Fortify Security Report

14 Mar 2022

Demo User

Executive Summary

Issues Overview

On 14 Mar 2022, a source code review was performed over the FortifyDemoApp code base. 18 files, 274 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 15 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Low	8
High	4
Critical	3

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/src
Number of Files: 18
Lines of Code: 274
Build Label: SNAPSHOT

Scan Information

Scan time: 00:34
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.WebAppDemo.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

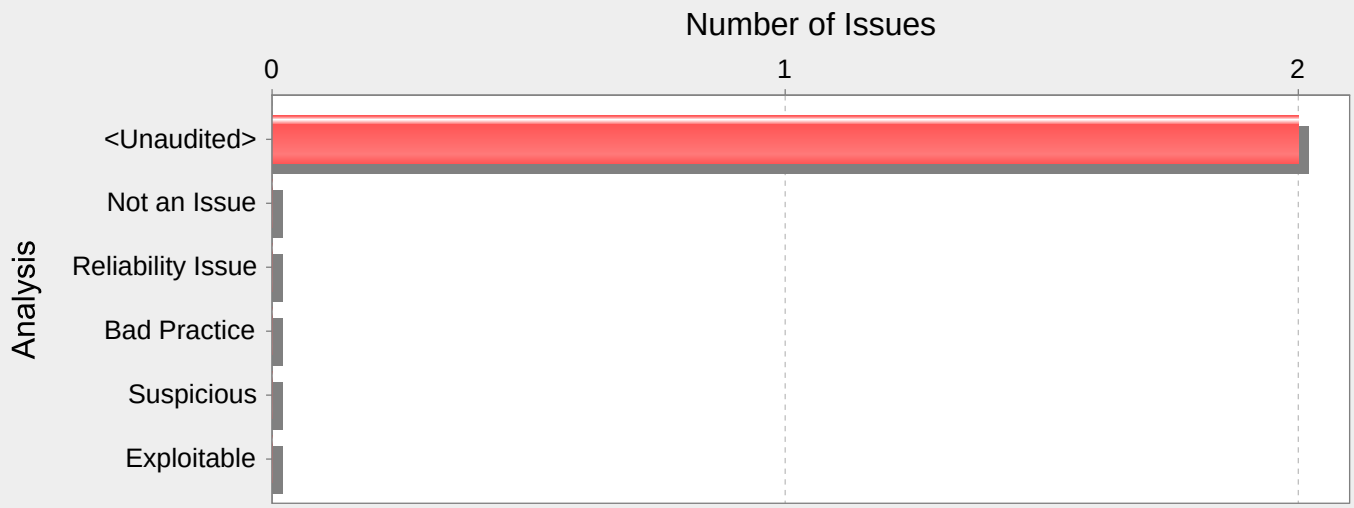
Results Outline

Overall number of results

The scan found 15 issues.

Vulnerability Examples by Category

Category: JSON Injection (2 Issues)



Abstract:

On line 41 of FsService.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();
```

FsService.java, line 41 (JSON Injection)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 41 of FsService.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.		

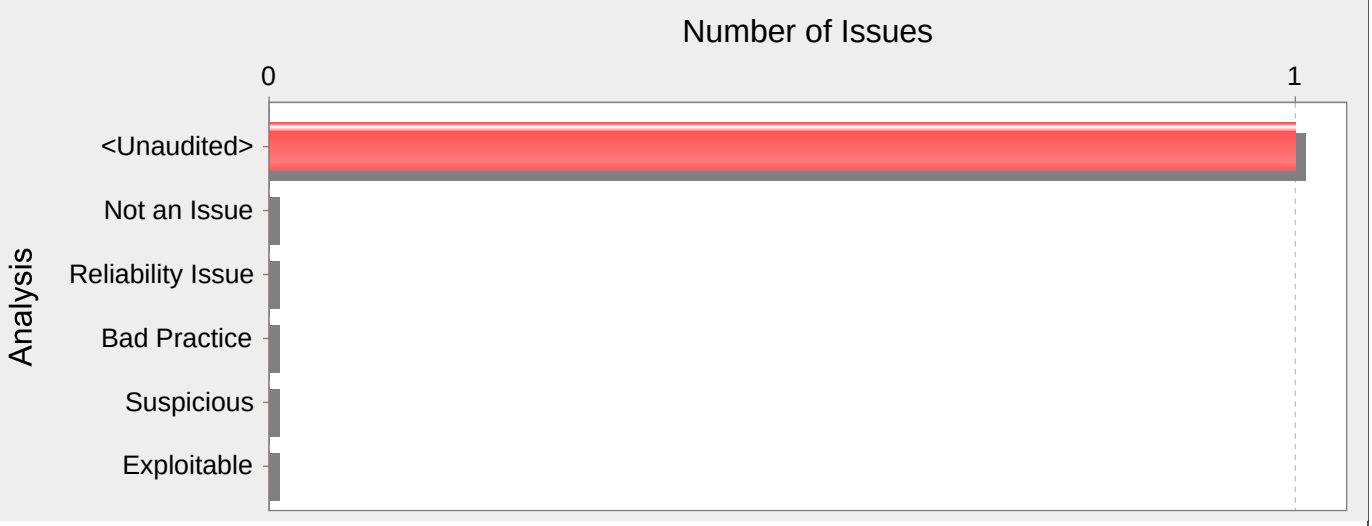
Source: DefaultController.java:60 subscribeUser(0)

```
58
59      @PostMapping(value = {"/api/subscribe-user"}, produces = {"application/json"},
consumes = {"application/json"})
60      public ResponseEntity<String> subscribeUser(@RequestBody SubscribeUserRequest
newUser) {
61          log.debug("API::Subscribing a user to the newsletter: " + newUser.toString());
62          try {
```

Sink: FsService.java:41 com.fasterxml.jackson.core.JsonGenerator.writeRawValue()

```
39
40    jGenerator.writeFieldName("name");
41    jGenerator.writeRawValue("\"" + user + "\"");
42
43    jGenerator.writeFieldName("email");
```

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.

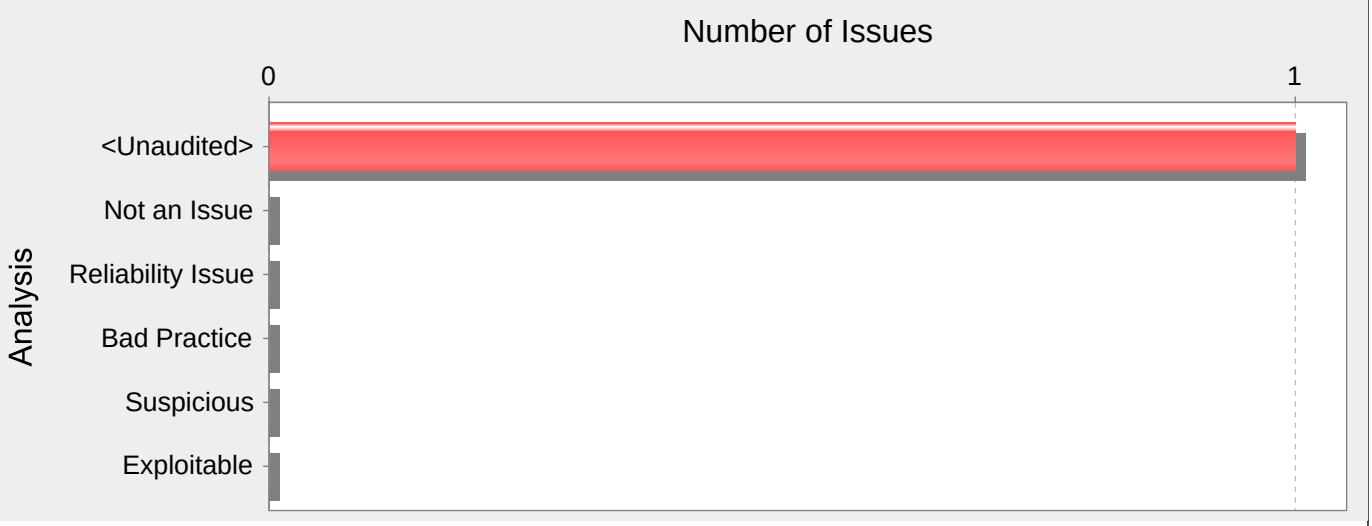
SubscribeNewsletter.js, line 39 (Insecure Randomness)

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: 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.

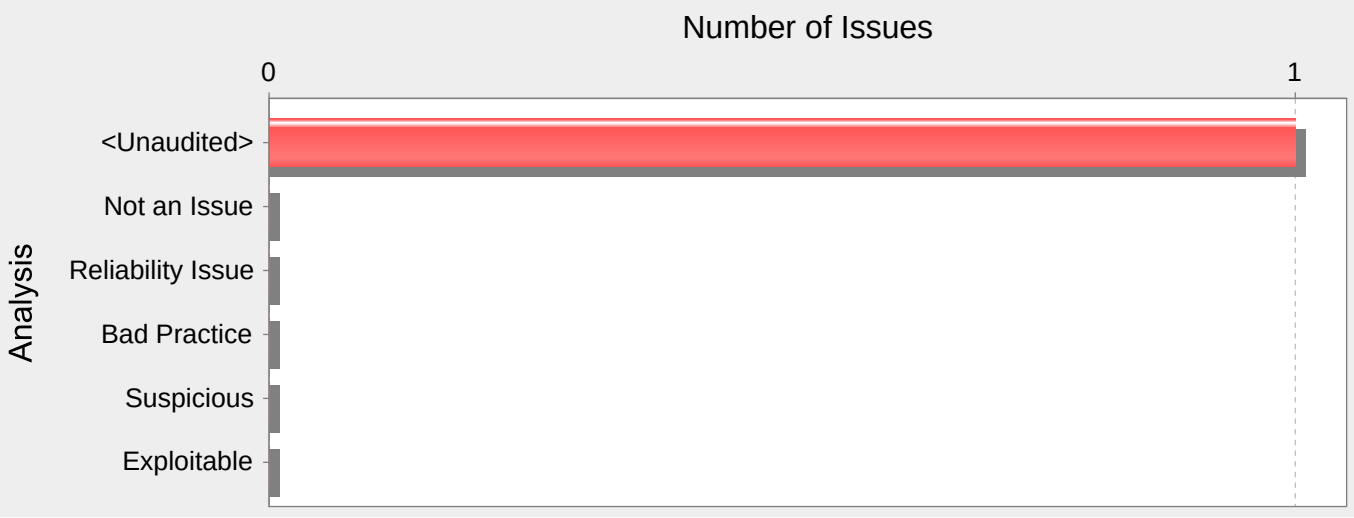
DefaultController.java, line 60 (Mass Assignment: Insecure Binder Configuration)

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:60 Function: subscribeUser()

```
58
59                  @PostMapping(value = {"/api/subscribe-user"}, produces = {"application/json"},
consumes = {"application/json"})
60                  public ResponseEntity<String> subscribeUser(@RequestBody SubscribeUserRequest
newUser) {
61                      log.debug("API::Subscribing a user to the newsletter: " + newUser.toString());
62                      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:

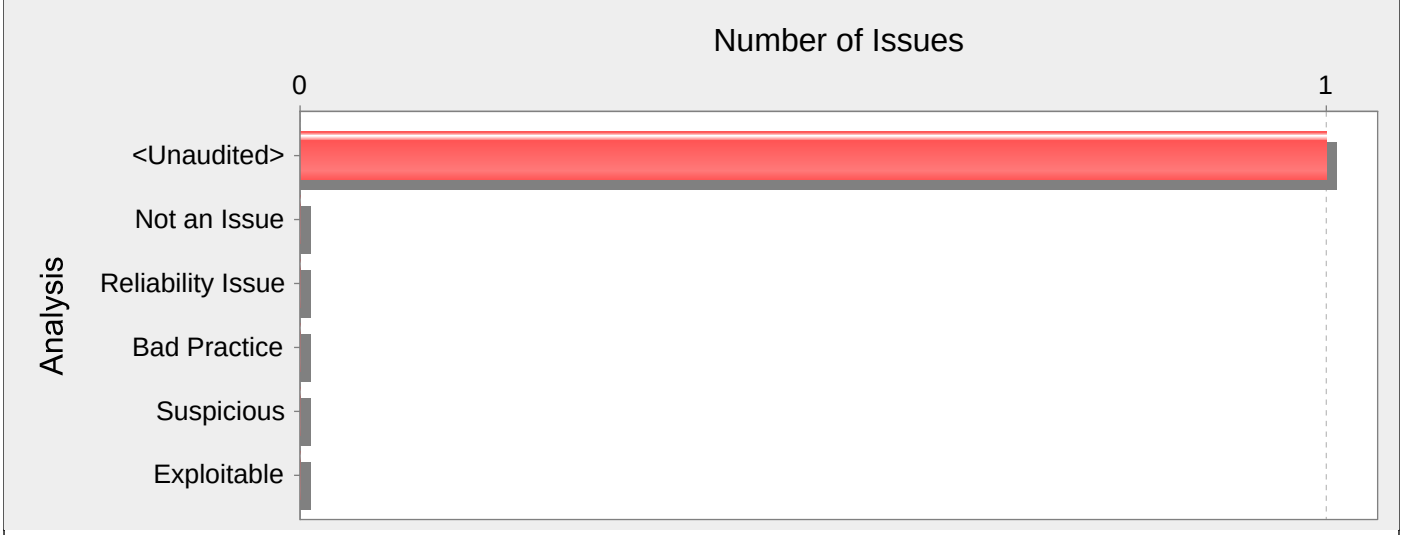
1. Fortify Static Code Analyzer searches configuration files for common names used for password properties. Audit these issues by verifying that the flagged entry is used as a password.

application.properties, line 9 (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:9 spring.datasource.password()		

```
7  spring.datasource.driverClassName=org.h2.Driver
8  spring.datasource.username=sa
9  spring.datasource.password=
10
11 ## Enables H2 console (A simple web interface to access H2 DB)
```

Category: Path Manipulation (1 Issues)



Abstract:

Attackers can control the file system path argument to File() at FsService.java line 26, 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.

FsService.java, line 26 (Path Manipulation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to File() at FsService.java line 26, which allows them to access or modify otherwise protected files.		

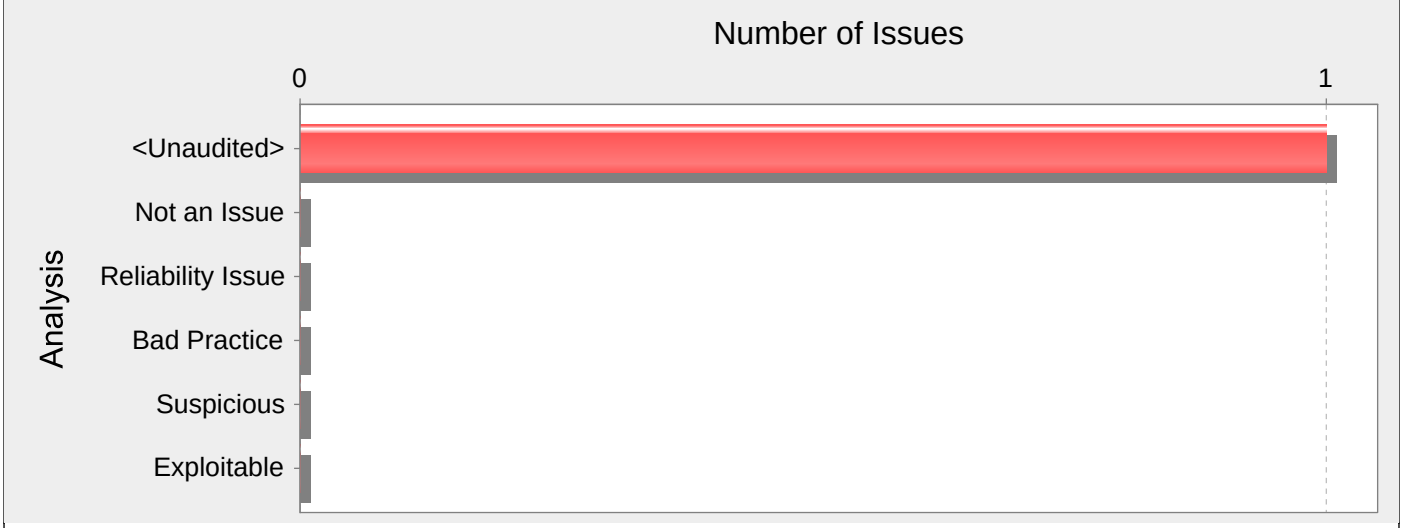
Source: FsService.java:55 java.lang.System.getProperty()

```
53
54                  private static String getFilePath(String relativePath) {
55                    return System.getProperty("user.home") + File.separatorChar + relativePath;
56                  }
57                }
```

Sink: FsService.java:26 java.io.File.File()

```
24                    JsonFactory jsonFactory = new JsonFactory();
25
26                    File dataFile = new File(getFilePath(USER_INFO_FILE));
27                    if (dataFile.createNewFile()){
28                      if (log.isDebugEnabled()) {
```

Category: SQL Injection (1 Issues)



Abstract:

On line 39 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 39 (SQL Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 39 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:34 showProductsPage(0)		

```
32
33      @GetMapping({"/", "/products"})
34      public ModelAndView showProductsPage(@RequestParam(value = "keywords", required = false)String keywords)
```

```
35                      {
36                          log.info("DefaultController:showProductsPage");
```

Sink: ProductRepository.java:39
org.springframework.jdbc.core.JdbcTemplate.query()

```
37                                  " OR lower(description) LIKE '%" + query + "%'";
38
39                          return jdbcTemplate.query(sqlQuery, new ProductMapper());
40                      }
```

Issue Count by Category

Issues by Category	
JSON Injection	2
System Information Leak	2
System Information Leak: HTML Comment in JSP	2
Trust Boundary Violation	2
Cross-Site Request Forgery	1
Insecure Randomness	1
Mass Assignment: Insecure Binder Configuration	1
Password Management: Empty Password in Configuration File	1
Path Manipulation	1
SQL Injection	1
System Information Leak: Internal	1

Issue Breakdown by Analysis

Issues by Analysis