Fortify Security Report

Jan 28, 2020

psdravid

Executive Summary

Issues Overview

On Jan 28, 2020, a source code review was performed over the reportingcompatibility code base. 482 files, 11,574 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 40 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
High	23
Critical	17

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: /srv/openmrs_code/org/openmrs/module/reportingcompatibility
Number of Files: 482
Lines of Code: 11574
Build Label: <No Build Label>

Scan Information

Scan time: 15:16
SCA Engine version: 19.1.0.2241
Machine Name: vclv99-200.hpc.ncsu.edu
Username running scan: psdravid

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:
File System:
java.io.FileInputStream.FileInputStream
java.io.FileInputStream.FileInputStream

Private Information:
java.util.Properties.getProperty

System Information:
null.null.null
java.lang.Throwable.getMessage

Filter Set Summary

Current Enabled Filter Set:
Quick 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
Visibility Filters:
If impact is not in range [2.5, 5.0] Then hide issue
If likelihood is not in range (1.0, 5.0] Then hide issue

Audit Guide Summary

Audit guide not enabled

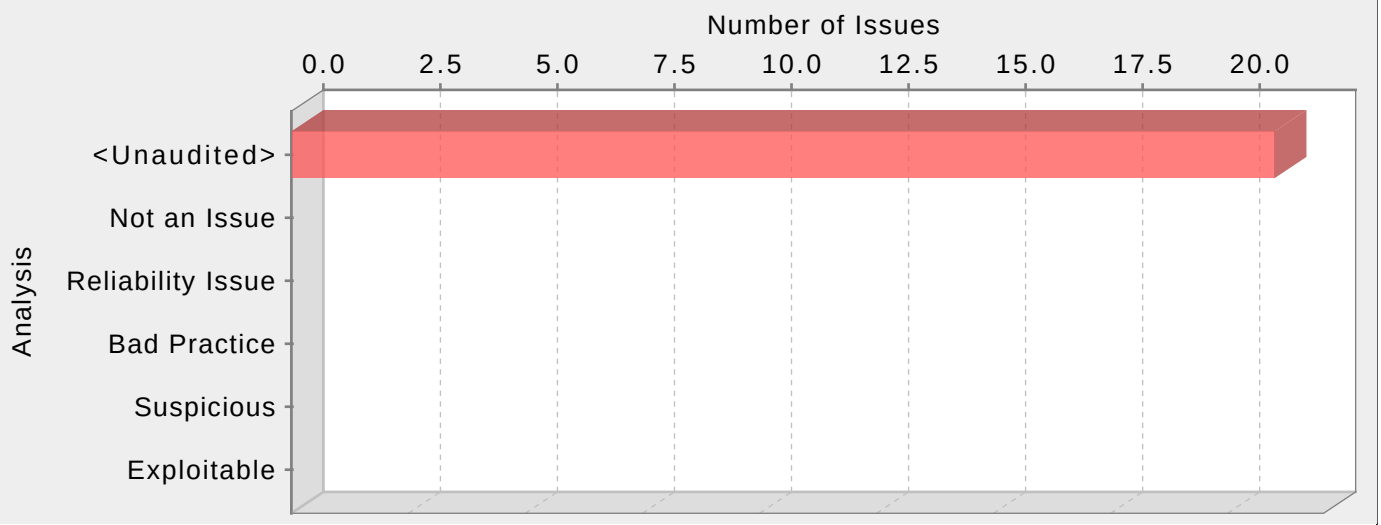
Results Outline

Overall number of results

The scan found 40 issues.

Vulnerability Examples by Category

Category: Log Forging (21 Issues)



Abstract:

The method createCompositionFilter() in CohortSearchHistory.java writes unvalidated user input to the log on line 404. 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 blacklisting. Blacklisting selectively rejects or escapes 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 whitelist of characters that are allowed 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 whitelist.

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, the Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by Fortify Static Code Analyzer 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.

CohortSearchHistory.java, line 404 (Log Forging)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method createCompositionFilter() in CohortSearchHistory.java writes unvalidated user input to the log on line 404. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log.		

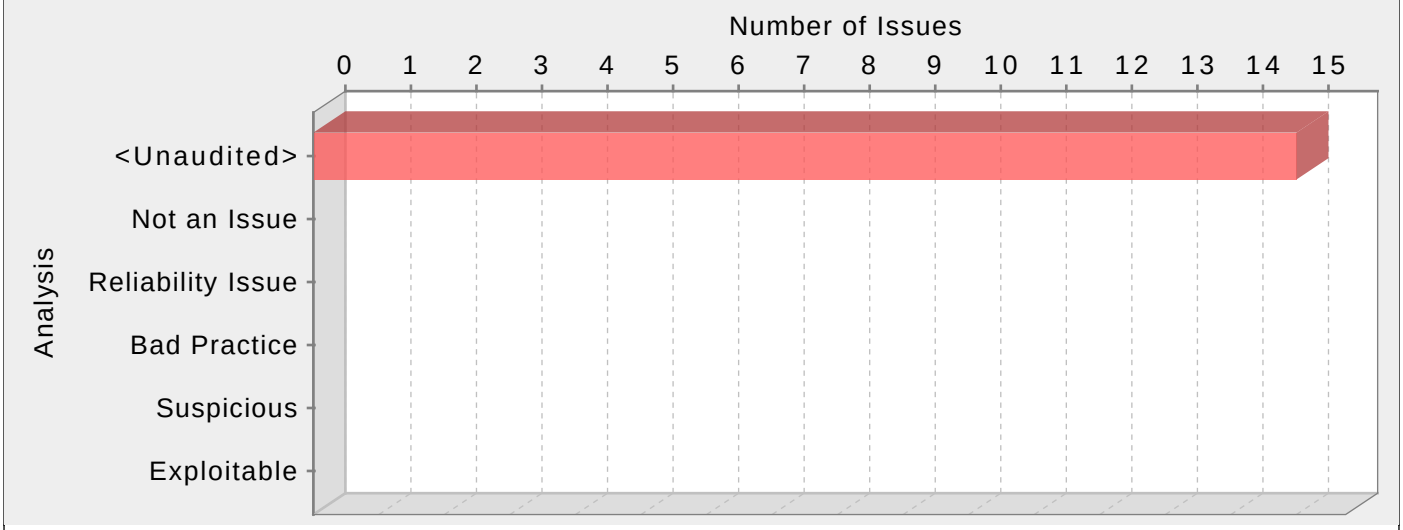
Source: CohortBuilderController.java:403 javax.servlet.ServletRequest.getParameter()

```
401                 log.warn("addCohort(id) didn't find " + cohortId);
402               }
403               temp = request.getParameter("composition");
404               if (temp != null) {
405                PatientSearch ps = history.createCompositionFilter(temp);
```

Sink: CohortSearchHistory.java:404 org.apache.commons.logging.Log.error()

```
402             }
403             catch (Exception ex) {
404               log.error("Error in description string: " + description, ex);
405               return null;
406             }
```

Category: Cross-Site Scripting: Reflected (15 Issues)



Abstract:

The method _jspService() in cohortReportForm.jsp sends unvalidated data to a web browser on line 61, 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 being validated.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may 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 meta-characters or source code, then the code will be 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 world, 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 will execute 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, the rulepacks dynamically re-prioritize the issues reported by Fortify Static Code Analyzer 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.

Since 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 may accept input through a shared data store or other trusted source, and that data store may 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 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 a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric 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 should still be considered valid input once 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 known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order 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 what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts, which is why we do not encourage the use of blacklists 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 with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, 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 in question.

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

- Semicolons, parentheses, curly braces, and new line characters should 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 may 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 in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may 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. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also 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 RTA adds protection against this category.

cohortReportForm.jsp, line 61 (Cross-Site Scripting: Reflected)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method _jspService() in cohortReportForm.jsp sends unvalidated data to a web browser on line 61, which can result in the browser executing malicious code.		

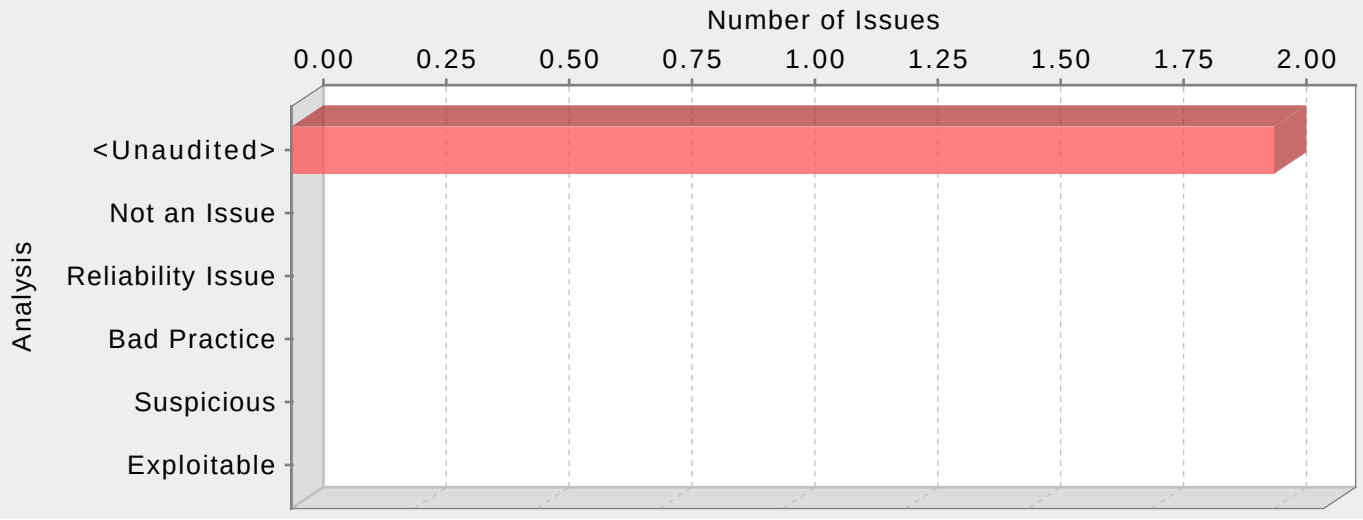
Source: cohortReportForm.jsp:61 javax.servlet.ServletRequest.getParameter()

```
59    <c:forEach var="clazz" items="${parameterClasses}">
60     opt = document.createElement("option");
61     <c:if test="${param.clazz == clazz}">
62      opt.setAttribute("selected", "true");
63     </c:if>
```

Sink: cohortReportForm.jsp:61 javax.servlet.jsp.JspWriter.print()

```
59    <c:forEach var="clazz" items="${parameterClasses}">
60     opt = document.createElement("option");
61     <c:if test="${param.clazz == clazz}">
62      opt.setAttribute("selected", "true");
63     </c:if>
```

Category: Denial of Service: Regular Expression (2 Issues)



Abstract:

Untrusted data is passed to the application and used as a regular expression. This can cause the thread to overconsume CPU resources.

Explanation:

There is a vulnerability in implementations of regular expression evaluators and related methods that can cause the thread to hang when evaluating regular expressions that contain a grouping expression that is itself repeated. Additionally, any regular expression that contains alternate subexpressions that overlap one another can also be exploited. This defect can be used to execute a Denial of Service (DoS) attack.

Example:

(e+)+

([a-zA-Z]+)*

(e|ee)+

There are no known regular expression implementations which are immune to this vulnerability. All platforms and languages are vulnerable to this attack.

Recommendations:

Do not allow untrusted data to be used as regular expression patterns.

EvaluationContext.java, line 354 (Denial of Service: Regular Expression)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	Untrusted data is passed to the application and used as a regular expression. This can cause the thread to overconsume CPU resources.		

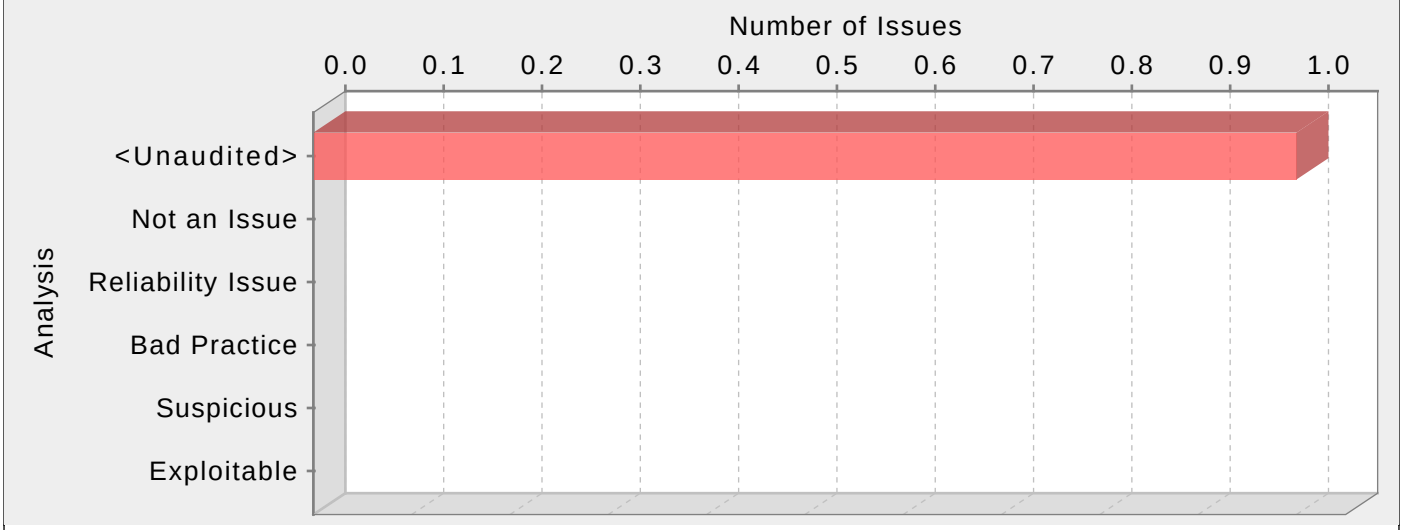
Source: CohortBuilderController.java:523 javax.servlet.ServletRequest.getParameter()

```
521      }
522      argValues
523              .add(new ArgHolder(c, name, isList ? request.getParameterValues(name) :
request.getParameter(name)));
524    }
525
```

Sink: EvaluationContext.java:354 java.lang.String.replaceAll()

```
352           log.debug("Calculated date of: " + foundDate);
353         }
354         replacement = replacement.replaceAll("\\Q" + m.group(0) + "\\E", foundDate);
355         log.debug("Modified to: " + replacement);
356       }
```

Category: Dynamic Code Evaluation: XMLDecoder Injection (1 Issues)



Abstract:

The file ReportObjectXMLDecoder.java deserializes unvalidated XML input using java.beans.XMLDecoder on line 33. Deserializing user-controlled XML documents at run-time can allow attackers to execute malicious arbitrary code on the server.

Explanation:

The JDK XMLEncoder and XMLDecoder classes provide the developer with an easy way to persist objects, serializing them to XML documents. But XMLEncoder also allows a developer to serialize method calls and if an attacker can provide the XML document to be deserialized by XMLDecoder, he may be able to execute any arbitrary code on the server.

Example: The following Java code shows an instance of XMLDecoder processing untrusted input.

```
XMLDecoder decoder = new XMLDecoder(new InputSource(new InputStreamReader(request.getInputStream(), "UTF-8")));
Object object = decoder.readObject();
decoder.close();
```

Example: The following XML document will instantiate a ProcessBuilder object and will invoke its static start() method to run the windows calculator.

```
<java>
<object class="java.lang.ProcessBuilder">
<array class="java.lang.String" length="1" >
<void index="0">
<string>c:\\windows\\system32\\calc.exe</string>
</void>
</array>
<void method="start"/>
</object>
</java>
```

Recommendations:

Unfortunately there is no way to avoid code execution during the XMLDecoder deserialization process. Never use XMLDecoder with user-controlled data, but if there is no alternative, run a strong validation routine to avoid malicious payloads.

Tips:

1. Please disregard this issue if it was found in a Restlet 2.1.4 or later application.

ReportObjectXMLDecoder.java, line 33 (Dynamic Code Evaluation: XMLDecoder Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The file ReportObjectXMLDecoder.java deserializes unvalidated XML input using java.beans.XMLDecoder on line 33. Deserializing user-controlled XML documents at run-time can allow attackers to execute malicious arbitrary code on the server.		
Source:	PatientSearchFormController.java:75 javax.servlet.ServletRequest.getParameter()		

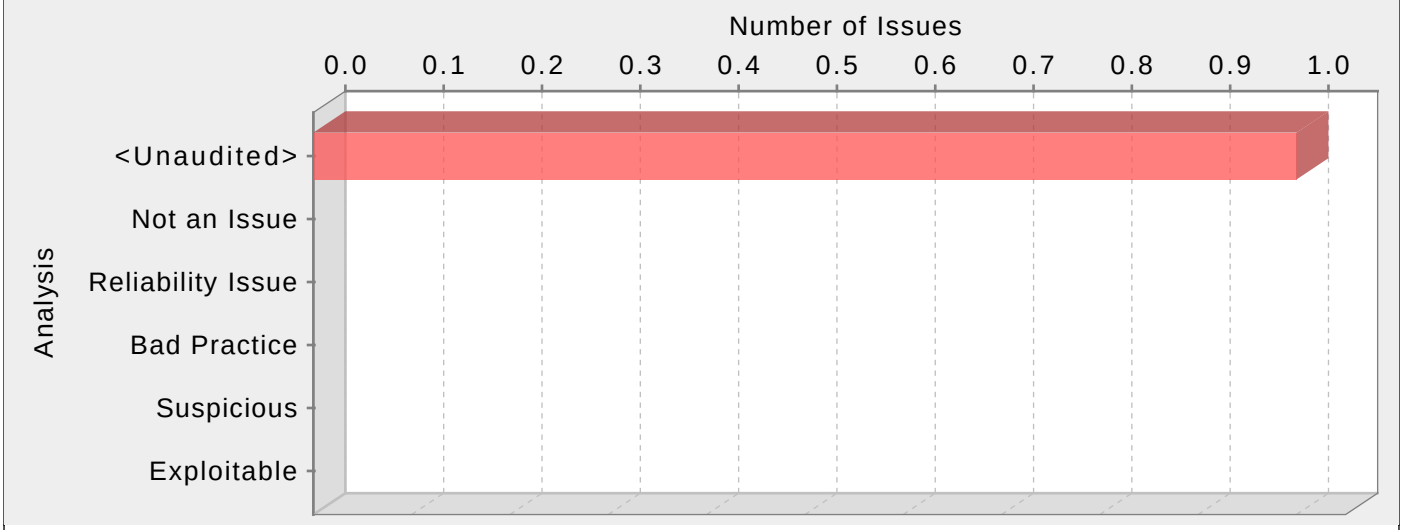
```
73          int hasXMLChanged = 0;
```

```
74                      hasXMLChanged = Integer.parseInt(request.getParameter("patientSearchXMLHasChanged"));
75                      String textAreaXML = request.getParameter("xmlStringTextArea");
76                      Integer argumentsLength = Integer.valueOf(request.getParameter("argumentsSize"));
77                      PatientSearch ps = null;
```

Sink: ReportObjectXMLDecoder.java:33 java.beans.XMLDecoder.XMLDecoder()

```
31                    public AbstractReportObject toAbstractReportObject() {
32                     ExceptionListener exListener = new ReportObjectWrapperExceptionListener();
33                     XMLDecoder dec = new XMLDecoder(new BufferedInputStream(new
                      ByteArrayInputStream(xmlToDecode.getBytes())), null,
34                              exListener);
35                     AbstractReportObject o = (AbstractReportObject) dec.readObject();
```

Category: XML External Entity Injection (1 Issues)



Abstract:

XML parser configured in ReportObjectXMLDecoder.java:33 does not prevent nor limit external entities resolution. This can expose the parser to an XML External Entities attack.

Explanation:

XML External Entities attacks benefit from an XML feature to build documents dynamically at the time of processing. An XML entity allows inclusion of data dynamically from a given resource. External entities allow an XML document to include data from an external URI. Unless configured to do otherwise, external entities force the XML parser to access the resource specified by the URI, e.g., a file on the local machine or on a remote system. This behavior exposes the application to XML External Entity (XXE) attacks, which can be used to perform denial of service of the local system, gain unauthorized access to files on the local machine, scan remote machines, and perform denial of service of remote systems.

The following XML document shows an example of an XXE attack.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

This example could crash the server (on a UNIX system), if the XML parser attempts to substitute the entity with the contents of the /dev/random file.

Recommendations:

The XML unmarshaller should be configured securely so that it does not allow external entities as part of an incoming XML document.

To avoid XXE injection do not use unmarshal methods that process an XML source directly as java.io.File, java.io.Reader or java.io.InputStream. Parse the document with a securely configured parser and use an unmarshal method that takes the secure parser as the XML source as shown in the following example:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.parse(<XML Source>);
Model model = (Model) u.unmarshal(document);
```

Tips:

1. Fortify RTA adds protection against this category.

ReportObjectXMLDecoder.java, line 33 (XML External Entity Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	XML parser configured in ReportObjectXMLDecoder.java:33 does not prevent nor limit external entities resolution. This can expose the parser to an XML External Entities attack.		
Source:	PatientSearchFormController.java:75 javax.servlet.ServletRequest.getParameter()		

```
73              int hasXMLChanged = 0;
```

```
74                      hasXMLChanged = Integer.parseInt(request.getParameter("patientSearchXMLHasChanged"));
75                      String textAreaXML = request.getParameter("xmlStringTextArea");
76                      Integer argumentsLength = Integer.valueOf(request.getParameter("argumentsSize"));
77                      PatientSearch ps = null;
```

Sink: ReportObjectXMLDecoder.java:33 java.beans.XMLDecoder.XMLDecoder()

```
31                    public AbstractReportObject toAbstractReportObject() {
32                      ExceptionListener exListener = new ReportObjectWrapperExceptionListener();
33                      XMLDecoder dec = new XMLDecoder(new BufferedInputStream(new
                      ByteArrayInputStream(xmlToDecode.getBytes())), null,
34                              exListener);
35                      AbstractReportObject o = (AbstractReportObject) dec.readObject();
```

Issue Count by Category

Issues by Category	
Log Forging	21
Cross-Site Scripting: Reflected	15
Denial of Service: Regular Expression	2
Dynamic Code Evaluation: XMLDecoder Injection	1
XML External Entity Injection	1

Issue Breakdown by Analysis

Issues by Analysis