Fortify Security Report

16 Mar 2022

Demo User

Executive Summary

Issues Overview

On 16 Mar 2022, a source code review was performed over the FortifyDemoApp code base. 21 files, 297 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 27 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Critical	9
Low	9
High	8
Medium	1

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: C:/Users/klee/source/repos/FortifyDemoApp
Number of Files: 21
Lines of Code: 297
Build Label: SNAPSHOT

Scan Information

Scan time: 00:38
SCA Engine version: 21.2.3.0005
Machine Name: GBklee01
Username running scan: klee

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
Command Line Arguments:
com.microfocus.app.FortifyDemoApp.main

Environment Variables:
null.null.null

Java Properties:
java.lang.System.getProperty

System Information:
null.null.null
null.null.lambda
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.System.getProperty

Filter Set Summary

Current Enabled Filter Set:
Security Auditor View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical
If [fortify priority order] contains high Then set folder to High
If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low

Audit Guide Summary

Audit guide not enabled

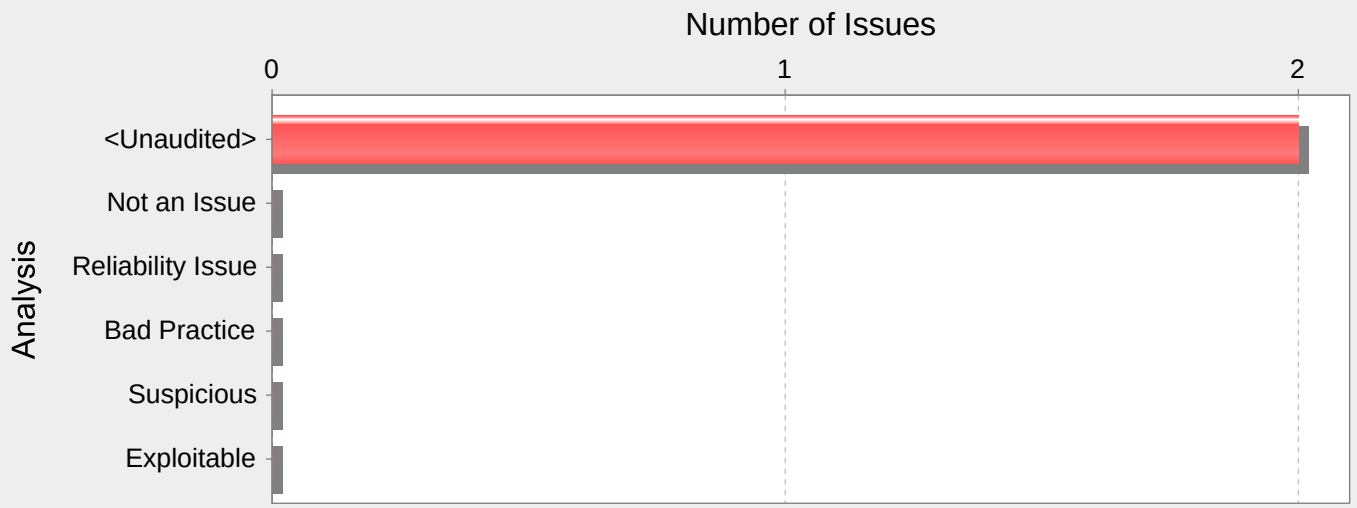
Results Outline

Overall number of results

The scan found 27 issues.

Vulnerability Examples by Category

Category: Cross-Site Scripting: Reflected (2 Issues)



Abstract:

The method _jspService() in index.jsp sends unvalidated data to a web browser on line 160, which can result in the browser executing malicious code.

Explanation:

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of reflected XSS, the untrusted source is typically a web request, while in the case of persisted (also known as stored) XSS it is typically a database or other back-end data store.

2. The data is included in dynamic content that is sent to a web user without validation.

The malicious content sent to the web browser often takes the form of a JavaScript segment, but can also include HTML, Flash or any other type of code that the browser executes. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

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

The code in this example operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes metacharacters or source code, then the code is executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL which causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use email or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

Example 2: The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

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

```
String name = rs.getString("name");
}
%>

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

As in Example 1, this code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. Again, this code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker may execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

Some think that in the mobile environment, classic web application vulnerabilities, such as cross-site scripting, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

Example 3: The following code enables JavaScript in Android's WebView (by default, JavaScript is disabled) and loads a page based on the value received from an Android intent.

```
...
WebView webview = (WebView) findViewById(R.id.webview);
webview.getSettings().setJavaScriptEnabled(true);
String url = this.getIntent().getExtras().getString("url");
webview.loadUrl(url);
...
```

If the value of url starts with javascript:, JavaScript code that follows executes within the context of the web page inside WebView.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or emailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- As in Example 2, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- As in Example 3, a source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

Recommendations:

The solution to XSS is to ensure that validation occurs in the correct places and checks are made for the correct properties.

Because XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application might accept input through a shared data store or other trusted source, and that data store might accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means that the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create an allow list of safe characters that are permitted to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alphanumeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser must be considered valid input after they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is to implement a deny list, which selectively rejects or escapes potentially dangerous characters before using the input. To form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines which characters have special meaning, many web browsers try to correct common mistakes in HTML and might treat other characters as special in certain contexts. This is why we do not recommend the use of deny lists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed in double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed in single quotes, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything greater than 127 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page.

Within the body of a <SCRIPT> </SCRIPT>:

- Semicolons, parentheses, curly braces, and new line characters must be filtered out in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and might bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

After you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and might be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. For any developed application, there are no guarantees about which application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will continue to stay in sync.

Tips:

1. The Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources. Nonetheless, it is often still a good idea to validate everything read from the database.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient to protect against cross-site scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. Fortify AppDefender adds protection against this category.

index.jsp, line 160 (Cross-Site Scripting: Reflected)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method _jspService() in index.jsp sends unvalidated data to a web browser on line 160, which can result in the browser executing malicious code.		

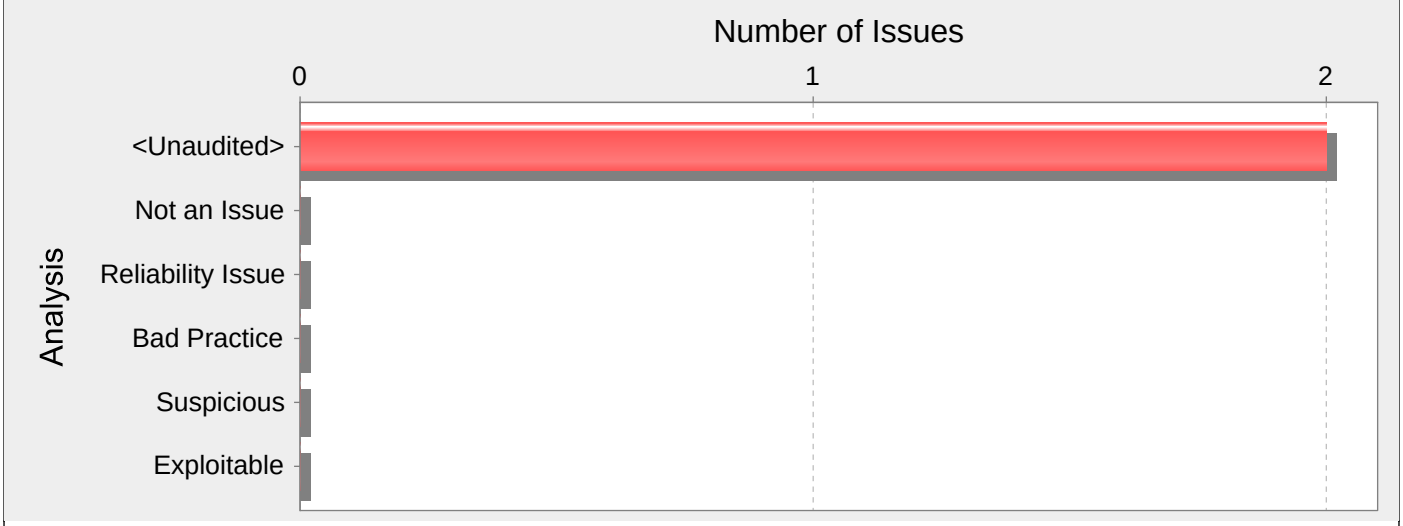
Source: index.jsp:160 javax.servlet.ServletRequest.getParameter()

```
158         <div class="input-group">
159          <input type="text" class="form-control" placeholder="Search products" name="keywords"
160              id="keywords" value="${param.keywords}">
161          <div class="input-group-append">
162           <button class="btn btn-secondary" type="submit">
```

Sink: index.jsp:160 javax.servlet.jsp.JspWriter.print()

```
158         <div class="input-group">
159          <input type="text" class="form-control" placeholder="Search products" name="keywords"
160              id="keywords" value="${param.keywords}">
161          <div class="input-group-append">
162           <button class="btn btn-secondary" type="submit">
```

Category: JSON Injection (2 Issues)



Abstract:

On line 43 of FileSystemService.java, the method writeUser() writes unvalidated input into JSON. This call could allow an attacker to inject arbitrary elements or attributes into the JSON entity.

Explanation:

JSON injection occurs when:

1. Data enters a program from an untrusted source.

2. The data is written to a JSON stream.

Applications typically use JSON to store data or send messages. When used to store data, JSON is often treated like cached data and may potentially contain sensitive information. When used to send messages, JSON is often used in conjunction with a RESTful service and can be used to transmit sensitive information such as authentication credentials.

The semantics of JSON documents and messages can be altered if an application constructs JSON from unvalidated input. In a relatively benign case, an attacker may be able to insert extraneous elements that cause an application to throw an exception while parsing a JSON document or request. In a more serious case, such as ones that involves JSON injection, an attacker may be able to insert extraneous elements that allow for the predictable manipulation of business critical values within a JSON document or request. In some cases, JSON injection can lead to cross-site scripting or dynamic code evaluation.

Example 1: The following Java code uses Jackson to serialize user account authentication information for non-privileged users (those with a role of "default" as opposed to privileged users with a role of "admin") from user-controlled input variables username and password to the JSON file located at ~/user_info.json:

```
...
JsonFactory jfactory = new JsonFactory();
JsonGenerator jGenerator = jfactory.createJsonGenerator(new File("~/user_info.json"), JsonEncoding.UTF8);
jGenerator.writeStartObject();
jGenerator.writeFieldName("username");
jGenerator.writeRawValue("\"" + username + "\"");
jGenerator.writeFieldName("password");
jGenerator.writeRawValue("\"" + password + "\"");
jGenerator.writeFieldName("role");
jGenerator.writeRawValue("\"default\"");
jGenerator.writeEndObject();
jGenerator.close();
```

Yet, because the JSON serialization is performed using JsonGenerator.writeRawValue(), the untrusted data in username and password will not be validated to escape JSON-related special characters. This allows a user to arbitrarily insert JSON keys, possibly changing the structure of the serialized JSON. In this example, if the non-privileged user mallory with password Evil123! were to append ","role":"admin to her username when entering it at the prompt that sets the value of the username variable, the resulting JSON saved to ~/user_info.json would be:

```
{
"username":"mallory",
```

"role":"admin",
"password":"Evil123!",
"role":"default"
}

If this serialized JSON file were then deserialized to an HashMap object with Jackson's JsonParser as so:

JsonParser jParser = jfactory.createJsonParser(new File("~/user_info.json"));

while (jParser.nextToken() != JsonToken.END_OBJECT) {

String fieldname = jParser.getCurrentName();

if ("username".equals(fieldname)) {
jParser.nextToken();
userInfo.put(fieldname, jParser.getText());
}

if ("password".equals(fieldname)) {
jParser.nextToken();
userInfo.put(fieldname, jParser.getText());
}

if ("role".equals(fieldname)) {
jParser.nextToken();
userInfo.put(fieldname, jParser.getText());
}

if (userInfo.size() == 3)
break;
}

jParser.close();

The resulting values for the username, password, and role keys in the HashMap object would be mallory, Evil123!, and admin respectively. Without further verification that the deserialized JSON values are valid, the application will incorrectly assign user mallory "admin" privileges.

Recommendations:

When writing user supplied data to JSON, follow these guidelines:

1. Do not create JSON attributes with names that are derived from user input.

2. Ensure that all serialization to JSON is performed using a safe serialization function that delimits untrusted data within single or double quotes and escapes any special characters.

Example 2: The following Java code implements the same functionality as that in Example 1, but instead uses JsonGenerator.writeString() rather than JsonGenerator.writeRawValue() to serialize the data, therefore ensuring that any untrusted data is properly delimited and escaped:

...

JsonFactory jfactory = new JsonFactory();

JsonGenerator jGenerator = jfactory.createJsonGenerator(new File("~/user_info.json"), JsonEncoding.UTF8);

jGenerator.writeStartObject();

jGenerator.writeFieldName("username");
jGenerator.writeString(username);

jGenerator.writeFieldName("password");
jGenerator.writeString(password);

jGenerator.writeFieldName("role");
jGenerator.writeString("default");

jGenerator.writeEndObject();

jGenerator.close();

FileSystemService.java, line 43 (JSON Injection)

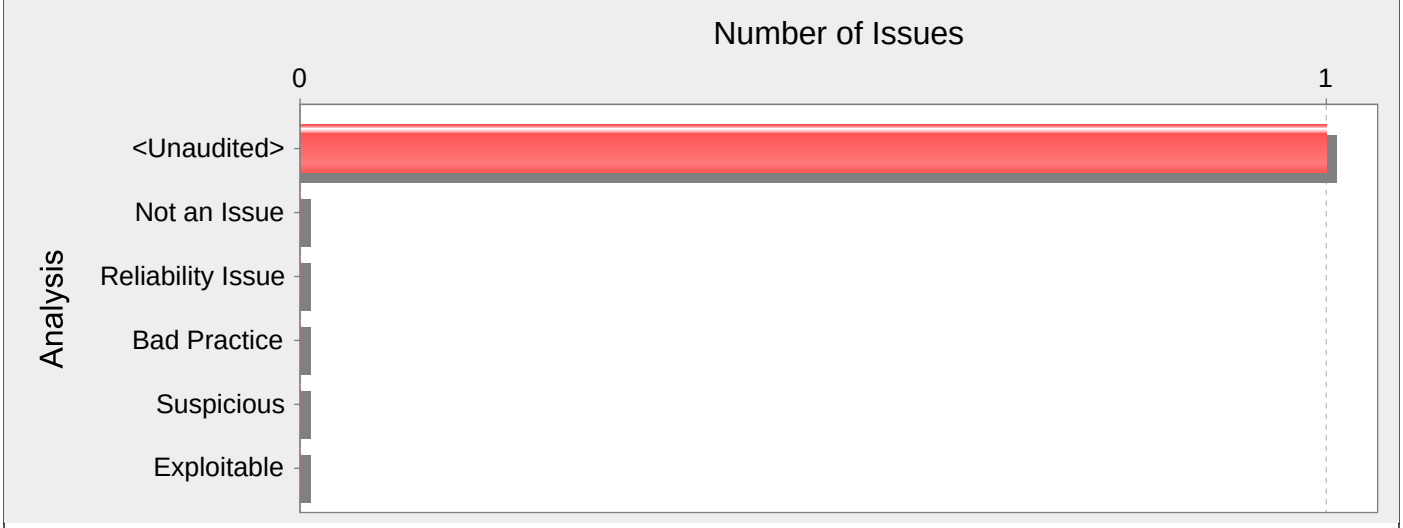
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 43 of FileSystemService.java, the method writeUser() writes unvalidated input into JSON. This call could allow an attacker to inject arbitrary elements or attributes into the JSON entity.		
Source:	DefaultController.java:64 subscribeUser(0)		

```
62
63      @PostMapping(value = {"/api/subscribe-user"}, produces = {"application/json"},
consumes = {"application/json"})
64      public ResponseEntity<String> subscribeUser(@RequestBody SubscribeUserRequest
newUser) {
65          log.debug("DefaultController:subscribeUser");
66          try {
```

Sink: FileSystemService.java:43 com.fasterxml.jackson.core.JsonGenerator.writeRawValue()

```
41
42    jGenerator.writeFieldName("name");
43    jGenerator.writeRawValue("\"" + user + "\"");
44
45    jGenerator.writeFieldName("email");
```

Category: Access Control: Azure Storage (1 Issues)



Abstract:

On line 45 of azuredeploy.json, the template defines an Azure storage account with unrestricted network access.

Explanation:

By default, Azure storage accounts accept connections from clients on any network.

Loose access configurations unnecessarily expose systems and broaden an organization's attack surface. Services open to interaction with the public are subjected to almost continuous scanning and probing by malicious entities.

Example 1: The following example template fails to restrict network access to an Azure storage account.

```
{
...
"resources": [
{
"name": "VulnerableStorageAccount",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2019-06-01",
"tags": {
"displayName": "vulnerablestore1"
},
"location": "[resourceGroup().location]",
...
"properties": {
"networkAcls": {
"bypass": "None",
"defaultAction": "Allow"
}
}
}
],
"outputs": {}
}
```

Recommendations:

Restrict critical services access to specific IP addresses, ranges, or virtual networks.

Configure Azure Storage firewall rules to block all traffic by default and specify the particular sources of trusted communication.

To specify trusted sources, configure the networkAcls/ipRules, networkAcls/resourceAccessRules or networkAcls/virtualNetworkRules. Then, to block all traffic by default, configure the networkAcls/defaultAction property to Deny.

Note that the networkAcls property was introduced in apiVersion 2017-06-01 and later.

Example 2: The following example fixes the problem in Example 1. It restricts network access to an Azure storage account to a specific virtual network.

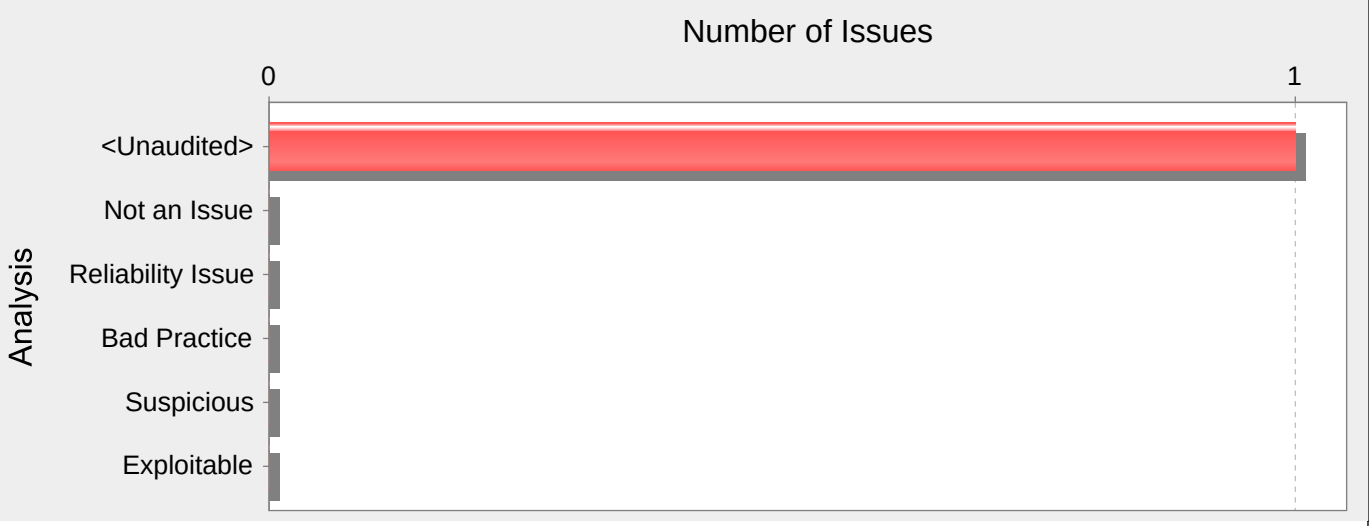
```
{
...
"resources": [
{
"name": "VulnerableStorageAccount",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2019-06-01",
"tags": {
"displayName": "vulnerablestore1"
},
"location": "[resourceGroup().location]",
...
"properties": {
"networkAcls": {
"bypass": "None",
"virtualNetworkRules": [
{
"id": "[variables('subnetId')[0]]",
"action": "Allow"
}
],
"defaultAction": "Deny"
}
}
}
],
"outputs": {}
}
```

azuredeploy.json, line 45 (Access Control: Azure Storage)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 45 of azuredeploy.json, the template defines an Azure storage account with unrestricted network access.		
Sink:	azuredeploy.json:45 ConfigMap()		

```
43          },
44          "resources": [
45              {
46                  "type": "Microsoft.Storage/storageAccounts",
47                  "sku": {
```

Category: Azure Resource Manager Misconfiguration: Insecure Transport (1 Issues)



Abstract:

On line 89 of azuredeploy.json, the template defines an Azure App Service that does not enforce HTTPS communication.

Explanation:

Unencrypted communication channels are prone to eavesdropping and tampering.

Serving web applications over HTTP allows attackers to perform man-in-the-middle attacks, giving them access to read or modify the data in transit over the channel.

Example 1: The following example shows a template that defines an Azure App Service that does not enforce HTTPS communication.

```
"resources": [
{
"name": "webSite",
"type": "Microsoft.Web/sites",
"apiVersion": "2020-12-01",
"location": "location1",
"tags": {},
"properties": {
"enabled": true
},
"resources": []
}
```

Recommendations:

Enforce transport encryption to ensure that all data exchange is performed over encrypted channels and inhibit man-in-the-middle attacks.

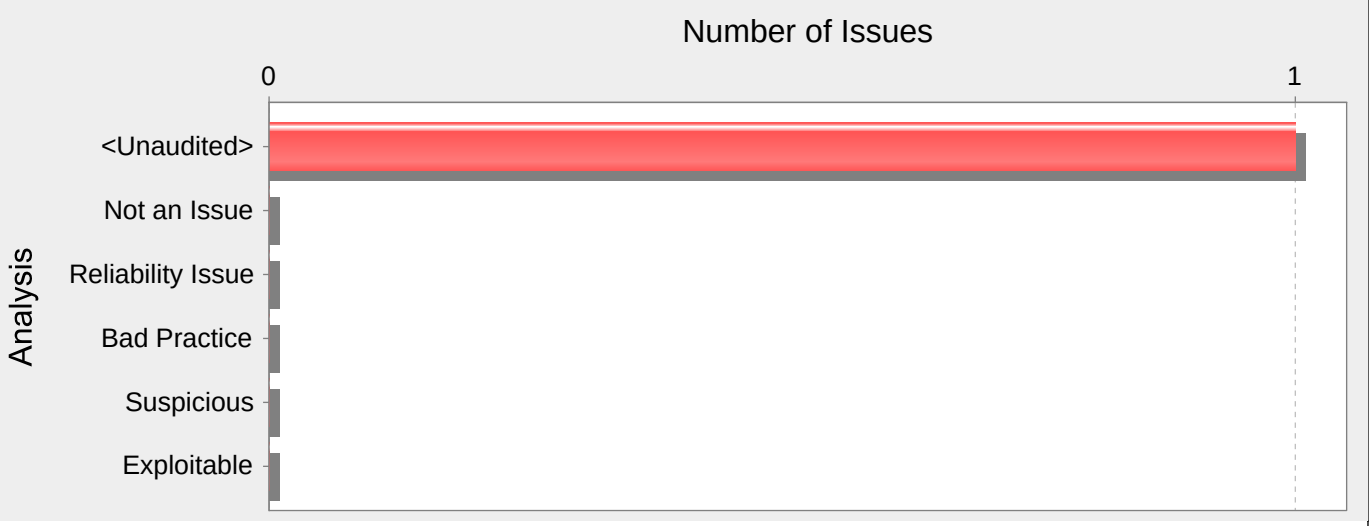
Example 2: The following example fixes the problem in Example 1. It enforces HTTPS communication by enabling the httpsOnly property.

```
"resources": [
{
"name": "webSite",
"type": "Microsoft.Web/sites",
"apiVersion": "2020-12-01",
"location": "location1",
"tags": {},
"properties": {
"enabled": true,
"httpsOnly": true
},
"resources": []
}
```

azuredeploy.json, line 89 (Azure Resource Manager Misconfiguration: Insecure Transport)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Environment		
Abstract:	On line 89 of azuredeploy.json, the template defines an Azure App Service that does not enforce HTTPS communication.		
Sink:	azuredeploy.json:89 ConfigMap()		

```
87                      }
88                  },
89                  {
90                      "type": "Microsoft.Web/sites",
91                      "kind": "app",
```

Category: Azure Resource Manager Misconfiguration: Overly Permissive CORS Policy (1 Issues)



Abstract:

On line 55 of azuredeploy.json, the template defines an overly permissive CORS policy.

Explanation:

Cross-Origin Resource Sharing, commonly referred to as CORS, is a technology that allows a domain to define a policy for its resources to be accessed by a web page hosted on a different domain. Historically, web browsers have restricted their domain resources from being accessed by scripts loaded from a different domain to abide by the same origin policy.

CORS provides a method for a domain to whitelist other domains and allow them to access its resources.

Be careful when defining a CORS policy because an overly permissive policy configured at the server level for a domain or a directory on a domain can expose more content for cross domain access than intended. CORS can enable a malicious application to communicate with a victim application inappropriately, which can lead to information disclosure, spoofing, data theft, relay, or other attacks.

Implementing CORS can increase an application's attack surface and should only be used when necessary.

Example 1: The following example template defines an overly permissive CORS policy for an Azure SignalR web application.

```
{
...
"type": "Microsoft.SignalRService/SignalR",
...
"properties": {
...
"cors": {
"allowedOrigins": ["*"]
},
...
}
```

Example 2: The following example template defines an overly permissive CORS policy for an Azure web application.

```
{
"apiVersion": "2020-12-01",
"type": "Microsoft.Web/sites",
...
"properties": {
...
"siteConfig": {
...
"cors": {
"allowedOrigins": [
"*"
]
},
```

```
...
}
```

Example 3: The following example template defines an overly permissive CORS policy for an Azure Maps account.

```
{
"apiVersion": "2021-12-01-preview",
"type": "Microsoft.Maps/accounts",
...
"properties":{
"cors":{
"allowedOrigins": ["*"]
}
},
...
}
```

Example 4: The following example template defines an overly permissive CORS policy for an Azure Cosmos DB account.

```
{
"type": "Microsoft.DocumentDB/databaseAccounts",
...
"properties": {
"cors": [{
"allowedOrigins":"*"
}],
...
}
```

Example 5: The following example template defines an overly permissive CORS policy for an Azure storage blob service.

```
{
"type": "Microsoft.Storage/storageAccounts/blobServices",
...
"properties": {
"cors": {
"corsRules": [
{
"allowedOrigins":["*"],
...
}
...
}
```

Recommendations:

Do not use a wildcard (*) as the value for the allowedOrigins property. Instead, provide an explicit list of trusted domains.

Example 6: The following example fixes the problem in Example 1 by specifying an explicitly trusted domain for an Azure SignalR web application.

```
{
...
"type": "Microsoft.SignalRService/SignalR",
...
"properties": {
...
"cors": {
"allowedOrigins": ["www.trusted.com"]
},
...
}
```

Example 7: The following example fixes the problem in Example 2 by specifying an explicitly trusted domain for an Azure web application.

```
{
"apiVersion": "2020-12-01",
"type": "Microsoft.Web/sites",
...
"properties": {
...
"siteConfig": {
...
"cors": {
"allowedOrigins": [
"www.trusted.com"
]
},
...
}
```

Example 8: The following example fixes the problem in Example 3 by specifying an explicitly trusted domain for an Azure Maps account.

```
{
"apiVersion": "2021-12-01-preview",
"type": "Microsoft.Maps/accounts",
...
"properties":{
"cors":{
"allowedOrigins": ["www.trusted.com"]
}
},
...
}
```

Example 9: The following example fixes the problem in Example 4 by specifying an explicitly trusted domain for an Azure Cosmos DB.

```
{
"type": "Microsoft.DocumentDB/databaseAccounts",
...
"properties": {
"cors": [{
"allowedOrigins":"www.trusted.com"
}],
...
}
```

Example 10: The following example fixes the problem in Example 5 by specifying an explicitly trusted domain for an Azure blob Service.

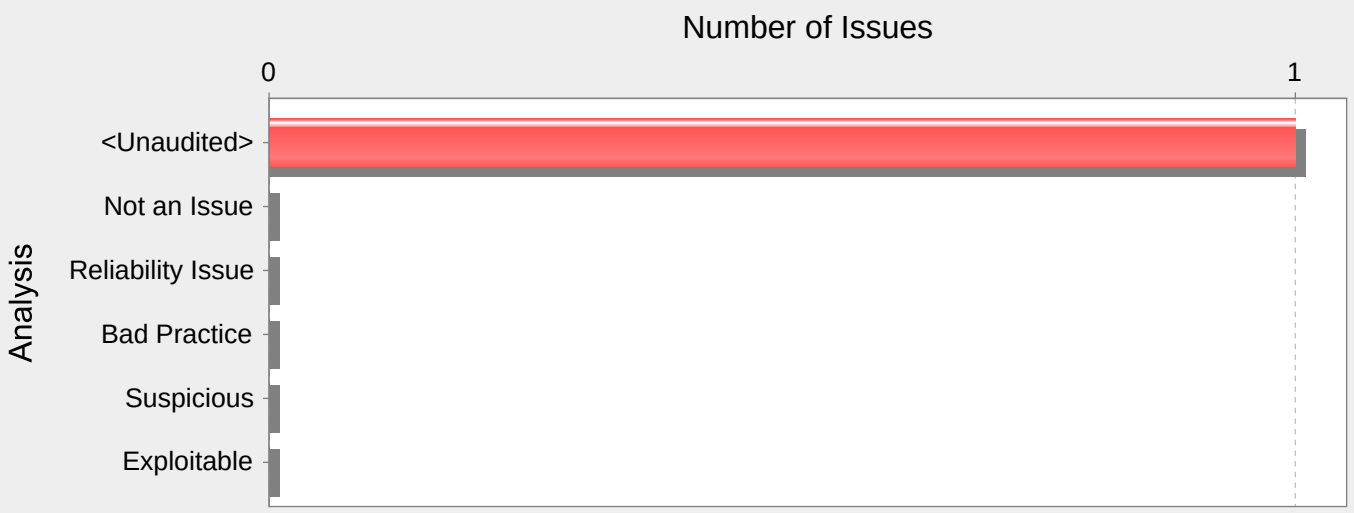
```
{
"type": "Microsoft.Storage/storageAccounts/blobServices",
...
"properties": {
"cors": {
"corsRules": [
{
"allowedOrigins":["www.trusted.com"],
...
}
...
```

```
}
```

azuredeploy.json, line 55 (Azure Resource Manager Misconfiguration: Overly Permissive CORS Policy)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 55 of azuredeploy.json, the template defines an overly permissive CORS policy.		
Sink:	azuredeploy.json:55 ConfigMap()		

```
53                  "location": "[parameters('location')]",
54                  "properties": {
55                  "cors": {
56                    "corsRules": [
57                      {
```

Category: Dockerfile Misconfiguration: Default User Privilege (1 Issues)



Abstract:

The Dockerfile sets a container to run with root user.

Explanation:

Docker containers run with super user privileges by default. These super user privileges are propagated to the code running inside the container, which is usually more permission than necessary.

Recommendations:

It is good practice to run your containers as a non-root user when possible.

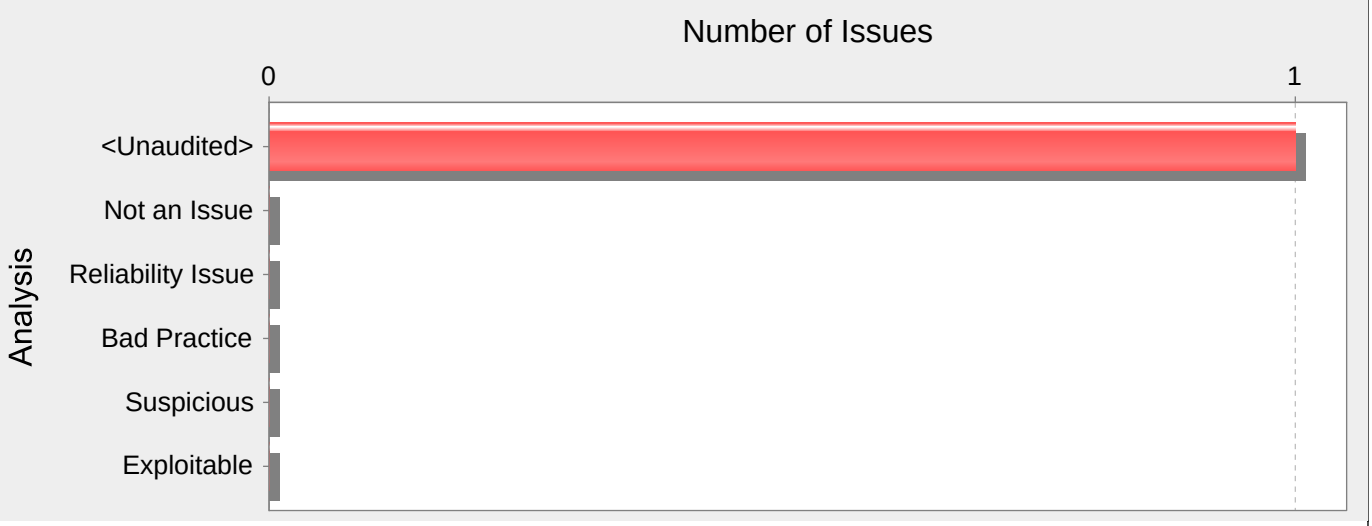
To modify a docker container to use a non-root user, the Dockerfile needs to specify a different user, such as:

RUN useradd myLowPrivilegeUser
USER myLowPrivilegeUser

Dockerfile, line 1 (Dockerfile Misconfiguration: Default User Privilege)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	The Dockerfile sets a container to run with root user.		
Sink:	Dockerfile:1 FROM()		
-1	`FROM openjdk:11-jdk-slim`		
0			
1	`LABEL maintainer="kevin.lee@microfocus.com"`		

Category: Insecure Randomness (1 Issues)



Abstract:

The random number generator implemented by random() cannot withstand a cryptographic attack.

Explanation:

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudorandom Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and form an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between the generated random value and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts, where its use can lead to serious vulnerabilities such as easy-to-guess temporary passwords, predictable cryptographic keys, session hijacking, and DNS spoofing.

Example: The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

```
function genReceiptURL (baseURL){
var randNum = Math.random();
var receiptURL = baseURL + randNum + ".html";
return receiptURL;
}
```

This code uses the Math.random() function to generate "unique" identifiers for the receipt pages it generates. Since Math.random() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendations:

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Do not use values such as the current time because it offers only negligible entropy.)

In JavaScript, the typical recommendation is to use the window.crypto.random() function in the Mozilla API. However, this method does not work in many browsers, including more recent versions of Mozilla Firefox. There is currently no cross-browser solution for a robust cryptographic PRNG. In the meantime, consider handling any PRNG functionality outside of JavaScript.

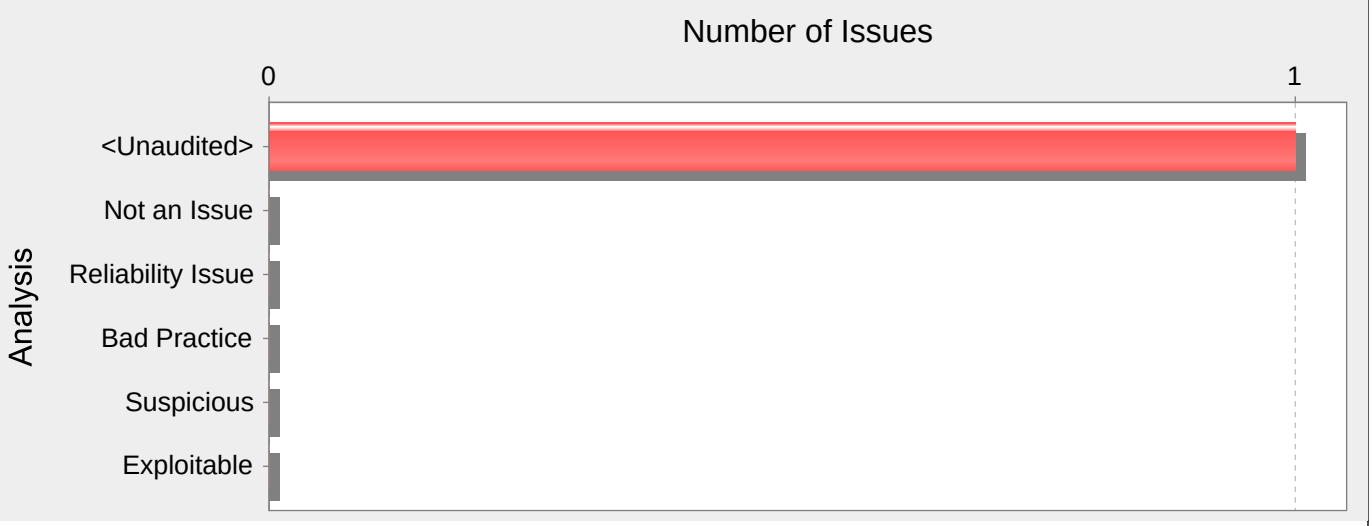
SubscribeNewsletter.js, line 39 (Insecure Randomness)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The random number generator implemented by random() cannot withstand a cryptographic attack.		
Sink:	SubscribeNewsletter.js:39 FunctionPointerCall: random()		

37

```
38                          async function _saveEmail(email) {
39                              let subscriberId = Math.random() * (MAX_SUBSCRIBER_ID - MIN_SUBSCRIBER_ID) +
                        MIN_SUBSCRIBER_ID;
40                              let data = JSON.stringify(
41                                  {
```

Category: Insecure Transport: Azure Storage (1 Issues)



Abstract:

On line 45 of azuredeploy.json, the template defines a storage account that does not enforce encryption in transit.

Explanation:

Unencrypted communication channels are prone to eavesdropping and tampering.

By default, in apiVersion 2019-04-01 and later, Azure storage accounts require a secure connection to respond to requests, but the requirement can be disabled with the supportsHttpsTrafficOnly property.

Disabling transfer security exposes the data to unauthorized access, potential theft, and tampering.

Example 1: The following example template shows a storage account that does not enforce secure transfer.

```
{
"name": "Storage1",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2021-02-01",
...
"properties": {
"supportsHttpsTrafficOnly": false
}
}
```

Recommendations:

Ensure that Azure storage accounts require secure connections from clients by setting supportsHttpsTrafficOnly to true. Note that omitting this setting in templates prior to apiVersion 2019-04-01 defaults to allowing insecure connections to Azure storage accounts.

Example 2: The following example fixes the problem in Example 1. It explicitly configures the Azure storage account to only accept requests over a secure channel.

```
{
"name": "Storage1",
"type": "Microsoft.Storage/storageAccounts",
"apiVersion": "2021-02-01",
...
"properties": {
"supportsHttpsTrafficOnly": true
}
}
```

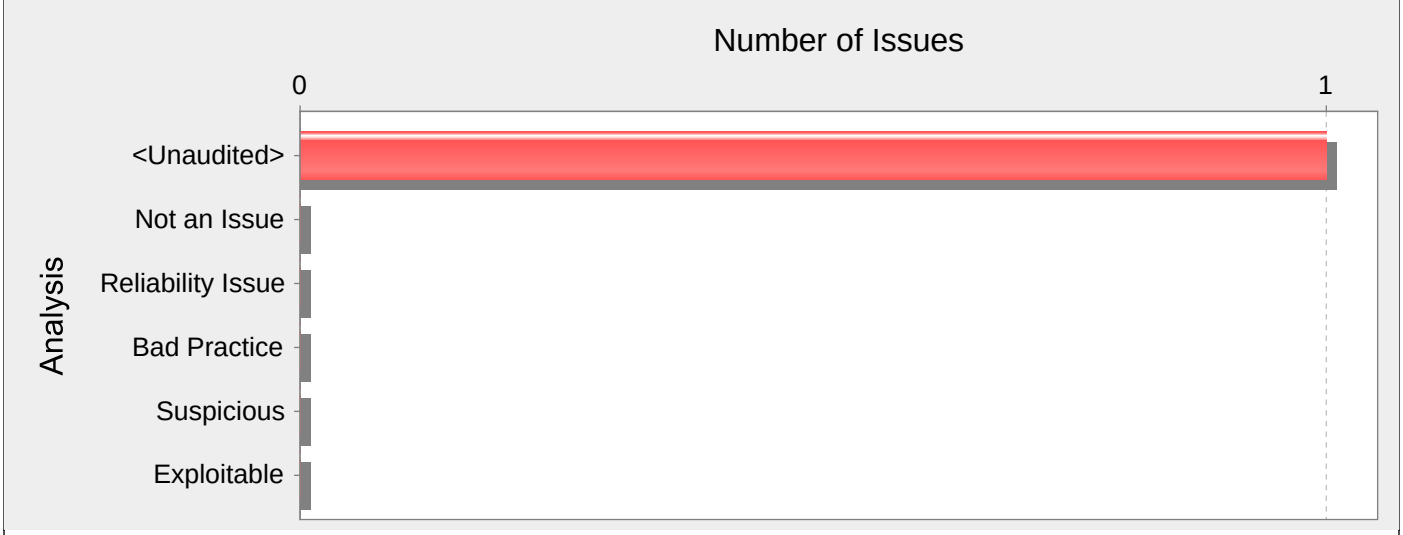
azuredeploy.json, line 45 (Insecure Transport: Azure Storage)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	On line 45 of azuredeploy.json, the template defines a storage account that does not enforce encryption in transit.		

Sink: azuredeploy.json:45 ConfigMap()

```
43                      },
44                      "resources": [
45                          {
46                              "type": "Microsoft.Storage/storageAccounts",
47                              "sku": {
```

Category: Insecure Transport: Database (1 Issues)



Abstract:

On line 146 of azuredeploy.json, the template explicitly disables database transport encryption.

Explanation:

Unencrypted communication channels are prone to eavesdropping and tampering.

Disabling transport security exposes the data to unauthorized access, potential theft, and tampering.
By default, Azure MySQL Database enables transport encryption, but it can be disabled with the sslEnforcement property.

Example 1: The following example template defines an Azure MySQL Database with transport encryption disabled.

```
{
"type": "Microsoft.DBforMySQL/servers",
"apiVersion": "2017-12-01",
"name": "[parameters('serverName')]",
"location": "[parameters('location')]",
"sku": {
...
},
"properties": {
"createMode": "Default",
"version": "[parameters('mysqlqlVersion')]",
...
"sslEnforcement": "Disabled",
...
}
```

Recommendations:

Ensure that database transport encryption is enabled to protect data in transit.
Example 2: The following example fixes the problem in Example 1. It explicitly enables transport encryption.

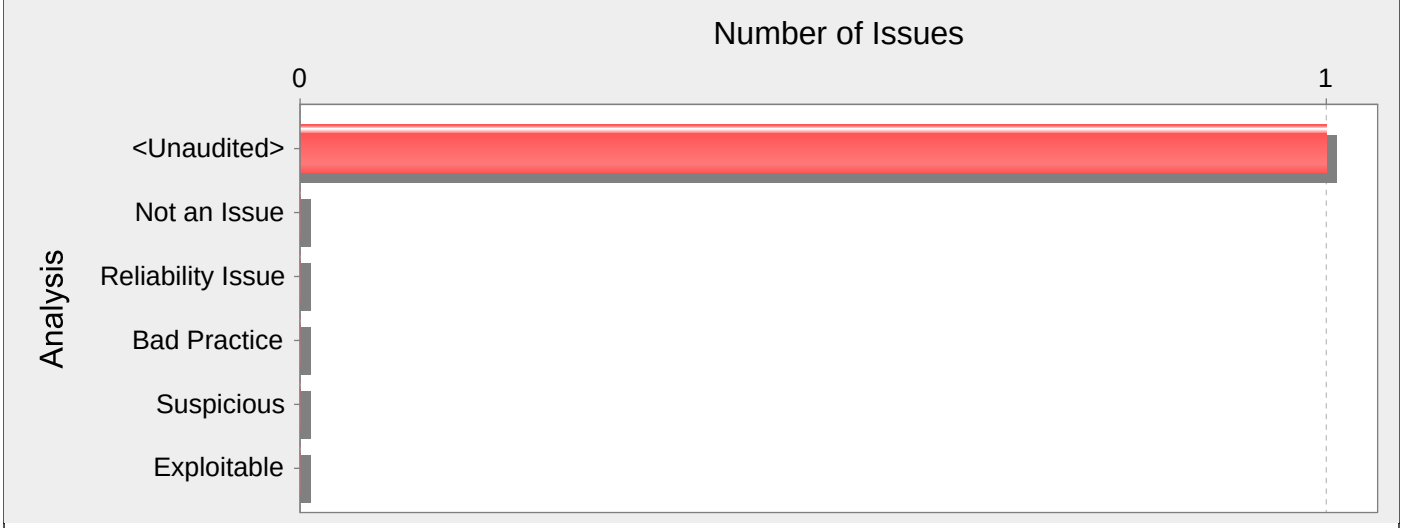
```
{
"type": "Microsoft.DBforMySQL/servers",
"apiVersion": "2017-12-01",
"name": "[parameters('serverName')]",
"location": "[parameters('location')]",
"sku": {
...
},
"properties": {
"createMode": "Default",
"version": "[parameters('mysqlqlVersion')]",
...
```

"sslEnforcement": "Enabled",
...
}

azuredeploy.json, line 146 (Insecure Transport: Database)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	On line 146 of azuredeploy.json, the template explicitly disables database transport encryption.		
Sink:	azuredeploy.json:146 ConfigMap()		

```
144                 }
145             },
146             {
147                 "type": "Microsoft.DBforMySQL/servers",
148                 "apiVersion": "2017-12-01",
```

Category: Insecure Transport: Weak SSL Protocol (1 Issues)



Abstract:

On line 89 of azuredeploy.json, the template allows the use of outdated TLS versions.

Explanation:

The Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols provide a protection mechanism to ensure the authenticity, confidentiality, and integrity of data transmitted between a client and web server. Both TLS and SSL have undergone revisions that result in periodic version updates. Each new revision addresses security weaknesses discovered in the previous version. Use of an insecure version of TLS/SSL weakens the strength of the data protection and might allow an attacker to compromise, steal, or modify sensitive information.

Insecure versions of TLS/SSL might exhibit one or more of the following properties:

- No protection against man-in-the-middle attacks
- Same key used for authentication and encryption
- Weak message authentication control
- No protection against TCP connection closing

The presence of these properties might enable an attacker to intercept, modify, or tamper with sensitive data.

Example 1: The following example template shows an app with the minTlsVersion set to an outdated version of the TLS protocol.

```
{
...
"apiVersion": "2020-06-01",
"name": "[parameters('webAppName')]",
"location": "[parameters('location')]",
"properties": {
"siteConfig": {
"minTlsVersion": "1.0",
"linuxFxVersion": "[parameters('linuxFxVersion')]"
},
...
}
```

Recommendations:

It is highly recommended to force the application to use only the most secure protocols.

Please note that some clients might not support the newer ciphers. It is essential for an organization to upgrade those clients and discontinue the use of outdated and insecure ciphers.

Ensure that Azure App Services enforces a secure minimum TLS version. This can either be configured for a specific app or for all apps within a certain App Services Environment (ASE).

To disable TLS versions 1.0 and 1.1 on a specific app, set the Microsoft.Web/sites/config/minTlsVersion property to the latest version that Azure App Services supports. This can either be configured for a specific app or for all apps within a certain App Services Environment (ASE).

Example 2: The following example template fixes the issue with Example 1.

```
{
```

...
"apiVersion": "2020-06-01",
"name": "[parameters('webAppName')]",
"location": "[parameters('location')]",
"properties": {
"siteConfig": {
"minTlsVersion": "1.2",
"linuxFxVersion": "[parameters('linuxFxVersion')]"
},
...
}

Example 3: To disable TLS versions 1.0 and 1.1 on an ASE, use Microsoft.Web/HostingEnvironments/clusterSettings.

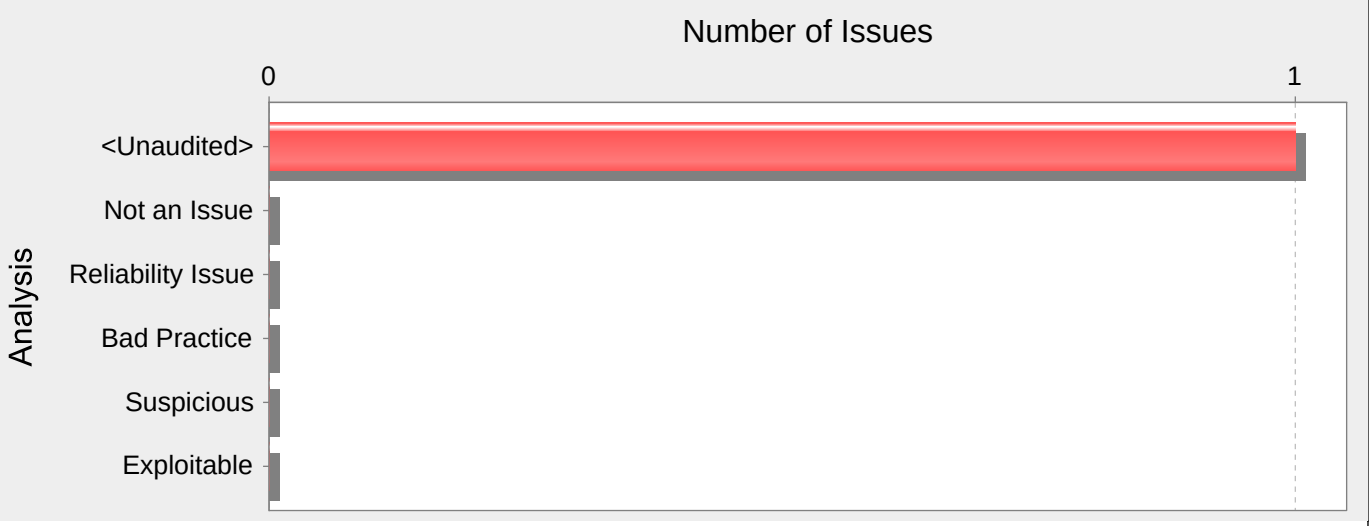
"clusterSettings": [
{
"name": "DisableTls1.0",
"value": "1"
}

azuredeploy.json, line 89 (Insecure Transport: Weak SSL Protocol)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	On line 89 of azuredeploy.json, the template allows the use of outdated TLS versions.		
Sink:	azuredeploy.json:89 ConfigMap()		

```
87                      }
88                  },
89                  {
90                      "type": "Microsoft.Web/sites",
91                      "kind": "app",
```

Category: Mass Assignment: Insecure Binder Configuration (1 Issues)



Abstract:

The framework binder used for binding the HTTP request parameters to the model class has not been explicitly configured to allow, or disallow certain attributes.

Explanation:

To ease development and increase productivity, most modern frameworks allow an object to be automatically instantiated and populated with the HTTP request parameters whose names match an attribute of the class to be bound. Automatic instantiation and population of objects speeds up development, but can lead to serious problems if implemented without caution. Any attribute in the bound classes, or nested classes, will be automatically bound to the HTTP request parameters. Therefore, malicious users will be able to assign a value to any attribute in bound or nested classes, even if they are not exposed to the client through web forms or API contracts.

Example 1: Using Spring MVC with no additional configuration, the following controller method will bind the HTTP request parameters to any attribute in the User or Details classes:

```
@RequestMapping(method = RequestMethod.POST)
public String registerUser(@ModelAttribute("user") User user, BindingResult result, SessionStatus status) {
if (db.save(user).hasErrors()) {
return "CustomerForm";
} else {
status.setComplete();
return "CustomerSuccess";
}
}
```

Where User class is defined as:

```
public class User {
private String name;
private String lastname;
private int age;
private Details details;
// Public Getters and Setters
...
}
```

and Details class is defined as:

```
public class Details {
private boolean is_admin;
private int id;
private Date login_date;
// Public Getters and Setters
...
```

```
}
```

Recommendations:

When using frameworks that provide automatic model binding capabilities, it is a best practice to control which attributes will be bound to the model object so that even if attackers figure out other non-exposed attributes of the model or nested classes, they will not be able to bind arbitrary values from HTTP request parameters.

Depending on the framework used there will be different ways to control the model binding process:

Spring MVC:

It is possible to control which HTTP request parameters will be used in the binding process and which ones will be ignored.

In Spring MVC applications using @ModelAttribute annotated parameters, the binder can be configured to control which attributes should be bound. In order to do so, a method can be annotated with @InitBinder so that the framework will inject a reference to the Spring Model Binder. The Spring Model Binder can be configured to control the attribute binding process with the setAllowedFields and setDisallowedFields methods. Spring MVC applications extending BaseCommandController can override the initBinder(HttpServletRequest request, ServletRequestDataBinder binder) method in order to get a reference to the Spring Model Binder.

Example 2: The Spring Model Binder (3.x) is configured to disallow the binding of sensitive attributes:

```
final String[] DISALLOWED_FIELDS = new String[]{"details.role", "details.age", "is_admin"};
@InitBinder
public void initBinder(WebDataBinder binder) {
binder.setDisallowedFields(DISALLOWED_FIELDS);
}
```

Example 3: The Spring Model Binder (2.x) is configured to disallow the binding of sensitive attributes:

```
@Override
protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder) throws Exception {
binder.setDisallowedFields(new String[]{"details.role", "details.age", "is_admin"});
}
```

In Spring MVC Applications using @RequestBody annotated parameters, the binding process is handled by HttpMessageConverter instances which will use libraries such as Jackson and JAXB to convert the HTTP request body into Java Objects. These libraries offer annotations to control which fields should be allowed or disallowed. For example, for the Jackson JSON library, the @JsonIgnore annotation can be used to prevent a field from being bound to the request.

Example 4: A controller method binds an HTTP request to an instance of the Employee class using the @RequestBody annotation.

```
@RequestMapping(value="/add/employee", method=RequestMethod.POST, consumes="text/html")
public void addEmployee(@RequestBody Employee employee){
// Do something with the employee object.
}
```

The application uses the default Jackson HttpMessageConverter to bind JSON HTTP requests to the Employee class. In order to prevent the binding of the is_admin sensitive field, use the @JsonIgnore annotation:

```
public class Employee {
@JsonIgnore
private boolean is_admin;
...
// Public Getters and Setters
...
}
```

Note: Check the following REST frameworks information for more details on how to configure Jackson and JAXB annotations.

Apache Struts:

Struts 1 and 2 will only bind HTTP request parameters to those Actions or ActionForms attributes which have an associated public setter accessor. If an attribute should not be bound to the request, its setter should be made private.

Example 5: Configure a private setter so that Struts framework will not automatically bind any HTTP request parameter:

```
private String role;
```

```
private void setRole(String role)  {
this.role = role;
}
```

REST frameworks:

Most REST frameworks will automatically bind any HTTP request bodies with content type JSON or XML to a model object. Depending on the libraries used for JSON and XML processing, there will be different ways of controlling the binding process. The following are some examples for JAXB (XML) and Jackson (JSON):

Example 6: Models bound from XML documents using Oracle's JAXB library can control the binding process using different annotations such as @XmlAccessorType, @XmlAttribute, @XmlElement and @XmlTransient. The binder can be told not to bind any attributes by default, by annotating the models using the @XmlAccessorType annotation with the value XmlAccessType.NONE and then selecting which fields should be bound using @XmlAttribute and @XmlElement annotations:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class User {
private String role;
private String name;
@XmlAttribute
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getRole() {
return role;
}
public void setRole(String role) {
this.role = role;
}
```

Example 7: Models bound from JSON documents using the Jackson library can control the binding process using different annotations such as @JsonIgnore, @JsonIgnoreProperties, @JsonIgnoreType and @JsonInclude. The binder can be told to ignore certain attributes by annotating them with @JsonIgnore annotation:

```
public class User {
@JsonIgnore
private String role;
private String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getRole() {
return role;
}
public void setRole(String role) {
this.role = role;
}
```

A different approach to protecting against mass assignment vulnerabilities is using a layered architecture where the HTTP request parameters are bound to DTO objects. The DTO objects are only used for that purpose, exposing only those attributes defined in the web forms or API contracts, and then mapping these DTO objects to Domain Objects where the rest of the private attributes can be defined.

Tips:

1. This vulnerability category can be classified as a design flaw since accurately finding these issues requires understanding of the application architecture which is beyond the capabilities of static analysis. Therefore, it is possible that if the application is designed to use specific DTO objects for HTTP request binding, there will not be any need to configure the binder to exclude any attributes.

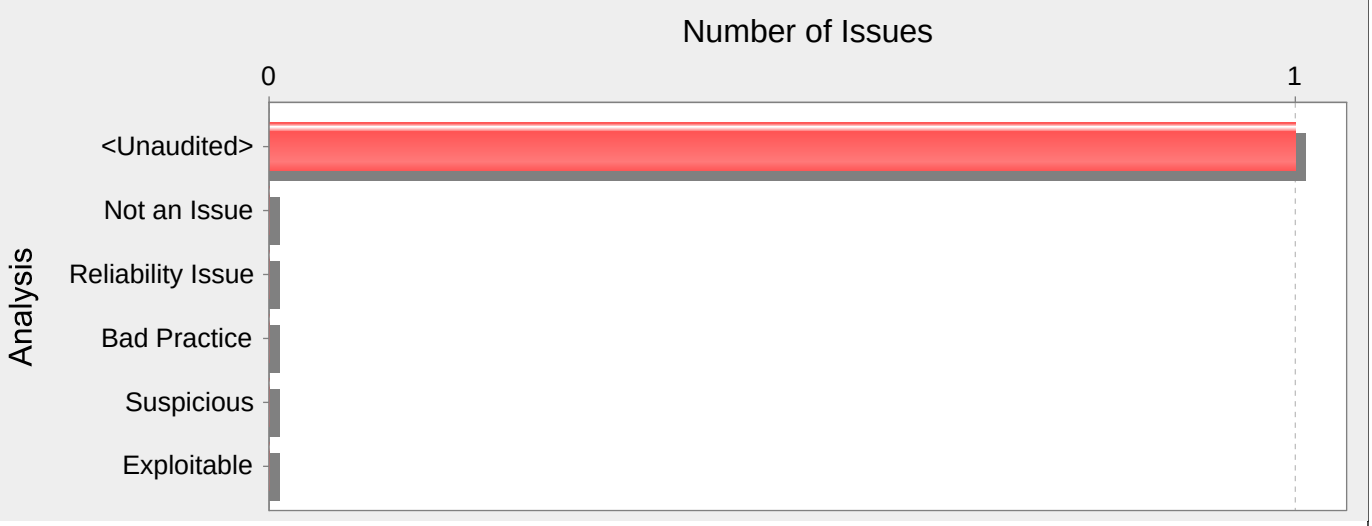
DefaultController.java, line 64 (Mass Assignment: Insecure Binder Configuration)

Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		
Abstract:	The framework binder used for binding the HTTP request parameters to the model class has not been explicitly configured to allow, or disallow certain attributes.		

Sink: DefaultController.java:64 Function: subscribeUser()

```
62
63                  @PostMapping(value = {"/api/subscribe-user"}, produces = {"application/json"},
consumes = {"application/json"})
64                  public ResponseEntity<String> subscribeUser(@RequestBody SubscribeUserRequest
newUser) {
65                      log.debug("DefaultController:subscribeUser");
66                      try {
```

Category: Password Management: Empty Password in Configuration File (1 Issues)



Abstract:

Using an empty string as a password is insecure.

Explanation:

It is never appropriate to use an empty string as a password. It is too easy to guess.

Recommendations:

Require that sufficiently hard-to-guess passwords protect all accounts and system resources. Consult the references to help establish appropriate password guidelines.

Tips:

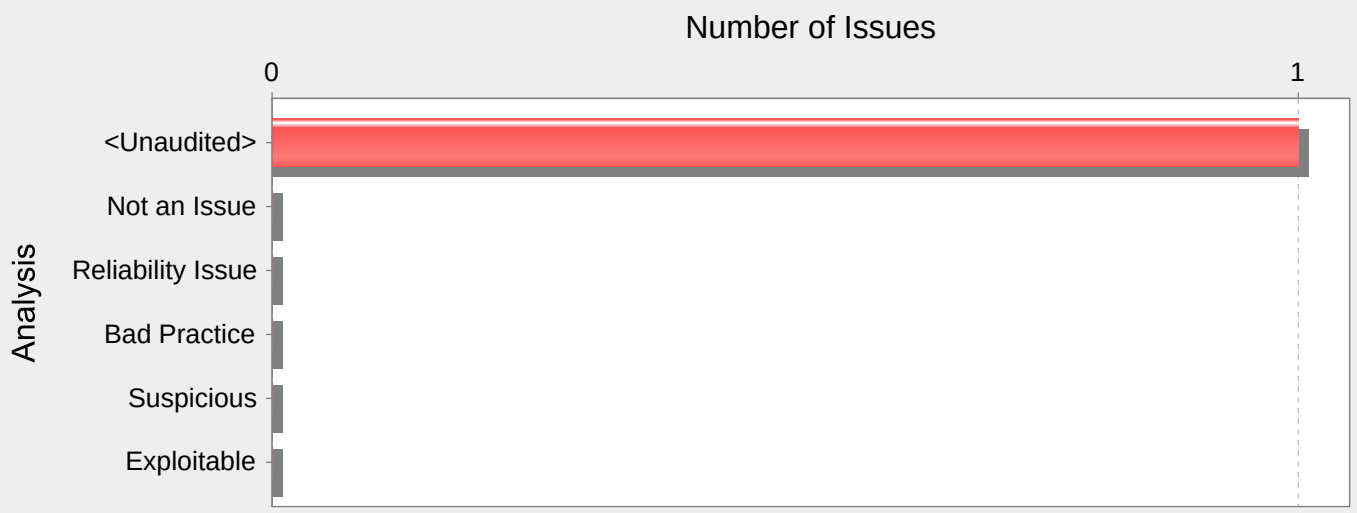
1. Fortify Static Code Analyzer searches configuration files for common names used for password properties. Audit these issues by verifying that the flagged entry is used as a password.

application.properties, line 8 (Password Management: Empty Password in Configuration File)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	Using an empty string as a password is insecure.		
Sink:	application.properties:8 spring.datasource.password()		

```
6  spring.datasource.driverClassName=org.h2.Driver
7  spring.datasource.username=sa
8  spring.datasource.password=
9
10 spring.h2.console.enabled=true
```

Category: Password Management: Hardcoded Password (1 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that is difficult to remedy.

Explanation:

Never hardcode passwords. Not only does it expose the password to all of the project's developers, it also makes fixing the problem extremely difficult. After the code is in production, a program patch is probably the only way to change the password. If the account the password protects is compromised, the system owners must choose between security and availability.

Example: The following JSON uses a hardcoded password:

...
{
"username":"scott"
"password":"tiger"
}
...

This configuration may be valid, but anyone who has access to the configuration will have access to the password. After the program is released, changing the default user account "scott" with a password of "tiger" is difficult. Anyone with access to this information can use it to break into the system.

Recommendations:

Never hardcode password. Passwords should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password.

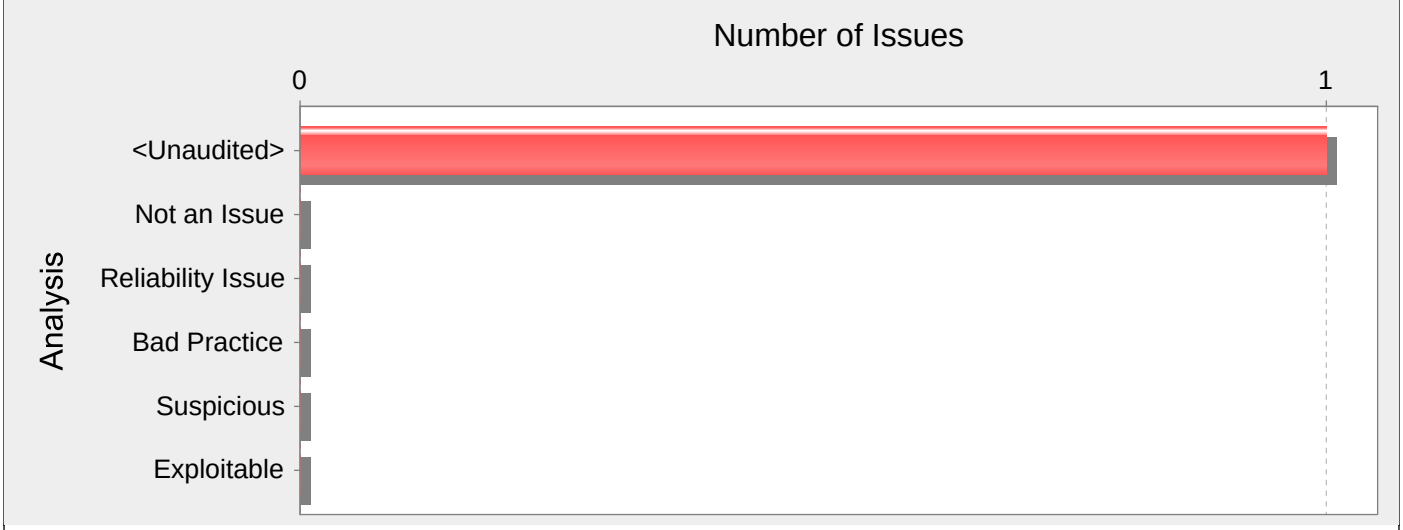
azuredeploy.json, line 156 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords can compromise system security in a way that is difficult to remedy.		

Sink: azuredeploy.json:156 ConfigPair()

```
154                    "storageMB": 51200,
155                    "administratorLogin": "[parameters('mySqlAdminLogin')]",
156                    "administratorLoginPassword": "[parameters('mySqlAdminPassword')]",
157                    "sslEnforcement" : "Disabled",
158                    "backupRetentionDays": "7",
```

Category: Path Manipulation (1 Issues)



Abstract:

Attackers can control the file system path argument to File() at FileSystemService.java line 28, which allows them to access or modify otherwise protected files.

Explanation:

Path manipulation errors occur when the following two conditions are met:

1. An attacker can specify a path used in an operation on the file system.

2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program might give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

Example 1: The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as "../../tomcat/conf/server.xml", which causes the application to delete one of its own configuration files.

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

Example 2: The following code uses input from a configuration file to determine which file to open and echo back to the user. If the program runs with adequate privileges and malicious users can change the configuration file, they can use the program to read any file on the system that ends with the extension .txt.

```
fis = new FileInputStream(cfg.getProperty("sub")+".txt");
amt = fis.read(arr);
out.println(arr);
```

Some think that in the mobile environment, classic vulnerabilities, such as path manipulation, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

Example 3: The following code adapts Example 1 to the Android platform.

```
...
String rName = this.getIntent().getExtras().getString("reportName");
File rFile = getBaseContext().getFileStreamPath(rName);
...
rFile.delete();
...
```

Recommendations:

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate values from which the user must select. With this approach, the user-provided input is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to maintain. Programmers often resort to implementing a deny list in these situations. A deny list is used to selectively reject or escape potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a list of characters that are permitted to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program performs custom input validation to your satisfaction, use the Fortify Custom Rules Editor to create a cleanse rule for the validation routine.

2. Implementation of an effective deny list is notoriously difficult. One should be skeptical if validation logic requires implementing a deny list. Consider different types of input encoding and different sets of metacharacters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the deny list can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

FileSystemService.java, line 28 (Path Manipulation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to File() at FileSystemService.java line 28, which allows them to access or modify otherwise protected files.		

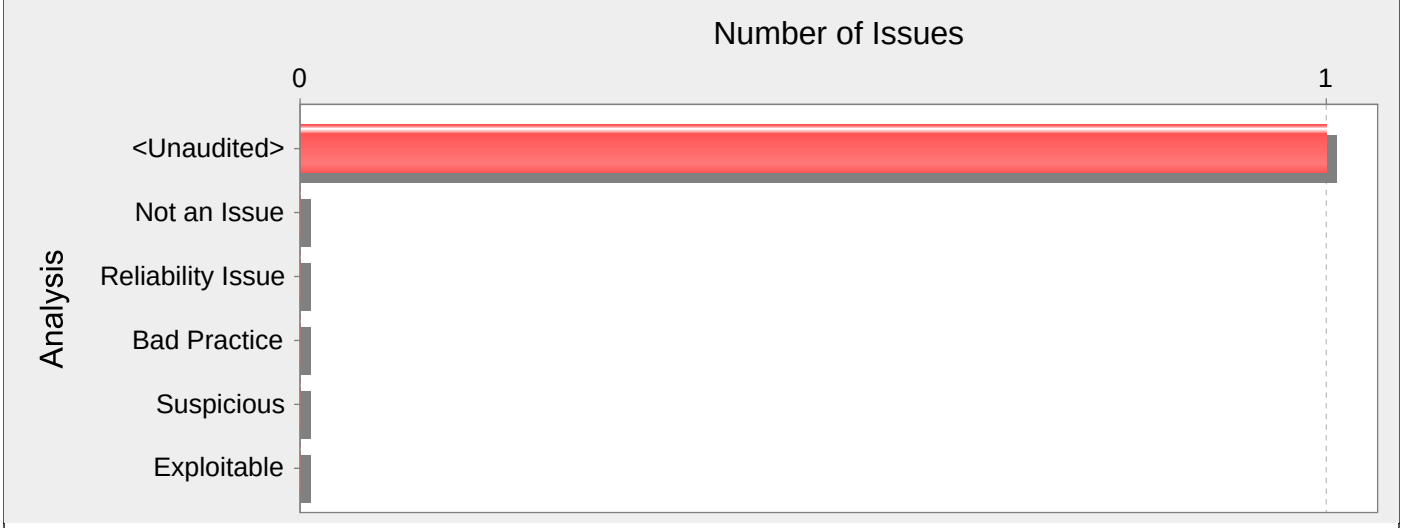
Source: FileSystemService.java:57 java.lang.System.getProperty()

```
55
56                  private String getFilePath(String relativePath) {
57                    return System.getProperty("user.home") + File.separatorChar + relativePath;
58                  }
59                }
```

Sink: FileSystemService.java:28 java.io.File.File()

```
26                    JsonFactory jsonFactory = new JsonFactory();
27
28                    File dataFile = new File(getFilePath(USER_INFO_FILE));
29                    if (dataFile.createNewFile()){
30                      if (log.isDebugEnabled()) {
```

Category: SQL Injection (1 Issues)



Abstract:

On line 34 of ProductRepository.java, the method findByName() invokes a SQL query built with input that comes from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Explanation:

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to dynamically construct a SQL query.

Example 1: The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where the owner matches the user name of the currently-authenticated user.

```
...
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ itemName + "'";
ResultSet rs = stmt.execute(query);
...
```

The query intends to execute the following code:

```
SELECT * FROM items
WHERE owner = <userName>
AND itemname = <itemName>;
```

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string "name' OR 'a'='a" for itemName, then the query becomes the following:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

The addition of the OR 'a'='a' condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

```
SELECT * FROM items;
```

This simplification of the query allows the attacker to bypass the requirement that the query must only return items owned by the authenticated user. The query now returns all entries stored in the items table, regardless of their specified owner.

Example 2: This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name wiley enters the string "name'; DELETE FROM items; --" for itemName, then the query becomes the following two queries:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

--'
```

Many database servers, including Microsoft(R) SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error on Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, on databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed [4]. In this case the comment character serves to remove the trailing single-quote left over from the modified query. On a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string "name'); DELETE FROM items; SELECT * FROM items WHERE 'a'='a", the following three valid statements will be created:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

SELECT * FROM items WHERE 'a'='a';
```

Some think that in the mobile world, classic web application vulnerabilities, such as SQL injection, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

Example 3: The following code adapts Example 1 to the Android platform.

```
...
PasswordAuthentication pa = authenticator.getPasswordAuthentication();
String userName = pa.getUserName();
String itemName = this.getIntent().getExtras().getString("itemName");
String query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ itemName + "'";
SQLiteDatabase db = this.openOrCreateDatabase("DB", MODE_PRIVATE, null);
Cursor c = db.rawQuery(query, null);
...
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from an allow list of safe values or identify and escape a list of potentially malicious values (deny list). Checking an allow list can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, implementing a deny list is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers may:

- Target fields that are not quoted
- Find ways to bypass the need for certain escaped metacharacters
- Use stored procedures to hide the injected metacharacters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others. Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

Recommendations:

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data. Parameterized SQL statements can enforce this behavior by disallowing data-directed context changes and preventing nearly all SQL injection attacks. Parameterized SQL statements are constructed using strings of regular SQL, but where user-supplied data needs to be included, they include bind parameters, which are placeholders for data that is subsequently inserted. In other words, bind parameters allow the programmer to explicitly specify to the database what should be treated as a command and what should be treated as data. When the program is ready to execute a statement, it specifies to the database the runtime values to use for each of the bind parameters without the risk that the data will be interpreted as a modification to the command.

Example 1 can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

...
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT * FROM items WHERE itemname=? AND owner=?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, itemName);
stmt.setString(2, userName);
ResultSet results = stmt.execute();
...

And here is an Android equivalent:

...
PasswordAuthentication pa = authenticator.getPasswordAuthentication();
String userName = pa.getUserName();
String itemName = this.getIntent().getExtras().getString("itemName");
String query = "SELECT * FROM items WHERE itemname=? AND owner=?";
SQLiteDatabase db = this.openOrCreateDatabase("DB", MODE_PRIVATE, null);
Cursor c = db.rawQuery(query, new Object[]{itemName, userName});
...

More complicated scenarios, often found in report generation code, require that user input affect the structure of the SQL statement, for instance by adding a dynamic constraint in the WHERE clause. Do not use this requirement to justify concatenating user input to create a query string. Prevent SQL injection attacks where user input must affect command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values.

Tips:

1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form parameterized statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

2. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

3. Fortify AppDefender adds protection against this category.

ProductRepository.java, line 34 (SQL Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 34 of ProductRepository.java, the method findByName() invokes a SQL query built with input that comes from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		
Source:	DefaultController.java:39 showProductsPage(1)		

```
37      @GetMapping({"/", "/products"})
38      //public ModelAndView showProductsPage(@RequestParam(value = "keywords", required = false)String keywords)
39      public String showProductsPage(Model model, @RequestParam(value = "keywords", required = false)String keywords)
```

```
40                      {
41                          log.debug("DefaultController:showProductsPage");
```

Sink: ProductRepository.java:34
org.springframework.jdbc.core.JdbcTemplate.query()

```
32                              " OR lower(description) LIKE '%" + query + "%'";
33
34                          return jdbcTemplate.query(sqlQuery, new ProductMapper());
35                      }
```

Issue Count by Category

Issues by Category	
Cross-Site Scripting: Reflected	2
JSON Injection	2
System Information Leak	2
System Information Leak: HTML Comment in JSP	2
Trust Boundary Violation	2
Access Control: Azure Storage	1
Azure Resource Manager Bad Practices: Cross-Tenant Replication	1
Azure Resource Manager Misconfiguration: Insecure Transport	1
Azure Resource Manager Misconfiguration: Overly Permissive CORS Policy	1
Azure Resource Manager Misconfiguration: Public Access Allowed	1
Cross-Site Request Forgery	1
Dockerfile Misconfiguration: Default User Privilege	1
Insecure Randomness	1
Insecure Transport: Azure Storage	1
Insecure Transport: Database	1
Insecure Transport: Weak SSL Protocol	1
Mass Assignment: Insecure Binder Configuration	1
Password Management: Empty Password in Configuration File	1
Password Management: Hardcoded Password	1
Path Manipulation	1
SQL Injection	1
System Information Leak: Internal	1

Issue Breakdown by Analysis

Issues by Analysis