestorsディレクティブを優先するX-Content-Type-Options、リファラーポリシー、および (CSPが利用できない場合) X-Frame-Options、その間 Options: nosniffを含むHTTPセキュリティヘッダーを活用します。
- 承認されたサニタイザーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

### 開発とテストの実践

- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はHTMLに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。

側。これにより、DOM ベースのクロスサイト スクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/webgoat/static/js/goatApp/view/LessonContentView.js (行: 178)

### データフロー

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

```
117:35     }).then(関数 (データ){ [ ] )  
117:35     }).then(関数 (データ){ [ ] )  
118:44 self.onSuccessResponse( データ, [failureCallbackFunctionName, successCallBackFunctionName, informationalCallbackFunctionName] ) onSuccessResponse:  
123:42     function ( データ, failureCallbackFunctionName, successCallBackFunctionName, informationalCallbackFunctionName  
124:37         this.renderFeedback(データ.feedback);  
124:42         this.renderFeedback(data.feedback); [ ]  
176:39     renderFeedback: 関数 (フィードバック){ [ ] )  
177:52     var s = this.removeSlashesFromJSON( フィードバック ); [ ]  
169:46     JSONからスラッシュを削除する: 関数 ( str){ [ ] )  
173:24     str を返します。replace(/\\"(.)/g, "$1");  
173:28     str を返します。replace(/\\"(.)/g, "$1");  
177:21     [ ] = this.removeSlashesFromJSON(フィードバック); var s  
178:51     this.$curFeedback.html(polyglot.t(s) || ""); this.  
178:49     $curFeedback.html(polyglot.t(s) || ""); [ ]  
178:40     this.$curFeedback.html( polyglot.t(s) || ""); [ ]  
178:35     this.$curFeedback.html([polyglot].t(s) || "");  
[ ] ソース 0  
[ ] 1  
[ ] 2  
[ ] 3  
[ ] 4  
[ ] 5  
[ ] 6  
[ ] 7  
[ ] 8  
[ ] 9  
[ ] 10  
[ ] 11  
[ ] 12  
[ ] 13  
[ ] 14  
シンク 15
```

### 修正分析

#### 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS) は、攻撃者が制御するデータ (例: location.search document.referrer

、場所ハッシュ)

、ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くのWAFは、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOM ベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

#### 攻撃の種類

タイプ			技術的な詳細
URLパラメータ	ナビゲーション	ソース カテゴリ 説明URL クエリ内に埋め込み された悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
注入			
フラグメント識別子	ナビゲーション	URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
擷取			
HTTPリファラー	リクエストコンテキストdocument.referrer値に含まれる		ユーザーを誘導するためにソーシャルエンジニアリングが必要
操作		悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトが脆弱なサイトにリンクしている 応用。	細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ	永続ストレージローカルストレージへの悪意のあるデータ注入		攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる
攻撃		sessionStorage、またはその後読み取られる IndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	(別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合せます。
クロスフレーム	インターフレーム	悪意のあるデータが投稿されたpostMessage APIの悪用	埋め込みiframeやポップアップのあるアプリケーションでよく見られる
コミュニケーション	メッセージング	フレームまたはウィンドウ間で送信され、処理される 適切なオリジン検証と入力サンタイズが行われていない。	クロスオリジン通信を実装するウィンドウ メカニズム。
ウェブソケット	リアルタイム	WebSocket経由で配信される悪意のあるペイロード	チャットなどのリアルタイムアプリケーションに特に効果的
メッセージインジェクション	コミュニケーション	クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共に機能を実装するソーシャルメディアプラットフォームとウェブ

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサンタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージデ、セッションストレージ、または他のクライアント側ストレージメカニズムを使用する前に  
データをサンタイズします。

### 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際に、文字列引数を持つinnerHTML、外側のHTML、insertAdjacentHTMLユー、ドキュメント.write()、ドキュメント.writeln()、評価()、関数()、  
setTimeout() / setInterval()などのリスクの高いDOMシンクを回避します。
- textContent実行可能コードなど、より安全なDOM操作方法を使用してください。setAttribute、テキストノードを作成()、DOM要素作成APIはコンテンツを解釈しない  
(!)はURL/イベント属性以外の属性にのみ使用し、信頼できないデータからon\*ハンドラーや危険なURL属性を設定することは避けてください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、  
データ転送用の JSON などの構造化データ形式。

### ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、  
合の影響が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティ ライブラリの改ざんを防止します。  
脆弱性。
- クリックジャッキング保護のために、CSP のframe-ancestorsディレクティブを優先するX-Content-Type-、リファラーポリシー、および (CSPが利用できない場合) X-Frame-Options、  
Options: nosniffを含むHTTPセキュリティヘッダーを活用します。
- 承認されたサンタイマーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

### 開発とテストの実践

- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立するセキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はHTMLに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。

側。これにより、DOM ベースのクロスサイト スクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/webgoat/static/js/goatApp/view/LessonContentView.js (行: 185)

### データフロー

```
117:35     }).then(関数 (データ){ [ ] 0
117:35     }).then(関数 (データ){ [ ] 1
118:44 self.onSuccessResponse( データ, failureCallbackName, successCallBackFunctionName, informationalCallbackFunctionName) onSuccessResponse: 2
123:42     function ( データ, failureCallbackFunctionName, successCallBackFunctionName, informationalCallbackFunctionName 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 || ""); renderOutput: 7
183:37         function (output) { var s = [ ] 8
184:52             this.removeSlashesFromJSON(output); [ ] 9
169:46             removeSlashesFromJSON: function (str) { [ ] 10
173:24                 str を返します。replace(/\\"(.)/g, "$1"); 11
173:28                 str を返します。replace(/\\"(.)/g, "$1"); 12
184:21                     [ ] = this.removeSlashesFromJSON(出力); var s 13
185:49             this.$curOutput.html(polyglot.t( s) || ""); this. 14
185:47             $curOutput.html(polyglot.t( s) || ""); this. 15
185:38             $curOutput.html( polyglot.t(s) || ""); this. 16
185:33             $curOutput.html( polyglot.t(s) || ""); 17
                                         シンク
```

### 修正分析

### 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS) は、攻撃者が制御するデータ (例: location.search 場所.ハッシュ)

ドキュメント.リファーー、ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くの WAF は、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOM ベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

### 攻撃の種類

URLパラメータ 注入	ナビゲーション	された悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
フラグメント識別子 擷取	ナビゲーション	URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる	悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトが脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ 攻撃	永続ストレージローカルストレージへの悪意のあるデータ注入	sessionStorage、またはその後読み取られる IndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。
クロスフレーム コミュニケーション	インターフレーム メッセージング	悪意のあるデータが投稿されたpostMessage APIの悪用 フレームまたはウィンドウ間で送信され、処理される 適切なオリジン検証と入力サニタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	WebSocket経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ ディレクトリ、セッションストレージ、または他のクライアント側ストレージメカニズムを使用する前に  
データをサニタイズします。

### 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際に、文字列引数を持つinnerHTML、外側のHTML、insertAdjacentHTMLユーティリティ、ドキュメント.write()、ドキュメント.writeln()、評価()、関数()、setTimeout() / setInterval()などのリスクの高いDOMシンクを回避します。
- textContent実行可能コードなど、より安全なDOM操作方法を使用してください。setAttribute、テキストノードを作成()、DOM要素作成APIはコンテンツを解釈しない()  
()はURL/イベント属性以外の属性にのみ使用し、信頼できないデータからon\*ハンドラーや危険なURL属性を設定することは避けてください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、  
データ転送用の JSON などの構造化データ形式。

### ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、  
合の影響が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティ ライブリージョンを防止します。
- クリックジャッキング保護のために、CSP のframe-ancestorsディレクティブを優先するX-Content-Type-Options、リファラーポリシー、および (CSPが利用できない場合) X-Frame-Options、その間  
Options: nosniffを含むHTTPセキュリティヘッダーを活用します。
- 承認されたサニタイザーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジン データの公開を制限するように適切に構成します。

### 開発とテストの実践

- JavaScript コード内の DOM XSS 脆弱性を検出し、ソースからシンクへの安全でないデータ フローを識別できる特殊なツールを使用して、自動静的分析を実装します。
- 特殊な DOM XSS 検出機能と手動侵入テストを使用して動的アプリケーション セキュリティ テスト (DAST) を実行し、実装されたセキュリティ制御の有効性を検証します。
- DOM XSS 攻撃ベクトル、安全な JavaScript プログラミング プラクティス、セキュリティ重視のライブラリとフレームワークの適切な使用に関するトレーニングを含む、開発チーム向けの安全なコーディング ガイドラインを確立します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/challenges/SolutionConstants.java (行: 34)

### データフロー

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

34:21 文字列 PASSWORD = '!!webgoat\_admin\_1234!!';

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/challenges/SolutionConstants.java (行: 35)

### データフロー

35:25 文字列 PASSWORD\_TOM = "thisisasecretfortomonly";

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/idor/IDORLogin.java (行: 51)

### データフロー

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

51:46 idorUserInfo.get("bill").put("password", "buffalo");

ソースシンク

0

### ✓修正分析

#### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (行: 61)

### データフロー

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

61:41 パブリック静的最終文字列 PASSWORD = "bm5nhSkxCXZkKRy4";

ソースシンク

0

### ✓修正分析

#### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionAC.java (行: 32)

### データフロー

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

32:53 パブリック静的最終文字列 PASSWORD\_SALT\_SIMPLE = "DeliberatelyInsecure1234";

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionAC.java (行: 33)

### データフロー

33:52 パブリック静的最終文字列 PASSWORD\_SALT\_ADMIN = "DeliberatelyInsecure1235";

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java (行: 62)

### データフロー

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

62:7 「トムのパスワードとして誰も入力できない非常にランダムな何か」;

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。イコールでハードコードされたパスワードが見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/insecurelogin/InsecureLoginTask.java (行: 36)

### データフロー

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

36:43 if ("CaptainJack".equals(ユーザー名) && "BlackPearl".equals(パスワード)) { }

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードは展開前に削除されることが想定されていますが、コード内にうっかり残ってしまうことがあります。これは深刻なセキュリティリスクをもたらし、特にパスワードが昇格された権限を付与している場合に顕著です。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。イコールでハードコードされたパスワードが見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpoint.java (行: 77)

### データフロー

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

77:43 if ("Jerry".equalsIgnoreCase(ユーザー) && PASSWORD.equals(パスワード)){

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード

コード内にパスワードをハードコードしないでください。イコールでハードコードされたパスワードが見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/passwordreset/ResetLinkAssignment.java (行: 88)

### データフロー

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

88:11 if ( passwordTom.equals( PASSWORD\_TOM\_9) {

ソースシンク

0

### 修正分析

### 詳細

権限が複数のシステム間で再利用される場合もあります。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限の昇格、機密データの窃取、サービスの可用性の妨害などの被害を受ける可能性があります。

パスワードがさまざまな環境やアプリケーション間で再利用されると、侵害が急速かつ広範囲に広がる可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## HTTP ヘッダー内の CRLF シーケンスの不適切な無効化

SNYK-CODE CWE-113 HttpResponsSplitting

HTTP パラメータからのサニタライズされていない入力が addCookie に流れ込み、ユーザーに返される HTTP ヘッダーに到達します。これにより、CR/LF を含む悪意のある入力により、http 応答が 2 つの応答に分割され、2 番目の応答が攻撃者によって制御される可能性があります。

これは、クロスサイトスクリプティングやキャッシュポイズニングなどのさまざまな攻撃を仕掛けるために使用される可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/hijacksession/HijackSessionAssignment.java (行: 89)

### データフロー

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

```
63:7  [ @RequestParam 文字列 ユーザー名、 ] ソース 0
63:7  [ @RequestParam 文字列 ユーザー名、 ] 1
72:45 Authentication.builder().name( ユーザー名 ).credentials( パスワード ).build(); 2
72:15 [ Authentication.builder().name( ) ユーザー名 ].credentials( パスワード ).build(); 3
72:15 [ Authentication.builder().name(ユーザー名).credentials( ) パスワード ].build(); 4
72:15 [ Authentication.builder().name(ユーザー名).credentials( パスワード ).build( ) ); 5
71:11 [ プロバイダー.認証( ) ] 6
70:7 認証 = 7
    プロバイダー.認証(
        Authentication.builder().name(ユーザー名).credentials( パスワード ).build(); )
73:27 setCookie( レスポンス、認証.getId() ); 8
73:27 setCookie( レスポンス、authentication.getId() ); 9
85:56 プライベート void setCookie( HttpServletResponse レスポンス、String cookieValue ) { } 10
86:45 Cookie cookie = 新しい Cookie( COOKIE_NAME、cookieValue ); 11
86:25 Cookie cookie = 新しい Cookie( COOKIE_NAME, cookieValue ); 12
86:12 Cookie cookie = 新しい Cookie( COOKIE_NAME、cookieValue ); 13
89:24 response.addCookie( cookie ); 14
89:5 [ response.addCookie( ) cookie ]; シング 15
```

### 修正分析

#### 詳細

CRLF は「キャリッジリターン」と「ラインフィード」の略語です。この 2 つの特殊文字は、初期の印刷端末で使用されていた旧式の印刷端末の名残です。

コンピュータ時代。しかし、今日でも両方ともデータ間の区切り文字としてよく使われています。この弱点が存在する場合、CR と LF 文字（それぞれ

\r と \n などのコード）が HTTP ヘッダー内に存在することが許可されていますが、これは通常、開発中のデータ処理の計画が不十分なためです。

HTTP ヘッダーの CRLF シーケンスは、これらの文字がブラウザからの応答を効果的に分割し、1 行を分割するため、「応答分割」として知られています。

サーバーによって複数行として受け入れられます（たとえば、1 行目の First Line\r\nSecond Line は、サーバーによって 2 行の入力として受け入れられます）。

レスポンス分割自体は攻撃ではなく、悪用されない限り完全に無害ですが、その存在はインジェクション攻撃（CRLF 攻撃として知られる）につながる可能性があります。

インジェクションや、予測不可能で潜在的に危険な様々な動作を引き起こす可能性があります。この脆弱性は、ページハイジャックやクロスユーザー攻撃など、さまざまな方法で悪用される可能性があります。

改ざんとは、攻撃者が偽のサイトコンテンツを表示したり、認証情報などの機密情報を取得したりする行為です。クロスサイトスクリプティング攻撃につながる可能性もあります。

攻撃者がユーザーのブラウザで悪意のあるコードを実行させる可能性があります。

たとえば、次のコードは脆弱です。

```

Cookie cookie = 新しい Cookie("name", request.getParameter("name"));
レスポンスにCookieを追加します(クッキー)。
}

```

ユーザーがXYZ\r\nHTTP/1.1 200 OK\nATTACKER CONTROLLEDのような値を持つ名前パラメータを指定する可能性があるからです。

この場合、2番目のHTTPが生成されます

応答：

HTTP/1.1 200 OK

攻撃者を制御

考えられる修正方法は、英数字以外の文字をすべて削除することです。

```

保護された void doGet(HttpServletRequest リクエスト、HttpServletResponse レスポンス) {
    文字列名 = request.getParameter("name")
        .replaceAll("[^a-zA-Z]", "");
    Cookie cookie = 新しい Cookie("name", name);
    レスポンスにCookieを追加します(クッキー)。
}

```

この場合、攻撃者は2番目のHTTP応答を生成できなくなります。

## 予防のためのベストプラクティス

- すべての入力は潜在的に悪意のあるものであると想定してください。可能な限り許容可能な応答を定義し、それが不可能な場合は、ヘッダーの改ざんを防ぐためにCRとLF文字をエンコードしてください。分割します。
- \r (キャリッジリターン)と\n (ラインフィード)の両方を"" (空文字列)に置き換えます。多くのプラットフォームではこれらの文字を互換的に扱うため、脆弱性は依然として存在する可能性があります。2つのうち1つが許可されている場合、この文字は存在します。ベストプラクティスに従い、可能な限り他のすべての特殊文字 ("、/、;など、およびスペース)を削除してください。必ずサニタイズしてください。ブラウザからサーバーへの方向と、ブラウザに返されるデータの両方向で特殊文字が使用されるようにします。理想的には、最新の開発リソースを採用します。言語やライブラリなど、ヘッダーへのCRとLFの挿入をブロックするツールが存在します。ユーザー側で改ざんまたは変更される可能性のあるすべての入力タイプには注意が必要です。

意図的または意図せずに、GET、POST、Cookie、その他のHTTPヘッダーなど、HTTPリクエストの終端を（意図的または意図せずに）変更し、インジェクション攻撃につながる可能性があります。

## HTTP ヘッダー内の CRLF シーケンスの不適切な無効化

SNYK-CODE CWE-113 HttpResponseSplitting

HTTP/パラメータからのサニタイズされていない入力がaddCookieに流れ込み、ユーザーに返されるHTTPヘッダーに到達します。これにより、CR/LFを含む悪意のある入力により、http応答が2つの応答に分割され、2番目の応答が攻撃者によって制御される可能性があります。これは、クロスサイトスクリプティングやキャッシュポイズニングなどのさまざまな攻撃を仕掛けるために使用される可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (行: 95)

データフロー

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

62:7	@RequestParam 文字列 ユーザー名、	ソース	0
62:7	@RequestParam 文字列 ユーザー名、		1
68:35	credentialsLoginFlow( ユーザー名、パスワード、応答 ) を返します。		2
82:7	[ 文字列ユーザー名、 ] 文字列パスワード、HttpServletResponse レスポンス){		3
83:33	文字列 lowerCasedUsername = username.toLowerCase();		4
83:33	文字列 lowerCasedUsername = username.toLowerCase();		5
83:12	文字列 lowerCasedUsername = username.toLowerCase();		6
91:45	文字列 newCookieValue = EncDec.encode( lowerCasedUsername );		7
91:31	文字列 newCookieValue = EncDec.encode( lowerCasedUsername );		8
91:14	文字列 newCookieValue = EncDec.encode(lowerCasedUsername);		9
92:50	Cookie newCookie = 新しい Cookie(COOKIE_NAME,newCookieValue);		10
92:30	Cookie newCookie = 新しい Cookie(COOKIE_NAME、newCookieValue);		11
92:14	Cookie newCookie = 新しい Cookie(COOKIE_NAME,newCookieValue);		12
95:26	レスポンス.addCookie(新しいCookie);		13
95:7	[ レスポンス.addCookie(新しいCookie);		14



## 詳細

CRLFは「キャリッジリターン」と「ラインフィード」の略称です。この2つの特殊文字は、コンピュータ黎明期に使用されていた旧式の印刷端末の名残です。しかし、今日でもどちらもデータ間の区切り文字としてよく使用されています。この脆弱性が存在する場合、CR文字とLF文字（コードではそれぞれ\rと\nで表されます）がHTTPヘッダー内に存在することが許可されます。これは通常、開発中のデータ処理計画が不十分なことが原因です。

HTTP ヘッダー内の CRLF シーケンスは、ブラウザからの応答を効果的に分割し、單一行をサーバーによって複数行として受け入れるため、「応答分割」と呼ばれます（たとえば、單一行の First Line\r\nSecond Line は、サーバーによって 2 行の入力として受け入れられます）。

レスポンス分割自体は攻撃ではなく、悪用されない限り全く無害ですが、その脆弱性はインジェクション攻撃（CRLFインジェクション）や、予測不可能で潜在的に危険な様々な動作につながる可能性があります。この脆弱性は、ページハイジャックやクロスユーザー改ざんなど、様々な方法で悪用される可能性があります。クロスユーザー改ざんとは、攻撃者が偽のサイトコンテンツを表示したり、認証情報などの機密情報を取得したりする攻撃です。さらに、クロスサイトスクリプティング攻撃（ユーザーのブラウザで悪意のあるコードを実行させる攻撃）にもつながる可能性があります。

たとえば、次のコードは脆弱です。

```
protected void doGet(HttpServletRequest リクエスト、HttpServletResponse レスポンス) { Cookie cookie = new  
Cookie("name", request.getParameter("name")); response.addCookie(cookie);  
}
```

ユーザーがXYZのような値を持つ名前パラメータを指定する可能性があるため\r\nHTTP/1.1 200 OK\r\n攻撃者が制御  
応答：

HTTP/1.1 200 OK 攻  
撃者が制御

考えられる修正方法は、英数字以外の文字をすべて削除することです。

```
保護された void doGet(HttpServletRequest リクエスト、HttpServletResponse レスポンス) {  
    文字列 name = request.getParameter("name") .replaceAll("[^a-  
zA-Z ]", "");  
    Cookie cookie = 新しい Cookie("name", name);  
    response.addCookie(cookie);  
}
```

この場合、攻撃者は 2 番目の HTTP 応答を生成できなくなります。

## 予防のためのベストプラクティス

- すべての入力は潜在的に悪意のあるものであると想定してください。可能な限り許容可能な応答を定義し、それが不可能な場合は、ヘッダーの分割を防ぐためにCR文字とLF文字をエンコードしてください。
- \r（キャリッジリターン）と\n（ラインフィード）の両方を""（空文字列）に置き換えてください。多くのプラットフォームではこれらの文字は互換的に処理されるため、どちらか一方が許可されている場合でも脆弱性が存在する可能性があります。ベストプラクティスに従い、可能な限り他のすべての特殊文字（"、/、；など、およびスペース）を削除してください。ブラウザからサーバーへの双方向、およびブラウザに返されるデータの両方で、特殊文字をサニタイズしてください。理想的には、ヘッダーへのCRおよびLFインジェクションをブロックする言語やライブラリなどの最新の開発リソースを採用してください。ユーザー側で（意図的または意図せずに）改ざんまたは変更される可能性があり、インジェクション攻撃につながる可能性のあるすべての入力タイプに注意してください。これには、GET、POST、Cookie、その他のHTTPヘッダーが含まれます。

## 保護されていない資格情報の保管

SNYK-CODE CWE-256 パスワードを返す

比較タイミングの露出により、攻撃者がパスワードの値を検出できる可能性があります。Arrays.equals() または String.equals() 関数が呼び出された場合、一致するバイト数が少ないほど、これらの関数は早期に終了します。パスワードの比較には、BCrypt などのパスワードエンコーダを使用してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java (行: 55)





## 詳細

認証情報が保護されていない場合、または強力な暗号化によって十分に保護されていない場合、攻撃者はさまざまな方法でこの情報にアクセスできます。開発者はシステムが攻撃から完全に安全である、あるいは内部関係者のみがアクセスできると信じているにもかかわらず、認証情報を平文で保存することに頼っている。この自信は誤ったものであり、危険です。悪意のある内部関係者（元従業員など）や、SQLインジェクション、XMLインジェクション、ブルートフォース攻撃などを使用する敵対的な攻撃者がシステムにアクセスした場合、この資格情報をアクセスして、システム内で不正な権限を取得したり、その他の機密情報や安全な情報をエクスポートしたりすることができます。

## 予防のためのベストプラクティス

- パスワードは、「純粋に内部」使用の場合でも、決してプレーンテキストで保存しないようにしてください。
- Base 64 エンコードなどのパスワードエンコードには絶対に依存せず、ソルト化とハッシュ化を含む複雑な暗号化アルゴリズムを選択してください。
- ユーザーが正当なビジネス目的に必要な情報のみにアクセスできるゼロトラスト アプローチを実装します。
- 可能な限り、インジェクション攻撃やその他の種類の脆弱性からアプリケーションを保護します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/passwordreset/SimpleMailAssignment.java (行: 65)



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

60:57 パブリックAttackResultログイン(@RequestParam String email, @RequestParam String password) {  
65:12 && StringUtils.reverse(ユーザー名).equals(パスワード)) {

ソース

0

シンク

1



## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリはアクセスを高速化するために、有効なセッションIDのロックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは攻撃者は、ロックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与えている場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。  
生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

パスワードの秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionLesson6b.java (行: 50)

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

```
58:12 文字列\\"スワード = "dave";  
50:9 if ( userid_6b.equals( getPassword() ) {
```

0

1



## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリはアクセスを高速化するために、有効なセッションIDのロックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは攻撃者は、ロックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

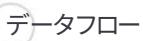
- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与える場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

getUserHashの秘密の値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。値を安全に比較するには java.security.MessageDigest.isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionACYourHash.java (行: 55)



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

```
55:25 if (userHash.equals( displayUser.getUserHash() )) {  
55:9 if ( userHash.equals( displayUser.getUserHash() )) {
```

0

1



## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリはアクセスを高速化するために、有効なセッションIDのロックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは攻撃者は、ロックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与える場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

getUserHashの秘密の値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/missingac/MissingFunctionACYourHashAdmin.java (行: 62)

## データフロー

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

```
62:25 if (userHash.equals( displayUser.getUserHash())) { } 0
62:9 if ( userHash.equals( displayUser.getUserHash())) { }
```

ソース

0

シンク

1

## ✓修正分析

## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのロックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは

攻撃者は、ロックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。

有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に

安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
  - パフォーマンスを最適化したいために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
  - アプリが攻撃者を助ける手がかりを与えていた場合、暗号化だけでは不十分であることを開発者に認識させましょう。
  - タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。
- 生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

uniqueCode の秘密値はequalsを使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/webwolfintroduction/LandingAssignment.java (行: 51)

## データフロー

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

```
50:29 パブリックAttackResultクリック(文字列ユニークコード){ } 0
51:9 if ( StringUtils.reverse(getWebSession().getUserName()).equals( uniqueCode)) { }
```

ソース

0

シンク

1

## ✓修正分析

## 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのロックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは

攻撃者は、ロックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。

有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に

安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

## 予防のためのベストプラクティス

• 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。

パフォーマンスを最適化するために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。

- アプリが攻撃者を助ける手がかりを与えていた場合、暗号化だけでは不十分であることを開発者に認識させましょう。

タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。

生産可能です。

- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## 観測可能なタイミングの不一致（タイミング攻撃）

SNYK-CODE CWE-208 タイミング攻撃

uniqueCode の秘密値は equals を使って比較されるため、攻撃者はそれを推測することができ、タイミング攻撃に対して脆弱です。

値を安全に比較するには java.security.MessageDigest isEqual を使用します。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/webwolfintroduction/MailAssignment.java (行: 86)

### データフロー

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

```
85:33 パブリックAttackResult完了(@RequestParamStringuniqueCode){}
86:9 if ( uniqueCode.equals( StringUtils.reverse(getWebSession().getUserName()) ) ) {
```

ソース

0

シンク

1

### ✓修正分析

#### 詳細

タイミング攻撃はサイドチャネル攻撃の一種であり、コードの構造を利用するのではなく、外部の手がかりを利用して攻撃者を攻撃することを意味します。

攻撃者はプログラムの状態を推測します。タイミング攻撃では、特定の操作の実行にかかる時間からプログラムの状態を推測します。例えば、アプリは

アクセスを高速化するために、有効なセッションIDのレックアップテーブルを使用します。これは認証済みのユーザーにとっては便利ですが、無効なセッションIDは拒否されるまでに非常に時間がかかります（無効なセッションIDは攻撃者は、レックアップテーブルを破壊し、ブルートフォース攻撃を仕掛ける貴重な手段を得る。攻撃者は、ランダムに生成された多数のセッションIDをテストするだけでよい。

有効なものを見つけられることを期待して。アクセスが許可されると、悪意のある攻撃者はセッションIDを通じて正当なユーザーになりすまし、アクションを実行したり、自由に

安全なデータへのアクセス。大規模なボット攻撃などのブルートフォース攻撃によって、この脆弱性が悪用され、非常に強力な暗号化アルゴリズムさえも回避される可能性があります。

### 予防のためのベストプラクティス

- 入力の有効性と返される結果に関係なくタイミングが同一であることを保証する定数時間アルゴリズムを実装します。
- パフォーマンスを最適化するために定数時間アルゴリズムが実用的でない場合は、ブラインド化などの別の手法を選択します。
- アプリが攻撃者を助ける手がかりを与えていた場合、暗号化だけでは不十分であることを開発者に認識させましょう。
- タイミング攻撃は多くの場合、本番環境に依存するため、開発中にテストすることは困難です。できるだけ本番環境に近いステージング環境を使用してください。
- 生産可能です。
- 独自の方法を実装するのではなく、サイドチャネル攻撃保護戦略を備えた、強化された信頼性の高い暗号化および認証ライブラリを選択します。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はHTMLに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。

側。これにより、DOM ベースのクロスサイト スクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/lessons/idor/js/idor.js (行: 4)

### データフロー

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

```
3:45 webgoat.customjs.idorViewProfile = 関数(データ) { }
3:45 webgoat.customjs.idorViewProfile = 関数(データ) { }
5:19 '名前:' + データ.名前 + '<br/>' +
5:24 '名前:' + データ.名前 + '<br/>' +
5:19 '名前:' + データ名 + '<br/>' +
```

ソース

0

1

2

3

4

Machine Translated by Google

```

5:9 前:' + データ名 + '<br/>' + '名
5:9 '名前:' + データ名 + '<br/>' +
      data.name + '<br/>' +
      '色 : ' + data.color + '<br/>' + '名前:' +
      '名前:' + データ名 + '<br/>' +
      '色:' + データ.色 + '<br/>' +
      '名前:' + データ名 + '<br/>' +
      '色:' + データ色 + '<br/>' +
      'サイズ : ' + データサイズ + '<br/>' +
      '名前:' + データ名 + '<br/>' +
      '色:' + データ色 + '<br/>' +
      'サイズ:' + データサイズ + '<br/>' +
      '名前:' + データ名 + '<br/>' +
      '色:' + データ色 + '<br/>' +
      'サイズ:' + データサイズ + '<br/>'

4:46 webgoat.customjs.jquery('#idor-profile').html( [ ] )

```

13 シンク

✓修正分析

## 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS)は、攻撃者が制御するデータ (例：location.search document.referrer

、場所ハッシュ、ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くのWAFは、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP)ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOMベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

## 攻撃の種類

タイプ	ソース カテゴリ 説明ナビゲーション URL	技術的な詳細
URLパラメータ 注入	クエリ内に埋め込まれた悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
フラグメント識別子 擷取	ナビゲーション URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる 悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ 攻撃	永続ストレージlocalStorageへの悪意のあるデータ注入、 sessionStorage、またはその後読み取られる IndexedDB アプリケーション JavaScriptによって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。
クロスフレーム コミュニケーション	インターフーム メッセージング	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSSの脆弱性は、さまざまなカテゴリのWebアプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブウェブアプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装

- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

## 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ データをサードパーティ、セッションストレージ、または他のクライアント側ストレージメカニズムを使用する前にサニタイズします。

## 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際には、文字列引数を持つinnerHTML、外側のHTML insertAdjacentHTML、document.write()、setTimeout() / setInterval()などのリスク、ドキュメント.writeln()、評価()、関数()、内部のHTMLの高いDOMシンクを避けてください。
- コンテンツを解釈しないtextContentやDOM要素作成APIなどのより安全なDOM操作方法を活用する、テキストノードを作成()  
実行可能コードsetAttribute()はURL属性やイベント属性以外の属性にのみ使用し、信頼できないデータからのon\*ハンドラーや危険なURL属性を設定するために使用しないでください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、データ転送用のJSONなどの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合の影響、オブジェクトソース、ベースURI、frame-ancestorsを使用してスクリプトの実行を制限し、が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティ ライブラリの改ざんを防止します。  
脆弱性。
- クリックジャッキング保護のために、CSP のframe-ancestorsディレクティブを優先するX-Content-Type-Options:、リファラーポリシー、および (CSPが利用できない場合) X-Frame-Options、その間nosniffを含むHTTPセキュリティヘッダーを活用します。
- 承認されたサニタイザーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。  
開発プロセス。
- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。  
発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、  
実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。  
セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はHTMLに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。

側。これにより、DOMベースのクロスサイトスクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/lessons/lessontemplate/js/idor.js (行: 4)

データフロー

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

3:45 webgoat.customjs.idorViewProfile = 関数(データ) {

[ ]

ソース

0

3:45 webgoat.customjs.idorViewProfile = 関数(データ) {

[ ]

1

5:19 Machine Translated by Google

```

5:24 '名前:' + データ名 + '<br/>' +
5:19 '名前:' + データ名 + '<br/>' +
5:9 前: + データ名 + '<br/>' + '名
5:9 '名前:' + データ名 + '<br/>' +
5:9 data.name + '<br/>' + data.color + '<br/>' + '名前:' +
      '色 : '
5:9 '名前:' + データ名 + '<br/>' + '色:' + データ.色 + '<br/>' +
5:9 '名前:' + データ名 + '<br/>' + '色:' + データ色 + '<br/>' +
5:9 '名前:' + データ名 + '<br/>' + '色:' + データ色 + '<br/>' +
      'サイズ : '
5:9 '名前:' + データ名 + '<br/>' + '色:' + データ色 + '<br/>' +
      'サイズ:' + データサイズ + '<br/>' +
5:9 '名前:' + データ名 + '<br/>' + '色:' + データ色 + '<br/>' +
      'サイズ:' + データサイズ + '<br/>' +
4:46 webgoat.customjs.jquery('#idor-profile').html( [ ] )

```

2

3

4

5

6

7

8

9

10

11

12

13

シンク

## 修正分析

### 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS)は、攻撃者が制御するデータ (例：location.search document.referrer) によって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来のサーバー側の防御や多くのWAFは、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用するAPI。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。DOMベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、ブラウザ環境内でユーザーを直接ターゲットにします。

### 攻撃の種類

タイプ	ソース カテゴリ 説明ナビゲーション URL	技術的な詳細
URLパラメータ 注入	クエリ内に埋め込まれた悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
フラグメント識別子 操作	ナビゲーション location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる 悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ 攻撃	永続ストレージlocalStorageへの悪意のあるデータ注入、 sessionStorage、またはその後読み取られる IndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な搾取のために他のベクトルと組み合わせます。
クロスフレーム コミュニケーション	インターフーム メッセージング	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

### 影響を受ける環境

- React, Angular, Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

## 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ データをサードパーティのセッションストレージ、または他のクライアント側ストレージメカニズムを使用する前に  
サニタイズします。

## 安全なDOM操作の実践

- 一が制御可能なデータを処理する際に、文字列引数を持つinnerHTML、外側のHTML、insertAdjacentHTMLユーザ、ドキュメント.write()、ドキュメント.writeln()、評価()、関数()、setTimeout() / setInterval()などのリスクの高い DOM シンクを回避します。
- textContent 実行可能コードなど、より安全なDOM操作方法を使用してください。setAttribute() はテキストノードを作成()、DOM要素作成APIはコンテンツを解釈しない URL/イベント属性以外の属性にのみ使用し、信頼できないデータからon\*ハンドラーや危険なURL属性を設定することは避けてください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、  
データ転送用の JSON などの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- 包括的なコンテンツ セキュリティ ポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合の影響、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、  
が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティ ライブラリの改ざんを防止します。  
脆弱性。
- クリックジャッキング保護のために、CSP の frame-ancestors ディレクティブを優先する X-Content-Type-Options: リファラーポリシー、および (CSP が利用できない場合) X-Frame-Options、その間 nosniff を含む HTTP セキュリティ ヘッダーを活用します。
- 承認されたサニタイザーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジン データの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的な DOM XSS 脆弱性を特定します。  
開発プロセス。
- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。  
発生源から吸収源へ。
- DOM XSS 検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、  
実装されたセキュリティ制御の有効性。
- DOM XSS 攻撃ベクトル、安全な JavaScript プログラミング プラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。  
セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYK コード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力は HTML に流れ込み、クライアント上で HTML ページを動的に構築するために使用されます。

側。これにより、DOM ベースのクロスサイトスクリプティング攻撃 (DOM XSS) が発生する可能性があります。

見つかった場所: src/main/resources/webgoat/static/js/goatApp/support/GoatUtils.js (行: 57)

```
56:69    $.get(goatConstants.cookieService, {}, 関数(返信) {  
56:69    $.get(goatConstants.cookieService, {}, function( reply) { $  
57:51    ("#lesson_cookies").html( reply);  
57:46    $("#lesson_cookies").html(返信);
```

ソース	0
	1
	2

✓修正分析

詳細

DOMベースのクロスサイトスクリプティング (DOM XSS) は、攻撃者が制御するデータ（例：location.search.document.referrer）を含む JavaScript を実行する脆弱性。

、`ブラウザストレージ`、`postMessage`、`WebSocket`などのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSSinkに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くの WAF は、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM / JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する。

API-さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー（CSP）ディレクティブやその他のプラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

#### DOM ベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな要意のある目的を達成するために頻繁に悪用されています。

クッキーの次難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のブリュート

配布・機密データの窃取など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ（防御）を回避して

ゴミ回収環境にてユーチューバーを直接名一矢ツッコミます。

## 攻撃の種類

タイプ	ソースカテゴリの説明		技術的な詳細
URL/パラメータ 注入	ナビゲーション	URLクエリ内に埋め込まれた悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
フラグメント識別子 擷取	ナビゲーション	URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作		リクエストコンテキストdocument.referrer値に含まれる 悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ 攻撃		永続ストレージローカルストレージセッションストレージへの悪意のあるデータ注入、 、またはその後読み取られるIndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。
クロスフレーム コミュニケーション	インターフレーム メッセージング	悪意のあるデータが投稿されたpostMessage APIの悪用 フレームまたはウィンドウ間で送信され、処理される 適切なオリジン検証と入力サニタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	WebSocket経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React, Angular, Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
  - 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
  - ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
  - コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
  - WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
  - 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
  - クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
  - クライアント側コンテンツの埋め込みと其の機能を実装するレシピライズ (プロジェクトコード) の実装

圣母のためのバフト・プラクティフ

このセクションでは、完全な開発プロトコルと技術的制御を通じて DOM XSS の脆弱性を防ぐ上に設計された包括的なセキュリティ対策について説明します。

## 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
  - HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。エスケープ、CSS 値のエンコード、および URI パラメータのエンコード。

信頼できないソースからの悪意のあるデータの挿入を防ぎます。

- 永続ストレージデータをサニタイズするには、localStorage、sessionStorage、その他のクライアント側ストレージメカニズムから取得したすべてのデータを検証してから、アプリケーション内での処理またはレンダリング。

## 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際には、文字列引数を持つ、外側のHTML、insertAdjacentHTML、document.write()、document.writeln()、eval()、Function()、innerHTML setTimeout() / setInterval()などのリスクの高いDOMシンクを避けてください。
- textContentやDOM要素作成APIなど、コンテンツを解釈しないより安全なDOM操作方法を活用してTextNode()、実行可能コード。setAttribute()はURL属性やイベント属性以外の属性にのみ使用し、信頼できないデータからのon\*ハンドラーや危険なURL属性を設定するために使用しないでください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、データ転送用のJSONなどの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー(CSP)ディレクティブ(例: script-src、object-src)を展開し、DOM XSSが存在する場所(ベースURI、frame-ancestors)を使用してスクリプトの実行を制限し、合の影響を軽減します。
- すべての外部JavaScriptリソースにサブリソース整合性(SRI)を実装し、DOM XSSを引き起こす可能性のあるサードパーティライブラリの改ざんを防止します。脆弱性。
- X-Content-Type-Options: nosniff、Referrer-Policy、(CSPが利用できない場合) X-Frame-OptionsなどのHTTPセキュリティヘッダーを活用し、クリックジャッキングの保護にはCSPのframe-ancestorsディレクティブを優先します。
- 承認されたサニタイザーを通過しない限りDOMシンクへの割り当てを防止するために、Trusted Types(サポートされている場合)などの最新のブラウザ防御を採用します。CORSをDOM XSS制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。開発プロセス。
- JavaScriptコードのDOM XSS脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト(DAST)を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング(XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はinnerHTMLに流れ込み、そこでHTMLページを動的に構築するために使用されます。

クライアント側DOMベースのクロスサイトスクリプティング攻撃(DOMXSS)が発生する可能性があります。

見つかった場所: src/main/resources/lessons/clientSideFiltering/js/clientSideFiltering.js (行: 38)

### データフロー

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

```
17時70分 $.get("clientSideFiltering/salaries?userId=" + userId, function (result, status) { [ ] 0
17時70分 $.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
28:20 html = html + '<td>' + result[i].UserID + '</td>'; [ ] 13
28:13 html = html + '<td>' + result[i].UserID + '</td>'; [ ]
```

Machine Translated by Google

```

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

```

シンク41

## 修正分析

### 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS)は、攻撃者が制御するデータ (例 : location.search document.referrer ) が場所ハッシュ

、ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くのWAFは、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOMベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

## 攻撃の種類

タイプ	ソース カテゴリ 説明ナビゲーション URL	技術的な詳細
URLパラメータ	クエリ内に埋め込まれた悪意のあるペイロード	一般的にURLSearchParamsの直接文字列解析を悪用する。
注入	脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
フラグメント識別子	ナビゲーション	特にシングルページアプリケーション (SPA)では、
操作	location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー	リクエストコンテキストdocument.referrer値に含まれる	ユーザーを誘導するためにソーシャルエンジニアリングが必要
操作	悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ	永続ストレージlocalStorageへの悪意のあるデータ注入、 攻撃	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、
	sessionStorage、またはその後読み取られるIndexedDB	

クロスフレーム	インターフレーム	悪意のあるデータが投稿されたpostMessage APIの悪用	段階的な擷取のために他のベクトルと組み合せます。
コミュニケーション	メッセージング	フレームまたはウィンドウ間で送信され、処理される 適切なオリジン検証と入力サニタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウメカニズム。
ウェブソケット	リアルタイム	WebSocket経由で配信される悪意のあるペイロード	チャットなどのリアルタイムアプリケーションに特に効果的
メッセージインジェクション	コミュニケーション	クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブウェブアプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツスクリプトの挿入とWebページインタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS値のエンコード、およびURLパラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージデータをセッションストレージ、または他のクライアント側ストレージメカニズムを使用する前に  
ニタイズします。

### 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際には、文字列引数を持つinnerHTML、outerHTML、document.write()、setTimeout() / setInterval()などのリスクドキュメント.writeln()、評価()、関数()、  
の高いDOMシンクを避けてください。
- コンテンツを解釈しないtextContentやDOM要素作成APIなどのより安全なDOM操作方法を活用するテキストノードを作成()  
実行可能コード。setAttribute()はURL属性やイベント属性以外の属性にのみ使用し、信頼できないデータからのon\*ハンドラーや危険なURL属性を設定するために使用しないでください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、  
データ転送用のJSONなどの構造化データ形式。

### ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合の影響、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、  
が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティライブラリの改ざんを防止します。  
脆弱性。
- クリックジャッギング保護のために、CSP のframe-ancestorsディレクティブを優先するX-Content-Type-Options:、リファラーポリシー、および (CSPが利用できない場合) X-Frame-Options  
nosniffを含むHTTPセキュリティヘッダーを活用します。
- 承認されたサニタイザーを通してない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

### 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。  
開発プロセス。
- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。  
発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、  
実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。  
セキュリティに重点を置いたライブラリとフレームワークの使用。

リモートリソースからのデータからのサニタイズされていない入力はappendに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。

側。これにより、DOM ベースのクロスサイト スクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/lessons/csrf/js/csrf-review.js (行: 41)

## データフロー

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

```

35:40     $.get('csrf/review', 関数 ( 結果, ステータス){[ ]})
35:40     $.get('csrf/review', 関数 ( 結果, ステータス){[ ]})
40:52     コメント = comment.replace('STARS', result[i].stars)
40:52     コメント = comment.replace('STARS', result[i].stars)
40:62     コメント = comment.replace('STARS', result[i].stars)
40:52     コメント = comment.replace('STARS', result[i].stars)
40:35     comment = comment.replace('STARS', result[i].stars)
40:17     [コメント = comment.replace('STARS', result[i].stars)]
41:35     $("#list").append(コメント);
41:28     $("#list"). append(コメント);

```

ソース

0

1

2

3

4

5

6

7

8

9

シンク

## 修正分析

### 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS) は、攻撃者が制御するデータ (例: location.search document.referrer

、場所.ハッシュ

、ブラウザストレージ、 postMessage 、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くの WAF は、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOM ベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

## 攻撃の種類

タイプ	ソース カテゴリ 説明URL クエリ内に埋め	技術的な詳細
URL/パラメータ 注入	ナビゲーション 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState )。
フラグメント識別子 擷取	ナビゲーション URLハッシュフラグメント (#)を利用して攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる 悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ 攻撃	永続ストレージlocalStorageへの悪意のあるデータ注入、 後で読み取られるsessionStorageまたはIndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。
クロスフレーム コミュニケーション	インターフーム メッセージング	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

- React, Angular, Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

## 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ データをサードパーティのセッションストレージ、または他のクライアント側ストレージメカニズムを使用する前にサニタイズします。

## 安全なDOM操作の実践

- 一が制御可能なデータを処理する際に、文字列引数を持つinnerHTML、外側のHTML、insertAdjacentHTMLユーザ、ドキュメント.write()、ドキュメント.writeln()、評価()、関数()、setTimeout() / setInterval()などのリスクの高いDOMシンクを回避します。
- textContent実行可能コードなど、より安全なDOM操作方法を使用してください。setAttribute()はテキストノードを作成()、DOM要素作成APIはコンテンツを解釈しないURL/イベント属性以外の属性にのみ使用し、信頼できないデータからon\*ハンドラーや危険なURL属性を設定することは避けてください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、データ転送用のJSONなどの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合の影響、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティライブラリの改ざんを防止します。  
脆弱性。
- クリックジャッキング保護のために、CSP の frame-ancestors ディレクティブを優先する X-Content-Type-Options:、リファラーポリシー、および (CSP が利用できない場合) X-Frame-Options、その間 nosniff を含む HTTP セキュリティヘッダーを活用します。
- 承認されたサニタイザーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。  
開発プロセス。
- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。  
発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、  
実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。  
セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はappend()に流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。  
側。これにより、DOMベースのクロスサイトスクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/lessons/jwt/js/jwt-final.js (行: 6)

5:23	}).then(関数 ( 結果 ) {	<input type="text"/>	ソース	0
5:23	}).then(関数 ( 結果 ) {	<input type="text"/>		1
6時28分	\$("#toast").append(結果);	<input type="text"/>		2
6時21分	\$("#toast"). append( 結果 );	<input type="text"/>	シンク	3

✓修正分析

## 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS) は、攻撃者が制御するデータ (例: location.search document.referrer) を場所.ハッシュに書き込まれる。

- ・ ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来のサーバー側の防御や多くのWAFは、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOMベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

## 攻撃の種類

タイプ	ソースカテゴリの説明	技術的な詳細	
URLパラメータ 注入	ナビゲーション 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。	
フラグメント識別子 擷取	ナビゲーション URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA) では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。	
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる 悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。	
ブラウザストレージ 攻撃	永続ストレージローカルストレージセッションストレージへの悪意のあるデータ注入 、 またはその後読み取られるIndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。	
クロスフレーム コミュニケーション	インターフーム メッセージング	悪意のあるデータが投稿されたpostMessage APIの悪用 フレームまたはウインドウ間で送信され、処理される 適切なオリジン検証と入力サニタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウインドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	WebSocket経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSSの脆弱性は、さまざまなカテゴリのWebアプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- ・ React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- ・ 広範なクライアント側機能とオフライン機能を備えたプログレッシブウェブアプリケーション (PWA)
- ・ ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- ・ コンテンツスクリプトの挿入とWebページインタラクション機能を備えたブラウザ拡張機能とアドオン
- ・ WebViewコンポーネントとJavaScriptプリミティブ実装を活用したハイブリッドモバイルアプリケーション
- ・ 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- ・ クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- ・ クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じてDOM XSSの脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- ・ URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- ・ HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS値のエンコード、およびURLパラメータのエンコード。

- 永続ストレージデータをサニタイズするには、localStorage、sessionStorage、その他のクライアント側ストレージメカニズムから取得したすべてのデータを検証してから、アプリケーション内での処理またはレンダリング。

## 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際には、文字列引数を持つ、外側のHTML、insertAdjacentHTML、document.write()、document.writeln()、eval()、Function()、innerHTML setTimeout() / setInterval()などのリスクの高いDOMシンクを避けてください。
- textContentやDOM要素作成APIなど、コンテンツを解釈しないより安全なDOM操作方法を活用するextNode()、実行可能コード。setAttribute()はURL属性やイベント属性以外の属性にのみ使用し、信頼できないデータからのon\*ハンドラーや危険なURL属性を設定するために使用しないでください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、データ転送用のJSONなどの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー(CSP)ディレクティブ(例: script-src、object-src)を展開し、DOM XSSが存在する場合、ベースURI、frame-ancestorsを使用してスクリプトの実行を制限し、影響を軽減します。
- すべての外部JavaScriptリソースにサブリソース整合性(SRI)を実装し、DOM XSSを引き起こす可能性のあるサードパーティライブラリの改ざんを防止します。脆弱性。
- X-Content-Type-Options: nosniff、Referrer-Policy、(CSPが利用できない場合) X-Frame-OptionsなどのHTTPセキュリティヘッダーを活用し、クリックジャッキングの保護にはCSPのframe-ancestorsディレクティブを優先します。
- 承認されたサニタイザーを通過しない限りDOMシンクへの割り当てを防止するために、Trusted Types(サポートされている場合)などの最新のブラウザ防御を採用します。CORSをDOM XSS制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。開発プロセス。
- JavaScriptコードのDOM XSS脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト(DAST)を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング(XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はappend()に流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。

側。これにより、DOMベースのクロスサイトスクリプティング攻撃(DOMXSS)が発生する可能性があります。

見つかった場所: src/main/resources/lessons/jwt/js/jwt-voting.js (行: 63)

### データフロー

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

```
43:36 $.get("JWT/votings", 関数(結果, ステータス){ }  
43:36 $.get("JWT/votings", 関数(結果, ステータス){ }  
56:60 投票テンプレート = 投票テンプレート.replace('AVERAGE', result[i].average || "");  
56:60 投票テンプレート = 投票テンプレート.replace('AVERAGE', result[i].average || "");  
56:70 voteTemplate = voteTemplate.replace('AVERAGE', result[i].average || ""); voteTemplate =  
56:60 voteTemplate.replace('AVERAGE', result[i].average || ""); voteTemplate =  
56:41 voteTemplate.replace('AVERAGE', result[i].average || "");  
56:13 [ 投票テンプレート = 投票テンプレート.replace('AVERAGE', result[i].average || ""); ]  
59:28 投票テンプレート = 投票テンプレート.replace(/HIDDEN_VIEW_VOTES/g, 非表示); 投票テンプレート  
59:41 レート = 投票テンプレート.replace(/HIDDEN_VIEW_VOTES/g, 非表示);  
59:13 [ 投票テンプレート = 投票テンプレート.replace(/HIDDEN_VIEW_VOTES/g, 非表示); ]  
61:28 投票テンプレート = voteTemplate.replace(/HIDDEN_VIEW_RATING/g, 非表示); 投票テンプレート  
61:41 ↵ = voteTemplate.replace(/HIDDEN_VIEW_RATING/g, 非表示);  
61:13 [ 投票テンプレート = 投票テンプレート.replace(/HIDDEN_VIEW_RATING/g, 非表示); ]
```

63:29 ("#votesList").append(投票テンプレート);

シンク

15

## ✓修正分析

### 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS) は、攻撃者が制御するデータ (例: location.search document.referrer 、場所.ハッシュ ) 、 ブラウザストレージ、 postMessage 、 WebSocket などの外部の通信が JavaScript によって読み取られ、適切な検証やエンコードが行われずに危険な DOM/JS シンクに書き込まれるため、コード全体がブラウザ内で実行されます。ペイロードはサーバーまたは他のチャネルから配信される可能性がありますが、反射型 XSS や保存型 XSS とは異なり、サーバーは実行をトリガーするインジェクション / エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来のサーバー側防御や多くの WAF では見落とされがちです。

DOM XSS 脆弱性を効果的に軽減するには、クライアント側の実行コンテキストに特化した包括的な入力検証と出力エンコーディング戦略を実装する必要があります。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、実行される特定の DOM 操作に基づいたコンテキスト認識型出力エンコーディングの実装、危険な DOM/JavaScript API を通じて露出される攻撃対象領域を最小限に抑える安全なコーディング プラクティスの採用が含まれます。さらに、最新の Web アプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、DOM ベースの攻撃に対する多層防御を確立する必要があります。

DOM XSS 攻撃はセキュリティ研究において広く文書化されており、Cookie 窃取によるセッションハイジャック、フィッシング攻撃による認証情報の窃取、認証済みユーザーを装った不正なアクションの実行、クライアント側でのマルウェア拡散、機密データの窃取など、様々な悪意ある目的を達成するために頻繁に悪用されています。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、ブラウザ環境内でユーザーを直接標的とするのに効果的です。

### 攻撃の種類

タイプ	ソース カテゴリ 説明ナビゲーション	技術的な詳細	
URLパラメータ 注入	location.search または 同様のプロパティにアクセスする脆弱な JavaScript コードによって 処理される URL クエリ パラメータ内に埋め込まれた悪意のあるペイロード。	一般的に、 URLSearchParams の直接文字列解析、またはクエリ パラメータを安全に処理しないフレームワーク ルーティング メカニズム ( pushState など の History API 更新を含む) が悪用されます。	
フラグメント識別子 擷取	ナビゲーション location.hash を介してアクセスでき、クライアント側のルーティングまたはコントンツ読み込みメカニズムによって処理される URL ハッシュ フラグメント (#) を利用する攻撃ベクトル。	ハッシュベースのナビゲーションを処理するクライアント側ルーティング フレームワークを備えたシングル ページ アプリケーション (SPA) で特に見られます。	
HTTPリファラー 操作	悪意のあるペイロードを含む document.referrer 値のリクエスト コンテキストの悪用。通常は、脆弱なアプリケーションにリンクする攻撃者が管理する Web サイトから発信されます。	細工されたリファラー値を使用して、攻撃者が管理するドメインからユーザーを誘導するには、ソーシャル エンジニアリングが必要です。	
ブラウザストレージ 攻撃	永続ストレージ localStorage 、 sessionStorage 、または IndexedDB への悪意のある データ挿入。その後、アプリケーションの JavaScript によって安全に読み取られず、処理されません。	攻撃者が (別のバグやソーシャル エンジニアリングを介して) データを保存できる場合、セッションをまたいで存続する可能性があり、段階的な悪用のために他のベクトルと組み合わせられる可能性があります。	
クロスフレーム コミュニケーション	インターフーム メッセージング	postMessage API を悪用し、悪意のあるデータがフレームまたはウィンドウ間で 送信され、適切なオリジン検証と入力サニタイズなしで処理されます。	クロスオリジン通信メカニズムを実装する埋め込み iframe またはポップアップ ウィンドウを持つアプリケーションでよく使用されます。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	適切な入力検証を行わずにクライアント側のメッセージ ハンドラー によって処理される、WebSocket 接続を介して配信される悪意のあるペイロード。	チャット システム、共同作業ツール、ライブ データ フィードなどのリアルタイム アプリケーションに対して特に効果的です。

### 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.js などのフレームワークやカスタム JavaScript 実装を活用したシングルページアプリケーション (SPA) 、広範なクライアント側機能とオフライン機能を備えた PWA
- 機能を備えたプログレッシブ Web アプリケーション (PWA) 、ハッシュベースおよび HTML5 History API ルーティング メカニズムを含むクライアント側実装、コンテンツスクリプトインジェクションと Web ページインタラクション機能を備えたブラウザ拡張機能とアドオン
- WebView コンポーネントと JavaScript ブリッジ 実装を活用したハイブリッドモバイルアプリケーション、複雑なクライアント側ビジュアル
- ネスロジックと DOM 操作を備えたリッチインターネット アプリケーション (RIA) 、クライアント側編集インターフェースと動的コンテンツ
- レンダリングを備えたコンテンツ管理システム (CMS) 、クライアント側コンテンツの埋め込みと共有機能を実装したソーシャルメディア
- プラットフォームと ウィジェット

### 予防のためのベストプラクティス

このセクションでは、安全な開発 プラクティス と技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- ブラックリスト方式ではなく、ホワイトリスト方式を使用して、URL パラメータ、フラグメント識別子、リファラー値、ストレージ メカニズム、フレーム間通信など、すべてのクライアント側データ ソースに対して厳密な入力検証を実装します。
- HTML エンティティ エンコーディング、JavaScript 文字列エスケープ、CSS 値エンコーディング、URL パラメータ エンコーディングなど、データが利用される特定の DOM コンテキストに適したコンテキスト出力エンコーディングを確立します。
- 信頼できないソースからの悪意のあるデータ挿入を防ぐために、厳密な発信元チェックとメッセージ形式の検証を実装して、フレーム間の postMessage 通信のデータの発信元と整合性を検証します。
- アプリケーション内の localStorage 处理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ 、セッションストレージ 、または他のクライアント側ストレージ メカニズムを使用する前にデータをサニタイズします。

- ユーザーが制御可能なデータを処理する際には、文字列引数を持つ外側のHTML、insertAdjacentHTML、document.write()、document.writeln()、eval()、Function()、innerHTML setTimeout() / setInterval()などのリスクの高いDOMシンクを避けてください。
- textContentやDOM要素作成APIなど、コンテンツを解釈しないより安全なDOM操作方法を活用する。TextNodes()、実行可能コードsetAttribute()はURL属性やイベント属性以外の属性にのみ使用し、信頼できないデータからのon\*ハンドラーや危険なURL属性を設定するために使用しないでください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、データ転送用のJSONなどの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー(CSP)ディレクティブ(例: script-src、object-src)を展開し、DOM XSSが存在する場合のベースURI、frame-ancestorsを使用してスクリプトの実行を制限し、影響を軽減します。
- すべての外部JavaScriptリソースにサブリソース整合性(SRI)を実装し、DOM XSSを引き起こす可能性のあるサードパーティライブラリの改ざんを防止します。脆弱性。
- X-Content-Type-Options: nosniff、Referrer-Policy、(CSPが利用できない場合) X-Frame-OptionsなどのHTTPセキュリティヘッダーを活用し、クリックジャッキングの保護にはCSPのframe-ancestorsディレクティブを優先します。
- 承認されたサニタイザーを通過しない限りDOMシンクへの割り当てを防止するために、Trusted Types(サポートされている場合)などの最新のブラウザ防御を採用します。CORSをDOM XSS制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。開発プロセス。
- JavaScriptコードのDOM XSS脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト(DAST)を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング(XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はappendに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。側。これにより、DOMベースのクロスサイトスクリプティング攻撃(DOM XSS)が発生する可能性があります。

見つかった場所: src/main/resources/lessons/sqlinjection/js/assignment13.js (行: 57)

### データフロー

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

	ソース	0
43:73    \$.get("SqlInjectionMitigations/servers?column=" + column, function (result, status) {		
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); serve[=		4
56:52    server.replace('DESCRIPTION', result[i].description); server =		5
56:29    server.replace('DESCRIPTION', result[i].description);		6
56:13    [server = server.replace('説明', result[i].説明);]		7
57:34    \$("#servers").append(サーバー[=]);		8
57:27    \$("#servers").append([サーバー]);		9

### ✓修正分析

#### 詳細

DOMベースのクロスサイトスクリプティング(DOM XSS)は、攻撃者が制御するデータ(例: location.search document.referrer

、場所.ハッシュ)

、ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くのWAFは、しばしば失敗します。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOMベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

## 攻撃の種類

タイプ	ソース カテゴリ 説明 URL クエリ内に埋め込	技術的な詳細	
URLパラメータ 注入	ナビゲーション 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState()。	
フラグメント識別子 擷取	ナビゲーション URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA) では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。	
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる 悪意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。	
ブラウザストレージ 攻撃	永続ストレージローカルストレージへの悪意のあるデータ注入 sessionStorage、またはその後読み取られる IndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。	
クロスフレーム コミュニケーション	インターフーム メッセージング	悪意のあるデータが投稿されたpostMessage APIの悪用 フレームまたはウインドウ間で送信され、処理される 適切なオリジン検証と入力サニタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	WebSocket経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ、セッションストレージ、または他のクライアント側ストレージメカニズムを使用する前に  
データをサニタイズします。

### 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際に、文字列引数を持つ、外側のHTML、insertAdjacentHTMLユー、ドキュメント.write()、ドキュメント.writeln()、評価()、関数()、innerHTML setTimeout() / setInterval()などのリスクの高いDOMシンクを回避します。
- textContent実行可能コードなど、より安全なDOM操作方法を使用してください。、テキストノードを作成()、DOM要素作成APIはコンテンツを解釈しないsetAttribute()はURL/イベント属性以外の属性にのみ使用し、信頼できないデータからon\*ハンドラーや危険なURL属性を設定することは避けてください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、  
データ転送用のJSONなどの構造化データ形式。

## ブラウザのセキュリティ機能の実装

- すべての外部JavaScriptリソースにサブリソース整合性(SRI)を実装し、DOM XSSを引き起こす可能性のあるサードパーティライブラリの改ざんを防止します。脆弱性。
- X-Content-Type-Options: nosniff、Referrer-Policy、(CSPが利用できない場合) X-Frame-OptionsなどのHTTPセキュリティヘッダーを活用し、クリックジャッキングの保護にはCSPのframe-ancestorsディレクティブを優先します。
- 承認されたサニタイザーを通過しない限りDOMシンクへの割り当てを防止するために、Trusted Types(サポートされている場合)などの最新のブラウザ防御を採用します。CORSをDOM XSS制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジンデータの公開を制限するように適切に構成します。

## 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。開発プロセス。
- JavaScriptコードのDOM XSS脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト(DAST)を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング(XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はappendに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。側。これにより、DOMベースのクロスサイトスクリプティング攻撃(DOMXSS)が発生する可能性があります。

見つかった場所: src/main/resources/lessons/xss/js/stored-xss.js (行: 40)

### データフロー

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

35:58	\$.get('CrossSiteScripting/stored-xss', 関数(結果, ステータス) {	[ ]	ソース	0
35:58	\$.get('CrossSiteScripting/stored-xss', 関数(結果, ステータス) {	[ ]		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	コメント = comment.replace('COMMENT', result[i].text);	[ ]		5
39:35	comment = comment.replace('COMMENT', result[i].text);	[ ]		6
39:17	コメント = comment.replace('COMMENT', result[i].text);	[ ]		7
40:35	\$("#list").append(コメント);	[ ]		8
40:28	\$("#list").append(コメント);	[ ]	シンク	9

### ✓修正分析

### 詳細

DOMベースのクロスサイトスクリプティング(DOM XSS)は、攻撃者が制御するデータ(例: location.search document.referrer

、場所ハッシュ)

、ブラウザストレージ、postMessage、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来のサーバー側の防御や多くのWAFは、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー(CSP)ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOMベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

### 攻撃の種類

URLパラメータ 注入	ナビゲーション	された悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState)。
フラグメント識別子 擷取	ナビゲーション	URLハッシュフラグメント (#)を利用した攻撃ベクトルは、 location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA)では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる	恶意のあるペイロードは、通常、 攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている 応用。	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。
ブラウザストレージ 攻撃	永続ストレージローカルストレージへの悪意のあるデータ注入 sessionStorage、またはその後読み取られる IndexedDB アプリケーション JavaScript によって安全に処理されない可能性があります。		攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な擷取のために他のベクトルと組み合わせます。
クロスフレーム コミュニケーション	インターフーム メッセージング	悪意のあるデータが投稿されたpostMessage APIの悪用 フレームまたはウィンドウ間で送信され、処理される 適切なオリジン検証と入力サンタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウィンドウ メカニズム。
ウェブソケット メッセージインジェクション	リアルタイム コミュニケーション	WebSocket経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React、Angular、Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサンタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ ディレクトリ、セッションストレージ、または他のクライアント側ストレージメカニズムを使用する前に  
データをサンタイズします。

### 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際に、文字列引数を持つinnerHTML、外側のHTML、insertAdjacentHTMLユーティリティ、ドキュメント.write()、ドキュメント.writeln()、評価()、関数()、setTimeout() / setInterval()などのリスクの高いDOMシンクを回避します。
- textContent実行可能コードなど、より安全なDOM操作方法を使用してください。setAttribute、テキストノードを作成()、DOM要素作成APIはコンテンツを解釈しない()  
()はURL/イベント属性以外の属性にのみ使用し、信頼できないデータからon\*ハンドラーや危険なURL属性を設定することは避けてください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、  
データ転送用の JSON などの構造化データ形式。

### ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、  
合の影響が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティ ライブリリースの改ざんを防止します。
- クリックジャッキング保護のために、CSP のframe-ancestorsディレクティブを優先するX-Content-Type-Options、リファラーポリシー、および (CSPが利用できない場合) X-Frame-Options、その間  
Options: nosniffを含むHTTPセキュリティヘッダーを活用します。
- 承認されたサンタイマーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジン データの公開を制限するように適切に構成します。

### 開発とテストの実践

- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。セキュリティに重点を置いたライブラリとフレームワークの使用。

## DOMベースのクロスサイトスクリプティング (XSS)

SNYKコード CWE-79 DOMXSS

リモートリソースからのデータからのサニタイズされていない入力はappendに流れ込み、クライアント上でHTMLページを動的に構築するために使用されます。

側。これにより、DOM ベースのクロスサイト スクリプティング攻撃 (DOMXSS) が発生する可能性があります。

見つかった場所: src/main/resources/lessons/xxe/js/xxe.js (行: 78)

### データフロー

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

```

72:37   $.get("xxe/comments", 関数(結果, ステータス) {
72:37     $.get("xxe/comments", function ( result, status) { comment =
77:50       comment.replace('COMMENT', result[ i].text); comment =
77:50       comment.replace('COMMENT', result[i].text); comment =
77:60       comment.replace('COMMENT', result[i].text); comment = [ ]
77:50       comment.replace('COMMENT', result[i].text); comment =
77:31       comment.replace('COMMENT', result[i].text);
77:13     [コメント = comment.replace('COMMENT', result[i].text); ]
78:29     $(フィールド).append([コメント]); $(オ
78:22     イールド).append([コメント]);

```

ソース

0

1

1

2

2

3

3

4

4

5

5

6

6

7

7

8

8

9

9

シンク

9

### 修正分析

### 詳細

DOMベースのクロスサイトスクリプティング (DOM XSS) は、攻撃者が制御するデータ (例: location.search document.referrer

場所.ハッシュ

、 ブラウザストレージ、 postMessage 、WebSocketなどのデータがJavaScriptによって読み取られ、適切な検証や検証なしに危険なDOM/JSシンクに書き込まれる。

エンコードにより、コード全体がブラウザ内で実行される。ペイロードはサーバーや他のチャネルから配信される可能性があるが、反射型や保存型XSSとは異なり、

サーバーは実行をトリガーするインジェクション/エコーを実行しません。重要な問題は、クライアント側ロジック内のソースからシンクへの汚染されたデータフローであり、従来の

サーバー側の防御や多くの WAF は、しばしば失敗します。

DOM XSS脆弱性を効果的に軽減するには、特に設計された包括的な入力検証と出力エンコード戦略を実装する必要があります。

クライアント側の実行コンテキスト。これには、ユーザーが制御可能なデータに対する厳格なホワイトリストメカニズムの確立、コンテキスト認識出力エンコーディングの実装が含まれます。

実行されている特定のDOM操作に基づいて、危険なDOM/JavaScriptを通じて露出される攻撃面を最小限に抑える安全なコーディングプラクティスを採用する

API。さらに、最新のウェブアプリケーションは、コンテンツセキュリティポリシー (CSP) ディレクティブやその他のブラウザセキュリティ機能を活用して、多層防御を確立する必要があります。

DOM ベースの攻撃に対する保護。

DOM XSS攻撃はセキュリティ研究で広く文書化されており、セッションの乗っ取りなど、さまざまな悪意のある目的を達成するために頻繁に悪用されています。

クッキーの盗難によるハイジャック、フィッシング攻撃による認証情報の収集、認証されたユーザーに代わって実行される不正なアクション、クライアント側のマルウェア

配布、機密データの窃盗など、様々な攻撃が存在します。これらの攻撃はクライアント側で行われるため、従来の境界セキュリティ制御を回避し、

ブラウザ環境内でユーザーを直接ターゲットにします。

### 攻撃の種類

タイプ	ソース カテゴリ 説明ナビゲーション URL	技術的な詳細
URLパラメータ 注入	クエリ内に埋め込まれた悪意のあるペイロード 脆弱なJavaScriptによって処理されるパラメータ location.searchまたは同様のプロパティにアクセスするコード。	一般的にURLSearchParamsの直接文字列解析を悪用する。 またはフレームワークルーティングメカニズムが安全でない処理を行っている クエリパラメータ (History APIの更新を含む) pushState )。
フラグメント識別子 操作	ナビゲーション location.hashを通じてアクセス可能であり、処理される クライアント側のルーティングまたはコンテンツ読み込みメカニズム。	特にシングルページアプリケーション (SPA) では、 ハッシュベースのルーティングを処理するクライアント側ルーティングフレームワーク ナビゲーション。
HTTPリファラー 操作	リクエストコンテキストdocument.referrer値に含まれる 悪意のあるペイロードは、通常、	ユーザーを誘導するためにソーシャルエンジニアリングが必要 細工されたリファラー値を持つ、攻撃者が管理するドメイン。

攻撃者が管理するウェブサイトから脆弱なサイトにリンクしている  
応用。

ブラウザストレージ攻撃	永続ストレージローカルストレージへの悪意のあるデータ注入	sessionStorage、またはその後読み取られる IndexedDB	攻撃者が保存したデータを取得できれば、セッションをまたいで存続できる (別のバグやソーシャルエンジニアリングを介して)そして、 段階的な搾取のために他のベクトルと組み合わせます。
クロスフレームコミュニケーション	インターフーム メッセージング	悪意のあるデータが投稿されたpostMessage APIの悪用 フレームまたはウインドウ間で送信され、処理される 適切なオリジン検証と入力サニタイズが行われていない。	埋め込みiframeやポップアップのあるアプリケーションでよく見られる クロスオリジン通信を実装するウインドウ メカニズム。
ウェブソケットメッセージインジェクション	リアルタイム コミュニケーション	WebSocket経由で配信される悪意のあるペイロード クライアント側メッセージによって処理される接続 適切な入力検証を行わないハンドラー。	チャットなどのリアルタイムアプリケーションに特に効果的 システム、コラボレーションツール、またはライブデータフィード。

## 影響を受ける環境

DOM XSS の脆弱性は、さまざまなカテゴリの Web アプリケーションやクライアント側環境に影響を及ぼす可能性があります。

- React, Angular, Vue.jsなどのフレームワークやカスタムJavaScript実装を活用したシングルページアプリケーション (SPA)
- 広範なクライアント側機能とオフライン機能を備えたプログレッシブ ウェブ アプリケーション (PWA)
- ハッシュベースおよびHTML5 History APIルーティングメカニズムを含むクライアント側ルーティング実装
- コンテンツ スクリプトの挿入と Web ページ インタラクション機能を備えたブラウザ拡張機能とアドオン
- WebViewコンポーネントとJavaScriptプリッジ実装を活用したハイブリッドモバイルアプリケーション
- 複雑なクライアント側ビジネスロジックとDOM操作を備えたリッチインターネットアプリケーション (RIA)
- クライアント側の編集インターフェースと動的なコンテンツレンダリングを備えたコンテンツ管理システム (CMS)
- クライアント側コンテンツの埋め込みと共有機能を実装するソーシャルメディアプラットフォームとウィジェット

## 予防のためのベストプラクティス

このセクションでは、安全な開発プラクティスと技術的制御を通じて DOM XSS の脆弱性を防ぐように設計された包括的なセキュリティ対策について説明します。

### 入力検証とサニタイズ

- URLパラメータ、フラグメント識別子、リファラ値、ストレージメカニズムなど、すべてのクライアント側データソースに対して厳密な入力検証を実装します。  
ブラックリスト方式ではなく、許可リスト方式を使用したクロスフレーム通信。
- HTMLエンティティエンコーディング、JavaScript文字列エンコーディングなど、データが利用される特定のDOMコンテキストに適したコンテキスト出力エンコーディングを確立します。  
エスケープ、CSS 値のエンコード、および URL パラメータのエンコード。
- 厳密な発信元チェックとメッセージ形式の検証を実装することで、フレーム間のポストメッセージ通信のデータの発信元と整合性を検証します。  
信頼できないソースからの悪意のあるデータの挿入を防ぎます。
- アプリケーション内のlocalStorage処理またはレンダリングから取得されたすべてのデータを検証して、永続ストレージ データ、セッションストレージ、または他のクライアント側ストレージメカニズムを使用する前に  
をサニタイズします。

### 安全なDOM操作の実践

- ユーザーが制御可能なデータを処理する際には、文字列引数を持つinnerHTML、outerHTML、AdjacentHTML、document.write()、setTimeout() / setInterval()などのリ、ドキュメント.writeln()、評価()、関数()、  
スクの高い DOM シンクを避けてください。
- コンテンツを解釈しないtextContentやDOM要素作成APIなどのより安全なDOM操作方法を活用するテキストノードを作成()  
実行可能コード。setAttribute()はURL 属性やイベント属性以外の属性にのみ使用し、信頼できないデータからのon\*ハンドラーや危険な URL 属性を設定するために使用しないでください。
- 手動の文字列ではなく、自動出力エンコーディングとXSS保護を提供するフレームワークまたはライブラリを使用して、安全なテンプレートメカニズムを実装します。  
動的コンテンツ生成のための連結。
- ユーザーが制御できるデータと文字列の連結によるJavaScriptコードの構築を避け、データとコードの分離を強制し、  
データ転送用の JSON などの構造化データ形式。

### ブラウザのセキュリティ機能の実装

- 包括的なコンテンツセキュリティポリシー (CSP) ディレクティブ (例: script-src) を展開すると、DOM XSS が存在する場合の影響、オブジェクトソース、ベースURI、frame-ancestors) を使用してスクリプトの実行を制限し、影響が軽減されます。
- すべての外部 JavaScript リソースにサブリソース整合性 (SRI) を実装し、DOM XSS を引き起こす可能性のあるサードパーティ ライブラリの改ざんを防止します。  
脆弱性。
- クリックジャッキング保護のために、CSP のframe-ancestorsディレクティブを優先するX-Content-Type-Options:、リファラーポリシー、および (CSPが利用できない場合) X-Frame-Options、その間nosniffを含むHTTP セキュリティ ヘッダーを活用します。
- 承認されたサニタイザーを通過しない限り DOM シンクへの割り当てを防止するために、Trusted Types (サポートされている場合) などの最新のブラウザ防御を採用します。  
CORS を DOM XSS 制御ではなくアクセス制御メカニズムとして扱い、不要なクロスオリジン データの公開を制限するように適切に構成します。

### 開発とテストの実践

- 定期的にセキュリティコードレビューを実施し、特にクライアント側のデータフロー分析に焦点を当てて、開発中の潜在的なDOM XSS脆弱性を特定します。  
開発プロセス。
- JavaScript コードの DOM XSS 脆弱性を検出し、安全でないデータフローを識別できる専用ツールを使用して、自動静的分析を実装します。  
発生源から吸収源へ。
- DOM XSS検出機能と手動侵入テストを備えた動的アプリケーションセキュリティテスト (DAST) を実行し、  
実装されたセキュリティ制御の有効性。
- DOM XSS攻撃ベクトル、安全なJavaScriptプログラミングプラクティス、適切なコーディング方法のトレーニングを含む開発チーム向けの安全なコーディングガイドラインを確立する。  
セキュリティに重点を置いたライブラリとフレームワークの使用。

コード内にパスワードをハードコードしないでください。イコールでハードコードされたパスワードが見つかりました。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/cryptography/XOREncodingAssignment.java (行: 40)

### データフロー

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

40:32 if (answer\_pwd1 != null && answer\_pwd1.equals("databasepassword")) {

ソースシンク

0

### 修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードは、展開前に削除する必要があるにもかかわらず、コード内に意図せず残ってしまうことがあります。特にパスワードが昇格権限を付与している場合、深刻なセキュリティリスクが生じます。権限が複数のシステム間で再利用される場合もあります。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限の昇格、機密データの窃取、サービスの可用性の妨害などの被害を受ける可能性があります。パスワードがさまざまな環境やアプリケーション間で再利用されると、侵害が急速かつ広範囲に広がる可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## HTTP ヘッダー内の CRLF シーケンスの不適切な無効化

SNYK-CODE CWE-113 HttpResponsesplitting

HTTPリクエストボディからのサニタイズされていない入力はsetHeaderに流れ込み、ユーザーに返されるHTTPヘッダーに到達します。これにより、CR/LF を含む悪意のある入力により、http 応答が 2 つの応答に分割され、2 番目の応答が攻撃者によって制御される可能性があります。

これは、クロスサイトスクリプティングやキャッシュポイズニングなどのさまざまな攻撃を仕掛けるために使用される可能性があります。

見つかった場所: src/main/java/org/owasp/webgoat/webwolf/jwt/JWTController.java (行: 39)

### データフロー

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

35:26 パブリック JWTToken エンコード (@RequestBody MultiValueMap<String, String> フォームデータ) {

ソース

0

35:26 パブリック JWTToken エンコード (@RequestBody MultiValueMap<String, String> フォームデータ) {

1

36:18 var ヘッダー = フォームデータ.getFirst("ヘッダー");

2

36:18 var header = formData.getFirst("header");

3

36:9 var header = formData.getFirst("header");

4

39:28 JWTToken.encode( ヘッダー、ペイロード、シークレットキー ) を返します。

5

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

66:33 パブリック 静的 JWTToken エンコード (String header, String payloadAsString, String secretKey) { var headers = parse

6

67:25 (header); プライベート 静的 Map<String,

7

45:44 Object> parse (String header) { return reader.readValue(header, TreeMap.class);

8

48:31 return reader.readValue(header, TreeMap.class);

9

48:14 }

10

Machine Translated by Google  
80:25 headers.forEach((k, v) -> jws.setHeader(k, v)); headers.forEach((k,  
80:48 v) -> jws.setHeader(k, v)); headers.forEach((k, v) ->  
80:31 jws.setHeader(k, v));

11

12

シンク13

## 修正分析

### 詳細

CRLFは「キャリッジリターン」と「ラインフィード」の略称です。この2つの特殊文字は、コンピュータ黎明期に使用されていた旧式の印刷端末の名残です。しかし、今日でもどちらもデータ間の区切り文字としてよく使用されています。この脆弱性が存在する場合、CR文字とLF文字（コードではそれぞれ\rと\nで表されます）がHTTPヘッダー内に存在することが許可されます。これは通常、開発中のデータ処理計画が不十分なことが原因です。

HTTP ヘッダー内の CRLF シーケンスは、ブラウザからの応答を効率的に分割し、單一行をサーバーによって複数行として受け入れるため、「応答分割」と呼ばれます（たとえば、單一行の First Line\r\nSecond Line は、サーバーによって 2 行の入力として受け入れられます）。

レスポンス分割自体は攻撃ではなく、悪用されない限り全く無害ですが、その脆弱性はインジェクション攻撃（CRLFインジェクション）や、予測不可能で潜在的に危険な様々な動作につながる可能性があります。この脆弱性は、ページハイジャックやクロスユーザー改ざんなど、様々な方法で悪用される可能性があります。クロスユーザー改ざんとは、攻撃者が偽のサイトコンテンツを表示したり、認証情報などの機密情報を取得したりする攻撃です。さらに、クロスサイトスクリプティング攻撃（ユーザーのブラウザで悪意のあるコードを実行させる攻撃）にもつながる可能性があります。

たとえば、次のコードは脆弱です。

```
protected void doGet(HttpServletRequest リクエスト、HttpServletResponse レスポンス) { Cookie cookie = new  
Cookie("name", request.getParameter("name")); response.addCookie(cookie);  
}
```

ユーザーがXYZのような値を持つ名前パラメータを指定する可能性があるため\r\nHTTP/1.1 200 OK\r\n攻撃者が制御  
応答：

HTTP/1.1 200 OK 攻  
撃者が制御

考えられる修正方法は、英数字以外の文字をすべて削除することです。

```
保護された void doGet(HttpServletRequest リクエスト、HttpServletResponse レスポンス) {  
    文字列 name = request.getParameter("name") .replaceAll("[^a-  
zA-Z ]", ""); Cookie cookie = new  
Cookie("name", name); response.addCookie(cookie);  
}
```

この場合、攻撃者は 2 番目の HTTP 応答を生成できなくなります。

### 予防のためのベストプラクティス

- すべての入力は潜在的に悪意のあるものであると想定してください。可能な限り許容可能な応答を定義し、それが不可能な場合は、ヘッダーの分割を防ぐためにCR文字とLF文字をエンコードしてください。
- \r（キャリッジリターン）と\n（ラインフィード）の両方を""（空文字列）に置き換えてください。多くのプラットフォームではこれらの文字は互換的に処理されるため、どちらか一方が許可されている場合でも脆弱性が存在する可能性があります。ベストプラクティスに従い、可能な限り他のすべての特殊文字（"、/、；など、およびスペース）を削除してください。ブラウザからサーバーへの双方向、およびブラウザに返されるデータの両方で、特殊文字をサニタイズしてください。理想的には、ヘッダーへのCRおよびLFインジェクションをブロックする言語やライブドリルなどの最新の開発リソースを採用してください。ユーザー側で（意図的または意図せずに）改ざんまたは変更される可能性があり、インジェクション攻撃につながる可能性のあるすべての入力タイプに注意してください。これには、GET、POST、Cookie、その他のHTTPヘッダーが含まれます。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798、CWE-259 ハードコードされたパスワードなし

コード内にパスワードをハードコードしないでください。パスワード内にハードコードされたパスワードが見つかりました。

見つかった場所: src/main/resources/lessons/jwt/js/jwt-refresh.js (行: 10)

## データフロー

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



## 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードは、展開前に削除する必要があるにもかかわらず、コード内に意図せず残ってしまうことがあります。特にパスワードが昇格権限を付与している場合、深刻なセキュリティリスクが生じます。

権限が複数のシステム間で再利用される場合もあります。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限の昇格、機密データの窃取、サービスの可用性の妨害などの被害を受ける可能性があります。

パスワードがさまざまな環境やアプリケーション間で再利用されると、侵害が急速かつ広範囲に広がる可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## 許容クロスドメインポリシー

SNYK-CODE CWE-942 TooPermissiveCorsPostMessage

postMessageでtargetOriginを「\*」に設定すると、悪意のある第三者がメッセージを傍受できる可能性があります。正確なターゲットオリジンを使用することを検討してください。  
その代わり。

見つかった場所: src/main/resources/webgoat/static/js/libs/ace.js (行: 1740)



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

1740:13 win.postMessage(メッセージ名, "\*");  
1740:13 win.postMessage(メッセージ名, "\*");

ソース

0

シンク

1



## 詳細

初期のウェブデザインとサイトの制限の遺産として、ほとんどのウェブアプリケーションはセキュリティ上の理由から「同一オリジンポリシー」をデフォルトとしています。これは、ブラウザが2つのサイトが同じドメインを共有している場合、別のサイトからデータを取得することはできません。しかし、今日の複雑なオンライン環境では、サイトやアプリケーションは多くの場合、他のドメインからのデータ。これは、「クロスオリジン・リソース・シェアリング」と呼ばれる同一オリジンポリシーの例外を通じて、かなり限定された条件下で行われます。

開発者は、絶対に必要な範囲を超えて信頼ドメインの定義を作成し、意図しないアクセスを開放してしまう可能性があります。この弱点は、

データの漏洩や損失が発生したり、攻撃者がサイトやアプリケーションを乗っ取ったりする可能性があります。

## 予防のためのベストプラクティス

- クロスオリジンリソース共有ではワイルドカードの使用は避けてください。代わりに、対象となるドメインを明示的に定義してください。
- クロスサイト スクリプティング攻撃 (XSS) に対してサイトまたはアプリが適切に防御されていることを確認します。XSS 攻撃は、クロスドメイン ポリシーが過度に許可されている場合に、乗っ取りにつながる可能性があります。
- クロスドメイン ポリシーを定義するときは、安全なプロトコルと安全でないプロトコルを混在させてください。
- どのドメインにリソース レベルのアクセスを許可するかを指定するための明確な承認済みリストを定義することを検討してください。この承認済みリストを使用して、すべてのドメイン アクセス要求を検証します。
- 不正使用を避けるために、各リソースおよびドメインに対して許可されるメソッド (表示、読み取り、更新) を明確に定義します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java (行: 108)

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

108:28 params.put("ユーザー名", "CaptainJack");

ソースシンク

0



## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

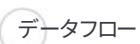
- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (行: 24)



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

24:26 userForm.setUsername( "test1234" );

ソースシンク

0



## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (行: 36)

36:26 userForm.setUsername( "test1234" );

ソースシンク

0



## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

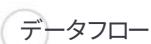
- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (行: 49)



## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

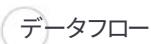
- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpointTest.java (行: 198)





修正分析

詳細

開発者は、ワークフローを簡素化するために、コーディング時にハードコードされた認証情報を使用する場合があります。開発者は、これらの認証情報を削除する責任を負いますが、本番環境では、このタスクが見落とされてしまうことがあります。また、複数のアプリケーション間で認証情報が再利用される場合、メンテナンス上の課題にもなります。

攻撃者がアクセスを獲得すると、権限レベルを利用してデータの削除や変更、サイトやアプリの停止、あるいは上記のいずれかを身代金目的で要求するといったことが起こります。リスクは複数の類似プロジェクトにまたがる場合は、さらに大きな問題となります。認証情報を含むコードが複数のプロジェクトで再利用されると、すべてのプロジェクトが侵害されることになります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTRefreshEndpointTest.java (行: 227)



データフロー

227:27 loginJson.put("user", "Jerry");

ソースシンク

0



修正分析

詳細

開発者は、ワークフローを簡素化するために、コーディング時にハードコードされた認証情報を使用する場合があります。開発者は、これらの認証情報を削除する責任を負いますが、本番環境では、このタスクが見落とされてしまうことがあります。また、複数のアプリケーション間で認証情報が再利用される場合、メンテナンス上の課題にもなります。

攻撃者がアクセスを獲得すると、権限レベルを利用してデータの削除や変更、サイトやアプリの停止、あるいは上記のいずれかを身代金目的で要求するといったことが起こります。リスクは複数の類似プロジェクトにまたがる場合は、さらに大きな問題となります。認証情報を含むコードが複数のプロジェクトで再利用されると、すべてのプロジェクトが侵害されることになります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (行: 70)



データフロー

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

ソースシンク

0

70:34 クレーム claims = createClaims( "WebGoat" );

## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (行: 81)

## データフロー

81:34 クレーム claims = createClaims( "webgoat" );

ソースシンク

0

## ✓修正分析

## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (行: 92)

## データフロー

92:34 クレーム claims = createClaims( "WebGoat" );

ソースシンク

0

## ✓修正分析

## 詳細

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java (行: 133)

### データフロー

133:34 クレーム claims = createClaims( "WebGoat" );

ソースシンク

0

### 修正分析

### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpointTest.java (行: 226)

### データフロー

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

226:24 claims.put("ユーザー", "侵入者");

ソースシンク

0

### 修正分析

### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpointTest.java (行: 244)

データフロー

244:24 claims.put("ユーザー", "侵入者");

ソースシンク

0

✓修正分析

## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignmentTest.java (行: 90)

データフロー

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

90:23 文字列ユーザー名 = "webgoat";

ソースシンク

0

✓修正分析

## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/webwolf/user/UserServiceTest.java (行: 48)

### データフロー

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

48:20 var ユーザー名 = "guest"; ]

ソースシンク

0

### ✓修正分析

#### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/webwolf/user/UserServiceTest.java (行: 61)

### データフロー

61:20 var ユーザー名 = "guest"; ]

ソースシンク

0

### ✓修正分析

#### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。

すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。

デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。

- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。

- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし/テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/webwolf/user/UserServiceTest.java (行: 70)

データフロー

70:20 var ユーザー名 = "guest"; ]

ソースシンク

0

✓修正分析

### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/challenges/challenge1/Assignment1.java (行: 54)

データフロー

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

54:9 「管理者」。equals(ユーザー名)

ソースシンク

0

✓修正分析

### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/insecurelogin/InsecureLoginTask.java (行: 36)

データフロー

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

36:9 if ("CaptainJack".equals(ユーザー名) && "BlackPearl".equals(パスワード)) {

ソースシンク

0

✓修正分析

詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpoint.java (行: 64)

データフロー

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

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

ソースシンク

0

✓修正分析

詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTFinalEndpoint.java (行: 106)

データフロー

106:13 if ( "Jerry".equals(ユーザー名)) {

ソースシンク

0

✓修正分析

### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JTRefreshEndpoint.java (行: 77)

データフロー

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

77:9 if ( "Jerry".equals(ignoreCase(user) && PASSWORD.equals(password)) {

ソースシンク

0

✓修正分析

### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

# Machine Translated by Google

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpoint.java (行: 54)

データフロー

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

54:46 プライベート静的最終文字列 WEBGOAT\_USER = "WebGoat" {

ソースシンク

0

✓修正分析

詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JTUVotesEndpoint.java (行: 158)

データフロー

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

158:13 if ( "ゲスト".equals(user) || !validUsers.contains(user) ) {

ソースシンク

0

✓修正分析

詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/logging/LogBleedingTask.java (行: 60)

#### データフロー

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

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

ソースシンク

0

#### 修正分析

#### 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

#### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

#### ハードコードされたパスワードの使用

SNYK-CODE CWE-798、CWE-259 ハードコードされたパスワード/テスト

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java (行: 109)

#### データフロー

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

109:28 params.put("パスワード", "BlackPearl");

ソースシンク

0

#### 修正分析

#### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

#### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/it/java/org/owasp/webgoat/GeneralLessonIntegrationTest.java (行: 119)

#### データフロー

119:28 params.put("パスワード", "ajnaeliclm^&&@kjn.");

ソースシンク

0

#### ✓修正分析

#### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

#### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (行: 25)

#### データフロー

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

25:26 userForm.setPassword( "test1234" );

ソースシンク

0

#### ✓修正分析

#### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

#### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード/テスト

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (行: 37)

### データフロー

37:26 userForm.setPassword( "test12345" );

ソースシンク

0

### ✓修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

SNYK-CODE CWE-798,CWE-259 ハードコードされたパスワード/テスト

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/test/java/org/owasp/webgoat/container/users/UserValidatorTest.java (行: 50)

### データフロー

50:26 userForm.setPassword( "test12345" );

ソースシンク

0

### ✓修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされたパスワードの使用

コード内にパスワードをハードコードしないでください。ここでハードコードされたパスワードが使用されています。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignmentTest.java (行: 91)

### データフロー

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

91:23 文字列パスワード = "webgoat";

ソースシンク

0

### ✓修正分析

### 詳細

開発者は、開発中にセットアップを効率化したり、テスト中の認証を簡素化したりするために、ハードコードされたパスワードを使用する場合があります。これらのパスワードはデプロイ前に削除されることが想定されていますが、コード内に意図せず残ってしまうことがあります。特に、パスワードが昇格された権限を付与する場合や、複数のシステムで再利用される場合、深刻なセキュリティリスクが生じます。

ハードコードされたパスワードを発見した攻撃者は、不正アクセス、権限昇格、機密データの窃取、サービスの可用性の妨害といった被害を受ける可能性があります。パスワードが異なる環境やアプリケーション間で再利用されている場合、侵害は迅速かつ広範囲に拡大する可能性があります。

### 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

### ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされたシークレット/テスト

ハードコードされた文字列値が暗号鍵として使用されています。代わりに、java.security.SecureRandomなどの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

見つかった場所: src/it/java/org/owasp/webgoat/JWTLessonIntegrationTest.java (行: 217)

### データフロー

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

227:49 .signWith(署名アルゴリズム.HS256, "Tomを削除しています")

ソース

0

217:9 Jwts.ビルダー()

署名アルゴリズム.HS256、「d

シンク

1

.setHeader(header) .setIssuer("WebGoat  
Token

Builder") .setAudience("webgoat.org") .setIssuedAt(Calendar.getInstance().getTime()) .setExpiration(Date.from(Instant.now().plusSeconds(60))) .setS

### ✓修正分析

### 詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが変更されない限り、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する：定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。  
将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされたシークレット/テスト

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/test/java/org/owasp/webgoat/lessons/jwt/JWTSecretKeyEndpointTest.java` (行: 122)

### データフロー

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

122:69 文字列トークン = `Jwts.builder().setClaims(claims).signWith(HS512, "wrong_password").compact();`

ソース

0

122:20 文字列トークン = `Jwts.builder().setClaims(claims).signWith(HS512, "wrong_password").compact();`

シンク

1

### 修正分析

### 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の单一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する：定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。  
将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされたシークレット/テスト

ハードコードされた値の文字列が暗号鍵として使用されます。この値は、次のような暗号的に強力な乱数生成器で生成されます。

代わりに `java.security.SecureRandom` を使用してください。

見つかった場所: `src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java` (行: 48)

### データフロー

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

44:18 文字列キー = `"qwertyqwerty1234";`

ソース

0

```
.setHeaderParam("kid",  
"webgoat_key") .setIssuedAt(新しい日付(System.currentTimeMillis() +  
TimeUnit.DAYS.toDays(10))) .setClaims(クレーム) .signWith(
```

✓修正分析

詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
  - ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
  - 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況(新しいハードウェアなど)に適応するために時間と費用がかかる可能性があります。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされたシークレット/テスト

ハードコードされた文字列値が暗号鍵として使用されています。代わりに、java.security.SecureRandomなどの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (行: 55)

データフロー

```
55:43 Jwt jwt = Jwts.parser().setSigningKey("qwertyqwerty1234").parse(トークン);  
55:15 Jwt jwt = Jwts.parser().setSigningKey("qwertyqwerty1234").parse(トークン);
```

- ソース 0
- シンク 1

✓修正分析

詳細

アプリケーションに定数がハードコードされている場合、その情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害された認証トークンがアプリケーション内の複数の場所にハードコードされている場合、すべてのインスタンスが変更されなければ、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコードすることのもう一つの悪影響は、開発チームがコード全体でハードコードされた定数のすべてのインスタンスを更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。これらの理由から、セキュリティに関する定数をハードコードすることは不適切なコーディング方法とみなされ、もし存在する場合は修正し、将来的には回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
  - ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
  - 「将来を見据えたコード」の考え方を採用する: 定数を使用すると、今は時間が少し節約され、短期的には開発が簡単になるかもしれません、将来的には規模やその他の予期しない状況 (新しいハードウェアなど) に適応するために時間と費用がかかる可能性があります。

ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされたシークレット/テスト

ハードコードされた文字列値が暗号鍵として使用されています。代わりに、java.security.SecureRandomなどの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (行: 76)

```

78:65 .signWith(io.jsonwebtoken.SignatureAlgorithm.HS512, "bm5n3SkxCX4kKRy4")
76:9 Jwts.ビルダー()
        .setClaims(クレーム)
        .signWith()

```

ソース 0  
シンク 1



## 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
- ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシーコード コンポーネントを調べて慎重にテストしてください。
- 「将来を見据えたコード」の考え方を採用する：定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。

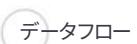
将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## パストラバーサル

SNYK-CODE CWE-23 PT/テスト

クッキーからのサニタライズされていない入力がexistsに流れ込み、パスとして使用されます。これによりパストラバーサルの脆弱性が生じ、攻撃者は条件式でアプリケーションのロジックをバイパスできます。

見つかった場所: src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java (行: 96)



src/it/java/org/owasp/webgoat/統合テスト.java

```

229:9 RestAssured.given()
        .いつも ()
        .relaxedHTTPSValidation()
        .クッキー (
                "WEBWOLFSESSION",getWebWolfCookie())
229:9 RestAssured.given()
        .いつも ()
        .relaxedHTTPSValidation()
        .クッキー (
                "WEBWOLFSESSION",getWebWolfCookie())
229:9 RestAssured.given()
        .いつも ()
        .relaxedHTTPSValidation()
        .cookie("WEBWOLFSESSION", getWebWolfCookie())
        .得る (
                webWolfUrl("/ファイルサーバーの場所"))
229:9 RestAssured.given()
        .いつも ()
        .relaxedHTTPSValidation()
        .cookie("WEBWOLFSESSION", getWebWolfCookie())
        .get(webWolfUrl("/ファイルサーバーの場所"))
        .それから (
229:9 RestAssured.given()
        .いつも ()
        .relaxedHTTPSValidation()
        .cookie("WEBWOLFSESSION", getWebWolfCookie())
        .get(webWolfUrl("/ファイルサーバーの場所"))
        .それから (
        .抽出する (

```

ソース 0  
シンク 1  
ソース 2  
シンク 3  
ソース 4

Machine Translated by Google

```

229:9 RestAssured.given()
    .if ()
        .relaxedHTTPSValidation()
        .cookie("WEBWOLFSESSION", getWebWolfCookie())
        .get(webWolfUrl("/ファイルサーバーの場所"))
    .then()
        .extract()
        .response()
        .body()

```

229:9 RestAssured.given()
 .if ()
 .relaxedHTTPSValidation()
 .cookie("WEBWOLFSESSION", getWebWolfCookie())
 .get(webWolfUrl("/ファイルサーバーの場所"))
 .then()
 .extract()
 .response()
 .body()
 .asString()

228:12 文字列結果 =
 RestAssured.given()
 .if ()
 .relaxedHTTPSValidation()
 .cookie("WEBWOLFSESSION", getWebWolfCookie())
 .get(webWolfUrl("/ファイルサーバーの場所"))
 .then()
 .extract()
 .response()
 .body()
 .asString();

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

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

```

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

66:22 webwolfFileDir = getWebWolfFileServerLocation();
66:5 [ webwolfFileDir = getWebWolfFileServerLocation(); ]
95:38 パス webWolfFilePath = Paths.get(webwolfFileDir);
95:28 パス webWolfFilePath = Paths.get(webwolfFileDir);
95:10 パス [ webWolfFilePath = Paths.get(webwolfFileDir); ]
96:9 webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists() の場合 {
96:9 if ( webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists() ) {
96:9 if ( webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists() ) {
96:9 if ( webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists() ) {
96:5 if (webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)).toFile().exists() ) {
    Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName)));
}

```

シンク21

✓修正分析

詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。



```
.relaxedHTTPSValidation()
.cookie("WEBWOLFSESSION", getWebWolfCookie())
.get(webWolfUrl("/ファイルサーバーの場所"))
.それから ()
.抽出する ()
.応答 (

```

```
229:9 RestAssured.given()
    .いつ ()
    .relaxedHTTPSValidation()
    .cookie("WEBWOLFSESSION", getWebWolfCookie())
    .get(webWolfUrl("/ファイルサーバーの場所"))
    .それから ()
    .抽出する ()
    .応答 ()
    .getBody()
```

```
229:9 RestAssured.given()
    .いつ ()
    .relaxedHTTPSValidation()
    .cookie("WEBWOLFSESSION", getWebWolfCookie())
    .get(webWolfUrl("/ファイルサーバーの場所"))
    .それから ()
    .抽出する ()
    .応答 ()
    .getBody()
    .asString()
```

```
228:12 文字列結果 =
    RestAssured.given()
        .いつ ()
        .relaxedHTTPSValidation()
        .cookie("WEBWOLFSESSION", getWebWolfCookie())
        .get(webWolfUrl("/ファイルサーバーの場所"))
        .それから ()
        .抽出する ()
        .応答 ()
        .getBody()
        .asString();
```

```
239:14 結果 = result.replace("%20", " ");
239:14 結果 = result.replace( "%20", " ") ;
239:5 結果 = result.replace("%20", " ");
```

## src/main/java/org/owasp/webgoat/XXEIntegrationTest.java

```
40:33 webWolfFileServerLocation = getWebWolfFileServerLocation();
40:5 webWolfFileServerLocation = getWebWolfFileServerLocation();
65:38 パス webWolfFilePath = Paths.get( webWolfFileServerLocation);
65:28 パス webWolfFilePath = Paths.get( webWolfFileServerLocation); //パス
65:10 webWolfFilePath = Paths.get( webWolfFileServerLocation);
66:9 webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists() の場合 {
66:9 if ( webWolfFilePath.resolve( Paths.get(this.getUser(), "blind.dtd")).toFile().exists() ) {
66:9 webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists() の場合
66:9 { webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists() の場合 {
66:5 if (webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")).toFile().exists() ) {
    Files.delete(webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")));
}
```

シングル

## ✓修正分析

### 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シケンとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります

- これを実現する方法の一つは、バストラバーサリファイル名を含む要素のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がクレジットの

抽送ノルマを検証するにハスクを書き下ろすこと、最終的なノルマはノーリットノンオフロード側になります。実行ノアルまたは設定ノアルか、要素のモードによって場合によって異なることをご了承ください。

恋愛の機会をもつた場合、恋愛運が簡単に上昇する「美男子」の恋愛運を変える可能性があるのです。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪

更多資訊請上網查詢：[www.104.com.tw](http://www.104.com.tw)

2018-04-15 22:04:29 19

見つかった場所: src/main/java/org/owasp/webgoat/CSRFIntegrationTest.java (行: 97)

ータフロー

src/main/java

。いじ。

```
.cookie("WEBWOLFSESSION", getWebWolfCookie())
.get(webWolfUrl("/ファイルサーバーの場所"))

.それから ()
.抽出する ()
.応答 ()
```

229:9 RestAssured.given()

```
.いつも ()
.relaxedHTTPSValidation()
.cookie("WEBWOLFSESSION", getWebWolfCookie())
.get(webWolfUrl("/ファイルサーバーの場所"))

.それから ()
.抽出する ()
.応答 ()
.getBody()
```

6

229:9 RestAssured.given()

```
.いつも ()
.relaxedHTTPSValidation()
.cookie("WEBWOLFSESSION", getWebWolfCookie())
.get(webWolfUrl("/ファイルサーバーの場所"))

.それから ()
.抽出する ()
.応答 ()
.getBody()
.asString()
```

7

228:12 文字列結果 =

```
RestAssured.given()
.いつも ()
.relaxedHTTPSValidation()
.cookie("WEBWOLFSESSION", getWebWolfCookie())
.get(webWolfUrl("/ファイルサーバーの場所"))

.それから ()
.抽出する ()
.応答 ()
.getBody()
asString();
```

8

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

9

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

10

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

11

## src/main/java/org/owasp/webgoat/CSRFIntegrationTest.java

```
66:22 webwolfFileDir = getWebWolfFileServerLocation();
66:5 [ webwolfFileDir = getWebWolfFileServerLocation();
95:38 パス webWolfFilePath = Paths.get( webwolfFileDir); パス webWolfFilePath
95:28 = Paths.get( webwolfFileDir); パス webWolfFilePath =
95:10 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 [ ]webWolfFilePath.resolve(Paths.get(this.getUser(), htmlName))); Files.delete(
```

12

13

14

15

16

17

18

シンク19



## 詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ（..）」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソースコード、構成、その他の重要なシステムファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限（Unix システムの setuid または setgid フラグなど）で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- 情報漏洩: 攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります。

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります。

注: %2eは URL エンコードされたバージョンです。 .(ドット)。

- 任意のファイルの書き込み: 攻撃者がファイルを作成したり、既存のファイルを置き換えたりできるようになります。このタイプの脆弱性は、Zip-Slipとも呼ばれます。

これを実現する方法の一つは、パストラバーサルファイル名を含む悪意のあるzipアーカイブを使用することです。zipアーカイブ内の各ファイル名がターゲットの抽出フォルダを検証せずにパスを上書きすると、最終的なパスはターゲットフォルダの外側になります。実行ファイルまたは設定ファイルが、悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。最終的に/root/.sshでauthorized\_keysファイルが上書きされます。

2018-04-15 22:04:29 ..... 19 19 良い.txt  
2018-04-15 22:04:42 ..... 20 20 ..../..../..../..../root/.ssh/承認キ-

パストラバーサル

SNYK-CODE CWE-23 PT/テスト

クッキーからのサニタライズされていない入力がjava.nio.file.Files.deleteに流れ込み、パスとして使用されます。これにより、パストラバーサルの脆弱性が発生する可能性があります。攻撃者が任意のファイルを削除できるようになります。

見つかった場所: src/it/java/org/owasp/webgoat/XXEIntegrationTest.java (行: 67)



src/main/java/org/owasp/webgoat/統合テスト.java

229:9	RestAssured.given() .いつ () .relaxedHTTPSValidation() .クッキー (	"WEBWOLFSESSION",getWebWolfCookie()	ソース
229:9	RestAssured.given() .いつ () .relaxedHTTPSValidation() .クッキー (	"WEBWOLFSESSION",getWebWolfCookie()	1
229:9	RestAssured.given() .いつ () .relaxedHTTPSValidation() .cookie("WEBWOLFSESSION", getWebWolfCookie()) .得る (	webWolfUrl("/ファイルサーバーの場所"))	2
229:9	RestAssured.given() .いつ () .relaxedHTTPSValidation() .cookie("WEBWOLFSESSION", getWebWolfCookie()) .get(webWolfUrl("/ファイルサーバーの場所")) .それから (	)	3
229:9	RestAssured.given() .いつ () .relaxedHTTPSValidation() .cookie("WEBWOLFSESSION", getWebWolfCookie()) .get(webWolfUrl("/ファイルサーバーの場所")) .それから () .抽出する (	)	4
229:9	RestAssured.given() .いつ () .relaxedHTTPSValidation() .cookie("WEBWOLFSESSION", getWebWolfCookie()) .get(webWolfUrl("/ファイルサーバーの場所")) .それから () .抽出する () .応答 (	)	5

Machine Translated by Google

```
BestAssured(given())
    .translatedBy Google
        .いつ_()
            .relaxedHTTPSValidation()
            .cookie("WEBWOLFSESSION", getWebWolfCookie())
            .get(webWolfUrl("/ファイルサーバーの場所"))
        .それから_()
            .抽出する_()
                .応答_()
                    .getBody()
```

```
229:9 RestAssured.given()
    .when()
        .get("http://www.webwolf.jp/api/v1/documents")
        .then()
            .statusCode(200)
            .body("documents[0].id", equalTo("1"))
            .body("documents[0].name", equalTo("test"))
            .body("documents[0].content", equalTo("test content"))
    .then()
        .log().all()
```

```
228:12 文字列結果 =  
  
RestAssured.given()  
    .いつも ()  
    .relaxedHTTPSValidation()  
    .cookie("WEBWOLFSESSION", getWebWolfCookie())  
    .get(webWolfUrl("/ファイルサーバーの場所"))  
    .それから ()  
    .抽出する ()  
    .応答 ()  
    .getBody()  
    .asString();
```

```
239:14 結果 = result.replace("%20", " ");  
239:14 結果 = result.replace( "%20", " ");  
239:5 結果 = result.replace("%20", " ");
```

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

```
40:33    webWolfFileServerLocation = getWebWolfFileServerLocation();  
40:5    [ webWolfFileServerLocation = getWebWolfFileServerLocation(); ]  
65:38    パス webWolfFilePath = Paths.get( webWolfFileServerLocation ); パス webWolfFilePath  
65:28    = Paths.get( webWolfFileServerLocation ); パス webWolfFilePath =  
65:10    Paths.get( webWolfFileServerLocation );  
67:20    Files.delete( webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd")));  
67:20    Files.delete( webWolfFilePath.resolve( Paths.get(this.getUser(), "blind.dtd")));  
67:7    [ ] webWolfFilePath.resolve(Paths.get(this.getUser(), "blind.dtd"))); Files.delete
```

シノク19



詳細

ディレクトリトラバーサル攻撃（パストラバーサルとも呼ばれる）は、意図したフォルダ外に保存されているファイルやディレクトリにアクセスすることを目的としています。

「ドット・ドット・スラッシュ (..)」シーケンスとそのバリエーション、または絶対ファイルパスを使用することで、ファイルシステムに保存されている任意のファイルやディレクトリにアクセスできる可能性があります。

アプリケーションのソース コード、構成、その他の重要なシステム ファイルなどが含まれます。

任意のパスにアクセスして操作できるということは、パスを提供するユーザーが持つべきではない権限でプログラムが実行されている場合に脆弱性につながる。

パストラバーサルの脆弱性を持つウェブサイトは、ユーザーがそのウェブサイトをホストするサーバー上の機密ファイルにアクセスできてしまう可能性があります。CLIプログラムもパストラバーサルの脆弱性を持つ可能性があります。

昇格された権限(Unix システムの setuid または setgid フラグなど)で実行されている場合は、トラバーサルが発生します。

ディレクトリ トラバーサルの脆弱性は、一般的に次の 2 つのタイプに分けられます。

- ・ 情報漏洩:攻撃者がフォルダー構造に関する情報を取得したり、システム上の機密ファイルの内容を読み取ったりできるようになります

stは、Web ページ上で静的ファイルを提供するためのモジュールであり、このタイプの脆弱性が含まれています。この例では、パブリックルートからファイルを提供します。

攻撃者が当社のサーバーから次の URL を要求すると、ルート ユーザーの機密性の高い秘密鍵が漏洩することになります

注: %2eはURl エンコードされたバージョンです。 (ドット)。

(ドット)

- ・ 任意のファイルの書き込み 攻撃者がファイルを作成したり 既存のファイルを置き換えるたりできるようになります このタイプの脆弱性は Zip-Slipとも呼ばれます

悪意のあるコードの場合、問題は簡単に任意のコード実行の問題に変わる可能性があります。

以下は、無害なファイルと悪意のあるファイルが1つずつ含まれたzipアーカイブの例です。悪意のあるファイルを抽出すると、対象フォルダから移動します。

最終的に/root/.ssh/でauthorized\_keysファイルが上書きされます。

```
2018-04-15 22:04:29 ....          19          19 良い.txt  
2018-04-15 22:04:42 ....          20          20 ..../..../..../..../root/.ssh/承認キー
```

## 「HttpOnly」フラグのない機密Cookie

SNYK-CODE CWE-1004 WebCookieMissesCallToSetHttpOnly

Cookie は setHttpOnly の呼び出しに失敗しました。クライアント側で悪意のあるコードが実行される可能性から Cookie を保護するには、HttpOnly フラグを true に設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/hijacksession/HijackSessionAssignment.java (行: 86)

### データフロー

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

86:25    Cookie cookie = 新しい Cookie(COOKIE\_NAME, cookieValue);

ソースシンク

0

### 修正分析

### 詳細

HttpOnly フラグは、ユーザー Cookie を設定する際に使用される単純なパラメータであり、機密セッションデータを含む Cookie がブラウザにのみ表示されるようにするものです。スク립トへの接続。これにより、攻撃者が機密セッション情報にアクセスし、その情報をを使って正当なユーザーを騙すクロスサイトスクリプティング攻撃を防ぐことができます。ウェブベースのアプリケーションが機密情報を漏洩したり、不正なリクエストを受け入れたりするのを防ぐことができます。開発者が HttpOnly フラグを使用して Cookie を設定すると、この機密性の高いセッション情報は、ブラウザ（読み取り）とサーバー（書き込み）以外では読み取りまたは書き込みできないようにする必要があります。ほとんどの最新のブラウザとバージョンでは HttpOnly フラグが認識されるようになりましたが、一部のレガシーブラウザとカスタムブラウザではまだ認識されません。

### 予防のためのベストプラクティス

- クライアント側で Cookie を設定する際は、レスポンスヘッダーに HttpOnly 属性を含めてください。ただし、この重要なステップは、部分的な効果しか提供しないことに注意してください。修復。
- ブラウザのバージョンを判断するクライアント側スクリプトを統合し、ブラウザの互換性を要求するか、サポートしていないブラウザに機密データを送信しないようにします。HttpOnly。
- クッキーを公開する可能性のあるサードパーティのコンポーネントまたはプラグインのリスクを理解し、評価します。
- 開発者にゼロトラストアプローチを教育し、クロスサイトスクリプティングを防ぐためのリスクとベストプラクティス（コードのすべてのユーザー入力をサニタイズするなど）を理解してもらうおよび特殊文字。

## 「HttpOnly」フラグのない機密Cookie

SNYK-CODE CWE-1004 WebCookieMissesCallToSetHttpOnly

Cookie は setHttpOnly の呼び出しに失敗しました。クライアント側で悪意のあるコードが実行される可能性から Cookie を保護するには、HttpOnly フラグを true に設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (行: 130)

### データフロー

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

130:27    Cookie cookie = 新しい Cookie("access\_token", token);

ソースシンク

0

### 修正分析

### 詳細

HttpOnly フラグは、ユーザー Cookie を設定する際に使用されるシンプルなパラメータです。このフラグにより、機密性の高いセッションデータを含むCookieが、スクリプトではなくブラウザのみに表示されるようになります。これは、クロスサイトスクリプティング攻撃を防ぐのに役立ちます。クロスサイトスクリプティング攻撃では、攻撃者が機密性の高いセッション情報をアクセスし、その情報をを利用して正規の Webベース アプリケーションを騙し、機密情報を開示させたり、不正なリクエストを受け入れさせたりします。開発者が HttpOnly フラグを使用してCookieを設定することで、この機密性の高いセッション情報は、ブラウザ（読み取り）とサーバー（書き込み）以外では読み取りまたは書き込み不可になります。ほとんどの最新ブラウザとバージョンはHttpOnly フラグを認識しますが、一部の旧式ブラウザやカスタムブラウザはまだ認識しません。

## 予防のためのベストプラクティス

- クライアント側でCookieを設定する際は、レスポンスヘッダーにHttpOnly属性を含めてください。ただし、この重要なステップは部分的な改善にしかならないことに注意してください。
- ブラウザのバージョンを判別するためにクライアント側スクリプトを統合し、ブラウザの互換性を要求するか、HttpOnlyをサポートしていないブラウザに機密データを送信しないようにします。
- クッキーを公開する可能性のあるサードパーティのコンポーネントまたはプラグインのリスクを理解し、評価します。
- 開発者にゼロトラスト アプローチを教育し、すべてのユーザー入力をコードや特殊文字に対してサニタイズするなど、クロスサイトスクリプティングを防止するためのリスクとベスト プラクティスを理解してもらいます。

## 「HttpOnly」フラグのない機密Cookie

SNYK-CODE CWE-1004 WebCookieMissesCallToSetHttpOnly

Cookie は setHttpOnly の呼び出しに失敗しました。クライアント側で悪意のあるコードが実行される可能性から Cookie を保護するには、HttpOnly フラグを true に設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (行: 135)

### データフロー

135:27 Cookie cookie = 新しい Cookie("access\_token", "");

ソースシンク

0

### 修正分析

### 詳細

HttpOnly フラグは、ユーザー Cookie を設定する際に使用されるシンプルなパラメータです。このフラグにより、機密性の高いセッションデータを含むCookieが、スクリプトではなくブラウザのみに表示されるようになります。これは、クロスサイトスクリプティング攻撃を防ぐのに役立ちます。クロスサイトスクリプティング攻撃では、攻撃者が機密性の高いセッション情報をアクセスし、その情報をを利用して正規の Webベース アプリケーションを騙し、機密情報を開示させたり、不正なリクエストを受け入れさせたりします。開発者が HttpOnly フラグを使用してCookieを設定することで、この機密性の高いセッション情報は、ブラウザ（読み取り）とサーバー（書き込み）以外では読み取りまたは書き込み不可になります。ほとんどの最新ブラウザとバージョンはHttpOnly フラグを認識しますが、一部の旧式ブラウザやカスタムブラウザはまだ認識しません。

## 予防のためのベストプラクティス

- クライアント側でCookieを設定する際は、レスポンスヘッダーにHttpOnly属性を含めてください。ただし、この重要なステップは部分的な改善にしかならないことに注意してください。
- ブラウザのバージョンを判別するためにクライアント側スクリプトを統合し、ブラウザの互換性を要求するか、HttpOnlyをサポートしていないブラウザに機密データを送信しないようにします。
- クッキーを公開する可能性のあるサードパーティのコンポーネントまたはプラグインのリスクを理解し、評価します。
- 開発者にゼロトラスト アプローチを教育し、すべてのユーザー入力をコードや特殊文字に対してサニタイズするなど、クロスサイトスクリプティングを防止するためのリスクとベスト プラクティスを理解してもらいます。

## 「HttpOnly」フラグのない機密Cookie

SNYK-CODE CWE-1004 WebCookieMissesCallToSetHttpOnly

Cookie は setHttpOnly の呼び出しに失敗しました。クライアント側で悪意のあるコードが実行される可能性から Cookie を保護するには、HttpOnly フラグを true に設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (行: 76)

### データフロー

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

76:25 Cookie cookie = 新しい Cookie(COOKIE\_NAME, "");

ソースシンク

0

### 修正分析

HttpOnlyフラグは、ユーザーCookieを設定する際に使用されるシンプルなパラメータです。このフラグにより、機密性の高いセッションデータを含むCookieが、スクリプトではなくブラウザのみに表示されるようになります。これは、クロスサイトスクリプティング攻撃を防ぐのに役立ちます。クロスサイトスクリプティング攻撃では、攻撃者が機密性の高いセッション情報をアクセスし、その情報を利用して正規のWebベースアプリケーションを騙し、機密情報を開示させたり、不正なリクエストを受け入れさせたりします。開発者がHttpOnlyフラグを使用してCookieを設定することで、この機密性の高いセッション情報は、ブラウザ（読み取り）とサーバー（書き込み）以外では読み取りまたは書き込み不可になります。ほとんどの最新ブラウザとバージョンはHttpOnlyフラグを認識しますが、一部の旧式ブラウザやカスタムブラウザはまだ認識しません。

## 予防のためのベストプラクティス

- クライアント側でCookieを設定する際は、レスポンスヘッダーにHttpOnly属性を含めてください。ただし、この重要なステップは部分的な改善にしかならないことに注意してください。
- ブラウザのバージョンを判別するためにクライアント側スクリプトを統合し、ブラウザの互換性を要求するか、HttpOnlyをサポートしていないブラウザに機密データを送信しないようにします。
- クッキーを公開する可能性のあるサードパーティのコンポーネントまたはプラグインのリスクを理解し、評価します。
- 開発者にゼロトラストアプローチを教育し、すべてのユーザー入力をコードや特殊文字に対してサニタイズするなど、クロスサイトスクリプティングを防止するためのリスクとベストプラクティスを理解してもらいます。

## 「HttpOnly」フラグのない機密Cookie

SNYK-CODE CWE-1004 WebCookieMissesCallToSetHttpOnly

CookieはsetHttpOnlyの呼び出しに失敗しました。クライアント側で悪意のあるコードが実行される可能性からCookieを保護するには、HttpOnlyフラグをtrueに設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (行: 92)

### データフロー

92:30 Cookie newCookie = 新しい Cookie( COOKIE\_NAME, newCookieValue);

ソースシンク

0

### ✓修正分析

#### 詳細

HttpOnlyフラグは、ユーザーCookieを設定する際に使用されるシンプルなパラメータです。このフラグにより、機密性の高いセッションデータを含むCookieが、スクリプトではなくブラウザのみに表示されるようになります。これは、クロスサイトスクリプティング攻撃を防ぐのに役立ちます。クロスサイトスクリプティング攻撃では、攻撃者が機密性の高いセッション情報をアクセスし、その情報を利用して正規のWebベースアプリケーションを騙し、機密情報を開示させたり、不正なリクエストを受け入れさせたりします。開発者がHttpOnlyフラグを使用してCookieを設定することで、この機密性の高いセッション情報は、ブラウザ（読み取り）とサーバー（書き込み）以外では読み取りまたは書き込み不可になります。ほとんどの最新ブラウザとバージョンはHttpOnlyフラグを認識しますが、一部の旧式ブラウザやカスタムブラウザはまだ認識しません。

## 予防のためのベストプラクティス

- クライアント側でCookieを設定する際は、レスポンスヘッダーにHttpOnly属性を含めてください。ただし、この重要なステップは部分的な改善にしかならないことに注意してください。
- ブラウザのバージョンを判別するためにクライアント側スクリプトを統合し、ブラウザの互換性を要求するか、HttpOnlyをサポートしていないブラウザに機密データを送信しないようにします。
- クッキーを公開する可能性のあるサードパーティのコンポーネントまたはプラグインのリスクを理解し、評価します。
- 開発者にゼロトラストアプローチを教育し、すべてのユーザー入力をコードや特殊文字に対してサニタイズするなど、クロスサイトスクリプティングを防止するためのリスクとベストプラクティスを理解してもらいます。

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass/テスト

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/main/java/org/owasp/webgoat/JWTLessonIntegrationTest.java (行: 68)

### データフロー

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

68:19 Jwt jwt = Jwts.parser().setSigningKey(TextCodec.BASE64.encode(key)).parse( トークン);

ソースシンク

0

## 詳細

io.jsonwebtoken.jwt ライブラリの一部の JSON Web Token (JWT) 解析メソッドは、パーサーに署名鍵が設定されているにもかかわらず、署名が空の JWT を受け入れてしまいます。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

## 予防のためのベストプラクティス

- parseClaimsJws または parsePlaintextJws メソッドを使用するか、JwtHandlerAdapter の onPlaintextJws または onClaimsJws メソッドをオーバーライドして、常に JWT 署名の検証を強制します。

## 予防のためのベストプラクティス

- [JWSを読む](#)

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass/テスト

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (行: 55)

## データフロー

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

55:15 Jwt jwt = Jwts.parser().setSigningKey("qwertyqwerty1234").parse(トークン);

ソースシンク

0

## ✓修正分析

## 詳細

io.jsonwebtoken.jwt ライブラリの一部の JSON Web Token (JWT) 解析メソッドは、パーサーに署名鍵が設定されているにもかかわらず、署名が空の JWT を受け入れてしまいます。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

## 予防のためのベストプラクティス

- parseClaimsJws または parsePlaintextJws メソッドを使用するか、JwtHandlerAdapter の onPlaintextJws または onClaimsJws メソッドをオーバーライドして、常に JWT 署名の検証を強制します。

## 予防のためのベストプラクティス

- [JWSを読む](#)

## JWT署名検証バイパス

SNYK-CODE CWE-347 JwtVerificationBypass/テスト

解析メソッドはJWT署名を検証しません。代わりに「parseClaimsJws」または「parsePlaintextJws」の使用を検討してください。

見つかった場所: src/test/java/org/owasp/webgoat/lessons/jwt/TokenTest.java (行: 57)

## データフロー

57:9 Jwts.パーサー().トークン) ;

```
.setSigningKeyResolver(新しい
    SigningKeyResolverAdapter() { @Override
        public byte[]
        resolveSigningKeyBytes(JwsHeader ヘッダー、 Claims クレーム) {
            TextCodec.BASE64.decode(キー) を返します。
        }
    }
}
```

ソースシンク

0



修正分析

詳細

io.jsonwebtoken.jwt ライブラリの一部の JSON Web Token (JWT) 解析メソッドは、パーサーに署名鍵が設定されているにもかかわらず、署名が空の JWT を受け入れてしまいます。つまり、これらのメソッドを使用すると、攻撃者は任意の JWT を作成し、それを受け入れてしまう可能性があります。

### 予防のためのベストプラクティス

- parseClaimsJws または parsePlaintextJws メソッドを使用するか、JwtHandlerAdapter の onPlaintextJws または onClaimsJws メソッドをオーバーライドして、常に JWT 署名の検証を強制します。

### 予防のためのベストプラクティス

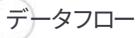
- [JWSを読む](#)

## 'Secure' 属性のない HTTPS セッション内の機密 Cookie

SNYK-CODE CWE-614 WebCookieMissesCallToSetSecure

Cookie は setSecure の呼び出しに失敗しました。中間者攻撃から Cookie を保護するには、Secure フラグを true に設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (行: 130)



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

130:27 Cookie cookie = 新しい Cookie("access\_token", token);

ソースシンク

0



修正分析

詳細

セッションハイジャック攻撃では、機密データを含むCookieがセキュア属性なしで設定されている場合、攻撃者がそのCookieを傍受できる可能性があります。攻撃者がこの情報を入手すると、ユーザーになりますし、機密データにアクセスし、通常は許可されていない操作を実行する可能性があります。攻撃者は、この機密Cookieデータが暗号化されてHTTPSセッションで送信されるのではなく、標準のHTTPセッションでプレーンテキストとして安全でない方法で送信されることで、多くの場合、この機密Cookieデータにアクセスできるようになります。機密性の高いセッションCookieを設定する際にベストプラクティスに従うことで、この種の攻撃は大幅に防止できます。

### 予防のためのベストプラクティス

- クライアント側で Cookie を設定するときに、応答ヘッダーに secure 属性を設定し、テスト ツールを使用して、安全な Cookie 送信が行われていることを確認します。
- すべてのログイン ページでは必ず HTTPS を使用し、HTTP から HTTPS にリダイレクトしないでください。HTTP から HTTPS にリダイレクトすると、安全なセッション データが傍受される可能性があります。
- セッション クッキーに関しては、HttpOnly フラグの設定や、時間制限の厳しいセッションの維持など、その他のベスト プラクティスに従ってください。
- ブラウザ チェックを実装し、厳格な Cookie セキュリティをサポートするブラウザ内でのみ安全なデータを提供することを検討してください。
- 簡単に予測できない方法でセッション ID を生成し、ログアウト時にセッションを無効にし、セッション ID を再利用しないでください。
- 開発者に対して、DIY アプローチを取るのではなく、開発環境内に組み込まれた安全なセッション管理機能を使用するように指導します。

## 'Secure' 属性のない HTTPS セッション内の機密 Cookie

SNYK-CODE CWE-614 WebCookieMissesCallToSetSecure

Cookie は setSecure の呼び出しに失敗しました。中間者攻撃から Cookie を保護するには、Secure フラグを true に設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/jwt/JWTVotesEndpoint.java (行: 135)



135:27 Cookie cookie = 新しい Cookie("access\_token", "");

ソースシンク

0

## 詳細

セッションハイジャック攻撃では、機密データを含むCookieがセキュア属性なしで設定されている場合、攻撃者がそのCookieを傍受できる可能性があります。攻撃者がこの情報を入手すると、ユーザーになりますし、機密データにアクセスし、通常は許可されていない操作を実行する可能性があります。攻撃者は、この機密Cookieデータが暗号化されてHTTPSセッションで送信されるのではなく、標準のHTTPセッションでplainテキストとして安全でない方法で送信されることで、多くの場合、この機密Cookieデータにアクセスできるようになります。機密性の高いセッションCookieを設定する際にベストプラクティスに従うことで、この種の攻撃は大幅に防止できます。

## 予防のためのベストプラクティス

- クライアント側で Cookie を設定するときに、応答ヘッダーに secure 属性を設定し、テスト ツールを使用して、安全な Cookie 送信が行われていることを確認します。
- すべてのログイン ページでは必ず HTTPS を使用し、HTTP から HTTPS にリダイレクトしないでください。HTTP から HTTPS にリダイレクトすると、安全なセッション データが傍受される可能性があります。
- セッション クッキーに関しては、HttpOnly フラグの設定や、時間制限の厳しいセッションの維持など、その他のベスト プラクティスに従ってください。
- ブラウザ チェックを実装し、厳格な Cookie セキュリティをサポートするブラウザ内でのみ安全なデータを提供することを検討してください。
- 簡単に予測できない方法でセッション ID を生成し、ログアウト時にセッションを無効にし、セッション ID を再利用しないでください。
- 開発者に対して、DIY アプローチを取るのではなく、開発環境内に組み込まれた安全なセッション管理機能を使用するように指導します。

## 'Secure' 属性のない HTTPS セッション内の機密 Cookie

SNYK-CODE CWE-614 WebCookieMissesCallToSetSecure

Cookie は setSecure の呼び出しに失敗しました。中間者攻撃から Cookie を保護するには、Secure フラグを true に設定してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/spoofcookie/SpoofCookieAssignment.java (行: 76)

## データフロー

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

76:25 Cookie cookie = 新しい Cookie(COOKIE\_NAME, "");

ソースリンク

0

## ✓修正分析

## 詳細

セッションハイジャック攻撃では、機密データを含むCookieがセキュア属性なしで設定されている場合、攻撃者がそのCookieを傍受できる可能性があります。攻撃者がこの情報を入手すると、ユーザーになりますし、機密データにアクセスし、通常は許可されていない操作を実行する可能性があります。攻撃者は、この機密Cookieデータが暗号化されてHTTPSセッションで送信されるのではなく、標準のHTTPセッションでplainテキストとして安全でない方法で送信されることで、多くの場合、この機密Cookieデータにアクセスできるようになります。機密性の高いセッションCookieを設定する際にベストプラクティスに従うことで、この種の攻撃は大幅に防止できます。

## 予防のためのベストプラクティス

- クライアント側で Cookie を設定するときに、応答ヘッダーに secure 属性を設定し、テスト ツールを使用して、安全な Cookie 送信が行われていることを確認します。
- すべてのログイン ページでは必ず HTTPS を使用し、HTTP から HTTPS にリダイレクトしないでください。HTTP から HTTPS にリダイレクトすると、安全なセッション データが傍受される可能性があります。
- セッション クッキーに関しては、HttpOnly フラグの設定や、時間制限の厳しいセッションの維持など、その他のベスト プラクティスに従ってください。
- ブラウザ チェックを実装し、厳格な Cookie セキュリティをサポートするブラウザ内でのみ安全なデータを提供することを検討してください。
- 簡単に予測できない方法でセッション ID を生成し、ログアウト時にセッションを無効にし、セッション ID を再利用しないでください。
- 開発者に対して、DIY アプローチを取るのではなく、開発環境内に組み込まれた安全なセッション管理機能を使用するように指導します。

## 計算量が不十分なパスワードハッシュの使用

SNYK-CODE CWE-916 安全でないハッシュ

MD5ハッシュ (java.security.MessageDigest.getInstanceで使用) は安全ではありません。安全なハッシュアルゴリズムに変更することを検討してください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java (行: 55)

## データフロー

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

✓修正分析

詳細

機密情報はブレーンテキストで保存しないでください。そうしないと、悪意のある内部者や外部の攻撃者など、権限のないユーザーが簡単に情報にアクセスできてしまいます。アクセス。ハッシュ化手法は、保存されたパスワードや他の機密データをユーザーが読み取れないようにするために使用されます。例えば、パスワードを初めて定義する場合、パスワードはハッシュ化されて保存されます。ユーザーが次回ログインを試みると、入力したパスワードは同じ手順でハッシュ化され、保存された値。この方法では、元のパスワードをシステムに保存する必要はありません。

ハッシュ化は一方向方式であるため、ハッシュ化されたパスワードはリバースエンジニアリングできません。ただし、古いハッシュ化方式やカスタムプログラムされたハッシュ化方式を使用すると、強力な最新計算能力を持つ攻撃者にとって、ハッシュへのアクセスは容易になります。これにより、保存されているすべてのパスワード情報にアクセスできるようになります。

セキュリティ侵害につながります。そのため、開発者は最新の安全なパスワードハッシュ技術を理解することが不可欠です。

## 予防のためのベストプラクティス

- ハッシュには、固有の弱点がある可能性のある、単純だが時代遅れの方法や DIY ハッシュスキームではなく、強力な標準アルゴリズムを使用します。
  - ハッシュを扱うすべてのコードにモジュール設計を使用し、時間の経過とともにセキュリティ標準が変わったときに交換できるようにします。
  - ソルトをハッシュと組み合わせて使用します（これによりリソースに対する要求は高まりますが、セキュリティを強化するために不可欠なステップです）
  - ゼロトラスト アーキテクチャを実装して、パスワードデータへのアクセスが正当なビジネス目的のみ許可されるようにします。
  - データセキュリティと暗号化の現在の標準に対する開発者の認識を高めます。

#### ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし / テスト

資格情報をコード内にハードコードしないでください。

見つかった場所: src/test/java/org/owasp/webgoat/ChallengeIntegrationTest.java (行: 32)

データフロー

src/main/java/org/owasp/webgoat/ChallengeCompositeTest.java

```
32:28     params.put("ユーザー名", "admin");
```

ソースシンク 0

## ✓修正分析

詳細

開発者は、ワークフローを簡素化するために、コーディング時にハードコードされた認証情報を使用する場合があります。開発者は、これらの認証情報を削除する責任を負いますが、本番環境では、このタスクが見落とされてしまうことがあります。また、複数のアプリケーション間で認証情報が再利用される場合、メンテナンス上の課題にもなります。

攻撃者がアクセスを獲得すると、権限レベルを利用してデータの削除や変更、サイトやアプリの停止、あるいは上記のいずれかを身代金目的で要求するといったことが起こります。リスクは複数の類似プロジェクトにまたがる場合は、さらに大きな問題となります。認証情報を含むコードが複数のプロジェクトで再利用されると、すべてのプロジェクトが侵害されることになります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
  - すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
  - デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
  - ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
  - 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

CHMIX CODE CHM 720.00 - 第二級社會文化政策研究（二二一）

资格审核表二-技术标书-技术-技术方案

見つかっても場所: `src/test/java/org/awesomewebhost/IntegrationTest.java` (行: 20)

src/main/java/org/owasp/webgoat/統合テスト.java

29:39 @Getter プライベート最終文字列 user = "webgoat";

ソースシンク

0



## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除するのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

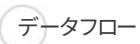
- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた資格情報の使用

SNYK-CODE CWE-798 ハードコードされた資格情報なし

資格情報をコード内にハードコードしないでください。

見つかった場所: src/main/java/org/owasp/webgoat/lessons/challenges/challenge7/PasswordResetLink.java (行: 15)



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

15時35分 if (username.equalsIgnoreCase( "admin" )) {

ソースシンク

0



## 詳細

開発者は、コーディング時にワークフローを簡素化するために、利便性のためにハードコードされた認証情報を使用する場合があります。本番環境導入前にこれらの認証情報を削除るのは開発者の責任ですが、この作業が見落とされてしまうことがあります。また、複数のアプリケーションで認証情報を再利用する場合、メンテナンス上の課題にもなります。

攻撃者がアクセス権限を取得すると、権限レベルを悪用してデータの削除や改ざん、サイトやアプリの停止、あるいはこれらのいずれかに対する身代金要求などを行う可能性があります。複数の類似プロジェクトにまたがる場合のリスクはさらに大きくなります。認証情報を含むコードが複数のプロジェクトで再利用された場合、すべてのプロジェクトが侵害される可能性があります。

## 予防のためのベストプラクティス

- 可能な限り、キーとパスワードが常にコードの外部に保存されるようにソフトウェア アーキテクチャを計画します。
- すべての資格情報に対するソフトウェア アーキテクチャへの暗号化を計画し、キー、資格情報、およびパスワードが適切に処理されるようにします。
- デフォルトのパスワードをハードコードするのではなく、最初のログイン時に安全なパスワードの入力を求めます。
- ハードコードされたパスワードまたは資格情報を使用する必要がある場合は、その使用を、たとえばネットワーク経由ではなくシステム コンソール ユーザーに制限します。
- 受信パスワード認証には強力なハッシュを使用します。理想的には、ブルートフォース攻撃の場合の難易度を上げるために、ランダムに割り当てられたソルトを使用します。

## ハードコードされた秘密

SNYK-CODE CWE-547 ハードコードされたシークレット/テスト

ハードコードされた文字列値が暗号鍵として使用されています。代わりに java.security.SecureRandom などの暗号的に強力な乱数ジェネレータを使用して値を生成してください。

## データフロー

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

```

33:18 文字列キー = "deletingTom";           ソース 0
37:9 Jwts.ビルダー()
      .setHeaderParam()
        「子供」、「ハッキングされた」UNION選択      "" + キー + INFORMATION_SCHEMA.SYSTEM_USERS から --
      .setIssuedAt(新しい日付(System.currentTimeMillis() + TimeUnit.DAYS.toDays(10)))
      .setClaims(クレーム)
      .signWith()

```

io.js シンク 1

## ✓修正分析

## 詳細

定数がアプリケーションにハードコードされている場合、この情報は簡単にリバースエンジニアリングされ、攻撃者に知られる可能性があります。例えば、侵害されたアプリケーションが認証トークンがアプリケーションの複数の場所にハードコードされている場合、すべてのインスタンスが認証されていない場合、アプリケーションのコンポーネントが脆弱なままになる可能性があります。定数をハードコーディングすることのもう一つの悪影響は、開発チームがすべての定数を更新しなかった場合、アプリケーションのパフォーマンスが予測不可能になる可能性があることです。コード全体を通して、ハードコードされた定数の単一のインスタンスのみを記述します。これらの理由から、セキュリティ関連定数をハードコードすることは、不適切なコーディング方法とみなされ、存在する場合は是正し、将来は回避する必要があります。

## 予防のためのベストプラクティス

- セキュリティ関連の定数をハードコードしないでください。シンボリック名または構成検索ファイルを使用してください。
  - ハードコーディングは小規模で一人で作業するコーダーによって行われることが多いため、スケーリングする際には、すべてのレガシー コード コンポーネントを調べて慎重にテストしてください。
  - 「将来を見据えたコード」の考え方を採用する: 定数の使用は、短期的には時間を節約し、開発を簡素化するかもしれません、時間とコストがかかる可能性があります。
- 将来規模やその他の予期しない状況（新しいハードウェアなど）に適応するための資金。

## 'Secure' 属性のない HTTPS セッション内の機密 Cookie

SNYK-CODE CWE-614 WebCookieSecureDisabledByDefault

CookieにSecure属性がありません（デフォルトではfalseです）。中間者攻撃からCookieを保護するには、これをtrueに設定してください。

見つかった場所: src/main/resources/lessons/spoofcookie/js/handler.js (行: 9)

## データフロー

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

```

9時11分 document.cookie = 'spoof_auth=;Max-Age=0;secure=true';           ソースシンク 0

```

## ✓修正分析

## 詳細

セッションハイジャック攻撃では、機密データを含むCookieがセキュア属性なしで設定されている場合、攻撃者はそのCookieを傍受できる可能性があります。攻撃者がこの情報を入手した場合、ユーザーになりすまして機密データにアクセスし、通常は許可されていないアクションを実行する可能性があります。攻撃者は、この機密性の高いCookieデータが暗号化されずに、標準のHTTPセッションでプレーンテキストとして安全でない方法で送信されると、アクセスできるようになることが多い。HTTPSセッション経由で送信されます。この種の攻撃は、機密性の高いセッションCookieを設定する際にベストプラクティスに従うことで、高い確率で防ぐことができます。

## 予防のためのベストプラクティス

- クライアント側で Cookie を設定するときに、応答ヘッダーに secure 属性を設定し、テスト ツールを使用して、安全な Cookie 送信が行われていることを確認します。
- すべてのログイン ページでは必ず HTTPS を使用し、HTTP から HTTPS にリダイレクトしないでください。HTTP から HTTPS にリダイレクトすると、安全なセッション データが傍受される可能性があります。
- セッション クッキーに関しては、HttpOnly フラグの設定や、時間制限の厳しいセッションの維持など、その他のベスト プラクティスに従ってください。
- ブラウザ チェックを実装し、厳格な Cookie セキュリティをサポートするブラウザ内でのみ安全なデータを提供することを検討してください。
- 簡単に予測できない方法でセッション ID を生成し、ログアウト時にセッションを無効にし、セッション ID を再利用しないでください。
- 開発者に対して、DIY アプローチを取るのではなく、開発環境内に組み込まれた安全なセッション管理機能を使用するように指導します。

jQueryプラグインへのオプションからのサニタイズされていない入力は\$に流れ込み、HTMLとして評価される可能性があります。入力がセレクターを使用する場合は、セレクターのみを受け入れるJQueryメソッドを使用する必要があります。入力をHTMLとして使用することを意図している場合、開発者は入力内容をサニタイズする責任はユーザーにあることが十分に文書化されていることを確認する必要があります。

見つかった場所: src/main/resources/webgoat/static/js/libs/jquery-ui-1.10.4.js (行: 12144)

## データフロー

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

12135:27	\$.fn.position = 関数(オプション) { [ ] }	ソース	0
12135:27	\$.fn.position = 関数(オプション) { [ ] }		1
12141:26	オプション = \$.extend( {}, オプション );		2
12141:14	オプション = \$.extend( {}, オプション );		3
12141:2	オプション = \$.extend( {}, オプション );		4
12144:15	ターゲット = \$( オプション.of ).		5
12144:23	ターゲット = \$( オプション.of ).		6
12144:15	ターゲット = \$( options.of ).		7
12144:12	ターゲット = \$( options.of ).	シンク	8

## 修正分析

## 詳細

JQueryなどの一般的なフレームワークは、サードパーティ製のプラグインによって拡張できます。サードパーティ製のプラグインは、例えばJQueryに追加の機能を追加するなどです。フレームワーク。このような関数は、クライアント（プラグインを使用する開発者）からのパラメータを受け入れる場合があります。

これらのプラグインのクライアントは、プラグインの実装の詳細を知らないため、入力と出力は、プラグイン開発者によって文書化される必要があります。

特に重要なのは、プラグイン開発者が特定の入力をクライアントがサニタイズする必要があるかどうかを文書化することです。

JQuery フレームワークは、次のもので構成される API を公開します。

- DOM内の要素を選択するために何らかの入力を使用するメソッド
- 何らかの入力を使用してDOMを変更するメソッド

その結果、プラグインは、例えば、ユーザー入力を受け取り、それを内部的にサニタイズせずにDOMに書き込む可能性があり、クライアントが必要な処理を実行していることを期待している。サニタイズ。この場合、クロスサイトスクリッピングの脆弱性が発生する可能性があります。

## 予防のためのベストプラクティス

- サニタイズすべき入力内容を文書化する
  - JQuery プラグイン開発者は、クロスサイトスクリッピング攻撃につながる可能性のある入力をすべて徹底的に文書化することが推奨されます。
- 曖昧なJQuery APIを避ける ドキュメント作成の責任に加えて、JQuery プラグイン開発者は、ユーザー入力が曖昧にならないように特に注意する必要があります。予期しない方法で処理されます。
  - JQuery には非常に柔軟な API があり、与えられた入力の種類に応じて選択または変更を実行するメソッドがいくつかあります。
  - その結果、プラグイン開発者は、プラグインがユーザー入力を受け入れて選択を実行することを意図しているが、特定のユーザー入力は代わりに動的なHTMLを実行する可能性がある。工事。
  - ユーザー入力が選択を実行するためだけに使われることを意図している場合は、選択を実行するだけのメソッドを使用する必要があります。