# Fortify Security Report

27 Apr 2022

Demo User

# Executive Summary

## Issues Overview

On 27 Apr 2022, a source code review was performed over the IWAPharmacyDirect code base. 271 files, 4,027 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 253 reviewed findings were uncovered during the analysis.

## Issues by Fortify Priority Order

| | |
|---|---|
| Low | 155 |
| High | 68 |
| Critical | 17 |
| Medium | 13 |

## 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/IWA/src/main

Number of Files: 271

Lines of Code: 4027

Build Label: SNAPSHOT

## Scan Information

Scan time: 01:12

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.example.Application.main

Environment Variables:
null.null.null
java.lang.System.getenv

File System:
java.io.FileReader.FileReader
java.io.FileReader.FileReader
java.util.zip.ZipFile.entries

Private Information:
null.null.null
java.lang.System.getenv
org.springframework.security.core.userdetails.UserDetails.getPassword

Java Properties:
null.null.null
java.lang.System.getProperty

System Information:
null.null.null
null.null.lambda
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.Throwable.getLocalizedMessage
java.lang.Throwable.getMessage
org.springframework.web.bind.MissingServletRequestParameterException.getParameterName

Web:
null.~JS_Generic.val
org.springframework.web.servlet.LocaleResolver.resolveLocale

## 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

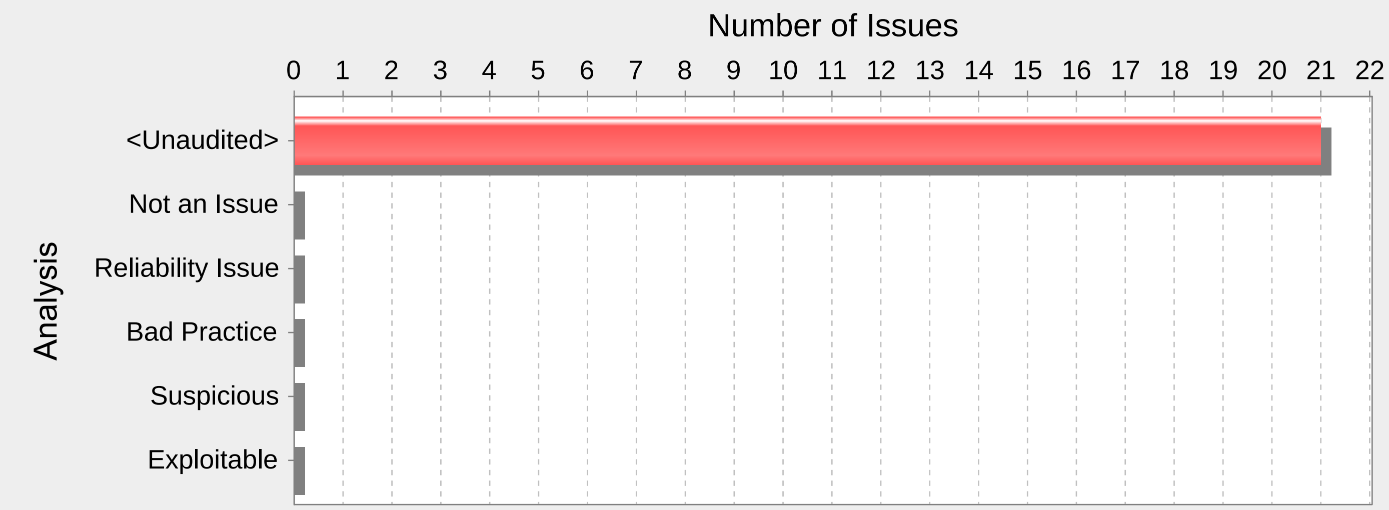
## Results Outline

### Overall number of results

The scan found 253 issues.

### Vulnerability Examples by Category

### Category: Mass Assignment: Insecure Binder Configuration (21 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:

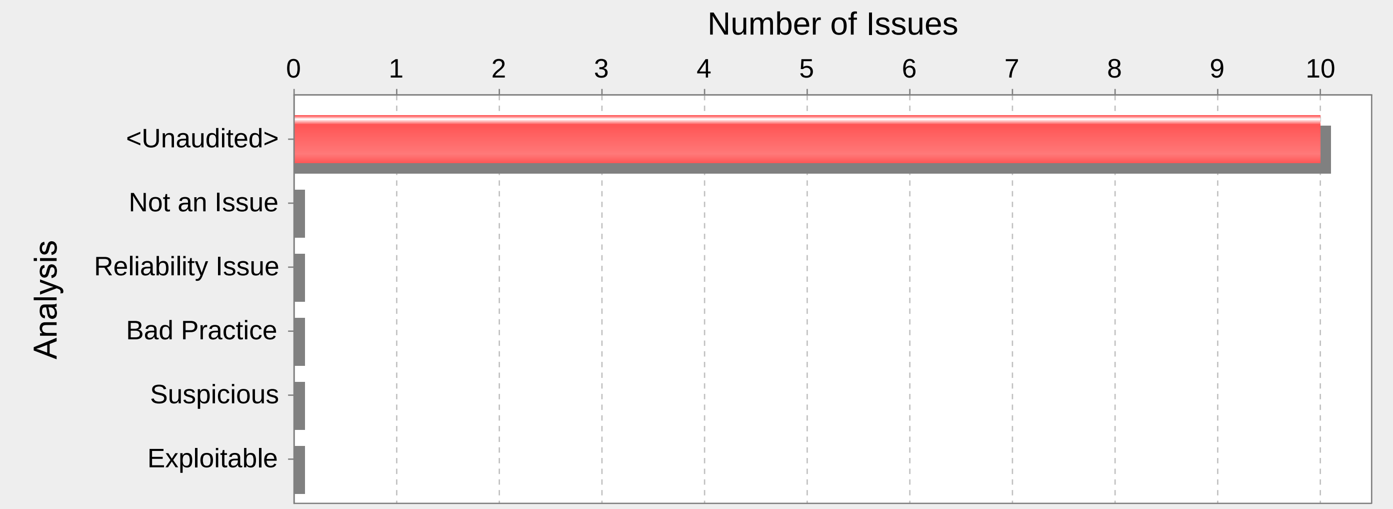
1. This vulnerability category can be classified as a design flaw since accurately finding these issues requires understanding of the application architecture which is beyond the capabilities of static analysis. Therefore, it is possible that if the application is designed to use specific DTO objects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

## ApiRoleController.java, line 114 (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:** | ApiRoleController.java:114 Function: createRole() | | |

```
112    @PostMapping(value = {""}, produces = {"application/json"}, consumes =
{"application/json"})
113    @ResponseStatus(HttpStatus.CREATED)
114    public ResponseEntity<Authority> createRole(
115            @io.swagger.v3.oas.annotations.parameters.RequestBody(description = "")
@Valid @RequestBody Authority newRole) {
116        //newRole.setId(0); // set to 0 for sequence id generation
```

## Category: Access Control: Database (10 Issues)



### Abstract:

Without proper access control, the method saveReviewFromAdminReviewForm() in ProductService.java can execute a SQL statement on line 242 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

### Explanation:

Database access control errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to specify the value of a primary key in a SQL query.

Example 1: The following code uses a parameterized statement, which escapes metacharacters and prevents SQL injection vulnerabilities, to construct and execute a SQL query that searches for an invoice matching the specified identifier [1]. The identifier is selected from a list of all invoices associated with the current authenticated user.

```
...
id = Integer.decode(request.getParameter("invoiceID"));
String query = "SELECT * FROM invoices WHERE id = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setInt(1, id);
ResultSet results = stmt.execute();
...
```

The problem is that the developer has failed to consider all of the possible values of id. Although the interface generates a list of invoice identifiers that belong to the current user, an attacker might bypass this interface to request any desired invoice. Because the code in this example does not check to ensure that the user has permission to access the requested invoice, it will display any invoice, even if it does not belong to the current user.

Some think that in the mobile world, classic web application vulnerabilities, such as database access control errors, 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 2: The following code adapts Example 1 to the Android platform.

```
...
String id = this.getIntent().getExtras().getString("invoiceID");
String query = "SELECT * FROM invoices WHERE id = ?";
SQLiteDatabase db = this.openOrCreateDatabase("DB", MODE_PRIVATE, null);
Cursor c = db.rawQuery(query, new Object[]{id});
...
```

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:

Rather than relying on the presentation layer to restrict values submitted by the user, access control should be handled by the application and database layers. Under no circumstances should a user be allowed to retrieve or modify a row in the database without the appropriate permissions. Every query that accesses the database should enforce this policy, which can often be accomplished by simply including the current authenticated username as part of the query.

Example 3: The following code implements the same functionality as Example 1 but imposes an additional constraint to verify that the invoice belongs to the currently authenticated user.

```
...
userName = ctx.getAuthenticatedUserName();
id = Integer.decode(request.getParameter("invoiceID"));
String query =
"SELECT * FROM invoices WHERE id = ? AND user = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setInt(1, id);
stmt.setString(2, userName);
ResultSet results = stmt.execute();
...
```

And here is an Android equivalent:

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

## ProductService.java, line 242 (Access Control: Database)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Security Features | | |

**Abstract:** Without proper access control, the method saveReviewFromAdminReviewForm() in ProductService.java can execute a SQL statement on line 242 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

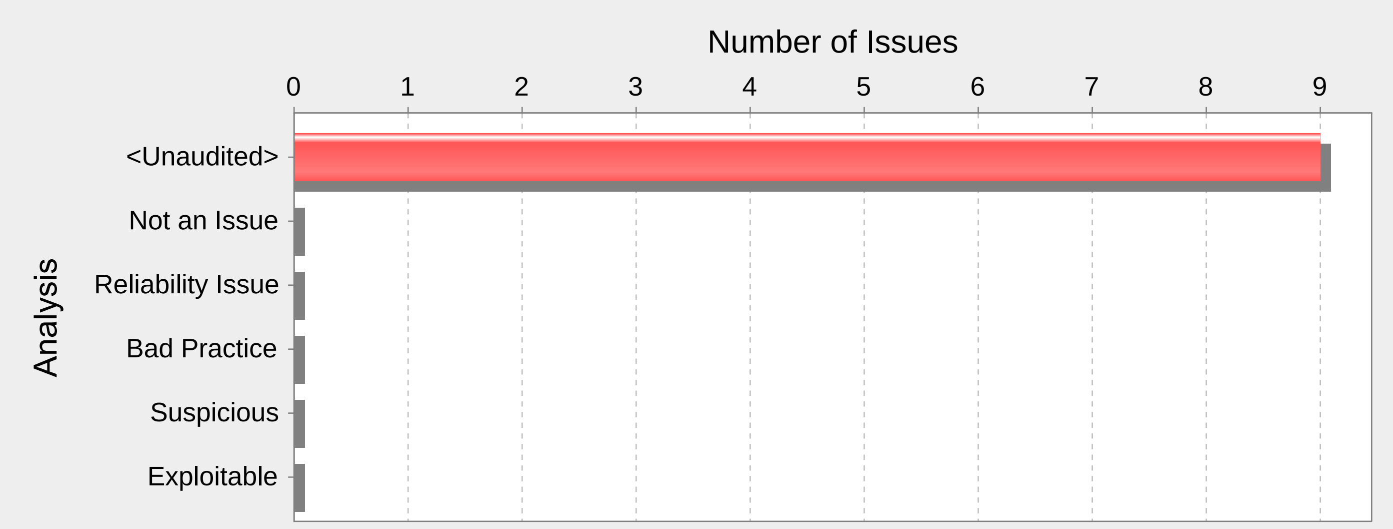
**Source:** AdminReviewController.java:100 saveReview(0)

```
98
99      @PostMapping("/{id}/save")
100     public String saveReview(@Valid @ModelAttribute("adminReviewForm") AdminReviewForm adminReviewForm,
101                              BindingResult bindingResult, Model model,
102                              RedirectAttributes redirectAttributes,
```

**Sink:** ProductService.java:242 org.springframework.data.repository.CrudRepository.findById()

```
240
241     public Review saveReviewFromAdminReviewForm(AdminReviewForm adminReviewForm) throws ReviewNotFoundException {
242         Optional<Review> optionalReview = reviewRepository.findById(adminReviewForm.getId());
243         if (optionalReview.isPresent()) {
244             Review rtmp = optionalReview.get();
```

## Category: Path Manipulation (9 Issues)



### Abstract:

Attackers can control the file system path argument to get() at FileSystemStorageService.java line 35, 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.

## FileSystemStorageService.java, line 35 (Path Manipulation)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Input Validation and Representation | | |

**Abstract:** Attackers can control the file system path argument to get() at FileSystemStorageService.java line 35, which allows them to access or modify otherwise protected files.

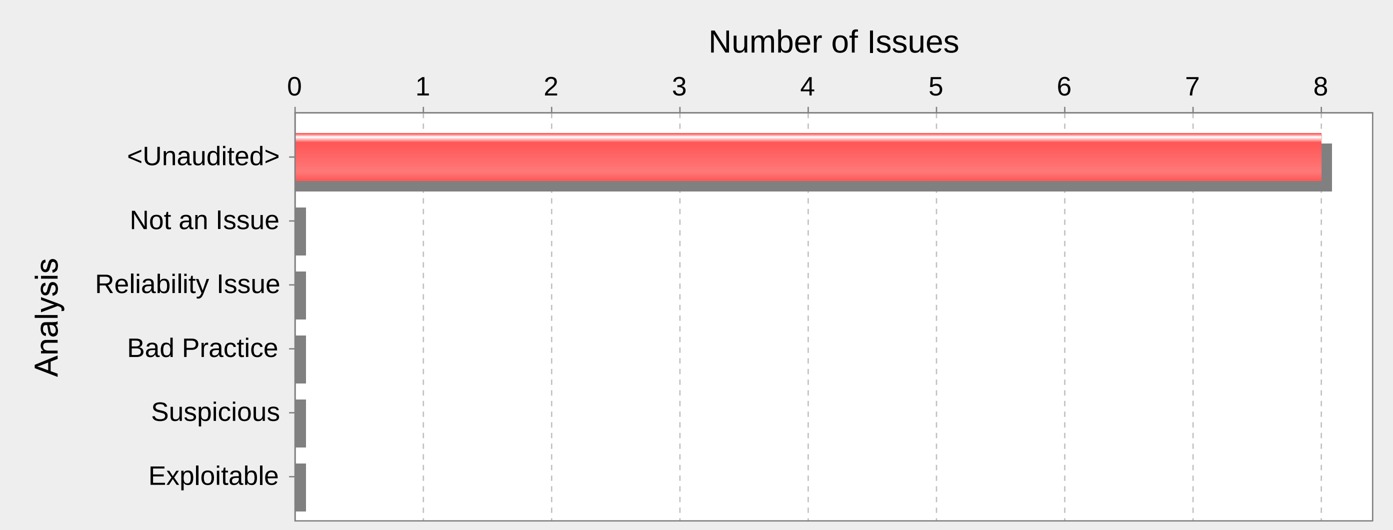
**Source:** StoragePropertiesjava:16 Read this.location()

```
14
15      public String getLocation() {
16          return location;
17      }
```

**Sink:** FileSystemStorageService.java:35 java.nio.file.Paths.get()

```
33      @Autowired
34      public FileSystemStorageService(StorageProperties properties) {
35          this.rootLocation = Paths.get(properties.getLocation());
36          if (!Files.exists(this.rootLocation)) {
37              log.debug("Creating storage service directory: " +
rootLocation.toString());
```

## Category: Log Forging (8 Issues)



### Abstract:

The method startDbBackup() in AdminUtils.java writes unvalidated user input to the log on line 67. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log.

### Explanation:

Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.

2. The data is written to an application or system log file.

Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.

Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker may be able to render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act [1]. In the worst case, an attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility [2].

Example 1: The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```
...
String val = request.getParameter("val");
try {
int value = Integer.parseInt(val);
}
catch (NumberFormatException nfe) {
log.info("Failed to parse val = " + val);
}
...
```

If a user submits the string "twenty-one" for val, the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

However, if an attacker submits the string "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy", the following entry is logged:

```
INFO: Failed to parse val=twenty-one

INFO: User logged out=badguy
```

Clearly, attackers may use this same mechanism to insert arbitrary log entries.

Some think that in the mobile world, classic web application vulnerabilities, such as log forging, 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 2: The following code adapts Example 1 to the Android platform.

```
...
String val = this.getIntent().getExtras().getString("val");
try {
int value = Integer.parseInt();
}
catch (NumberFormatException nfe) {
Log.e(TAG, "Failed to parse val = " + val);
}
...
```

## Recommendations:

Prevent log forging attacks with indirection: create a set of legitimate log entries that correspond to different events that must be logged and only log entries from this set. To capture dynamic content, such as users logging out of the system, always use server-controlled values rather than user-supplied data. This ensures that the input provided by the user is never used directly in a log entry.

Example 1 can be rewritten to use a pre-defined log entry that corresponds to a NumberFormatException as follows:

```
...
public static final String NFE = "Failed to parse val. The input is required to be an integer value."
...
String val = request.getParameter("val");
try {
int value = Integer.parseInt(val);
}
catch (NumberFormatException nfe) {
log.info(NFE);
}
..
```

And here is an Android equivalent:

```
...
public static final String NFE = "Failed to parse val. The input is required to be an integer value."
...
String val = this.getIntent().getExtras().getString("val");
try {
int value = Integer.parseInt();
}
catch (NumberFormatException nfe) {
Log.e(TAG, NFE);
}
...
```

In some situations this approach is impractical because the set of legitimate log entries is too large or complicated. In these situations, developers often fall back on implementing a deny list. A deny list is used to selectively reject or escape potentially dangerous characters before using the input. However, a list of unsafe characters can quickly become incomplete or outdated. A better approach is to create a list of characters that are permitted to appear in log entries and accept input composed exclusively of characters in the approved set. The most critical character in most log forging attacks is the '\n' (newline) character, which should never appear on a log entry allow list.

## Tips:

1. Many logging operations are created only for the purpose of debugging a program during development and testing. In our experience, debugging will be enabled, either accidentally or purposefully, in production at some point. Do not excuse log forging vulnerabilities simply because a programmer says "I don't have any plans to turn that on in production".

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.

## AdminUtils.java, line 67 (Log Forging)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Input Validation and Representation | | |
| **Abstract:** | The method startDbBackup() in AdminUtils.java writes unvalidated user input to the log on line 67. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log. | | |

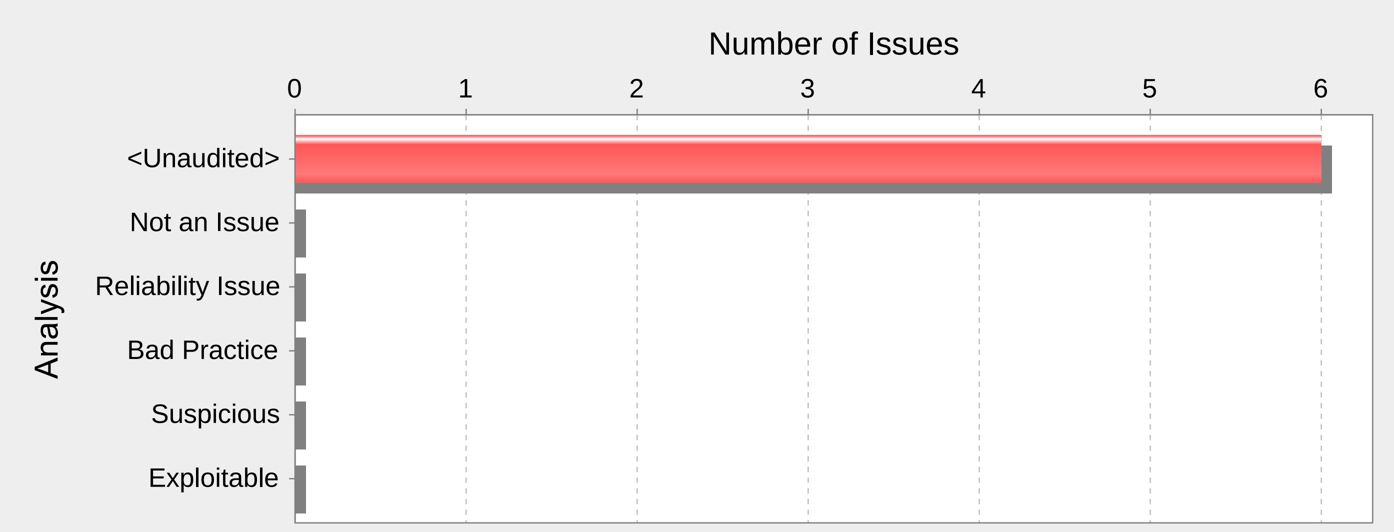
**Source:** AdminDefaultController.java:80 runDbBackup(0)

```
78
79      @PostMapping("/runDbBackup")
80      public String runDbBackup(@Valid @ModelAttribute("backupForm") BackupForm
   backupForm,
81                                BindingResult bindingResult, Model model,
82                                RedirectAttributes redirectAttributes,
```

**Sink:** AdminUtils.java:67 org.slf4j.Logger.info()

```
65                  "cmd.exe", "/K", "c:\\util\\cleanup.bat"
66          };
67          log.info("Running: " + Arrays.toString(backupCommand));
68          // call backup tool API
69          log.info("Running: " + Arrays.toString(cleanupCommand));
```

## Category: Cross-Site Scripting: Persistent (6 Issues)



### Abstract:

The method getMessagesByKeywords() in ApiMessageController.java sends unvalidated data to a web browser on line 85, 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 persistent (also known as stored) XSS, the untrusted source is typically a database or other back-end data store, while in the case of reflected XSS it is typically a web request.

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 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 %>
```

This code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. 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.

Example 2: 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 %>
```

As in Example 1, this code 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.

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, 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 2, 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 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.

### ApiMessageController.java, line 85 (Cross-Site Scripting: Persistent)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |

**Abstract:** The method getMessagesByKeywords() in ApiMessageController.java sends unvalidated data to a web browser on line 85, which can result in the browser executing malicious code.

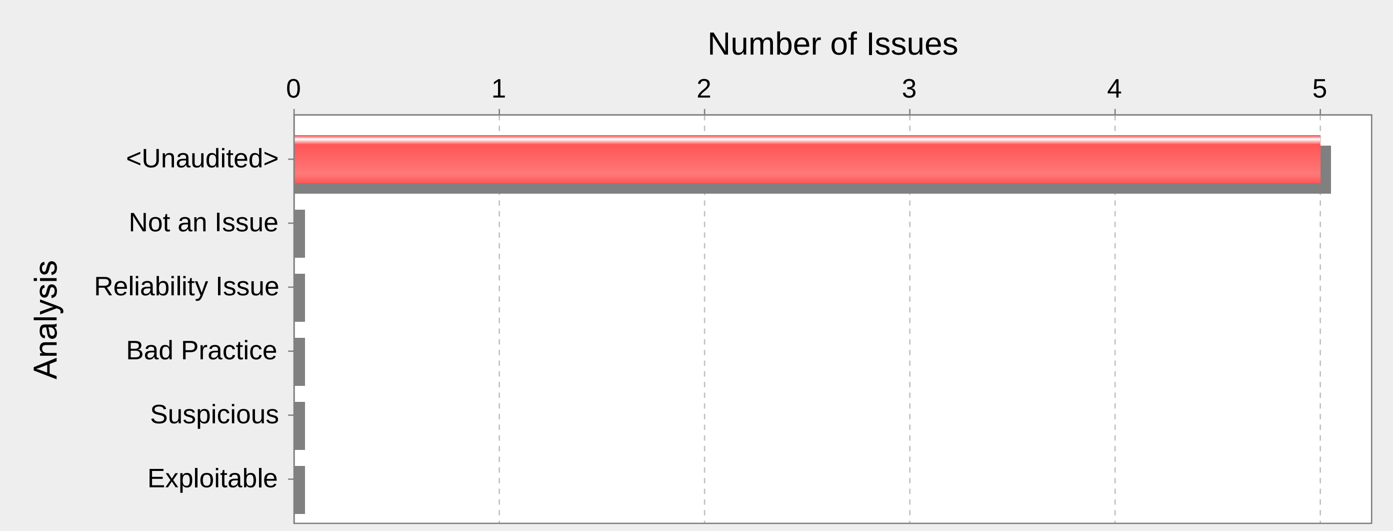
**Source:** UserService.java:352
org.springframework.data.jpa.repository.JpaRepository.findAll()

```
350
351         public List<Message> getAllMessages() {
352             return messageRepository.findAll();
353         }
```

**Sink:** ApiMessageController.java:85
org.springframework.http.ResponseEntity.BodyBuilder.body()

```
83                      userService.getAllMessages().stream()
84                          .map(MessageResponse::new)
85                          .collect(Collectors.toList()));
86              } else {
87                  return new ResponseEntity<>(
```

## Category: JSON Injection (5 Issues)



### Abstract:

On line 97 of UserUtils.java, the method registerUser() 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();
```

## UserUtils.java, line 97 (JSON Injection)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Input Validation and Representation | | |

**Abstract:** On line 97 of UserUtils.java, the method registerUser() writes unvalidated input into JSON. This call could allow an attacker to inject arbitrary elements or attributes into the JSON entity.

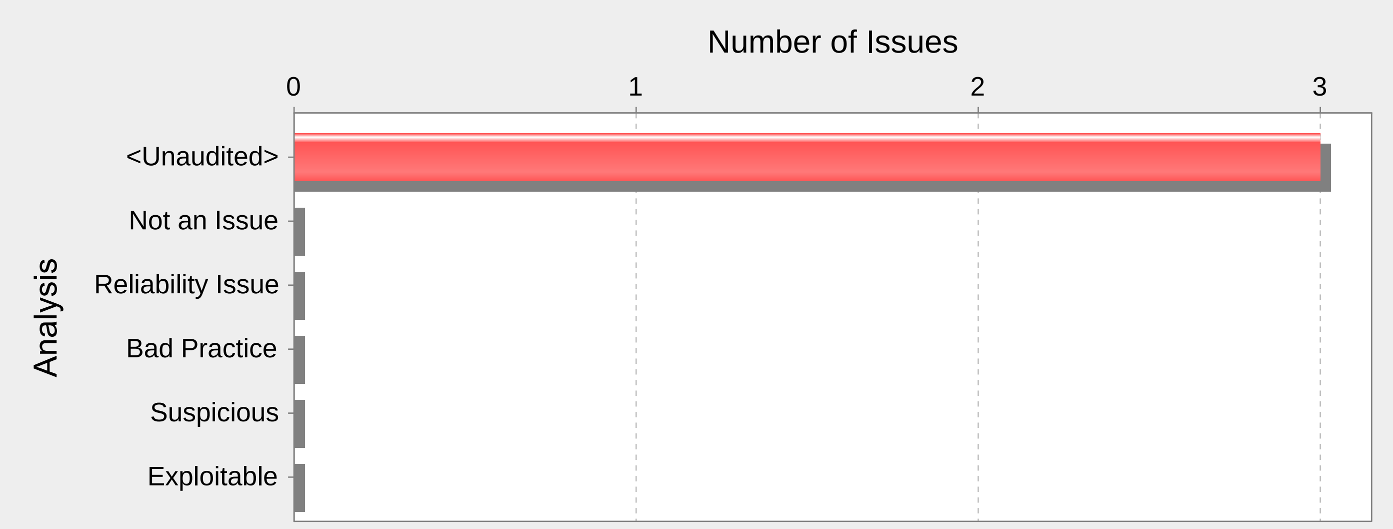
**Source:** UserUtils.java:81 java.io.FileReader.FileReader()

```
79         File dataFile = new File(getFilePath(NEWSLETTER_USER_FILE));
80         if (dataFile.exists()) {
81             jsonArray = (JSONArray) jsonParser.parse(new
FileReader(getFilePath(NEWSLETTER_USER_FILE)));
82         } else {
83             dataFile.createNewFile();
```

**Sink:** UserUtils.java:97 com.fasterxml.jackson.core.JsonGenerator.writeRawValue()

```
95                 JSONObject person = (JSONObject) jsonObject;
96                 jGenerator.writeFieldName("firstName");
97                 jGenerator.writeRawValue("\"" + (String) person.get("firstName") +
"\"");
98                 jGenerator.writeFieldName("lastName");
99                 jGenerator.writeRawValue("\"" + (String) person.get("lastName") +
"\"");
```

## Category: Insecure Randomness (3 Issues)



### Abstract:

The random number generator implemented by nextInt() 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.

```
String GenerateReceiptURL(String baseUrl) {
Random ranGen = new Random();
ranGen.setSeed((new Date()).getTime());
return (baseUrl + ranGen.nextInt(400000000) + ".html");
}
```

This code uses the Random.nextInt() function to generate "unique" identifiers for the receipt pages it generates. Since Random.nextInt() 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.)

The Java language provides a cryptographic PRNG in java.security.SecureRandom. As is the case with other algorithm-based classes in java.security, SecureRandom provides an implementation-independent wrapper around a particular set of algorithms. When you request an instance of a SecureRandom object using SecureRandom.getInstance(), you can request a specific implementation of the algorithm. If the algorithm is available, then it is given as a SecureRandom object. If it is unavailable or if you do not specify a particular implementation, then you are given a SecureRandom implementation selected by the system.

Sun provides a single SecureRandom implementation with the Java distribution named SHA1PRNG, which Sun describes as computing:

"The SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used [1]."

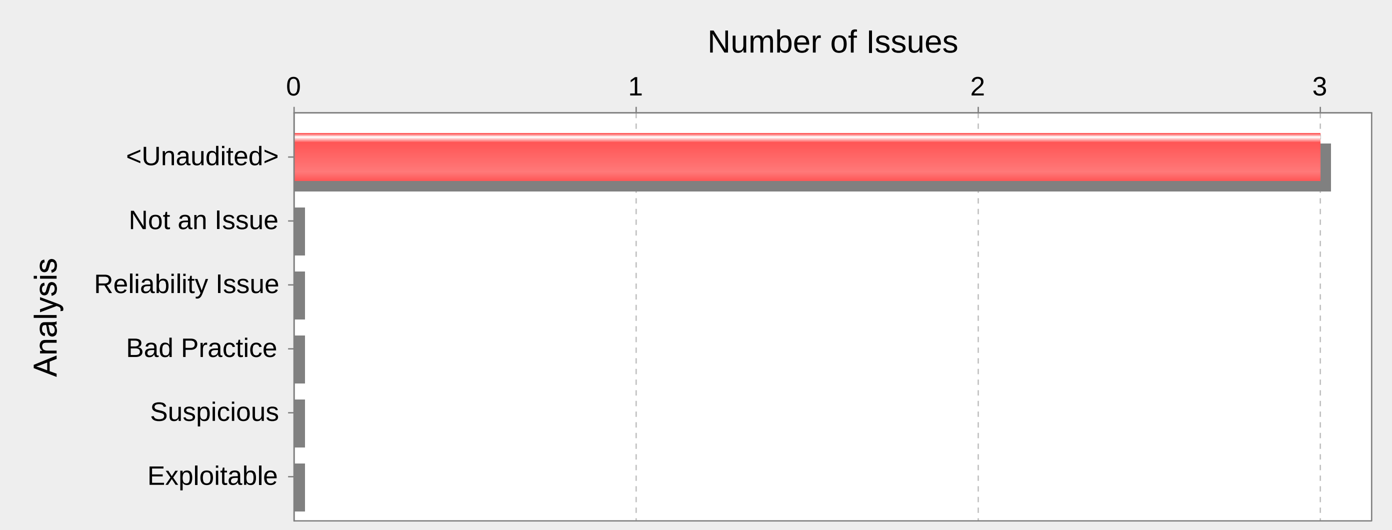
However, the specifics of the Sun implementation of the SHA1PRNG algorithm are poorly documented, and it is unclear what sources of entropy the implementation uses and therefore what amount of true randomness exists in its output. Although there is speculation on the Web about the Sun implementation, there is no evidence to contradict the claim that the algorithm is cryptographically strong and can be used safely in security-sensitive contexts.

## ProductService.java, line 293 (Insecure Randomness)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Security Features | | |
| **Abstract:** | The random number generator implemented by nextInt() cannot withstand a cryptographic attack. | | |
| **Sink:** | ProductService.java:293 nextInt() | | |

```
291                 int low = 10;
292                 int high = 100;
293                 int result = r.nextInt(high-low) + low;
294                 String formatted = String.format("%03d", result);
295                 otmp.setOrderNum("OID-P100-"+formatted);
```

## Category: Password Management: Password in Configuration File (3 Issues)



### Abstract:

Storing a plain text password in a configuration file may result in a system compromise.

### Explanation:

Storing a plain text password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plain text.

### Recommendations:

A password should never be stored in plain text. An administrator should be required to enter the password when the system starts. If that approach is impractical, a less secure but often adequate solution is to obfuscate the password and scatter the de-obfuscation material around the system so that an attacker has to obtain and correctly combine multiple system resources to decipher the password.

Some third-party products claim the ability to manage passwords in a more secure way. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. For a secure solution the only viable option is a proprietary one.

### 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 and that the password entry contains plain text.

2. If the entry in the configuration file is a default password, require that it be changed in addition to requiring that it be obfuscated in the configuration file.

## application-dev.yml, line 36 (Password Management: Password in Configuration File)

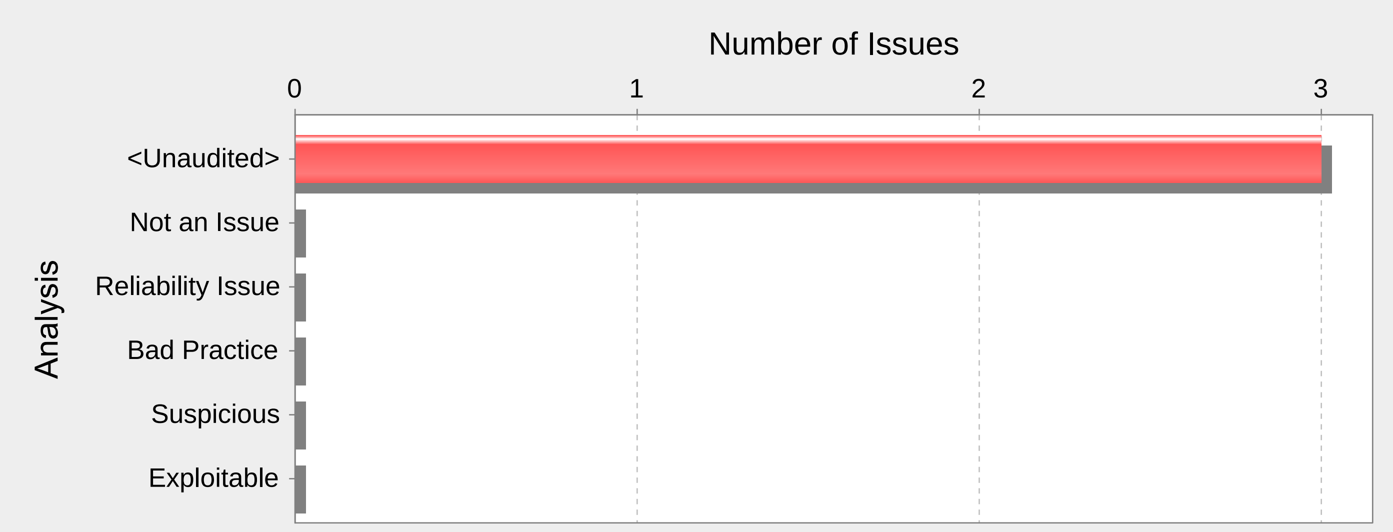
| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Environment | | |

**Abstract:** Storing a plain text password in a configuration file may result in a system compromise.

**Sink:** application-dev.yml:36 spring.datasource.password()

```
34          url: jdbc:h2:mem:iwa_dev
35          username: sa
36          password: password
37          initialization-mode: always
38        jpa:
```

## Category: System Information Leak: External (3 Issues)



### Abstract:

The function handle() in ApiAccessDeniedHandler.java reveals system data or debug information by calling println() on line 66. The information revealed by println() could help an adversary form a plan of attack.

### Explanation:

An external information leak occurs when system data or debug information leaves the program to a remote machine via a socket or network connection. External leaks can help an attacker by revealing specific data about operating systems, full pathnames, the existence of usernames, or locations of configuration files, and are more serious than internal information leaks, which are more difficult for an attacker to access.

Example 1: The following code leaks Exception information in the HTTP response:

```
protected void doPost (HttpServletRequest req, HttpServletResponse res) throws IOException {
...
PrintWriter out = res.getWriter();
try {
...
} catch (Exception e) {
out.println(e.getMessage());
}
}
```

This information can be exposed to a remote user. In some cases, the error message provides the attacker with the precise type of attack to which the system is vulnerable. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In Example 1, the leaked information could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Information leaks are also a concern in a mobile computing environment. With mobile platforms, applications are downloaded from various sources and are run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which is why application authors need to be careful about what information they include in messages addressed to other applications running on the device.

Example 2: The following code broadcasts the stack trace of a caught exception to all the registered Android receivers.

```
...
try {
...
} catch (Exception e) {
String exception = Log.getStackTraceString(e);
Intent i = new Intent();
i.setAction("SEND_EXCEPTION");
i.putExtra("exception", exception);
view.getContext().sendBroadcast(i);
}
...
```

This is another scenario specific to the mobile environment. Most mobile devices now implement a Near-Field Communication (NFC) protocol for quickly sharing information between devices using radio communication. It works by bringing devices in close proximity or having the devices touch each other. Even though the communication range of NFC is limited to just a few centimeters, eavesdropping, data modification and various other types of attacks are possible, because NFC alone does not ensure secure communication.

Example 3: The Android platform provides support for NFC. The following code creates a message that gets pushed to the other device within range.

```
...
public static final String TAG = "NfcActivity";
private static final String DATA_SPLITTER = "__:DATA:__";
private static final String MIME_TYPE = "application/my.applications.mimetype";
...
TelephonyManager tm = (TelephonyManager)Context.getSystemService(Context.TELEPHONY_SERVICE);
String VERSION = tm.getDeviceSoftwareVersion();
...
NfcAdapter nfcAdapter = NfcAdapter.getDefaultAdapter(this);
if (nfcAdapter == null)
return;

String text = TAG + DATA_SPLITTER + VERSION;
NdefRecord record = new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
MIME_TYPE.getBytes(), new byte[0], text.getBytes());
NdefRecord[] records = { record };
NdefMessage msg = new NdefMessage(records);
nfcAdapter.setNdefPushMessage(msg, this);
...
```

An NFC Data Exchange Format (NDEF) message contains typed data, a URI, or a custom application payload. If the message contains information about the application, such as its name, MIME type, or device software version, this information could be leaked to an eavesdropper.

## Recommendations:

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Debug traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system. Because of this, never send information to a resource directly outside the program.

Example 4: The following code broadcasts the stack trace of a caught exception within your application only, so that it cannot be leaked to other apps on the system. Additionally, this technique is more efficient than globally broadcasting through the system.

```
...
try {
...
} catch (Exception e) {
String exception = Log.getStackTraceString(e);
Intent i = new Intent();
i.setAction("SEND_EXCEPTION");
i.putExtra("exception", exception);
LocalBroadcastManager.getInstance(view.getContext()).sendBroadcast(i);
}
...
```

If you are concerned about leaking system data via NFC on an Android device, you could do one of the following three things. Do not include system data in the messages pushed to other devices in range, encrypt the payload of the message, or establish a secure communication channel at a higher layer.

## Tips:

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use Audit Guide to filter out this category from your scan results.

### ApiAccessDeniedHandler.java, line 66 (System Information Leak: External)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Encapsulation | | |

**Abstract:** The function handle() in ApiAccessDeniedHandler.java reveals system data or debug information by calling println() on line 66. The information revealed by println() could help an adversary form a plan of attack.

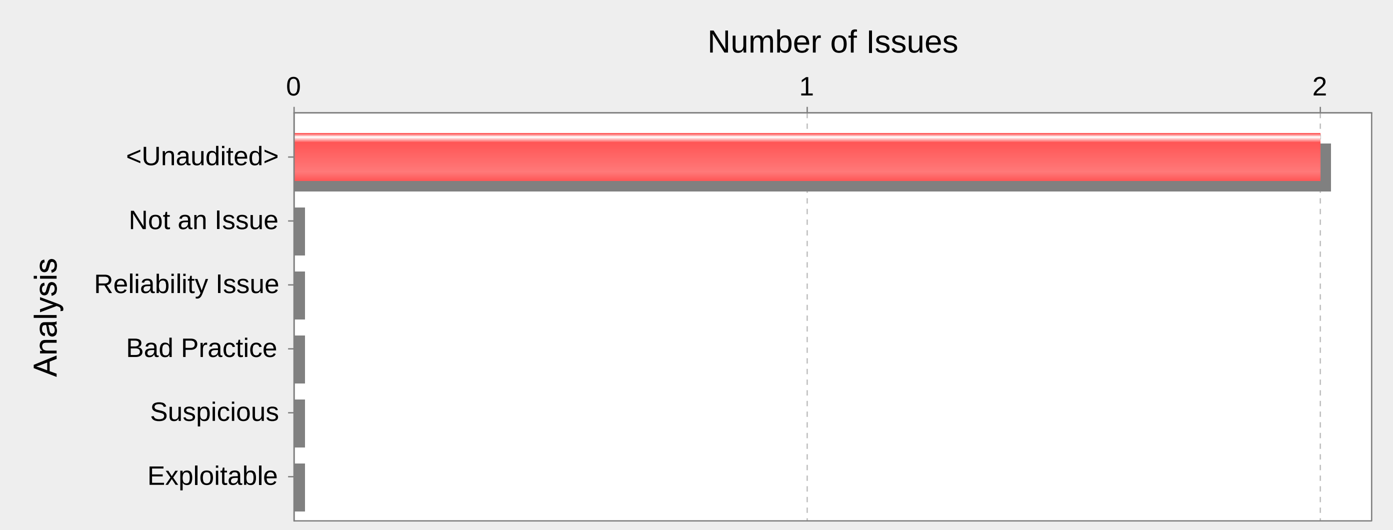
**Source:** ApiAccessDeniedHandler.java:53 java.lang.Throwable.getLocalizedMessage()

```
51            response.setStatus(HttpServletResponse.SC_FORBIDDEN);
52            ArrayList<String> errors = new ArrayList<>();
53            errors.add(ex.getLocalizedMessage());
54            ApiStatusResponse apiStatusResponse = new ApiStatusResponse
55                    .ApiResponseBuilder()
```

**Sink:** ApiAccessDeniedHandler.java:66 java.io.PrintWriter.println()

```
64            String jsonString = mapper.writeValueAsString(apiError.getBody());
65            PrintWriter writer = response.getWriter();
66            writer.println(jsonString);
67        }
```

## Category: Header Manipulation (2 Issues)



### Abstract:

The method downloadFile() in ProductController.java includes unvalidated data in an HTTP response header on line 138. This enables attacks such as cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.

### Explanation:

Header Manipulation vulnerabilities occur when:

1. Data enters a web application through an untrusted source, most frequently an HTTP request.

2. The data is included in an HTTP response header sent to a web user without being validated.

As with many software security vulnerabilities, Header Manipulation is a means to an end, not an end in itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to a vulnerable application, and the application includes the data in an HTTP response header.

One of the most common Header Manipulation attacks is HTTP Response Splitting. To mount a successful HTTP Response Splitting exploit, the application must allow input that contains CR (carriage return, also given by %0d or \r) and LF (line feed, also given by %0a or \n)characters into the header. These characters not only give attackers control of the remaining headers and body of the response the application intends to send, but also allows them to create additional responses entirely under their control.

Many of today's modern application servers will prevent the injection of malicious characters into HTTP headers. For example, recent versions of Apache Tomcat will throw an IllegalArgumentException if you attempt to set a header with prohibited characters. If your application server prevents setting headers with new line characters, then your application is not vulnerable to HTTP Response Splitting. However, solely filtering for new line characters can leave an application vulnerable to Cookie Manipulation or Open Redirects, so care must still be taken when setting HTTP headers with user input.

Example: The following code segment reads the name of the author of a weblog entry, author, from an HTTP request and sets it in a cookie header of an HTTP response.

```
String author = request.getParameter(AUTHOR_PARAM);
...
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
```

Assuming a string consisting of standard alphanumeric characters, such as "Jane Smith", is submitted in the request the HTTP response including this cookie might take the following form:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

However, because the value of the cookie is formed of unvalidated user input the response will only maintain this form if the value submitted for AUTHOR_PARAM does not contain any CR and LF characters. If an attacker submits a malicious string, such as "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n...", then the HTTP response would be split into two responses of the following form:

```
HTTP/1.1 200 OK
```

```
...
Set-Cookie: author=Wiley Hacker

HTTP/1.1 200 OK
...
```

Clearly, the second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability of attacker to construct arbitrary HTTP responses permits a variety of resulting attacks, including: cross-user defacement, web and browser cache poisoning, cross-site scripting, and page hijacking.

Cross-User Defacement: An attacker will be able to make a single request to a vulnerable server that will cause the server to create two responses, the second of which may be misinterpreted as a response to a different request, possibly one made by another user sharing the same TCP connection with the server. This can be accomplished by convincing the user to submit the malicious request themselves, or remotely in situations where the attacker and the user share a common TCP connection to the server, such as a shared proxy server. In the best case, an attacker may leverage this ability to convince users that the application has been hacked, causing users to lose confidence in the security of the application. In the worst case, an attacker may provide specially crafted content designed to mimic the behavior of the application but redirect private information, such as account numbers and passwords, back to the attacker.

Cache Poisoning: The impact of a maliciously constructed response can be magnified if it is cached either by a web cache used by multiple users or even the browser cache of a single user. If a response is cached in a shared web cache, such as those commonly found in proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged. Similarly, if the response is cached in the browser of an individual user, then that user will continue to receive the malicious content until the cache entry is purged, although only the user of the local browser instance will be affected.

Cross-Site Scripting: Once attackers have control of the responses sent by an application, they have a choice of a variety of malicious content to provide users. Cross-site scripting is common form of attack where malicious JavaScript or other code included in a response is executed in the user's browser. 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. The most common and dangerous attack vector against users of a vulnerable application uses JavaScript to transmit session and authentication information back to the attacker who can then take complete control of the victim's account.

Page Hijacking: In addition to using a vulnerable application to send malicious content to a user, the same root vulnerability can also be leveraged to redirect sensitive content generated by the server and intended for the user to the attacker instead. By submitting a request that results in two responses, the intended response from the server and the response generated by the attacker, an attacker may cause an intermediate node, such as a shared proxy server, to misdirect a response generated by the server for the user to the attacker. Because the request made by the attacker generates two responses, the first is interpreted as a response to the attacker's request, while the second remains in limbo. When the user makes a legitimate request through the same TCP connection, the attacker's request is already waiting and is interpreted as a response to the victim's request. The attacker then sends a second request to the server, to which the proxy server responds with the server generated request intended for the victim, thereby compromising any sensitive information in the headers or body of the response intended for the victim.

Cookie Manipulation: When combined with attacks like Cross-Site Request Forgery, attackers may change, add to, or even overwrite a legitimate user's cookies.

Open Redirect: Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

### Recommendations:

The solution to Header Manipulation is to ensure that input validation occurs in the correct places and checks for the correct properties.

Since Header Manipulation 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 responses dynamically, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for Header Manipulation.

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 Header Manipulation is generally relatively easy. Despite its value, input validation for Header Manipulation 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 Header Manipulation vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for Header Manipulation is to create an allow list of safe characters that are permitted to appear in HTTP response headers and accept input composed exclusively of characters in the approved set. For example, a valid name might only include alphanumeric characters or an account number might only include digits 0-9.

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 in HTTP response headers. Although the CR and LF characters are at the heart of an HTTP response splitting attack, other characters, such as ':' (colon) and '=' (equal), have special meaning in response headers as well.

After you identify the correct points in an application to perform validation for Header Manipulation attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. The application should reject any input destined to be included in HTTP response headers that contains special characters, particularly CR and LF, as invalid.

Many application servers attempt to limit an application's exposure to HTTP response splitting vulnerabilities by providing implementations for the functions responsible for setting HTTP headers and cookies that perform validation for the characters essential to an HTTP response splitting 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. Many HttpServletRequest implementations return a URL-encoded string from getHeader(), will not cause a HTTP response splitting issue unless it is decoded first because the CR and LF characters will not carry a meta-meaning in their encoded form. However, this behavior is not specified in the J2EE standard and varies by implementation. Furthermore, even encoded user input returned from getHeader() can lead to other vulnerabilities, including open redirects and other HTTP header tampering.

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.

## ProductController.java, line 138 (Header Manipulation)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Input Validation and Representation | | |

**Abstract:** The method downloadFile() in ProductController.java includes unvalidated data in an HTTP response header on line 138. This enables attacks such as cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.

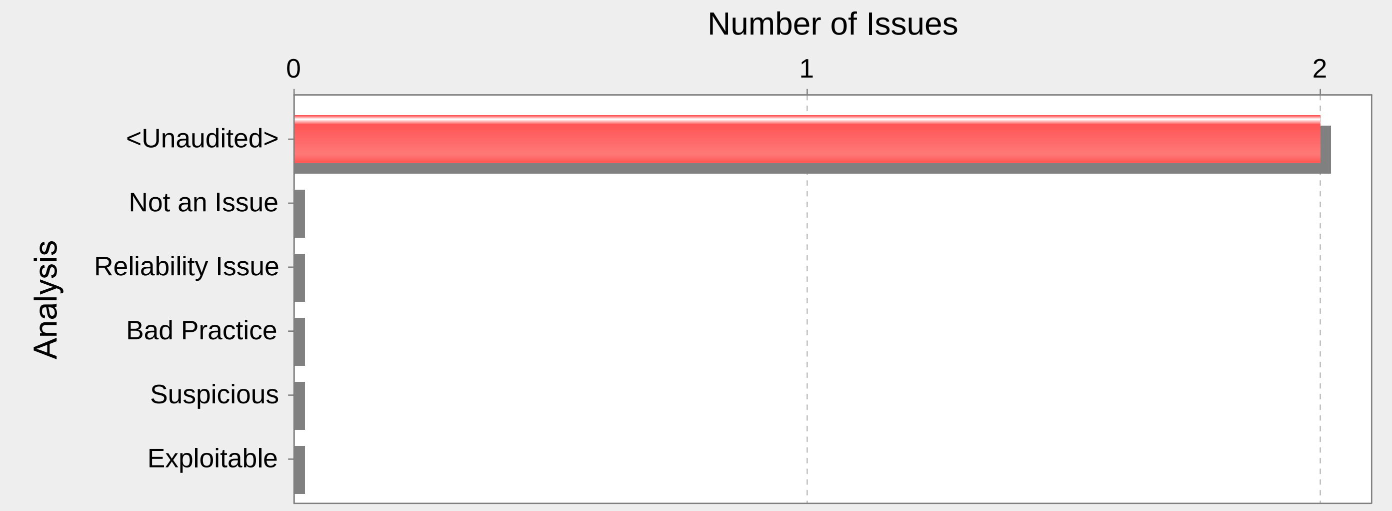
**Source:** ProductController.java:104 downloadFile(1)

```
102     @GetMapping("/{id}/download/{fileName:.+}")
103     public ResponseEntity<Resource> downloadFile(@PathVariable(value = "id") UUID productId,
104                                                  @PathVariable String fileName, HttpServletRequest request) {
105         Resource resource;
106         File dataDir = null;
```

**Sink:** ProductController.java:138 org.springframework.http.ResponseEntity.HeadersBuilder.header()

```
136
137         return ResponseEntity.ok().contentType(MediaType.parseMediaType(contentType))
138                 .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" + resource.getFilename() + "\"")
139                 .body(resource);
140     }
```

## Category: HTML5: Missing Content Security Policy (2 Issues)



### Abstract:

Content Security Policy (CSP) is not configured.

### Explanation:

Content Security Policy (CSP) is a declarative security header that enables developers to dictate which domains the site is allowed to load content from or initiate connections to when rendered in the web browser. It provides an additional layer of security from critical vulnerabilities such as cross-site scripting, clickjacking, cross-origin access and the like, on top of input validation and checking an allow list in code.

Spring Security and other frameworks do not add Content Security Policy headers by default. The web application author must declare the security policy/policies to enforce or monitor for the protected resources to benefit from this additional layer of security.

### Recommendations:

Configure a Content Security Policy to mitigate possible injection vulnerabilities.

Example: The following code sets a Content Security Policy in a Spring Security protected application:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
...
String policy = getCSPolicy();
http.headers().contentSecurityPolicy(policy);
...
}
```

Content Security Policy is not intended to solve all content injection vulnerabilities. Instead, you can leverage CSP to help reduce the harm caused by content injection attacks. Use regular defensive coding,above, current such as input validation and output encoding.

## WebSecurityConfiguration.java, line 97 (HTML5: Missing Content Security Policy)

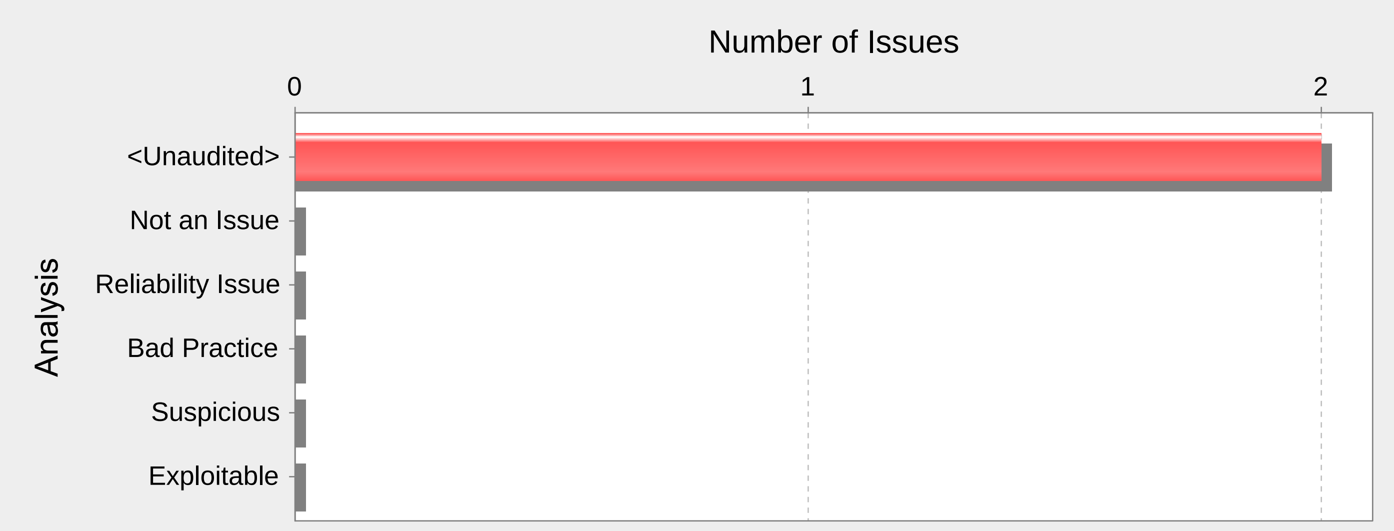
| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Encapsulation | | |

**Abstract:** Content Security Policy (CSP) is not configured.

**Sink:** WebSecurityConfiguration.java:97 Function: configure()

```
95
96                          @Override
97                          protected void configure(HttpSecurity httpSecurity) throws Exception {
98
99                              /*http.cors().and().csrf().disable()
```

## Category: Mass Assignment: Request Parameters Bound into Persisted Objects (2 Issues)



### Abstract:

The class in Authority.java is both a database persistent entity and a dynamically bound request object. Allowing database persistent entities to be auto-populated by request parameters will let an attacker create unintended database records in association entities or update unintended fields in the entity object.

### Explanation:

Persistent objects are bound to the underlying database and updated automatically by the persistence framework, such as Hibernate or JPA. Allowing these objects to be dynamically bound to the request by Spring MVC will let an attacker inject unexpected values into the database by providing additional request parameters.

Example 1: The Order, Customer, and Profile are Hibernate persisted classes.

```
public class Order {
String ordered;
List lineItems;
Customer cust;
...
}
public class Customer {
String customerId;
...
Profile p;
...
}
public class Profile {
String profileId;
String username;
String password;
...
}
```

OrderController is the Spring controller class handling the request:

```
@Controller
public class OrderController {
...
@RequestMapping("/updateOrder")
public String updateOrder(Order order) {
...
session.save(order);
}
}
```

Because command classes are automatically bound to the request, an attacker may use this vulnerability to update another user's password by adding the following request parameters to the request: "http://www.yourcorp.com/webApp/updateOrder?order.customer.profile.profileId=1234&order.customer.profile.password=urpowned"

### Recommendations:

Do not use persistent entity objects as your request bound objects. Manually copy the attributes which you are interested in persisting from your request bound objects to your persistent entity objects. An alternative would be to explicitly define which attributes on the request bound object are settable via request parameters.

## Authority.java, line 36 (Mass Assignment: Request Parameters Bound into Persisted Objects)

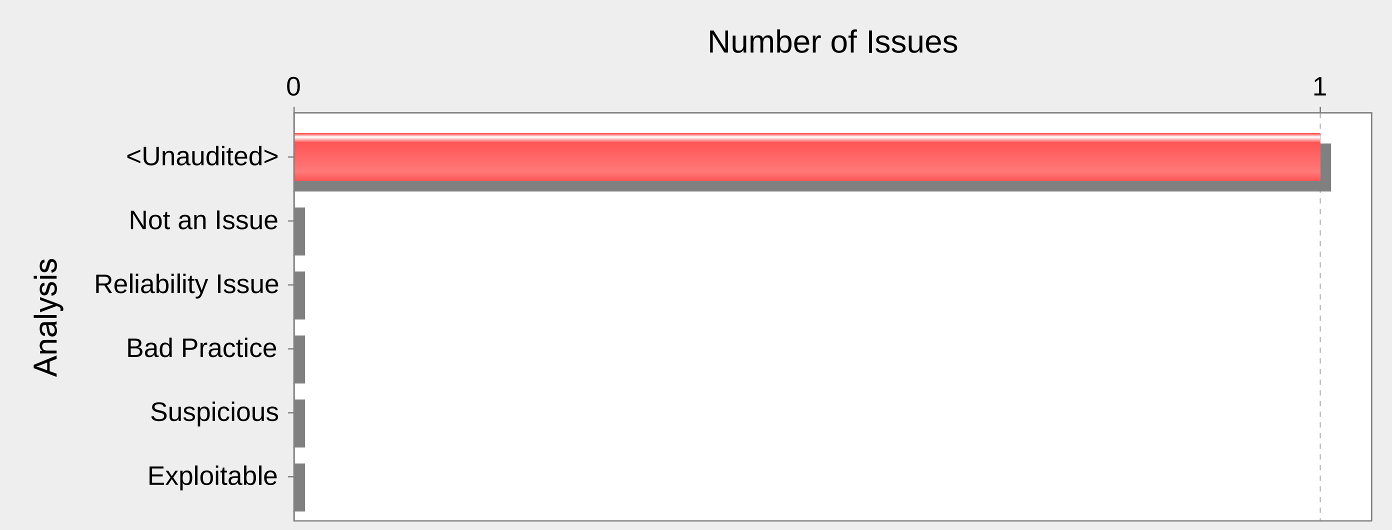
| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | API Abuse | | |

**Abstract:** The class in Authority.java is both a database persistent entity and a dynamically bound request object. Allowing database persistent entities to be auto-populated by request parameters will let an attacker create unintended database records in association entities or update unintended fields in the entity object.

**Sink:** Authority.java:36 Class: Authority()

```
34    @Table(name = "authorities")
35    @JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class, property =
      "id")
36    public class Authority {
37
38        private static final long serialVersionUID = 1L;
```

## Category: Cross-Site Scripting: Reflected (1 Issues)



### Abstract:

The method serveFile() in UserController.java sends unvalidated data to a web browser on line 416, 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.

## UserController.java, line 416 (Cross-Site Scripting: Reflected)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |

**Abstract:** The method serveFile() in UserController.java sends unvalidated data to a web browser on line 416, which can result in the browser executing malicious code.

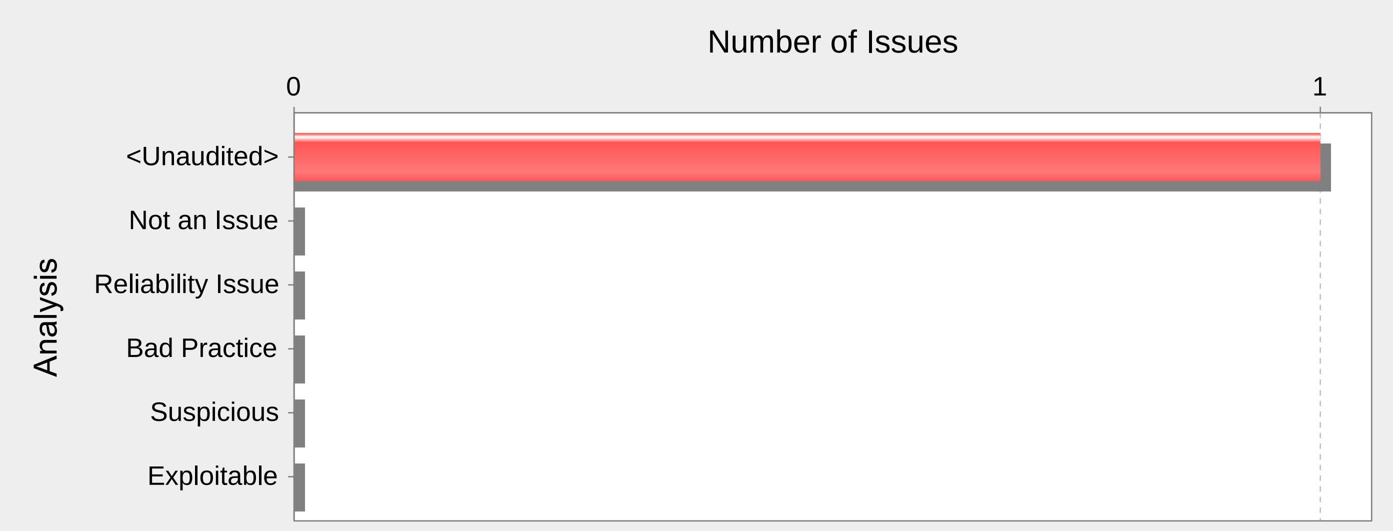
**Source:** UserController.java:412 serveFile(0)

```
410     @GetMapping("/files/{filename:.+}")
411     @ResponseBody
412     public ResponseEntity<Resource> serveFile(@PathVariable String filename) {
413 
414         Resource file = storageService.loadAsResource(filename);
```

**Sink:** UserController.java:416 org.springframework.http.ResponseEntity.BodyBuilder.body()

```
414         Resource file = storageService.loadAsResource(filename);
415         return ResponseEntity.ok().header(HttpHeaders.CONTENT_DISPOSITION,
416                 "attachment; filename=\"" + file.getFilename() + "\"").body(file);
417     }
```

## Category: HTML5: Missing Framing Protection (1 Issues)



### Abstract:

The application does not restrict browsers from letting third-party sites render its content.

### Explanation:

Allowing your website to be added to a frame can be a security issue. For example, it may lead to clickjacking vulnerabilities or allow undesired cross-frame communications.

By default, frameworks such as Spring Security include the X-Frame-Options header to instruct the browser whether the application should be framed. Disabling or not setting this header can lead to cross-frame related vulnerabilities.

Example 1: The following code configures a Spring Security protected application to disable the X-Frame-Options header:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
...
http.headers().frameOptions().disable();
...
}
```

### Recommendations:

Set the X-Frame-Options header as an additional layer of protection. If your application never needs to be framed, set its value to DENY, otherwise if it needs to be framed by a different application or page from the same origin, set its value to SAMEORIGIN.

By default, Spring Security disables rendering within an iframe. You can customize X-Frame-Options header to fit your application's requirements.

Example 2: The following code configures a Spring Security protected application to use a SAMEORIGIN policy:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
...
http.headers().frameOptions().sameOrigin();
...
}
```

## WebSecurityConfiguration.java, line 141 (HTML5: Missing Framing Protection)

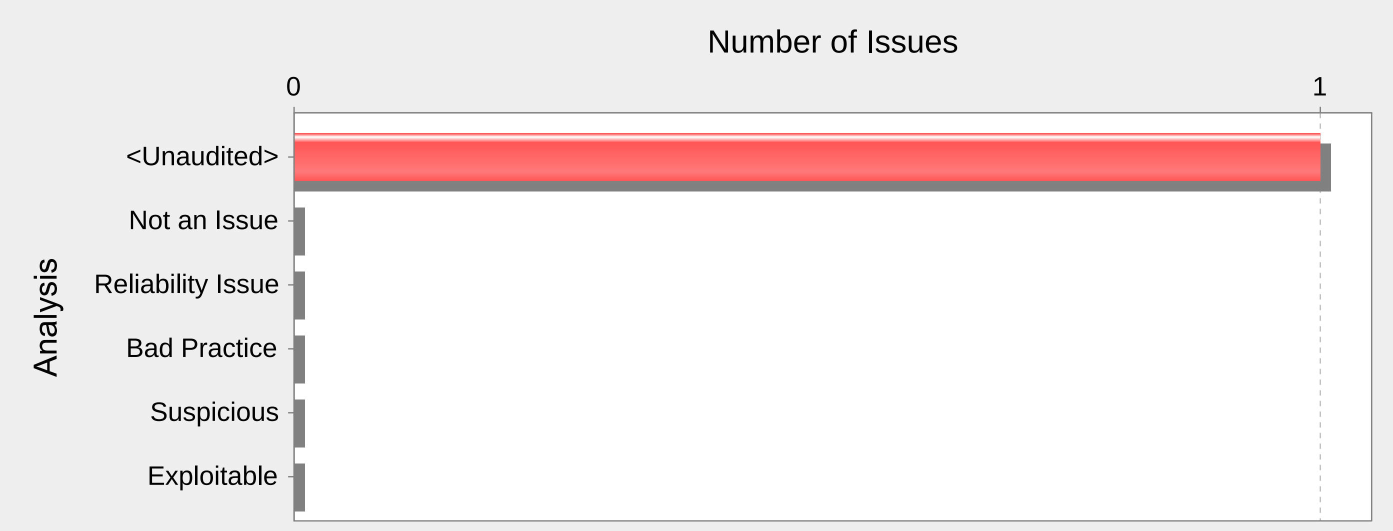
| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Encapsulation | | |

**Abstract:** The application does not restrict browsers from letting third-party sites render its content.

**Sink:** WebSecurityConfiguration.java:141 FunctionCall: disable()

```
139                         log.info("Running development profile");
140                         httpSecurity.csrf().disable();
141                         httpSecurity.headers().frameOptions().disable();
142                         httpSecurity.cors().disable();
143                         httpSecurity.headers().xssProtection().disable();
```

## Category: Insecure Randomness: Hardcoded Seed (1 Issues)



### Abstract:

The function genId() in AdminUtils.java is passed a constant value for the seed. Functions that generate random or pseudorandom values, which are passed a seed, should not be called with a constant argument.

### Explanation:

Functions that generate random or pseudorandom values, which are passed a seed, should not be called with a constant argument. If a pseudorandom number generator (such as Random) is seeded with a specific value (using a function such as Random.setSeed()), the values returned by Random.nextInt() and similar methods which return or assign values are predictable for an attacker that can collect a number of PRNG outputs.

Example 1: The values produced by the Random object s are predictable from the Random object r.

```
Random r = new Random();
r.setSeed(12345);
int i = r.nextInt();
byte[] b = new byte[4];
r.nextBytes(b);

Random s = new Random();
s.setSeed(12345);
int j = s.nextInt();
byte[] c = new byte[4];
s.nextBytes(c);
```

In this example, pseudorandom number generators: r and s were identically seeded, so i == j, and corresponding values of arrays b[] and c[] are equal.

### Recommendations:

Use a cryptographic PRNG seeded with hardware-based sources of randomness, such as ring oscillators, disk drive timing, thermal noise, or radioactive decay. Doing so makes the sequence of data produced by Random.nextInt() and similar methods much harder to predict than setting the seed to a constant.

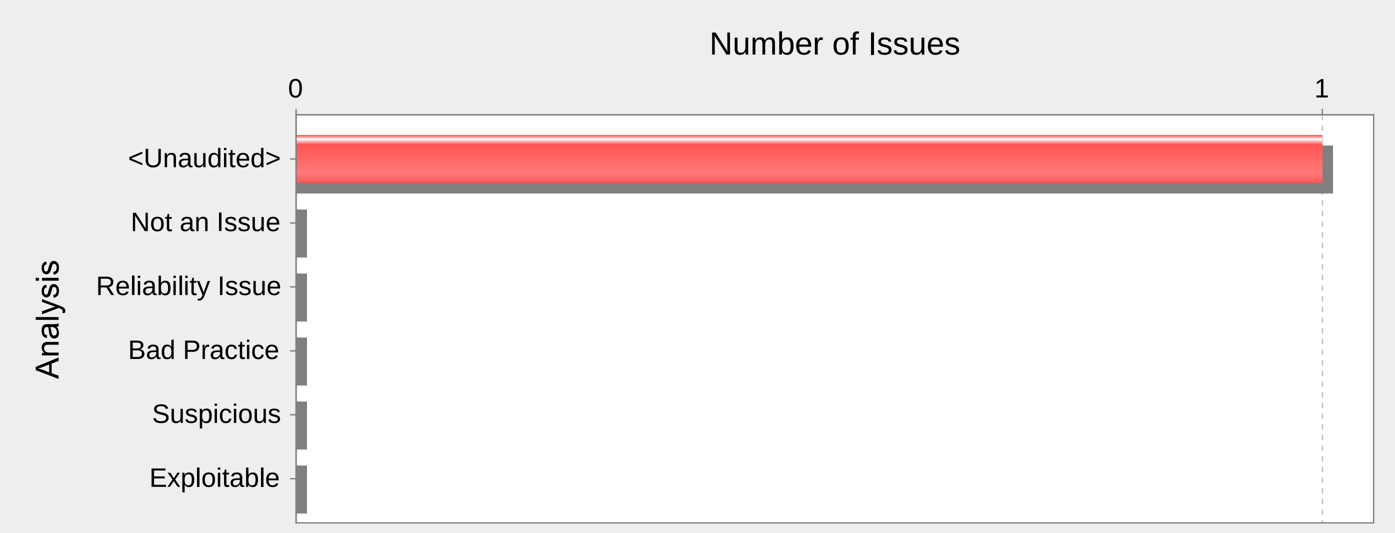
### AdminUtils.java, line 119 (Insecure Randomness: Hardcoded Seed)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Security Features | | |

| | |
|---|---|
| **Abstract:** | The function genId() in AdminUtils.java is passed a constant value for the seed. Functions that generate random or pseudorandom values, which are passed a seed, should not be called with a constant argument. |
| **Sink:** | AdminUtils.java:119 setSeed() |

```
117
118             Random r=new Random();
119             r.setSeed(12345);
120             return r.nextInt();
121         }
```

## Category: J2EE Bad Practices: Non-Serializable Object Stored in Session (1 Issues)



### Abstract:

The method generateAndSetSession() in JwtUtils.java stores a non-serializable object as an HttpSession attribute, which can damage application reliability.

### Explanation:

A J2EE application can make use of multiple JVMs in order to improve application reliability and performance. In order to make the multiple JVMs appear as a single application to the end user, the J2EE container can replicate an HttpSession object across multiple JVMs so that if one JVM becomes unavailable another can step in and take its place without disrupting the flow of the application.

In order for session replication to work, the values the application stores as attributes in the session must implement the Serializable interface.

Example 1: The following class adds itself to the session, but because it is not serializable, the session can no longer be replicated.

```
public class DataGlob {
String globName;
String globValue;

public void addToSession(HttpSession session) {
session.setAttribute("glob", this);
}
}
```

### Recommendations:

In many cases, the easiest way to fix this problem is simply to have the offending object implement the Serializable interface.

Example 2: The code in Example 1 could be rewritten in the following way:

```
public class DataGlob implements java.io.Serializable {
String globName;
String globValue;

public void addToSession(HttpSession session) {
session.setAttribute("glob", this);
}
}
```

Note that for complex objects, the transitive closure of the objects stored in the session must be serializable. If object A references object B and object A is stored in the session, then both A and B must implement Serializable.

While implementing the Serializable interface is often easy (since the interface does not force the class to define any methods), some types of objects will cause complications. Watch out for objects that hold references to external resources. For example, both streams and JNI are likely to cause complications.

Example 3: Use type checking to require serializable objects. Instead of this:

```
public static void addToSession(HttpServletRequest req,
String attrib, Object obj)
```

```
{
HttpSession sess = req.getSession(true);
sess.setAttribute(attrib, obj);
}
```

write this:

```
public static void addToSession(HttpServletRequest req,
String attrib, Serializable ser) {
HttpSession sess = req.getSession(true);
sess.setAttribute(attrib, ser);
}
```

## JwtUtils.java, line 111 (J2EE Bad Practices: Non-Serializable Object Stored in Session)

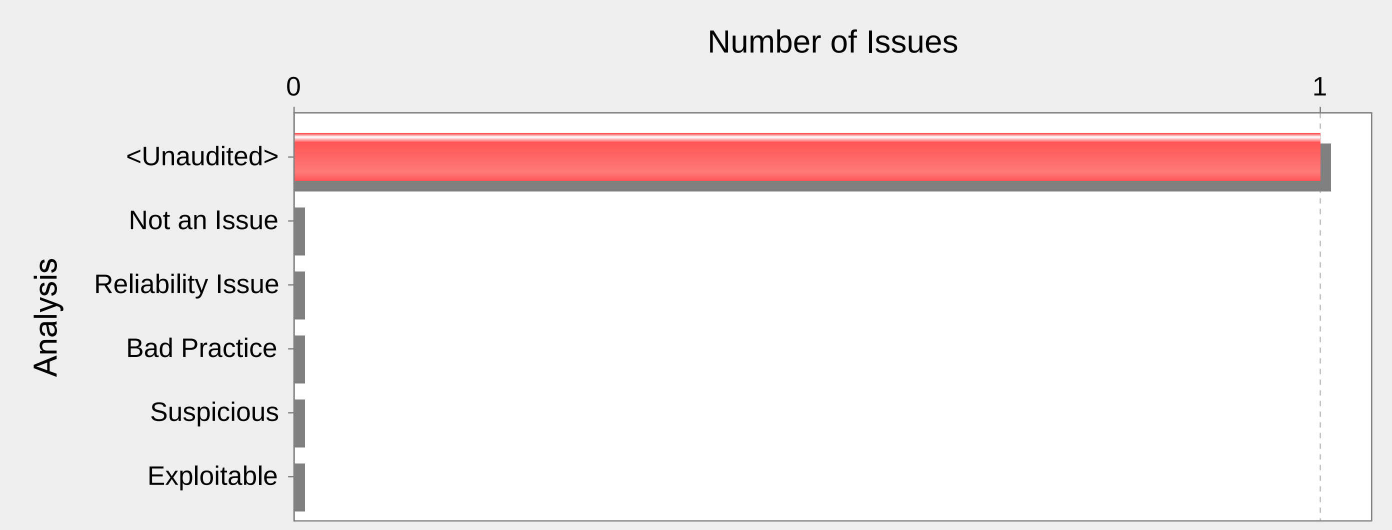
| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Time and State | | |

**Abstract:** The method generateAndSetSession() in JwtUtils.java stores a non-serializable object as an HttpSession attribute, which can damage application reliability.

**Sink:** JwtUtils.java:111 FunctionCall: setAttribute()

```
109                 session.setAttribute("userId", user.getId());
110                 session.setAttribute("username", user.getUsername());
111                 session.setAttribute("authorities", authentication.getAuthorities());
112                 session.setAttribute("jwtToken", jwtToken);
113                 return jwtToken;
```

## Category: Null Dereference (1 Issues)



### Abstract:

The method getTargetUrl() in CustomAuthenticationSuccessHandler.java can crash the program by dereferencing a null-pointer on line 113.

### Explanation:

Null-pointer exceptions usually occur when one or more of the programmer's assumptions is violated. A dereference-after-store error occurs when a program explicitly sets an object to null and dereferences it later. This error is often the result of a programmer initializing a variable to null when it is declared.

Most null-pointer issues result in general software reliability problems, but if attackers can intentionally trigger a null-pointer dereference, they can use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

Example: In the following code, the programmer explicitly sets the variable foo to null. Later, the programmer dereferences foo before checking the object for a null value.

```
Foo foo = null;
...
foo.setBar(val);
...
}
```

### Recommendations:

Implement careful checks before dereferencing objects that might be null. When possible, abstract null checks into wrappers around code that manipulates resources to ensure that they are applied in all cases and to minimize the places where mistakes can occur.

## CustomAuthenticationSuccessHandler.java, line 113 (Null Dereference)

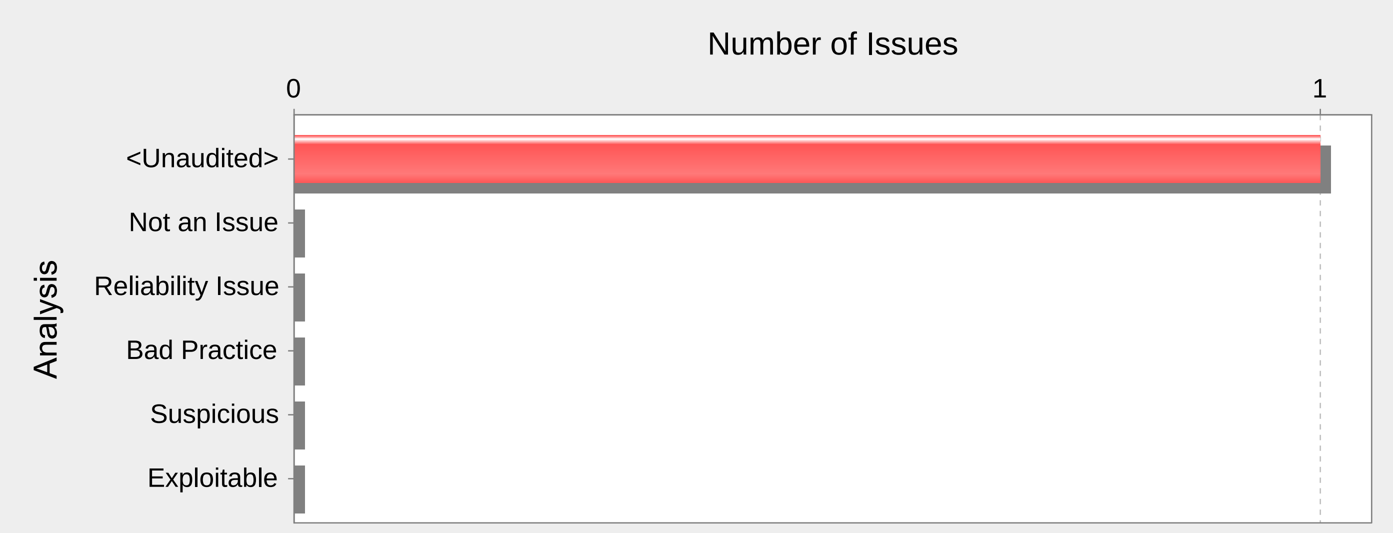
| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Code Quality | | |

**Abstract:** The method getTargetUrl() in CustomAuthenticationSuccessHandler.java can crash the program by dereferencing a null-pointer on line 113.

**Sink:** CustomAuthenticationSuccessHandler.java:113 Dereferenced : targetPath()

```
111                             }
112                             if (targetUrl.contains("?")) targetUrl = targetUrl.substring(0,
            targetUrl.indexOf("?"));
113                             if (targetPath.endsWith("/cart")) {
114                                 targetUrl = targetUrl.replace("/cart", "/cart/checkout");
115                             } else if (targetPath.endsWith("/login")) {
```

## Category: Often Misused: Boolean.getBoolean() (1 Issues)



### Abstract:

The method Boolean.getBoolean() is often confused with Boolean.valueOf() or Boolean.parseBoolean() method calls.

### Explanation:

In most cases, a call to Boolean.getBoolean() is often misused as it is assumed to return the boolean value represented by the specified string argument. However, as stated in the Javadoc Boolean.getBoolean(String) method "Returns true if and only if the system property named by the argument exists and is equal to the string 'true'."

Most often what the developer intended to use was a call to Boolean.valueOf(String) or Boolean.parseBoolean(String) method.

Example 1: The following code will not behave as expected. It will print "FALSE" as Boolean.getBoolean(String) does not translate a String primitive. It only translates system property.

```
...
String isValid = "true";
if ( Boolean.getBoolean(isValid) ) {
System.out.println("TRUE");
}
else {
System.out.println("FALSE");
}
...
```

### Recommendations:

Please ensure that you intend to call the method Boolean.getBoolean(String) and the specified string argument is a system property. Else the method call you are most likely looking for is Boolean.valueOf(String) or Boolean.parseBoolean(String).

## AdminUtils.java, line 91 (Often Misused: Boolean.getBoolean())

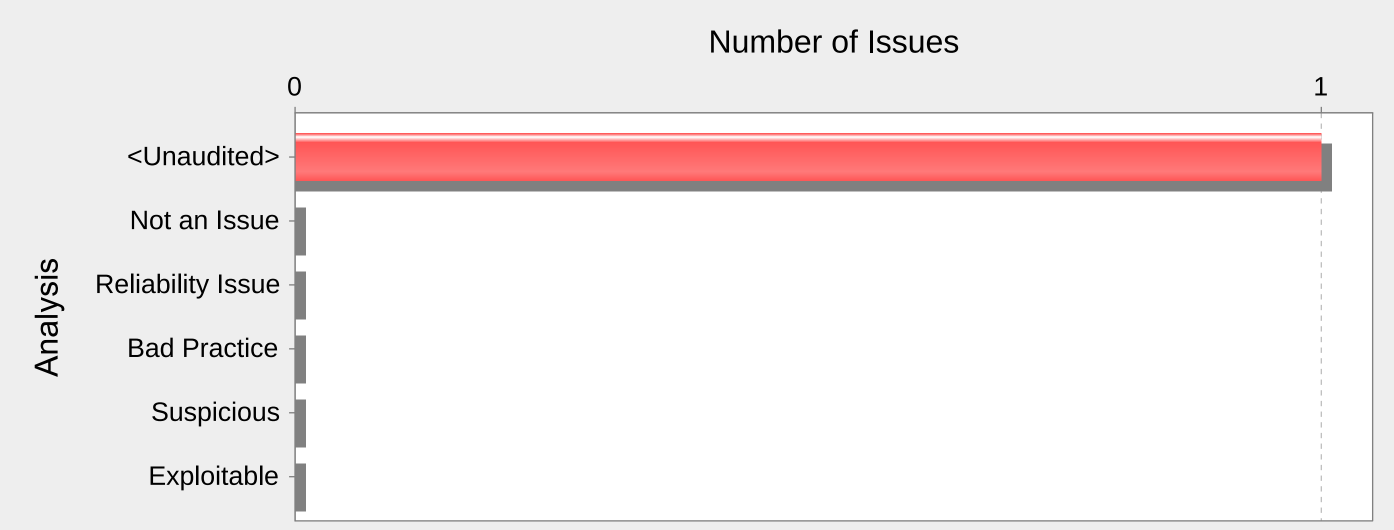
| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | API Abuse | | |

**Abstract:** The method Boolean.getBoolean() is often confused with Boolean.valueOf() or Boolean.parseBoolean() method calls.

**Sink:** AdminUtils.java:91 getBoolean()

```
89      */
90
91              if(Boolean.getBoolean(isLocked(backupId))){
92                  return"LOCKED";
93              }
```

## Category: Open Redirect (1 Issues)



### Abstract:

The file DefaultController.java passes unvalidated data to an HTTP redirect function on line 93. Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

### Explanation:

Redirects allow web applications to direct users to different pages within the same application or to external sites. Applications utilize redirects to aid in site navigation and, in some cases, to track how users exit the site. Open redirect vulnerabilities occur when a web application redirects clients to any arbitrary URL that can be controlled by an attacker.

Attackers may utilize open redirects to trick users into visiting a URL to a trusted site and redirecting them to a malicious site. By encoding the URL, an attacker is able to make it more difficult for end-users to notice the malicious destination of the redirect, even when it is passed as a URL parameter to the trusted site. Open redirects are often abused as part of phishing scams to harvest sensitive end-user data.

Example 1: The following JSP code instructs the user's browser to open a URL parsed from the dest request parameter when a user clicks the link.

```
<%
...
String strDest = request.getParameter("dest");
pageContext.forward(strDest);
...
%>
```

If a victim received an email instructing them to follow a link to "http://trusted.example.com/ecommerce/redirect.asp?dest=www.wilyhacker.com", the user would likely click on the link believing they would be transferred to the trusted site. However, when the victim clicks the link, the code in Example 1 will redirect the browser to "http://www.wilyhacker.com".

Many users have been educated to always inspect URLs they receive in emails to make sure the link specifies a trusted site they know. However, if the attacker Hex encoded the destination url as follows:

"http://trusted.example.com/ecommerce/redirect.asp?dest=%77%69%6C%79%68%61%63%6B%65%72%2E%63%6F%6D"

then even a savvy end-user may be fooled into following the link.

### Recommendations:

Unvalidated user input should not be allowed to control the destination URL in a redirect. Instead, use a level of indirection: create a list of legitimate URLs that users are allowed to specify and only allow users to select from the list. With this approach, input provided by users is never used directly to specify a URL for redirects.

Example 2: The following code references an array populated with valid URLs. The link the user clicks passes in the array index that corresponds to the desired URL.

```
<%
...
try {
int strDest = Integer.parseInt(request.getParameter("dest"));
if((strDest >= 0) && (strDest <= strURLArray.length -1 ))
{
```

```
strFinalURL = strURLArray[strDest];
pageContext.forward(strFinalURL);
}
}
catch (NumberFormatException nfe) {
// Handle exception
...
}
...
%>
```

In some situations this approach is impractical because the set of legitimate URLs is too large or too hard to keep track of. In such cases, use a similar approach to restrict the domains that users can be redirected to, which can at least prevent attackers from sending users to malicious external sites.

**Tips:**

1. 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.

2. Fortify AppDefender adds protection against this category.

### DefaultController.java, line 93 (Open Redirect)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |

**Abstract:** The file DefaultController.java passes unvalidated data to an HTTP redirect function on line 93. Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

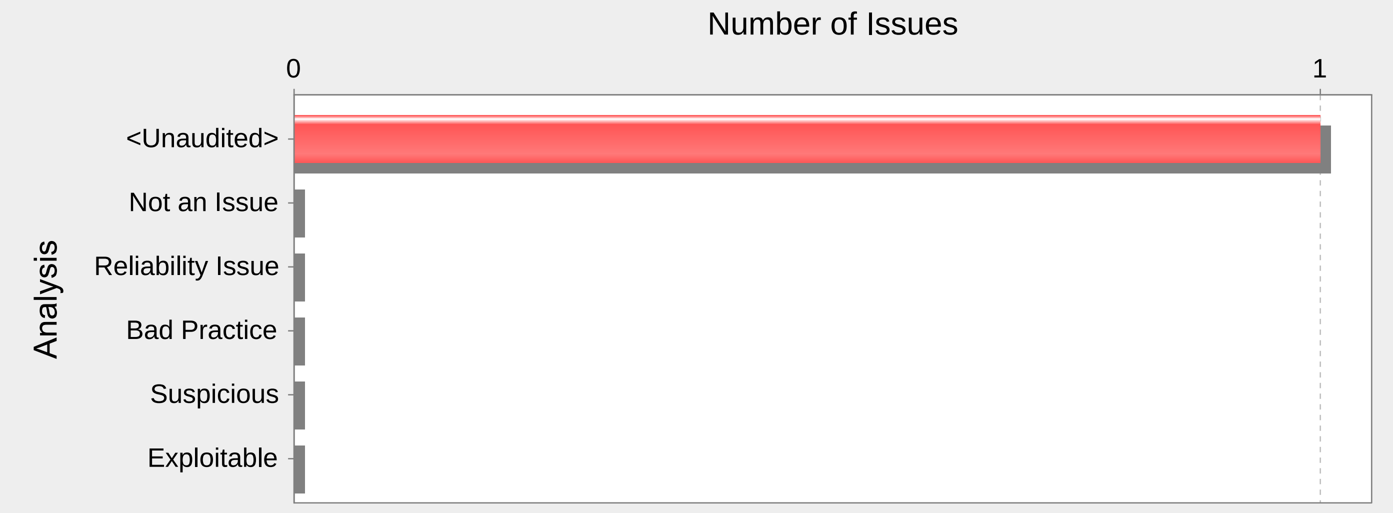
**Source:** DefaultController.java:77 javax.servlet.http.HttpServletRequest.getHeader()

```
75      public String login(HttpServletRequest request, Model model, Principal principal)
{
76          HttpSession session = request.getSession(false);
77          String referer = (String) request.getHeader("referer");
78          session.setAttribute("loginReferer", referer);
79          return "login";
```

**Sink:** DefaultController.java:93 Return()

```
91          String jwtToken = jwtUtils.generateAndSetSession(request, response,
authentication);
92          String targetUrl = CustomAuthenticationSuccessHandler.getTargetUrl(request,
response, authentication);
93          return "redirect:"+targetUrl;
94      }
```

## Category: Password Management: Hardcoded Password (1 Issues)



### Abstract:

Hardcoded passwords can compromise system security in a way that is not easy to remedy.

### Explanation:

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. After the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system must choose between security and availability.

Example 1: The following code uses a hardcoded password to connect to a database:

```
...
DriverManager.getConnection(url, "scott", "tiger");
...
```

This code will run successfully, but anyone who has access to it will have access to the password. After the program ships, there is likely no way to change the database user "scott" with a password of "tiger" unless the program is patched. An employee with access to this information can use it to break into the system. Even worse, if attackers have access to the bytecode for the application they can use the javap -c command to access the disassembled code, which will contain the values of the passwords used. The result of this operation might look something like the following for Example 1:

```
javap -c ConnMngr.class

22: ldc   #36; //String jdbc:mysql://ixne.com/rxsql
24: ldc   #38; //String scott
26: ldc   #17; //String tiger
```

In the mobile environment, password management is especially important given that there is such a high chance of device loss.

Example 2: The following code uses hardcoded username and password to setup authentication for viewing protected pages with Android's WebView.

```
...
webview.setWebViewClient(new WebViewClient() {
public void onReceivedHttpAuthRequest(WebView view,
HttpAuthHandler handler, String host, String realm) {
handler.proceed("guest", "allow");
}
});
...
```

Similar to Example 1, this code will run successfully, but anyone who has access to it will have access to the password.

### Recommendations:

Passwords should never be hardcoded and 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. At the very least, hash passwords before storing them.

Some third-party products claim the ability to securely manage passwords. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. Today, the best option for a secure generic solution is to create a proprietary mechanism yourself.

For Android, as well as any other platform that uses SQLite database, SQLCipher is a good alternative. SQLCipher is an extension to the SQLite database that provides transparent 256-bit AES encryption of database files. Thus, credentials can be stored in an encrypted database.

Example 3: The following code demonstrates how to integrate SQLCipher into an Android application after downloading the necessary binaries, and store credentials into the database file.

```
import net.sqlcipher.database.SQLiteDatabase;
...
SQLiteDatabase.loadLibs(this);
File dbFile = getDatabasePath("credentials.db");
dbFile.mkdirs();
dbFile.delete();
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(dbFile, "credentials", null);
db.execSQL("create table credentials(u, p)");
db.execSQL("insert into credentials(u, p) values(?, ?)", new Object[]{username, password});
...
```

Note that references to android.database.sqlite.SQLiteDatabase are substituted with those of net.sqlcipher.database.SQLiteDatabase.

To enable encryption on the WebView store, you must recompile WebKit with the sqlcipher.so library.

**Tips:**

1. You can use the Fortify Java Annotations FortifyPassword and FortifyNotPassword to indicate which fields and variables represent passwords.

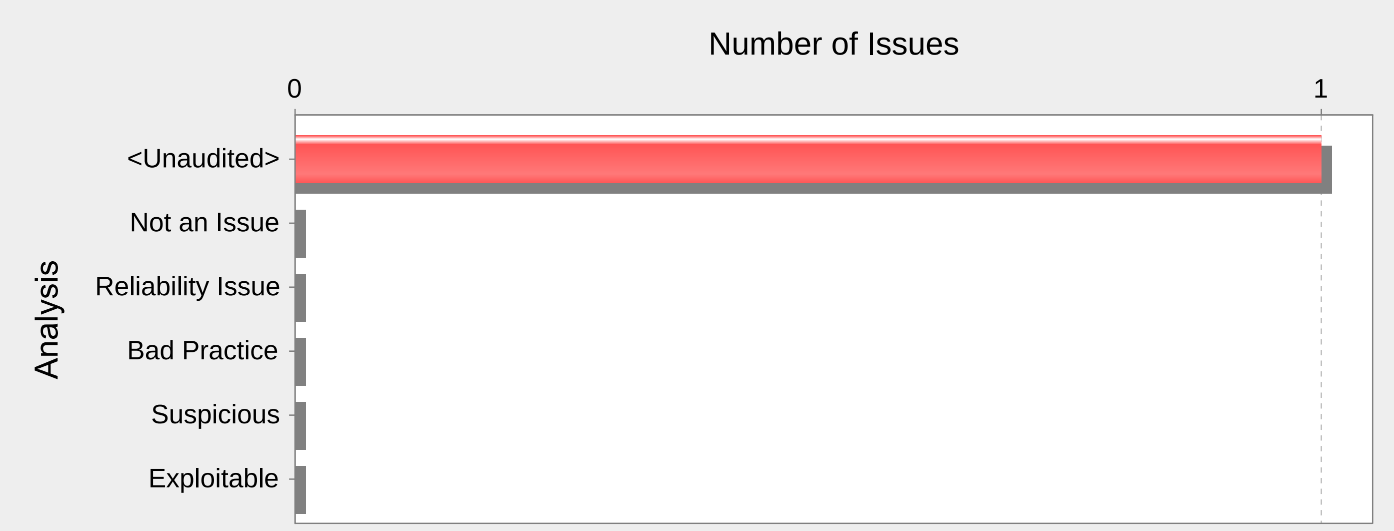
2. To identify null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Fortify Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

## PasswordConstraintValidator.java, line 59 (Password Management: Hardcoded Password)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Security Features | | |
| **Abstract:** | Hardcoded passwords can compromise system security in a way that is not easy to remedy. | | |
| **Sink:** | PasswordConstraintValidator.java:59 FieldAccess: invalidPasswordList() | | |

```
57
58        @Value("${app.invalidPasswordList}")
59        private String invalidPasswordList = "/invalid-password-list.txt";
60
61        @Override
```

## Category: Race Condition: Singleton Member Field (1 Issues)



### Abstract:

The class ProductService is a singleton, so the member field pageSize is shared between users. The result is that one user could see another user's data.

### Explanation:

Many Servlet developers do not understand that a Servlet is a singleton. There is only one instance of the Servlet, and that single instance is used and re-used to handle multiple requests that are processed simultaneously by different threads.

A common result of this misunderstanding is that developers use Servlet member fields in such a way that one user may inadvertently see another user's data. In other words, storing user data in Servlet member fields introduces a data access race condition.

Example 1: The following Servlet stores the value of a request parameter in a member field and then later echoes the parameter value to the response output stream.

```
public class GuestBook extends HttpServlet {

String name;

protected void doPost (HttpServletRequest req, HttpServletResponse res) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
```

While this code will work perfectly in a single-user environment, if two users access the Servlet at approximately the same time, it is possible for the two request handler threads to interleave in the following way:

Thread 1: assign "Dick" to name
Thread 2: assign "Jane" to name
Thread 1: print "Jane, thanks for visiting!"
Thread 2: print "Jane, thanks for visiting!"

Thereby showing the first user the second user's name.

### Recommendations:

Do not use Servlet member fields for anything but constants. (i.e. make all member fields static final).

Developers are often tempted to use Servlet member fields for user data when they need to transport data from one region of code to another. If this is your aim, consider declaring a separate class and using the Servlet only to "wrap" this new class.

Example 2: The bug in Example 1 can be corrected in the following way:

```
public class GuestBook extends HttpServlet {

protected void doPost (HttpServletRequest req, HttpServletResponse res) {
GBRequestHandler handler = new GBRequestHandler();
handler.handle(req, res);
}
}
public class GBRequestHandler {
String name;
public void handle(HttpServletRequest req, HttpServletResponse res) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
```

Alternatively, a Servlet can utilize synchronized blocks to access servlet instance variables but using synchronized blocks may cause significant performance problems.

Please notice that wrapping the field access within a synchronized block will only prevent the issue if all read and write operations on that member are performed within the same synchronized block or method.

Example 3: Wrapping the Example 1 write operation (assignment) in a synchronized block will not fix the problem since the threads will have to get a lock to modify name field, but they will release the lock afterwards, allowing a second thread to change the value again. If, after changing the name value, the first thread resumes execution, the value printed will be the one assigned by the second thread:

```
public class GuestBook extends HttpServlet {
String name;
protected void doPost (HttpServletRequest req, HttpServletResponse res) {
synchronized(name) {
name = req.getParameter("name");
}
...
out.println(name + ", thanks for visiting!");
}
}
```

In order to fix the race condition, all the write and read operations on the shared member field should be run atomically within the same synchronized block:

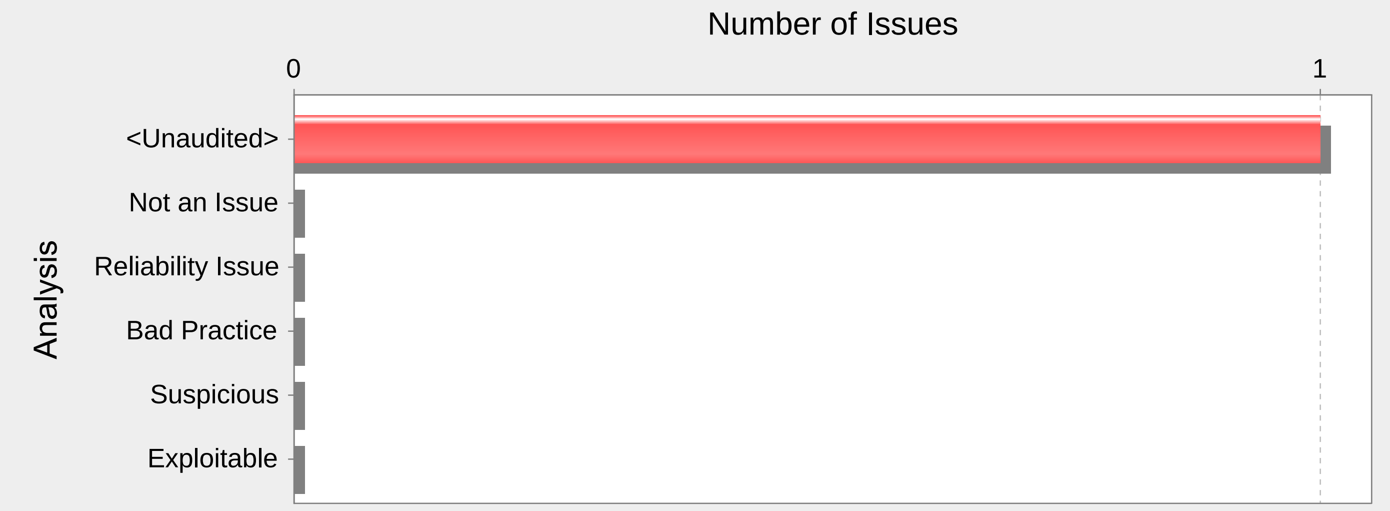
```
public class GuestBook extends HttpServlet {
String name;
protected void doPost (HttpServletRequest req, HttpServletResponse res) {
synchronized(name) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
}
```

### ProductService.java, line 81 (Race Condition: Singleton Member Field)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Time and State | | |
| **Abstract:** | The class ProductService is a singleton, so the member field pageSize is shared between users. The result is that one user could see another user's data. | | |
| **Sink:** | ProductService.java:81 AssignmentStatement() | | |

```
79
80          public void setPageSize(Integer pageSize) {
81              this.pageSize = pageSize;
82          }
```

## Category: Unreleased Resource: Files (1 Issues)



### Abstract:

The function logZipContents() in UserUtils.java sometimes fails to release a file handle allocated by ZipFile() on line 129.

### Explanation:

The program can potentially fail to release a file handle.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems. However, if an attacker can intentionally trigger a resource leak, the attacker may be able to launch a denial of service attack by depleting the resource pool.

Example 1: The following method never closes the file handle it opens. The finalize() method for ZipFile eventually calls close(), but there is no guarantee as to how long it will take before the finalize() method will be invoked. In a busy environment, this can result in the JVM using up all of its file handles.

```
public void printZipContents(String fName)
throws ZipException, IOException, SecurityException, IllegalStateException, NoSuchElementException
{
ZipFile zf = new ZipFile(fName);
Enumeration<ZipEntry> e = zf.entries();

while (e.hasMoreElements()) {
printFileInfo(e.nextElement());
}
}
```

Example 2: Under normal conditions, the following fix properly closes the file handle after printing out all the zip file entries. But if an exception occurs while iterating through the entries, the zip file handle will not be closed. If this happens often enough, the JVM can still run out of available file handles.

```
public void printZipContents(String fName)
throws ZipException, IOException, SecurityException, IllegalStateException, NoSuchElementException
{
ZipFile zf = new ZipFile(fName);
Enumeration<ZipEntry> e = zf.entries();

while (e.hasMoreElements()) {
printFileInfo(e.nextElement());
}
}
```

### Recommendations:

1. Never rely on finalize() to reclaim resources. In order for an object's finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for Example 2 should be rewritten as follows:

```
public void printZipContents(String fName)
throws ZipException, IOException, SecurityException, IllegalStateException, NoSuchElementException
{
ZipFile zf;
try {
zf = new ZipFile(fName);
Enumeration<ZipEntry> e = zf.entries();
...
}
finally {
if (zf != null) {
safeClose(zf);
}
}
}

public static void safeClose(ZipFile zf) {
if (zf != null) {
try {
zf.close();
} catch (IOException e) {
log(e);
}
}
}
```

This solution uses a helper function to log the exceptions that might occur when trying to close the file. Presumably this helper function will be reused whenever a file needs to be closed.

Also, the printZipContents method does not initialize the zf object to null. Instead, it checks to ensure that zf is not null before calling safeClose(). Without the null check, the Java compiler reports that zf might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If  zf is initialized to null in a more complex method, cases in which zf is used without being initialized will not be detected by the compiler.

## UserUtils.java, line 129 (Unreleased Resource: Files)

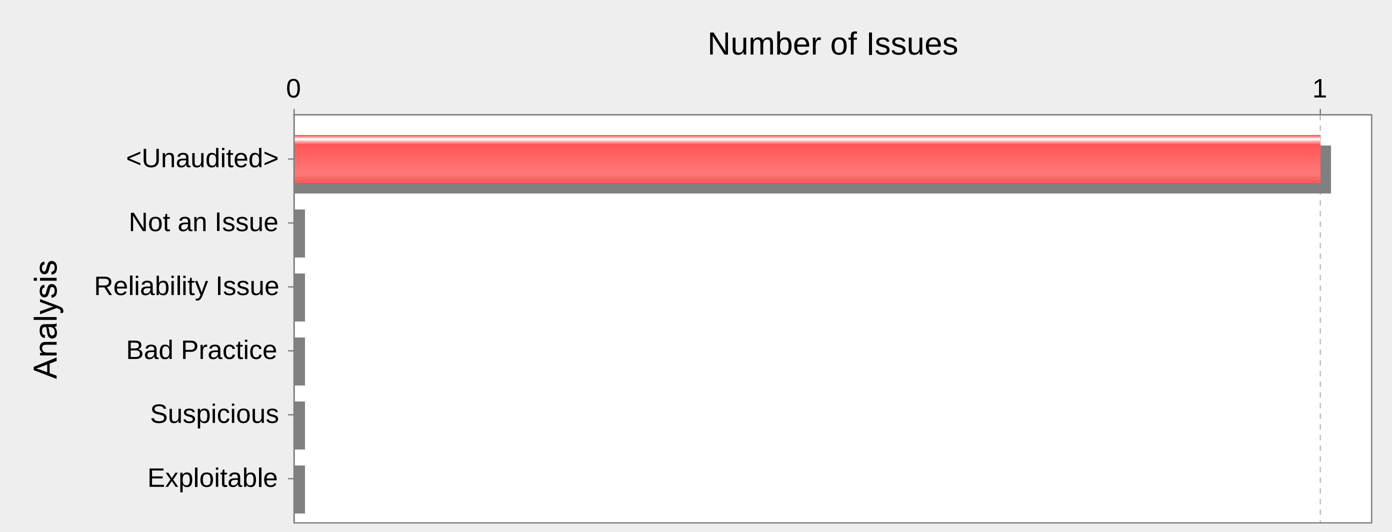
| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Code Quality | | |

**Abstract:** The function logZipContents() in UserUtils.java sometimes fails to release a file handle allocated by ZipFile() on line 129.

**Sink:** UserUtils.java:129 zf = new ZipFile(...)

```
127         public void logZipContents(String fName)
128                 throws IOException, SecurityException, IllegalStateException,
        NoSuchElementException {
129             ZipFile zf = new ZipFile(fName);
130             @SuppressWarnings("unchecked")
131       Enumeration<ZipEntry> e = (Enumeration<ZipEntry>) zf.entries();
```

## Category: Unreleased Resource: Streams (1 Issues)



### Abstract:

The function registerUser() in UserUtils.java sometimes fails to release a system resource allocated by FileReader() on line 81.

### Explanation:

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems. However, if an attacker can intentionally trigger a resource leak, the attacker may be able to launch a denial of service attack by depleting the resource pool.

Example: The following method never closes the file handle it opens. The finalize() method for FileInputStream eventually calls close(), but there is no guarantee as to how long it will take before the finalize() method will be invoked. In a busy environment, this can result in the JVM using up all of its file handles.

```
private void processFile(String fName) throws FileNotFoundException, IOException {
FileInputStream fis = new FileInputStream(fName);
int sz;
byte[] byteArray = new byte[BLOCK_SIZE];
while ((sz = fis.read(byteArray)) != -1) {
processBytes(byteArray, sz);
}
}
```

### Recommendations:

1. Never rely on finalize() to reclaim resources. In order for an object's finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network to a database, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for the Example should be rewritten as follows:

```
public void processFile(String fName) throws FileNotFoundException, IOException {
FileInputStream fis;
try {
fis = new FileInputStream(fName);
int sz;
byte[] byteArray = new byte[BLOCK_SIZE];
while ((sz = fis.read(byteArray)) != -1) {
processBytes(byteArray, sz);
}
}
finally {
if (fis != null) {
safeClose(fis);
}
}
}

public static void safeClose(FileInputStream fis) {
if (fis != null) {
try {
fis.close();
} catch (IOException e) {
log(e);
}
}
}
```

This solution uses a helper function to log the exceptions that might occur when trying to close the stream. Presumably this helper function will be reused whenever a stream needs to be closed.

Also, the processFile method does not initialize the fis object to null. Instead, it checks to ensure that fis is not null before calling safeClose(). Without the null check, the Java compiler reports that fis might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If fis is initialized to null in a more complex method, cases in which fis is used without being initialized will not be detected by the compiler.

## UserUtils.java, line 81 (Unreleased Resource: Streams)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Code Quality | | |

**Abstract:** The function registerUser() in UserUtils.java sometimes fails to release a system resource allocated by FileReader() on line 81.

**Sink:** UserUtils.java:81 new FileReader(...)

```
79              File dataFile = new File(getFilePath(NEWSLETTER_USER_FILE));
80              if (dataFile.exists()) {
81                  jsonArray = (JSONArray) jsonParser.parse(new
FileReader(getFilePath(NEWSLETTER_USER_FILE)));
82              } else {
83                  dataFile.createNewFile();
```

## Issue Count by Category

| Issues by Category | |
|---|---|
| Trust Boundary Violation | 37 |
| System Information Leak: Internal | 22 |
| Mass Assignment: Insecure Binder Configuration | 21 |
| Log Forging (debug) | 20 |
| System Information Leak: External | 18 |
| Access Control: Database | 12 |
| Cross-Site Scripting: Content Sniffing | 11 |
| Password Management: Password in Comment | 11 |
| Path Manipulation | 9 |
| Log Forging | 8 |
| Cross-Site Request Forgery | 7 |
| Cross-Site Scripting: Persistent | 6 |
| JavaScript Hijacking: Vulnerable Framework | 6 |
| Poor Style: Value Never Read | 6 |
| Session Puzzling: Spring | 6 |
| JSON Injection | 5 |
| SQL Injection | 4 |
| System Information Leak | 4 |
| Database Bad Practices: Use of Restricted Accounts | 3 |
| Dead Code: Unused Field | 3 |
| Insecure Randomness | 3 |
| JavaScript Hijacking | 3 |
| Password Management: Password in Configuration File | 3 |
| Header Manipulation | 2 |
| HTML5: Missing Content Security Policy | 2 |
| Mass Assignment: Request Parameters Bound into Persisted Objects | 2 |
| Poor Error Handling: Overly Broad Throws | 2 |
| Cross-Site Scripting: Reflected | 1 |
| HTML5: Cross-Site Scripting Protection | 1 |
| HTML5: Missing Framing Protection | 1 |
| Insecure Randomness: Hardcoded Seed | 1 |
| J2EE Bad Practices: Non-Serializable Object Stored in Session | 1 |
| Null Dereference | 1 |
| Often Misused: Boolean.getBoolean() | 1 |
| Often Misused: File Upload | 1 |
| Open Redirect | 1 |
| Password Management: Hardcoded Password | 1 |
| Poor Error Handling: Overly Broad Catch | 1 |
| Privacy Violation | 1 |
| Race Condition: Singleton Member Field | 1 |
| Resource Injection | 1 |
| Spring Security Misconfiguration: Lack of Fallback Check | 1 |
| Unreleased Resource: Files | 1 |
| Unreleased Resource: Streams | 1 |

## Issue Breakdown by Analysis

### Issues by Analysis