Fortify Developer Workbook

Jan 30, 2019

VHAHAMWandlJ

Report Overview

Report Summary

On Jan 28, 2019, a source code review was performed over the Inbound code base. 2,175 files, 36,222 LOC (Executable) were scanned. A total of 329 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of issues found in this project. Specific examples and source code are provided for each issue type.

Issues by Fortify Priority Order	
Low	316
Medium	8
High	5

Issue Summary

Overall number of results

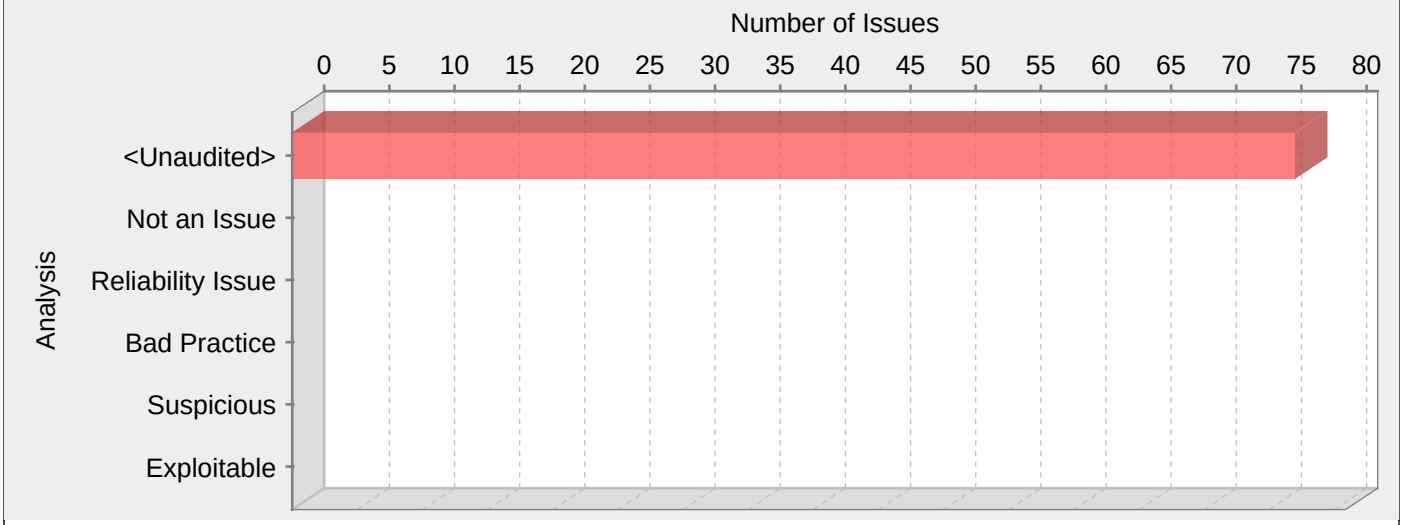
The scan found 329 issues.

Issues by Category	
Obsolete: Deprecated by ESAPI	77
Cross-Site Scripting: Poor Validation	40
Poor Style: Value Never Read	31
Poor Error Handling: Overly Broad Catch	29
Trust Boundary Violation	29
System Information Leak: HTML Comment in JSP	27
Password Management: Password in Comment	18
Poor Error Handling: Overly Broad Throws	17
Hidden Field	11
Missing Check against Null	11
System Information Leak: External	8
Build Misconfiguration: External Maven Dependency Repository	6
Cross-Site Request Forgery	6
Password Management	5
Code Correctness: Misleading Method Signature	4
Password Management: Hardcoded Password	4
Access Control: Database	2
Denial of Service	1
J2EE Misconfiguration: Excessive Session Timeout	1
Poor Error Handling: Throw Inside Finally	1
SQL Injection	1

Results Outline

Vulnerability Examples by Category

Category: Obsolete: Deprecated by ESAPI (77 Issues)



Abstract:

The call to print() in StreamUtilities.java on line 15 should be replaced with an ESAPI call.

Explanation:

The ESAPI secure coding guidelines contain a list of banned APIs for which a safer alternative is available in ESAPI.

The list of banned and substitute API's:

Banned 001 System.out.println()
Banned 002 Throwable.printStackTrace()
Banned 003 Runtime.exec()
Banned 004 Session.getId()
Banned 005 ServletRequest.getUserPrincipal()
Banned 006 ServletRequest.isUserInRole()
Banned 007 Session.invalidate()
Banned 008 Math.Random.*
Banned 009 File.createTempFile()
Banned 010 ServletResponse.setContentType()
Banned 011 ServletResponse.sendRedirect()
Banned 012 RequestDispatcher.forward()
Banned 013 ServletResponse.addHeader()
Banned 014 ServletResponse.addCookie()
Banned 015 ServletRequest.isSecure()
Banned 016 Properties.*
Banned 017 ServletContext.log()
Banned 018 java.security and javax.crypto
Banned 019 java.net.URLEncoder/Decoder
Banned 021 ServletResponse.encodeURL
Banned 022 ServletResponse.encodeRedirectURL
Banned 023 javax.servlet.ServletInputStream.readLine

Recommendations:

Replace banned API calls with the recommended safer version provided by ESAPI.

The list of banned and replacement API's:

Banned 001 System.out.println()
ESAPI Replacement: Logger.*
Banned 002 Throwable.printStackTrace()

ESAPI Replacement: Logger.*
Banned 003 Runtime.exec()
ESAPI Replacement: Executor.executeSystemCommand()
Banned 004 Session.getId()
ESAPI Replacement: Randomizer.getRandom* (better not to use at all)
Banned 005 ServletRequest.getUserPrincipal()
ESAPI Replacement: Authenticator.getCurrentUser()
Banned 006 ServletRequest.isUserInRole()
ESAPI Replacement: AccessController.isAuthorized*()
Banned 007 Session.invalidate()
ESAPI Replacement: Authenticator.logout()
Banned 008 Math.Random.*
ESAPI Replacement: Randomizer.*
Banned 009 File.createTempFile()
ESAPI Replacement: Randomizer.getRandomFilename()
Banned 010 ServletResponse.setContentType()
ESAPI Replacement: HTTPUtilities.setContentType()
Banned 011 ServletResponse.sendRedirect()
ESAPI Replacement: HTTPUtilities.sendRedirect()
Banned 012 RequestDispatcher.forward()
ESAPI Replacement: HTTPUtilities.sendForward()
Banned 013 ServletResponse.addHeader()
ESAPI Replacement: HTTPUtilities.setHeader()/addHeader()
Banned 014 ServletResponse.addCookie()
ESAPI Replacement: HTTPUtilities.addCookie()
Banned 015 ServletRequest.isSecure()
ESAPI Replacement: HTTPUtilties.assertSecureChannel()
Banned 016 Properties.*
ESAPI Replacement: EncryptedProperties.*
Banned 017 ServletContext.log()
ESAPI Replacement: Logger.*
Banned 018 java.security and javax.crypto
ESAPI Replacement: Encryptor.*
Banned 019 java.net.URLEncoder/Decoder
ESAPI Replacement: Encoder.encodeForURL()/decodeForURL()
Banned 021 ServletResponse.encodeURL
ESAPI Replacement: Encoder.encodeForURL()/decodeForURL()
Banned 022 ServletResponse.encodeRedirectURL
ESAPI Replacement: Encoder.encodeForURL()/decodeForURL()
ESAPI Replacement: HTTPUtilities.sendRedirect()
Banned 023 javax.servlet.ServletInputStream.readLine
ESAPI Replacement: Validator.safeReadLine()

VistaLinkConnectionUtility.java, line 76 (Obsolete: Deprecated by ESAPI)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The call to load() in VistaLinkConnectionUtility.java on line 76 should be replaced with an ESAPI call.		
Sink:	VistaLinkConnectionUtility.java:76 load()		

```
74
75                      if(null!=inputStream){
76                       properties.load(inputStream);
77                       proxyUser = properties.getProperty(VISTALINK_PROXYUSER).trim();
```

```
78                    }
```

StreamUtilities.java, line 35 (Obsolete: Deprecated by ESAPI)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The call to print() in StreamUtilities.java on line 35 should be replaced with an ESAPI call.		
Sink:	StreamUtilities.java:35 FunctionCall: print()		

```
33                    bufferedReader.close();
34                  } catch (IOException e) {
35                    System.out.print("StreamUtilities: Unable to close bufferedReader");
36                  }
37                }
```

StreamUtilities.java, line 15 (Obsolete: Deprecated by ESAPI)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The call to print() in StreamUtilities.java on line 15 should be replaced with an ESAPI call.		
Sink:	StreamUtilities.java:15 FunctionCall: print()		

```
13                    fileInputStream.close();
14                  } catch (IOException e) {
15                    System.out.print("StreamUtilities: Unable to close fileInputStream");
16                  }
17                }
```

VistaLinkConnectionUtility.java, line 68 (Obsolete: Deprecated by ESAPI)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The call to Properties() in VistaLinkConnectionUtility.java on line 68 should be replaced with an ESAPI call.		
Sink:	VistaLinkConnectionUtility.java:68 Properties()		

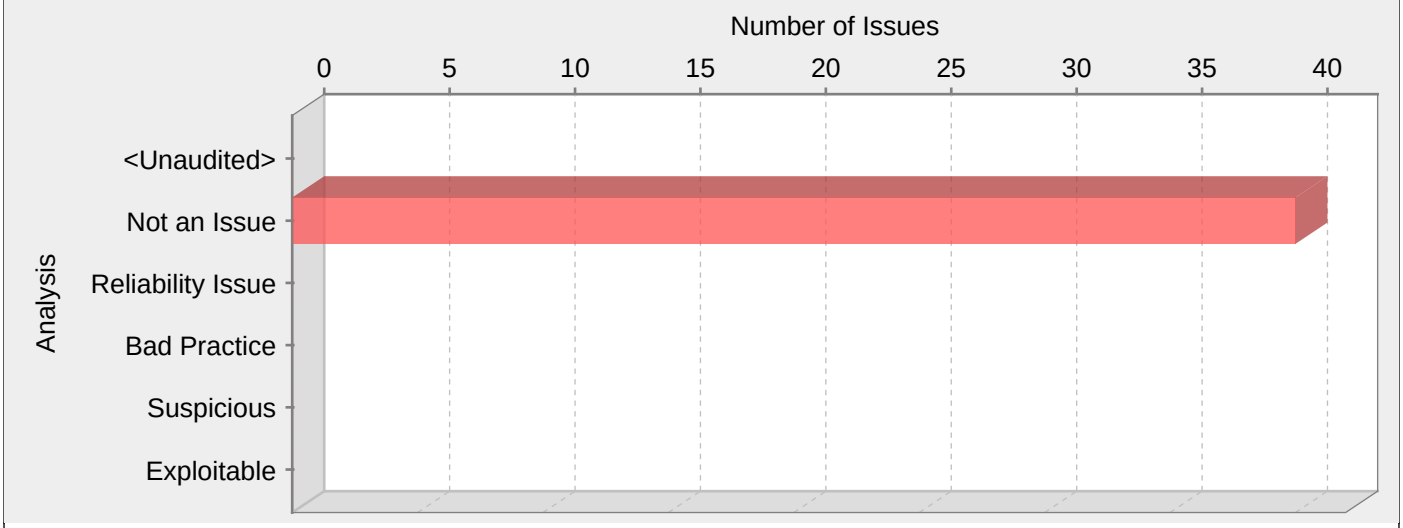
```
66
67
68                    Properties properties = new Properties();
69
70                  InputStream inputStream = null;
```

StreamUtilities.java, line 25 (Obsolete: Deprecated by ESAPI)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The call to print() in StreamUtilities.java on line 25 should be replaced with an ESAPI call.		
Sink:	StreamUtilities.java:25 FunctionCall: print()		

```
23                    inputStream.close();
24                  } catch (IOException e) {
25                  System.out.print("StreamUtilities: Unable to close inputStream");
26                  }
27                }
```

Category: Cross-Site Scripting: Poor Validation (40 Issues)



Abstract:

The method _jspService() in addpharmacy.jsp uses HTML, XML or other type of encoding that is not always enough to prevent malicious code from reaching the web browser.

Explanation:

The use of certain encoding constructs, such as the <c:out/> tag with the escapeXml="true" attribute (the default behavior), will prevent some, but not all cross-site scripting attacks. Depending on the context in which the data appear, characters beyond the basic <, >, &, and " that are HTML-encoded and those beyond <, >, &, ", and ' that are XML-encoded may take on meta-meaning. Relying on such encoding constructs is equivalent to using a weak blacklist to prevent cross-site scripting and might allow an attacker to inject malicious code that will be executed in the browser. Because accurately identifying the context in which the data appear statically is not always possible, Fortify Static Code Analyzer reports cross-site scripting findings even when encoding is applied and presents them as Cross-Site Scripting: Poor Validation issues.

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of reflected XSS, an untrusted source is most frequently a web request, and in the case of persistent (also known as stored) XSS -- it is the results of a database query.

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 via the <c:out/> tag.

Employee ID: <c:out value="${param.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 that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail 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 via the <c:out/> tag.

```
<%...
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: <c:out value="${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(URLEncoder.encode(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 e-mailed 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 for performing validation of user input. Struts and Spring MVC are two examples. 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 above 128 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. To differentiate between the data that are encoded and those that are not, use the Data Validation project template that groups the issues into folders based on the type of encoding applied to their source of input.

4. Fortify RTA adds protection against this category.

addpharmacy.jsp, line 119 (Cross-Site Scripting: Poor Validation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The method _jspService() in addpharmacy.jsp uses HTML, XML or other type of encoding that is not always enough to prevent malicious code from reaching the web browser.		
Source:	PharmacyManagementController.java:476 addNewPharmacy(1)		

```
474
475    @RequestMapping(value = "/addNewPharmacy",  method = RequestMethod.POST)
476    public ModelAndView addNewPharmacy(HttpServletRequest request,@Valid
       @ModelAttribute("pharmacyAddForm") PharmacyForm pharmacyForm, BindingResult
       bindingResult) throws  IOException {
477
478      LOG.info("Trying to persist pharmacy information.");
```

Sink: addpharmacy.jsp:119 javax.servlet.jsp.JspWriter.print()

```
117    Pharmacy Address Line 2:
118    </label>
119    <input id="pharmacyAddressLine2" class="editable pharmAddScreen" type="text" value="${
       pharmacyInfo.pharmacyAddressLine2}"  size="20" maxlength="35" style="display: block;
       opacity: 1;" name="pharmacyAddressLine2" title="Second line of pharmacy's address">
120    </div>
121
```

Analysis: Not an Issue

Comments:	*VHADURButtS 2018-11-07 2:15 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.

addpharmacy.jsp, line 61 (Cross-Site Scripting: Poor Validation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The method _jspService() in addpharmacy.jsp uses HTML, XML or other type of encoding that is not always enough to prevent malicious code from reaching the web browser.		
Source:	PharmacyManagementController.java:476 addNewPharmacy(1)		

```
474
475    @RequestMapping(value = "/addNewPharmacy",  method = RequestMethod.POST)
476    public ModelAndView addNewPharmacy(HttpServletRequest request,@Valid
       @ModelAttribute("pharmacyAddForm") PharmacyForm pharmacyForm, BindingResult
       bindingResult) throws  IOException {
477
478      LOG.info("Trying to persist pharmacy information.");
```

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

```
59                 </span>
60              </label>
61              <input id="storeName" class="editable pharmAddScreen" type="text" value="${
pharmacyInfo.storeName}"  size="20" maxlength="35" style="display: block; opacity: 1;"
name="storeName" title="Pharmacy's name. This is the published name visible to outside
providers">
62              </div>
63
```

Analysis:	Not an Issue		
Comments:	*VHADURButtS 2018-11-07 2:15 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.	

addpharmacy.jsp, line 112 (Cross-Site Scripting: Poor Validation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The method _jspService() in addpharmacy.jsp uses HTML, XML or other type of encoding that is not always enough to prevent malicious code from reaching the web browser.		
Source:	PharmacyManagementController.java:476 addNewPharmacy(1)		

```
474
475             @RequestMapping(value = "/addNewPharmacy",  method = RequestMethod.POST)
476             public ModelAndView addNewPharmacy(HttpServletRequest request,@Valid
@ModelAttribute("pharmacyAddForm") PharmacyForm pharmacyForm, BindingResult
bindingResult) throws  IOException {
477
478               LOG.info("Trying to persist pharmacy information.");
```

Sink:	addpharmacy.jsp:112 javax.servlet.jsp.JspWriter.print()

```
110                 </span>
111              </label>
112              <input id="pharmacyAddressLine1" class="editable pharmAddScreen" type="text" value="${
pharmacyInfo.pharmacyAddressLine1}"  size="20" maxlength="35" style="display: block;
opacity: 1;" name="pharmacyAddressLine1" title="First line of pharmacy's address">
113              </div>
114
```

Analysis:	Not an Issue		
Comments:	*VHADURButtS 2018-11-07 2:15 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.	

addpharmacy.jsp, line 102 (Cross-Site Scripting: Poor Validation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The method _jspService() in addpharmacy.jsp uses HTML, XML or other type of encoding that is not always enough to prevent malicious code from reaching the web browser.		
Source:	PharmacyManagementController.java:476 addNewPharmacy(1)		

```
474
475             @RequestMapping(value = "/addNewPharmacy",  method = RequestMethod.POST)
476             public ModelAndView addNewPharmacy(HttpServletRequest request,@Valid
@ModelAttribute("pharmacyAddForm") PharmacyForm pharmacyForm, BindingResult
bindingResult) throws  IOException {
477
478               LOG.info("Trying to persist pharmacy information.");
```

Sink:	addpharmacy.jsp:102 javax.servlet.jsp.JspWriter.print()

```
100                 </span>
101              </label>
102              <input id="divisionName" class="editable pharmAddScreen" type="text" value="${
pharmacyInfo.divisionName}"  size="20" maxlength="35" style="display: block; opacity:
1;" name="divisionName" title="Pharmacy's division name">
```

```
103                 </div>
```

Analysis:	Not an Issue	
Comments:	*VHADURButtS 2018-11-07 2:15 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.

addpharmacy.jsp, line 92 (Cross-Site Scripting: Poor Validation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The method _jspService() in addpharmacy.jsp uses HTML, XML or other type of encoding that is not always enough to prevent malicious code from reaching the web browser.		

Source: PharmacyManagementController.java:476 addNewPharmacy(1)

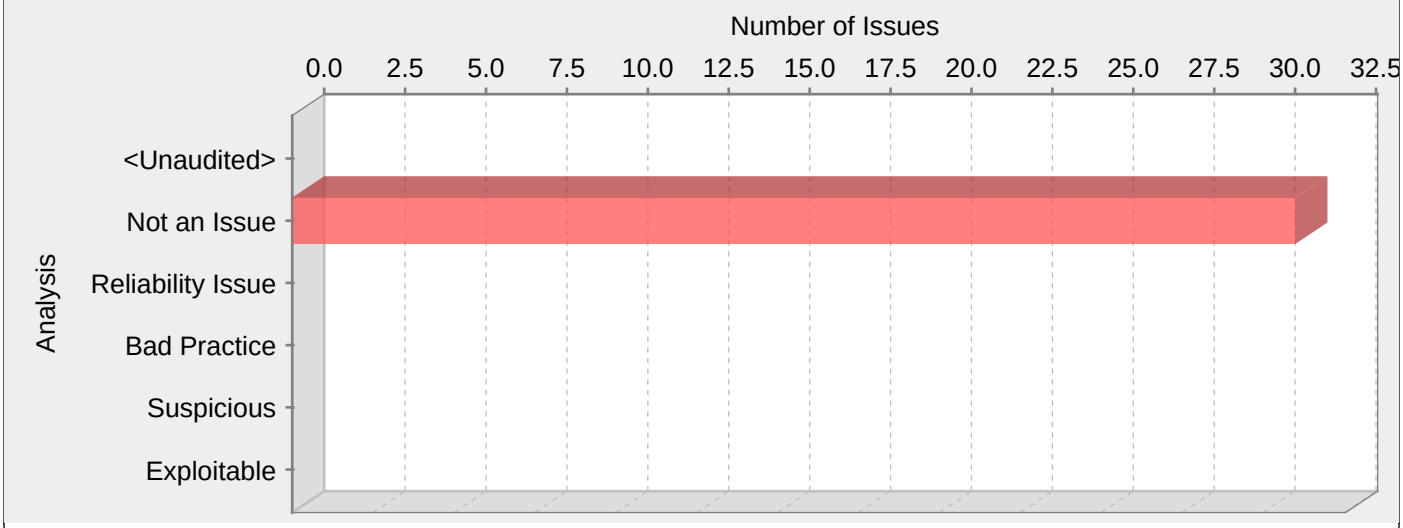
```
474
475                 @RequestMapping(value = "/addNewPharmacy",  method = RequestMethod.POST)
476                 public ModelAndView addNewPharmacy(HttpServletRequest request,@Valid
@ModelAttribute("pharmacyAddForm") PharmacyForm pharmacyForm, BindingResult
bindingResult) throws  IOException {
477
478                   LOG.info("Trying to persist pharmacy information.");
```

Sink: addpharmacy.jsp:92 javax.servlet.jsp.JspWriter.print()

```
90                     </span>:
91                  </label>
92                  <input id="vaStationId" class="editable pharmAddScreen" type="text" value="${
pharmacyInfo.vaStationId}"  size="20" maxlength="10" style="display: block; opacity:
1;" name="vaStationId" title="VA Station ID of pharmacy">
93                  </div>
94
```

Analysis:	Not an Issue	
Comments:	*VHADURButtS 2018-11-07 2:15 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.

Category: Poor Style: Value Never Read (31 Issues)



Abstract:

The method getAllStationIds() in UserManagementController.java never uses the value it assigns to the variable tempBuf on line 134.

Explanation:

This variable's value is not used. After the assignment, the variable is either assigned another value or goes out of scope.

Example: The following code excerpt assigns to the variable r and then overwrites the value without using it.

```
r = getName();
r = getNewBuffer(buf);
```

Recommendations:

Remove unnecessary assignments in order to make the code easier to understand and maintain.

EESummary_Client.java, line 173 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method setupTLS() in EESummary_Client.java never uses the value it assigns to the variable file on line 173.		
Sink:	EESummary_Client.java:173 VariableAccess: file()		

```
171     }
172
173     file = null;
174     tlsCP = null;
175     fileIn2 = null;
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2018-03-13 2:32 PM*	Explicitly assigning a null value to variables that are no longer needed to aid garbage collection.	

EESummary_Client.java, line 175 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method setupTLS() in EESummary_Client.java never uses the value it assigns to the variable fileIn2 on line 175.		
Sink:	EESummary_Client.java:175 VariableAccess: fileIn2()		

```
173     file = null;
174     tlsCP = null;
175     fileIn2 = null;
176   }
177
```

Analysis:	Not an Issue

Comments:	*VHADURButtS 2018-11-07 2:20 PM*	Explicitly assigning a null value to variables that are no longer needed to aid garbage collection.	

InboundNCPDPMessageServiceImpl.java, line 404 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method getInboundMessage() in InboundNCPDPMessageServiceImpl.java never uses the value it assigns to the variable inboundeRx on line 404.		
Sink:	InboundNCPDPMessageServiceImpl.java:404 VariableAccess: inboundeRx()		

```
402    StreamUtilities.safeClose(inputStream);
403
404    inboundeRx = null;
405    relatesToMessageID = null;
406    messageFrom = null;
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2018-03-13 2:32 PM*	Explicitly assigning a null value to variables that are no longer needed to aid garbage collection.	

EESummary_Client.java, line 174 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method setupTLS() in EESummary_Client.java never uses the value it assigns to the variable tlsCP on line 174.		
Sink:	EESummary_Client.java:174 VariableAccess: tlsCP()		

```
172
173    file = null;
174    tlsCP = null;
175    fileIn2 = null;
176   }
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2018-03-13 2:32 PM*	Explicitly assigning a null value to variables that are no longer needed to aid garbage collection.	

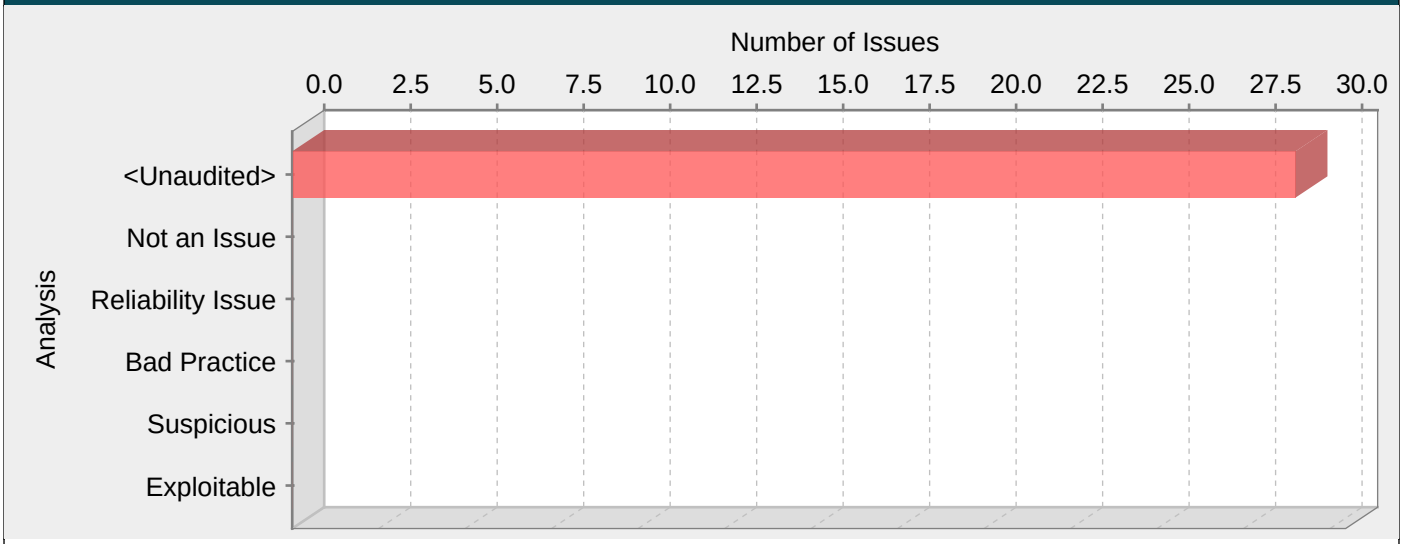
UserManagementController.java, line 134 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method getAllStationIds() in UserManagementController.java never uses the value it assigns to the variable tempBuf on line 134.		
Sink:	UserManagementController.java:134 VariableAccess: tempBuf()		

```
132    }
133
134    tempBuf = null;
135   return stationIdsList;
136  }
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2018-03-13 2:32 PM*	Explicitly assigning a null value to variables that are no longer needed to aid garbage collection.	

Category: Poor Error Handling: Overly Broad Catch (29 Issues)



Abstract:

The catch block at ESAPIValidator.java line 98 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.

Explanation:

Multiple catch blocks can get repetitive, but "condensing" catch blocks by catching a high-level class such as Exception can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. Catching an overly broad exception essentially defeats the purpose of Java's typed exceptions, and can become particularly dangerous if the program grows and begins to throw new types of exceptions. The new exception types will not receive any attention.

Example: The following code excerpt handles three types of exceptions in an identical fashion.

```
try {
doExchange();
}
catch (IOException e) {
logger.error("doExchange failed", e);
}
catch (InvocationTargetException e) {
logger.error("doExchange failed", e);
}
catch (SQLException e) {
logger.error("doExchange failed", e);
}
```

At first blush, it may seem preferable to deal with these exceptions in a single catch block, as follows:

```
try {
doExchange();
}
catch (Exception e) {
logger.error("doExchange failed", e);
}
```

However, if doExchange() is modified to throw a new type of exception that should be handled in some different kind of way, the broad catch block will prevent the compiler from pointing out the situation. Further, the new catch block will now also handle exceptions derived from RuntimeException such as ClassCastException, and NullPointerException, which is not the programmer's intent.

Recommendations:

Do not catch broad exception classes such as Exception, Throwable, Error, or RuntimeException except at the very top level of the program or thread.

Tips:

1. The Fortify Secure Coding Rulepacks will not flag an overly broad catch block if the catch block in question immediately throws a new exception.

VAIdM.java, line 30 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at VAIdM.java line 30 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: VAIdM.java:30 CatchBlock()

```
28                  try {
29                      url = VAIdM.class.getResource("wsdl/mvi/PREeRX_IdmHl7v3_FlatFile.wsdl");
30                  } catch (Exception e) {
31                      java.util.logging.Logger.getLogger(VAIdM.class.getName())
32                          .log(java.util.logging.Level.INFO,
```

EeSummaryPortService.java, line 33 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at EeSummaryPortService.java line 33 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: EeSummaryPortService.java:33 CatchBlock()

```
31                  try {
32                      url = EeSummaryPortService.class.getResource("wsdl/eAnde/eeSummary.wsdl");
33                  } catch (Exception e) {
34                      java.util.logging.Logger.getLogger(EeSummaryPortService.class.getName())
35                          .log(java.util.logging.Level.INFO,
```

PharmacyMigrationServiceImpl.java, line 36 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at PharmacyMigrationServiceImpl.java line 36 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: PharmacyMigrationServiceImpl.java:36 CatchBlock()

```
34                    pharmacyMigration = pharmacyMigrationDao.findByNCPDPId(NCPDPId);
35
36                  } catch (Exception e) {
37
38
```

ESAPIValidator.java, line 257 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The catch block at ESAPIValidator.java line 257 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		

Sink: ESAPIValidator.java:257 CatchBlock()

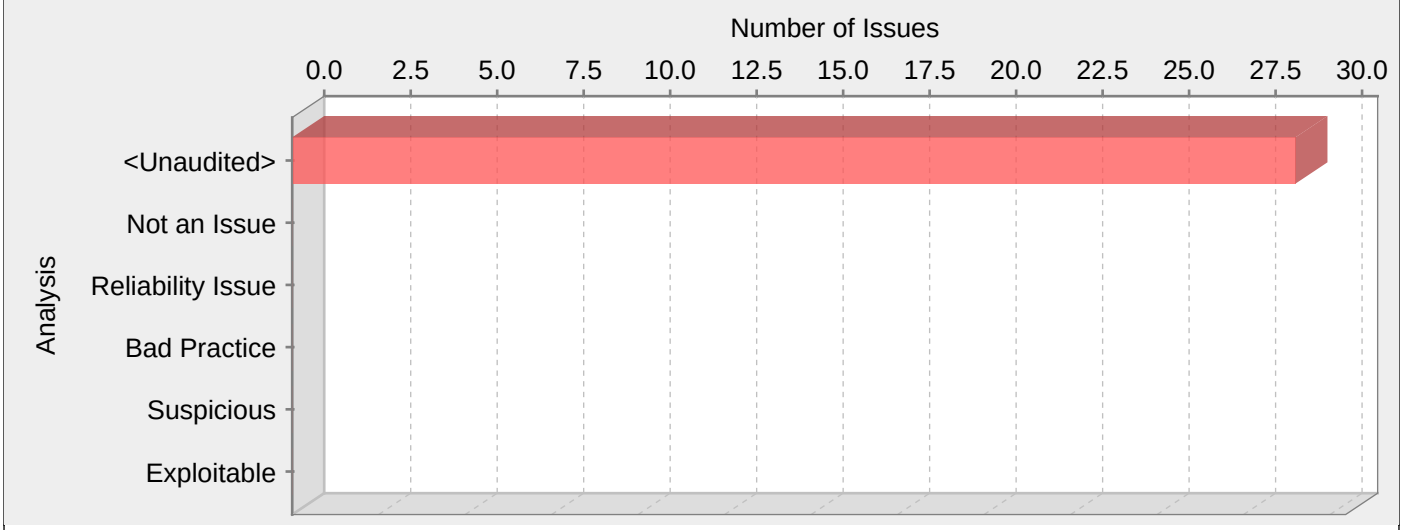
```
255                     validateStringInput(input, type);
256                   return true;
257                 } catch( Exception e ) {
258                   return false;
259                 }
```

ESAPIValidator.java, line 98 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		

Abstract:	The catch block at ESAPIValidator.java line 98 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.
Sink:	ESAPIValidator.java:98 CatchBlock()

```
96                  try {
97                      return validator().getValidInput("logForging", input, "logForging",
Integer.MAX_VALUE, false, false);
98                  } catch (Exception e) {
99                      return null;
100                 }
```

Category: Trust Boundary Violation (29 Issues)



Abstract:

The method getMainPage() in PharmacyManagementController.java commingles trusted and untrusted data in the same data structure, which encourages programmers to mistakenly trust unvalidated data.

Explanation:

A trust boundary can be thought of as line drawn through a program. On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy. The purpose of validation logic is to allow data to safely cross the trust boundary--to move from untrusted to trusted.

A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted. The most common way to make this mistake is to allow trusted and untrusted data to commingle in the same data structure.

Example: The following Java code accepts an HTTP request and stores the usrname parameter in the HTTP session object before checking to ensure that the user has been authenticated.

```
usrname = request.getParameter("usrname");
if (session.getAttribute(ATTR_USR) != null) {
session.setAttribute(ATTR_USR, usrname);
}
```

Without well-established and maintained trust boundaries, programmers will inevitably lose track of which pieces of data have been validated and which have not. This confusion will eventually allow some data to be used without first being validated.

Recommendations:

Define clear trust boundaries in the application. Do not use the same data structure to hold trusted data in some contexts and untrusted data in other contexts. Minimize the number of ways that data can move across a trust boundary.

Trust boundary violations sometimes occur when input needs to be built up over a series of user interactions before being processed. It may not be possible to do complete input validation until all of the data has arrived. In these situations, it is still important to maintain a trust boundary. The untrusted data should be built up in a single untrusted data structure, validated, and then moved into a trusted location.

Tips:

1. Do not feel that you need to find a "smoking gun" situation in which data that has not been validated is assumed to be trusted. If trust boundaries are not clearly delineated and respected, validation errors are inevitable. Instead of spending time searching for an exploitable scenario, concentrate on teaching programmers to create good trust boundaries.

2. Most programs have trust boundaries that are defined by the semantics of the application. Consider writing custom rules to check for other places where user input crosses a trust boundary.

3. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Spring MVC are two examples. 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.

PharmacyManagementController.java, line 109 (Trust Boundary Violation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		

Abstract:	The method getMainPage() in PharmacyManagementController.java commingles trusted and untrusted data in the same data structure, which encourages programmers to mistakenly trust unvalidated data.
Source:	VisnSelectDaoImpl.java:94 org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate.query()

```
92
93                    try {
94                     visnList = jdbcTemplate.query(sql,params, new VisnListRowMapper());
95                    } catch (DataAccessException e) {
96
```

Sink:	PharmacyManagementController.java:109 org.springframework.web.servlet.ModelAndView.addObject()

```
107                   List<VisnSelectModel>  visnList = getVisnSelect(currentUser.getVaStationIds());
108
109                   view.addObject("visnList", visnList);
110
111                   // when redirected after add pharmacy.
```

PharmacyManagementController.java, line 109 (Trust Boundary Violation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The method getMainPage() in PharmacyManagementController.java commingles trusted and untrusted data in the same data structure, which encourages programmers to mistakenly trust unvalidated data.		
Source:	VisnSelectDaoImpl.java:66 org.springframework.jdbc.core.JdbcTemplate.query()		

```
64                    try {
65
66                     visnList = jdbcTemplate.query(sql,new VisnListRowMapper());
67                    } catch (DataAccessException e) {
68
```

Sink:	PharmacyManagementController.java:109 org.springframework.web.servlet.ModelAndView.addObject()

```
107                   List<VisnSelectModel>  visnList = getVisnSelect(currentUser.getVaStationIds());
108
109                   view.addObject("visnList", visnList);
110
111                   // when redirected after add pharmacy.
```

PharmacyManagementController.java, line 103 (Trust Boundary Violation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The method getMainPage() in PharmacyManagementController.java commingles trusted and untrusted data in the same data structure, which encourages programmers to mistakenly trust unvalidated data.		
Source:	UserDaoImpl.java:34 org.hibernate.Criteria.uniqueResult()		

```
32                    crit.add(Restrictions.eq("vaUserid", networkID));
33
34                          VaUser users = (VaUser)crit.uniqueResult();
35
36                          return users;
```

Sink:	PharmacyManagementController.java:103 javax.servlet.http.HttpSession.setAttribute()

```
101                   session.removeAttribute("USER_STATIONS_IDS");
102
103                   session.setAttribute("USER_STATIONS_IDS", currentUser.getVaStationIds());
104
105                   ModelAndView view = new ModelAndView("managepharmacy.homepage");
```

PharmacyManagementController.java, line 342 (Trust Boundary Violation)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The method updatePharmacy() in PharmacyManagementController.java commingles trusted and untrusted data in the same data structure, which encourages programmers to mistakenly trust unvalidated data.		

Source: PharmacyManagementController.java:197 updatePharmacy(1)

```
195
196                 @RequestMapping(value = "/updatePharmacy",  method = RequestMethod.POST)
197                 public ModelAndView updatePharmacy(HttpServletRequest request,@Valid
                    @ModelAttribute("pharmacyEditForm") PharmacyForm pharmacyForm, BindingResult
                    bindingResult) throws  IOException {
198
199                   LOG.info("Trying to persist pharmacy information.");
```

Sink: PharmacyManagementController.java:342 org.springframework.web.servlet.ModelAndView.addObject()

```
340
341
342                   view.addObject("pharmacyInfo",pharmacyInfo );
343
344                   Map<String, String> statesMap = getStatesHashMap();
```

PharmacyManagementController.java, line 163 (Trust Boundary Violation)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The method getPharmacyList() in PharmacyManagementController.java commingles trusted and untrusted data in the same data structure, which encourages programmers to mistakenly trust unvalidated data.		

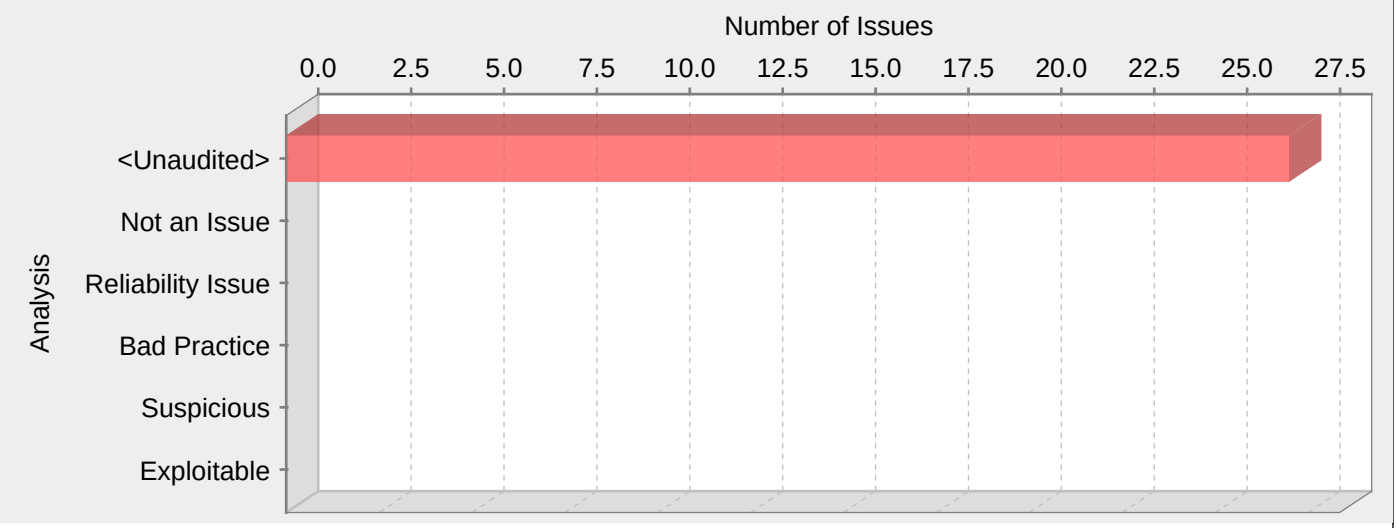
Source: PharmacyDaoImpl.java:331 org.hibernate.Criteria.list()

```
329                 criteria.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
330
331                 List<PharmacyEntity> pharmacies =  criteria.list();
332
333                 return pharmacies;
```

Sink: PharmacyManagementController.java:163 org.springframework.web.servlet.ModelAndView.addObject()

```
161                   }
162
163                   mav.addObject("items", pharmacyList);
164
165                   return mav;
```

Category: System Information Leak: HTML Comment in JSP (27 Issues)



Abstract:

Any information revealed in the HTML comment at help.jsp line 44 could help an adversary learn about the system and form a plan of attack.

Explanation:

HTML comments provide an attacker with an easy source of information about a dynamically generated web page.

Example 1:

<!-- TBD: this needs a security audit -->
<form method="POST" action="recalcOrbit">
...

Even comments that seem innocuous may be useful to someone trying to understand the way the system is built.

Recommendations:

Replace HTML comments with JSP comments (which will not be transmitted to the user).

Example 2: The previous example is rewritten to use JSP comments, which will not be displayed to the user.

<%-- TBD: this needs a security audit --%>
<form method="POST" action="recalcOrbit">
...

help.jsp, line 44 (System Information Leak: HTML Comment in JSP)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Any information revealed in the HTML comment at help.jsp line 44 could help an adversary learn about the system and form a plan of attack.		
Sink:	help.jsp:44 Comment()		

```
42  }
43
44  <!-- /* Font Definitions */
45  @font-face {
46   font-family: Wingdings;
```

help.jsp, line 1086 (System Information Leak: HTML Comment in JSP)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Any information revealed in the HTML comment at help.jsp line 1086 could help an adversary learn about the system and form a plan of attack.		
Sink:	help.jsp:1086 Comment()		

```
1084      </p>
1085      <div >
```

```
1086                    <!-- ordered list -->
1087
1088                    <ol>
```

help.jsp, line 1176 (System Information Leak: HTML Comment in JSP)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Any information revealed in the HTML comment at help.jsp line 1176 could help an adversary learn about the system and form a plan of attack.		
Sink:	help.jsp:1176 Comment()		

```
1174
1175                    </div>
1176                    <!-- ordered list End -->
1177
1178                    <h2>
```

help.jsp, line 1051 (System Information Leak: HTML Comment in JSP)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Any information revealed in the HTML comment at help.jsp line 1051 could help an adversary learn about the system and form a plan of attack.		
Sink:	help.jsp:1051 Comment()		

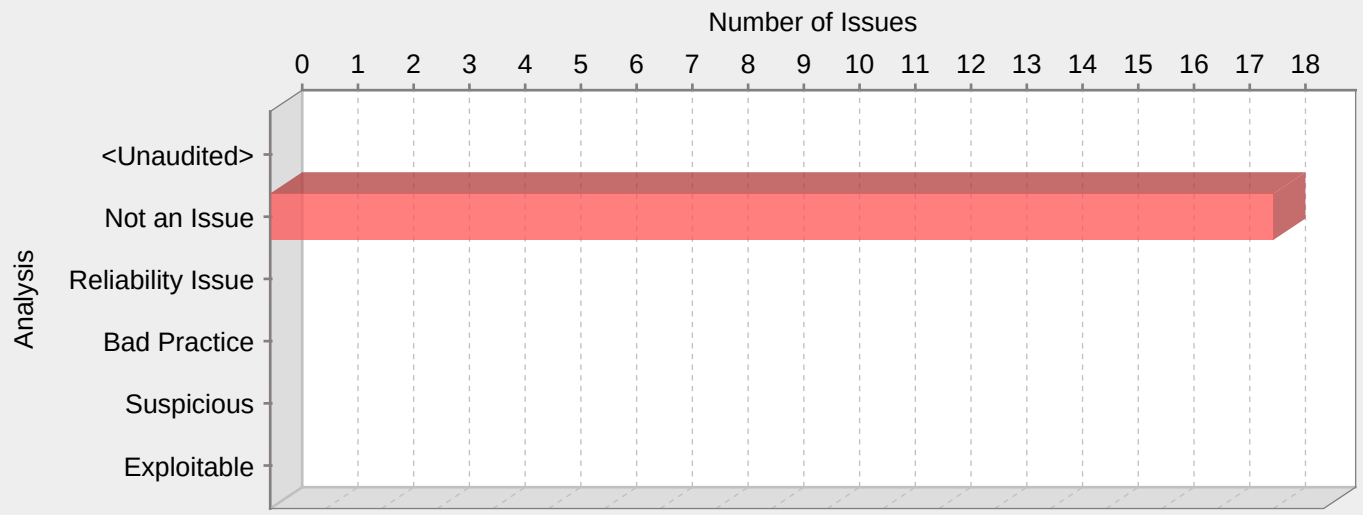
```
1049
1050                  <div role="main">
1051                   <!-- role main -->
1052
1053                   <span
```

help.jsp, line 1189 (System Information Leak: HTML Comment in JSP)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Any information revealed in the HTML comment at help.jsp line 1189 could help an adversary learn about the system and form a plan of attack.		
Sink:	help.jsp:1189 Comment()		

```
1187
1188                    <div >
1189                     <!-- ordered list -->
1190                     <ol>
1191                      <li ><p class=BodyTextNumbered1
```

Category: Password Management: Password in Comment (18 Issues)



Abstract:

Storing passwords or password details in plain text anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to hardcode a password. Storing password details within comments is equivalent to hardcoding passwords. Not only does it 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 is now leaked to the outside world and cannot be protected or 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: The following comment specifies the default password to connect to a database:

...
// Default username for database connection is "scott"
// Default password for database connection is "tiger"
...

This code will run successfully, but anyone who has access to it will have access to the password. After the program has shipped, 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.

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.

ObjectFactory.java, line 524 (Password Management: Password in Comment)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plain text anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		
Sink:	ObjectFactory.java:524 Comment()		

```
522      }
523
524      /**
525       * Create an instance of {@link PasswordRequestType }
526       *
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2017-09-22 2:56 PM*	Password is not stored in comments. Comment is only documenting a field/variable called "Password".	

BodyType.java, line 17 (Password Management: Password in Comment)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		

Abstract:	Storing passwords or password details in plain text anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.
Sink:	BodyType.java:17 Comment()

```
15
16
17  /**
18   * <p>Java class for BodyType complex type.
19   *
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-09-22 2:56 PM*	Password is not stored in comments. Comment is only documenting a field/variable called "Password".

ObjectFactory.java, line 516 (Password Management: Password in Comment)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plain text anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		
Sink:	ObjectFactory.java:516 Comment()		

```
514     }
515
516     /**
517      * Create an instance of {@link PasswordChange }
518      *
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-09-22 2:56 PM*	Password is not stored in comments. Comment is only documenting a field/variable called "Password".

BodyType.java, line 361 (Password Management: Password in Comment)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plain text anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		
Sink:	BodyType.java:361 Comment()		

```
359     }
360
361     /**
362      * Sets the value of the passwordChange property.
363      *
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-09-22 2:56 PM*	Password is not stored in comments. Comment is only documenting a field/variable called "Password".

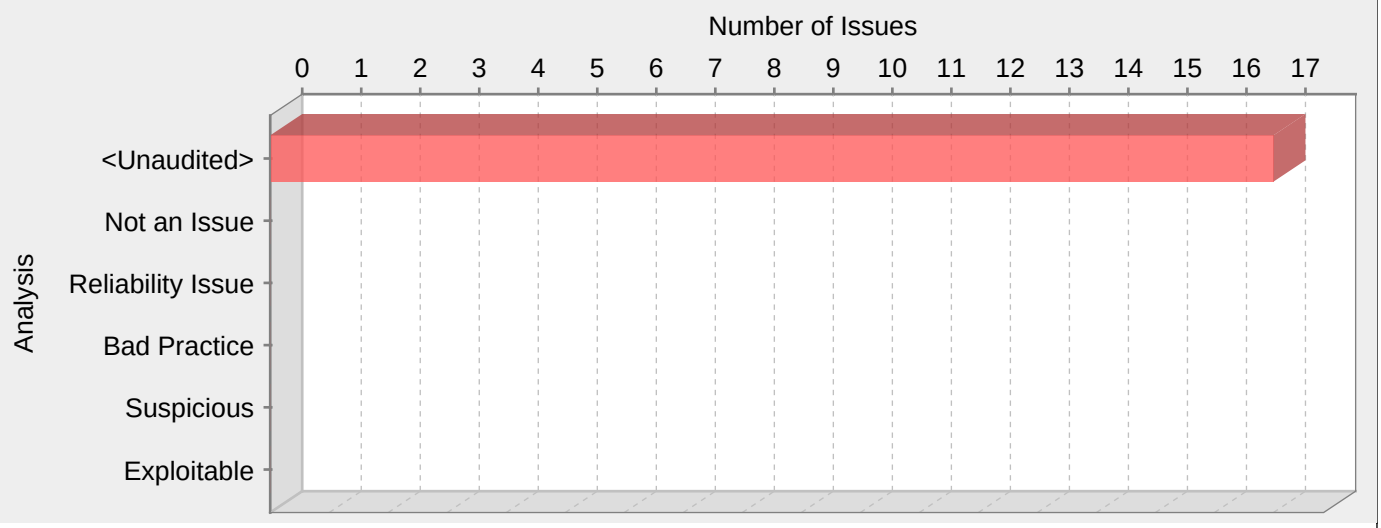
BodyType.java, line 349 (Password Management: Password in Comment)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Storing passwords or password details in plain text anywhere in the system or system code may compromise system security in a way that cannot be easily remedied.		
Sink:	BodyType.java:349 Comment()		

```
347     }
348
349     /**
350      * Gets the value of the passwordChange property.
351      *
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-09-22 2:56 PM*	Password is not stored in comments. Comment is only documenting a field/variable called "Password".

Category: Poor Error Handling: Overly Broad Throws (17 Issues)



Abstract:

The method saveInboundERx() in InboundNcpdpMsgService.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.

Explanation:

Declaring a method to throw Exception or Throwable makes it difficult for callers to do good error handling and error recovery. Java's exception mechanism is set up to make it easy for callers to anticipate what can go wrong and write code to handle each specific exceptional circumstance. Declaring that a method throws a generic form of exception defeats this system.

Example: The following method throws three types of exceptions.

```
public void doExchange()
throws IOException, InvocationTargetException,
SQLException {
...
}
```

While it might seem tidier to write

```
public void doExchange()
throws Exception {
...
}
```

doing so hampers the caller's ability to understand and handle the exceptions that occur. Further, if a later revision of doExchange() introduces a new type of exception that should be treated differently than previous exceptions, there is no easy way to enforce this requirement.

Recommendations:

Do not declare methods to throw Exception or Throwable. If the exceptions thrown by a method are not recoverable or should not generally be caught by the caller, consider throwing unchecked exceptions rather than checked exceptions. This can be accomplished by implementing exception classes that extend RuntimeException or Error instead of Exception, or add a try/catch wrapper in your method to convert checked exceptions to unchecked exceptions.

InboundNcpdpMsgService.java, line 7 (Poor Error Handling: Overly Broad Throws)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method saveInboundERx() in InboundNcpdpMsgService.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		

Sink: InboundNcpdpMsgService.java:7 Function: saveInboundERx()

```
5  public interface InboundNcpdpMsgService {
6
7   void saveInboundERx(InboundNcpdpMsgEntity inboundeRx) throws Exception ;
8
9  }
```

OutboundNcpdpMsgService.java, line 7 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method saveOutboundMsg() in OutboundNcpdpMsgService.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	OutboundNcpdpMsgService.java:7 Function: saveOutboundMsg()		

```
5  public interface OutboundNcpdpMsgService {
6
7   public long saveOutboundMsg(OutboundNcpdpMsgEntity outboundMsg) throws Exception ;
8
9   public OutboundNcpdpMsgEntity findById(long id);
```

PharmacyService.java, line 18 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method findById() in PharmacyService.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	PharmacyService.java:18 Function: findById()		

```
16  public PharmacyEntity findByNCPDPId(String NCPDPId);
17
18  public PharmacyEntity findById(Long pharmacyId) throws Exception;
19
20  public void updatePharmacyInfo(PharmacyEntity pharmacy) throws HibernateException;
```

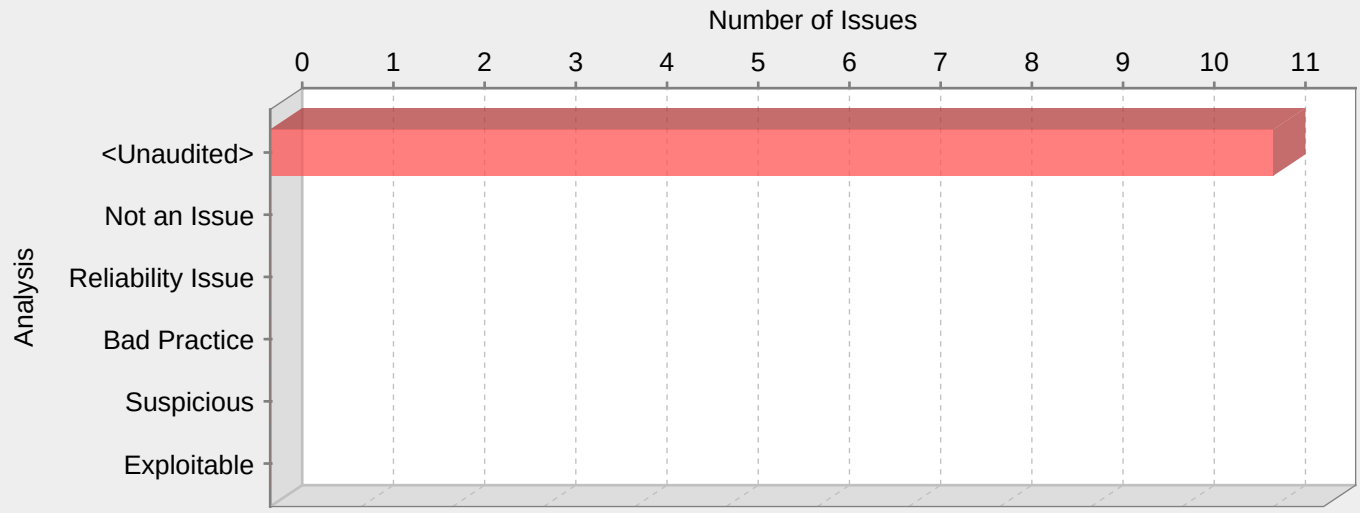
UserService.java, line 13 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method addNewVAUser() in UserService.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	UserService.java:13 Function: addNewVAUser()		

```
11  public VaUser findByVAUserID(String networkID);
12
13  public void addNewVAUser(VaUser user) throws Exception;
14
15  public void updateVAUser(VaUser user) throws Exception;
```

PharmacyService.java, line 14 (Poor Error Handling: Overly Broad Throws)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Errors		
Abstract:	The method find() in PharmacyService.java throws a generic exception making it harder for callers to do a good job of error handling and recovery.		
Sink:	PharmacyService.java:14 Function: find()		

```
12  public interface PharmacyService {
13
14  public List<PharmacyEntity> find(ManagePharmacyFilter managePharmacyFilter) throws
    Exception;
15
16  public PharmacyEntity findByNCPDPId(String NCPDPId);
```

Category: Hidden Field (11 Issues)



Abstract:

A hidden form field is used in landing.jsp at line 50.

Explanation:

Programmers often trust the contents of hidden fields, expecting that users will not be able to view them or manipulate their contents. Attackers will violate these assumptions. They will examine the values written to hidden fields and alter them or replace the contents with attack data.

Example: An <input> tag of type hidden indicates the use of a hidden field.

<input type="hidden">

If hidden fields carry sensitive information, this information will be cached the same way the rest of the page is cached. This can lead to sensitive information being tucked away in the browser cache without the user's knowledge.

Recommendations:

Expect that attackers will study and decode all uses of hidden fields in the application. Treat hidden fields as untrusted input. Don't store information in hidden fields if the information should not be cached along with the rest of the page.

manageusers.jsp, line 32 (Hidden Field)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	A hidden form field is used in manageusers.jsp at line 32.		
Sink:	manageusers.jsp:32 null()		

```
30
31                <form:hidden   path="modifiedIds" value="" />
32                <form:hidden   path="modifiedFields" value="" />
33                <form:hidden   path="enableDisableRecords" value="" />
34                <form:hidden   path="stationIdsSelected" value="" />
```

manageusers.jsp, line 34 (Hidden Field)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	A hidden form field is used in manageusers.jsp at line 34.		
Sink:	manageusers.jsp:34 null()		

```
32                <form:hidden   path="modifiedFields" value="" />
33                <form:hidden   path="enableDisableRecords" value="" />
34                <form:hidden   path="stationIdsSelected" value="" />
35                <form:hidden   path="modifiedStationIds" value="" />
36
```

manageusers.jsp, line 31 (Hidden Field)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		

Abstract:	A hidden form field is used in manageusers.jsp at line 31.
Sink:	manageusers.jsp:31 null()

```
29                  </c:if>
30
31                    <form:hidden   path="modifiedIds" value="" />
32                    <form:hidden   path="modifiedFields" value="" />
33                    <form:hidden   path="enableDisableRecords" value="" />
```

manageusers.jsp, line 33 (Hidden Field)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	A hidden form field is used in manageusers.jsp at line 33.		
Sink:	manageusers.jsp:33 null()		

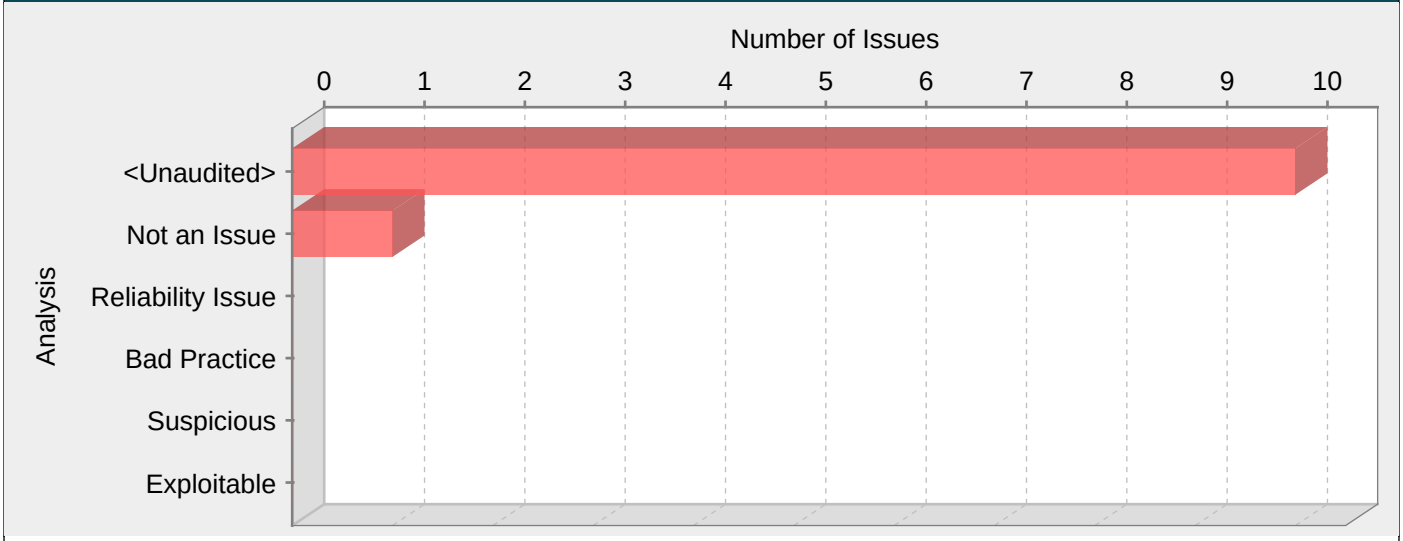
```
31                    <form:hidden   path="modifiedIds" value="" />
32                    <form:hidden   path="modifiedFields" value="" />
33                    <form:hidden   path="enableDisableRecords" value="" />
34                    <form:hidden   path="stationIdsSelected" value="" />
35                    <form:hidden   path="modifiedStationIds" value="" />
```

landing.jsp, line 50 (Hidden Field)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	A hidden form field is used in landing.jsp at line 50.		
Sink:	landing.jsp:50 null()		

```
48                      </div>
49
50                   <input  type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
51                    </form>
52                     </div>
```

Category: Missing Check against Null (11 Issues)



Abstract:

The method sendRequest() in VistaLinkConnectionUtility.java can dereference a null pointer on line 73 because it does not check the return value of getClassLoader(), which might return null.

Explanation:

Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break.

Two dubious assumptions that are easy to spot in code are "this function call can never fail" and "it doesn't matter if this function call fails". When a programmer ignores the return value from a function, they implicitly state that they are operating under one of these assumptions.

Example 1: The following code does not check to see if the string returned by getParameter() is null before calling the member function compareTo(), potentially causing a null dereference.

```
String itemName = request.getParameter(ITEM_NAME);
if (itemName.compareTo(IMPORTANT_ITEM)) {
...
}
...
```

Example 2:. The following code shows a system property that is set to null and later dereferenced by a programmer who mistakenly assumes it will always be defined.

```
System.clearProperty("os.name");
...
String os = System.getProperty("os.name");
if (os.equalsIgnoreCase("Windows 95") )
System.out.println("Not supported");
```

The traditional defense of this coding error is:

"I know the requested value will always exist because.... If it does not exist, the program cannot perform the desired behavior so it doesn't matter whether I handle the error or simply allow the program to die dereferencing a null value."

But attackers are skilled at finding unexpected paths through programs, particularly when exceptions are involved.

Recommendations:

If a function can return an error code or any other evidence of its success or failure, always check for the error condition, even if there is no obvious way for it to occur. In addition to preventing security errors, many initially mysterious bugs have eventually led back to a failed method call with an unchecked return value.

Create an easy to use and standard way for dealing with failure in your application. If error handling is straightforward, programmers will be less inclined to omit it. One approach to standardized error handling is to write wrappers around commonly-used functions that check and handle error conditions without additional programmer intervention. When wrappers are implemented and adopted, the use of non-wrapped equivalents can be prohibited and enforced by using custom rules.

Example 3: The following code implements a wrapper around getParameter() that checks the return value of getParameter() against null and uses a default value if the requested parameter is not defined.

String safeGetParameter (HttpRequest request, String name)
{
String value = request.getParameter(name);
if (value == null) {
return getDefaultValue(name)
}
return value;
}

Tips:

1. Watch out for programmers who want to explain away this type of issue by saying "that can never happen because ...". Chances are good that they have developed their intuition about the way the system works by using their development workstation. If your software will eventually run under different operating systems, operating system versions, hardware configurations, or runtime environments, their intuition may not apply.

VistaLinkConnectionUtility.java, line 73 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The method sendRequest() in VistaLinkConnectionUtility.java can dereference a null pointer on line 73 because it does not check the return value of getClassLoader(), which might return null.		
Sink:	VistaLinkConnectionUtility.java:73 getClassLoader() : Class.getClassLoader may return NULL()		

```
71                  try{
72
73                      inputStream =
this.getClass().getClassLoader().getResourceAsStream(VISTALINK_PROPERTIES_FILE);
74
75                      if(null!=inputStream){
```

XSSRequestWrapper.java, line 436 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The method getParameter() in XSSRequestWrapper.java can dereference a null pointer on line 436 because it does not check the return value of getParameterMap(), which might return null.		
Sink:	XSSRequestWrapper.java:436 getParameterMap() : XSSRequestWrapper.getParameterMap may return NULL()		

```
434                 String parameter = null;
435
436                 String[] vals = getParameterMap().get(name);
437
438                 if (vals != null && vals.length > 0) {
```

XSSRequestWrapper.java, line 414 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The method getParameterValues() in XSSRequestWrapper.java can dereference a null pointer on line 414 because it does not check the return value of getParameterMap(), which might return null.		
Sink:	XSSRequestWrapper.java:414 getParameterMap() : XSSRequestWrapper.getParameterMap may return NULL()		

```
412             @Override
413             public String[] getParameterValues(String name) {
414                 return getParameterMap().get(name);
415             }
416
```

EESummary_Client.java, line 67 (Missing Check against Null)

Fortify Priority:	Low	Folder	Low
Kingdom:	API Abuse		
Abstract:	The method returnResponse() in EESummary_Client.java can dereference a null pointer on line 67 because it does not check the return value of getClassLoader(), which might return null.		
Sink:	EESummary_Client.java:67 getClassLoader() : Class.getClassLoader may return NULL()		

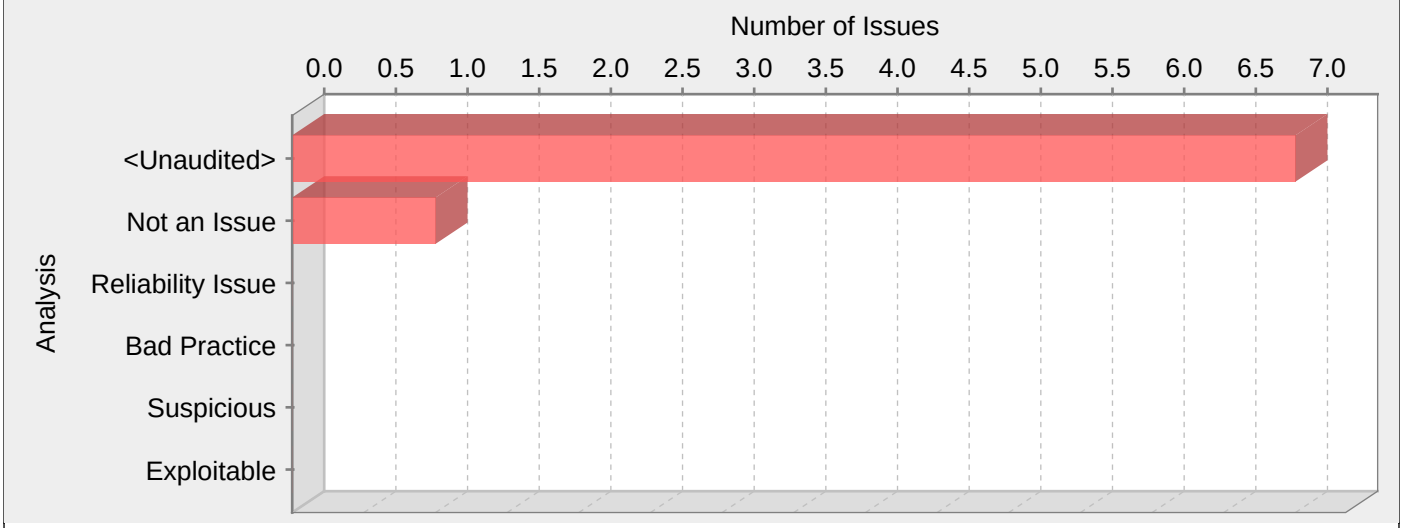
```
65    InputStream inputStream = null;
66    try{
67     inputStream = this.getClass().getClassLoader().getResourceAsStream(WSCLIENTS_PROPERTIES_FILE);
68     if(null != inputStream){
69      properties.load(inputStream);
```

ClientPasswordCallback.java, line 31 (Missing Check against Null)

Fortify Priority:	Low	Folder	Low
Kingdom:	API Abuse		
Abstract:	The method handle() in ClientPasswordCallback.java can dereference a null pointer on line 31 because it does not check the return value of getClassLoader(), which might return null.		
Sink:	ClientPasswordCallback.java:31 getClassLoader() : Class.getClassLoader may return NULL()		

```
29    String password =******
30
31    inputStream = this.getClass().getClassLoader().getResourceAsStream(WSCLIENTS_PROPERTIES_FILE);
32
33    try{
```

Category: System Information Leak: External (8 Issues)



Abstract:

The function _jspService() in footer.jsp might reveal system data or debugging information by calling print() on line 15. The information revealed by print() could help an adversary form a plan of attack.

Explanation:

An external information leak occurs when system data or debugging 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 tells the attacker precisely what sort of an attack the system is vulnerable to. 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 the example above, 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.

In the mobile world, information leaks are also a concern. 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 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 code below 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);
}
...
```

Here is another scenario specific to the mobile world. 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 to close proximity or simply having them 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, since 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 the 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);
...
```

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. Be careful, debugging 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. Due to this, it's advised to always keep information instead of sending it to a resource directly outside the program.

Example 4: The code below broadcasts the stack trace of a caught exception within your app only, so that it cannot be leaked to other apps on the system. There is also the added bonus that this 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. Either do not include system data in the messages pushed to other devices in range, or encrypt the payload of the message, or establish 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 AuditGuide to filter out this category.

PatientSearchImpl.java, line 101 (System Information Leak: External)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.		

Source: PatientSearchImpl.java:98 java.lang.Throwable.getMessage()

```
96                searchResponse.setSuccess(false);
97
98                searchResponse.setErrorMessage("Error while searching patient: " + e.getMessage());
99              }
```

Sink: PatientSearchImpl.java:101 Return searchResponse()

```
99              }
100
101             return searchResponse;
102           }
```

ProviderSearchImpl.java, line 62 (System Information Leak: External)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.		

Source: ProviderSearchImpl.java:59 java.lang.Throwable.getMessage()

```
57                searchResponse.setSuccess(false);
58
59                searchResponse.setErrorMessage("Error while searching provider: " + e.getMessage());
60              }
```

Sink: ProviderSearchImpl.java:62 Return searchResponse()

```
60              }
61
62              return searchResponse;
63            }
```

OutboundNCPDPMessageServiceImpl.java, line 207 (System Information Leak: External)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.		

Source: OutboundNCPDPMessageServiceImpl.java:204 java.lang.Throwable.getMessage()

```
202               wsResponse.setSuccess(false);
203
204               wsResponse.setErrorMessage("Error while processing outbound message: " +
e.getMessage());
205             }
```

Sink: OutboundNCPDPMessageServiceImpl.java:207 Return wsResponse()

```
205             }
206
207             return wsResponse;
208           }
```

DrugSearchImpl.java, line 62 (System Information Leak: External)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.		

Source:	DrugSearchImpl.java:57 java.lang.Throwable.getMessage()

```
55                searchResponse.setSuccess(false);
56
57                searchResponse.setErrorMessage("Error while searching drug: " + e.getMessage());
58
59                LOG.error("Error in DrugSearchImpl:"+e.getMessage());
```

Sink:	DrugSearchImpl.java:62 Return searchResponse()

```
60              }
61
62              return searchResponse;
63             }
```

footer.jsp, line 15 (System Information Leak: External)

Fortify Priority:	High	Folder	High
Kingdom:	Encapsulation		
Abstract:	The function _jspService() in footer.jsp might reveal system data or debugging information by calling print() on line 15. The information revealed by print() could help an adversary form a plan of attack.		
Source:	LoginController.java:72 org.springframework.core.env.PropertyResolver.getProperty()		

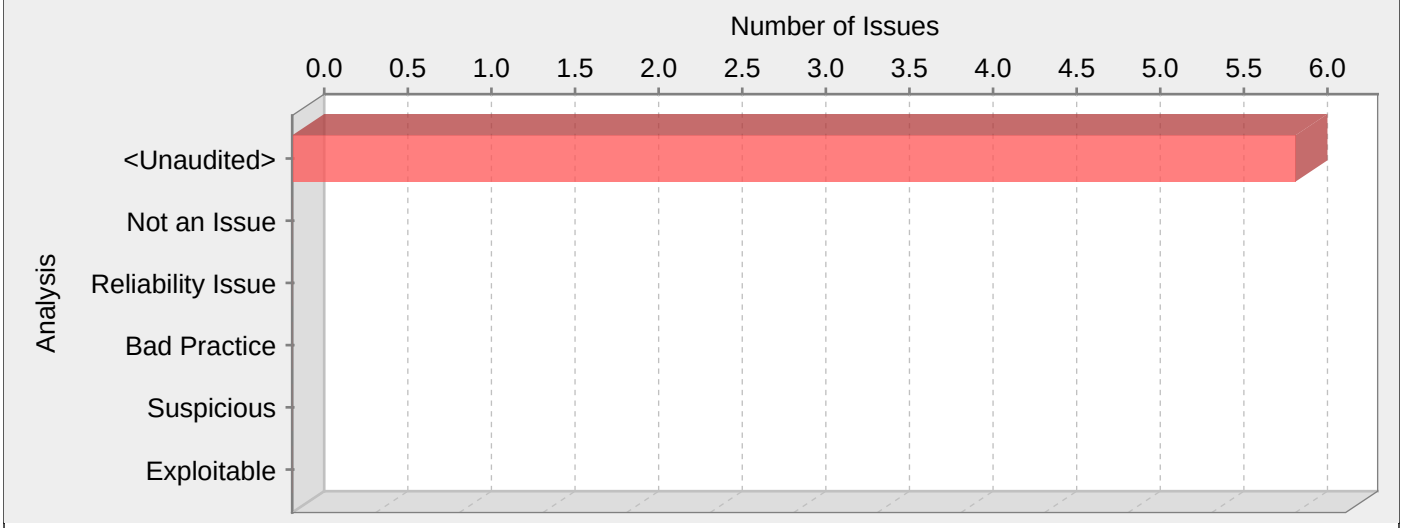
```
70                if(environment!=null){
71
72                  buildInfo = environment.getProperty("buildinformation");
73
74                  session.removeAttribute(Constants.BUILI_NFO);
```

Sink:	footer.jsp:15 javax.servlet.jsp.JspWriter.print()

```
13              <c:if test="${ buildInfo != null }">
14                <p>
15                 <label style="font-weight: bold;">Build:</label> <b>${buildInfo}</b>
16                </p>
17              </c:if>
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-10-17 9:28 AM*	The statement on line 15 only displays the current version and build number of the application and does not display other system, debugging or error information.

Category: Build Misconfiguration: External Maven Dependency Repository (6 Issues)



Abstract:

This maven build script relies on external sources, which could allow an attacker to insert malicious code into the final product or to take control of the build machine.

Explanation:

Several tools exist within the Java development world to aid in dependency management: both Apache Ant and Apache Maven build systems include functionality specifically designed to help manage dependencies and Apache Ivy is developed explicitly as a dependency manager. Although there are differences in their behavior, these tools share the common functionality that they automatically download external dependencies specified in the build process at build time. This makes it much easier for developer B to build software in the same manner as developer A. Developers just store dependency information in the build file, which means that each developer and build engineer has a consistent way to obtain dependencies, compile the code, and deploy without the dependency management hassles involved in manual dependency management. The following examples illustrate how Ivy, Ant and Maven can be used to manage external dependencies as part of a build process.

Under Maven, instead of listing explicit URLs from which to retrieve the dependencies, developers specify the dependency names and versions and Maven relies on its underlying configuration to identify the server(s) from which to retrieve the dependencies. For commonly used components this saves the developer from having to researching dependency locations.

Example 1: The following except from a Maven pom.xml file shows how a developer can specify multiple external dependencies using their name and version:

```
<dependencies>
<dependency>
<groupId>commons-logging</groupId>
<artifactId>commons-logging</artifactId>
<version>1.1</version>
</dependency>
<dependency>
<groupId>javax.jms</groupId>
<artifactId>jms</artifactId>
<version>1.1</version>
</dependency>
...
</dependencies>
```

Two distinct types of attack scenarios affect these systems: An attacker could either compromise the server hosting the dependency or compromise the DNS server the build machine uses to redirect requests for hostname of the server hosting the dependency to a machine controlled by the attacker. Both scenarios result in the attacker gaining the ability to inject a malicious version of a dependency into a build running on an otherwise uncompromised machine.

Regardless of the attack vector used to deliver the Trojan dependency, these scenarios share the common element that the build system blindly accepts the malicious binary and includes it in the build. Because the build system has no recourse for rejecting the malicious binary and existing security mechanisms, such as code review, typically focus on internally-developed code rather than external dependencies, this type of attack has a strong potential to go unnoticed as it spreads through the development environment and potentially into production.

Although there is some risk of a compromised dependency being introduced into a manual build process, by the tendency of automated build systems to retrieve the dependency from an external source each time the build system is run in a new environment greatly increases the window of opportunity for an attacker. An attacker need only compromise the dependency server or the DNS server during one of the many times the dependency is retrieved in order to compromise the machine on which the build is occurring.

Recommendations:

The simplest solution is to refrain from adopting automated dependency management systems altogether. Managing dependencies manually eliminates the potential for unexpected behavior caused by the build system. Obviously, the an attacker could still mount one of the attacks described above to coincide with the manual retrieval of a dependency, but limiting the frequency with which the dependency must be retrieved significantly reduces the window of opportunity for an attacker. Finally, this solution forces the development organization to rely on what is ostensibly an antiquated build system. A system based on manual dependency management is often more difficult to use and maintain, and may be unacceptable in some software development environments.

The second solution is a hybrid of the traditional manual dependency management approach and the fully automated solution that is popular today. The biggest advantage of the manual build process is the decreased window of attack, which can be achieved in a semi-automated system by replicating external dependency servers internally. Any build system that requires an external dependency can then point to the internal server using a hard-coded internal IP address to bypass the risk of DNS-based attacks. As new dependencies are added and new versions released, they can be downloaded once and included on the internal repository. This solution reduces the attack opportunities and allows the organization leverage existing internal network security infrastructure.

To implement this solution using Maven, a project should have the IP address for an internal repository hard coded the pom.xml. Specifying the IP address in the pom.xml ensures the internal repository will be used by the corresponding build, but is tied to a specific project. Alternatively, the IP address can be specified in settings.xml, which makes the configuration easier to share across multiple projects.

Example 2: The following Maven pom.xml demonstrates the use of an explicit internal IP address (the entries can also be used in settings.xml):

```
<project>
...
<repositories>
<repository>
<releases>
<enabled>true</enabled>
<updatePolicy>always</updatePolicy>
<checksumPolicy>warn</checksumPolicy>
</releases>
<snapshots>
<enabled>true</enabled>
<updatePolicy>never</updatePolicy>
<checksumPolicy>fail</checksumPolicy>
</snapshots>
<id>central</id>
<name>Internal Repository</name>
<url>http://172.16.1.13/maven2</url>
<layout>default</layout>
</repository>
</repositories>
<pluginRepositories>
...
</pluginRepositories>
...
</project>
```

pom.xml, line 17 (Build Misconfiguration: External Maven Dependency Repository)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Environment		
Abstract:	This maven build script relies on external sources, which could allow an attacker to insert malicious code into the final product or to take control of the build machine.		
Sink:	pom.xml:17 //project/repositories()		

15

```
16
17                  <repositories>
18                   <repository>
19                    <id>project.local</id>
```

pom.xml, line 16 (Build Misconfiguration: External Maven Dependency Repository)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Environment		
Abstract:	This maven build script relies on external sources, which could allow an attacker to insert malicious code into the final product or to take control of the build machine.		

Sink: pom.xml:16 //project/repositories()

```
14                   </parent>
15
16                      <repositories>
17                    <repository>
18                     <id>project.local</id>
```

pom.xml, line 2 (Build Misconfiguration: External Maven Dependency Repository)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Environment		
Abstract:	This maven build script relies on external sources, which could allow an attacker to insert malicious code into the final product or to take control of the build machine.		

Sink: pom.xml:2 //project/repositories()

```
0                    <project xmlns="http://maven.apache.org/POM/4.0.0"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1                     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
                     v4_0_0.xsd">
2
3                     <modelVersion>4.0.0</modelVersion>
```

pom.xml, line 21 (Build Misconfiguration: External Maven Dependency Repository)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Environment		
Abstract:	This maven build script relies on external sources, which could allow an attacker to insert malicious code into the final product or to take control of the build machine.		

Sink: pom.xml:21 //project/repositories()

```
19                   </modules>
20
21                   <repositories>
22                    <repository>
23                     <id>project.local</id>
```

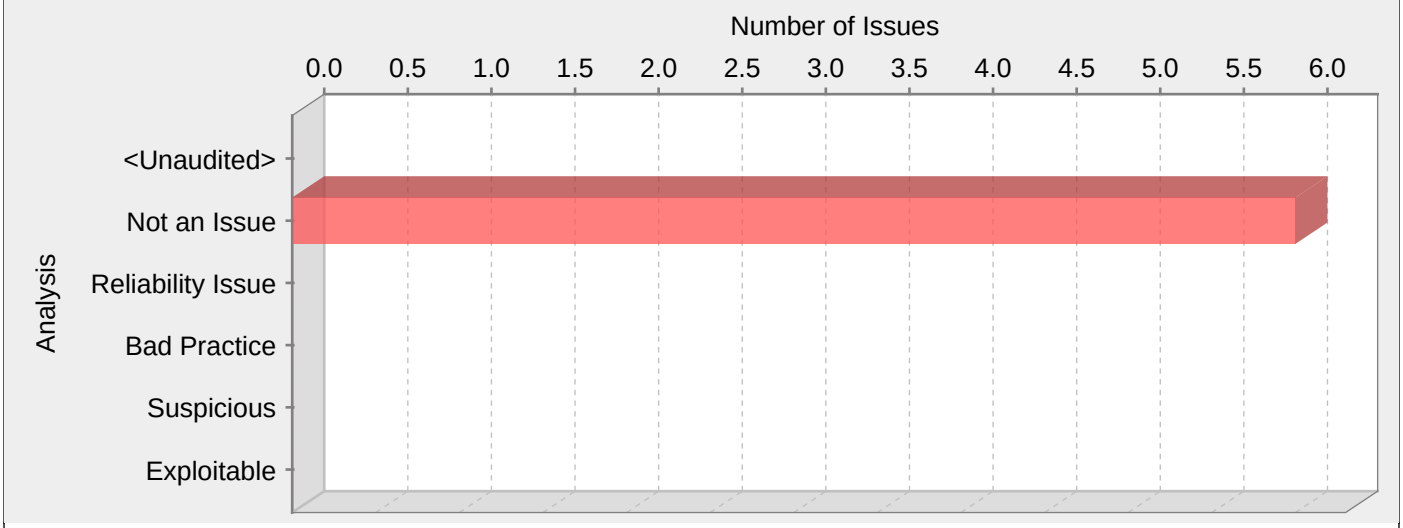
pom.xml, line 2 (Build Misconfiguration: External Maven Dependency Repository)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Environment		
Abstract:	This maven build script relies on external sources, which could allow an attacker to insert malicious code into the final product or to take control of the build machine.		

Sink: pom.xml:2 //project/repositories()

```
0                    <project xmlns="http://maven.apache.org/POM/4.0.0"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1                     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
                     v4_0_0.xsd">
2
3                     <modelVersion>4.0.0</modelVersion>
```

Category: Cross-Site Request Forgery (6 Issues)



Abstract:

The form post at addpharmacy.jsp line 11 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.

Explanation:

A cross-site request forgery (CSRF) vulnerability occurs when:

1. A Web application uses session cookies.

2. The application acts on an HTTP request without verifying that the request was made with the user's consent.

A nonce is a cryptographic random value that is sent with a message to prevent replay attacks. If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application). This means a Web application that uses session cookies has to take special precautions in order to ensure that an attacker can't trick users into submitting bogus requests. Imagine a Web application that allows administrators to create new accounts by submitting this form:

```
<form method="POST" action="/new_user" >
Name of new user: <input type="text" name="username">
Password for new user: <input type="password" name="user_passwd">
<input type="submit" name="action" value="Create User">
</form>
```

An attacker might set up a Web site with the following:

```
<form method="POST" action="http://www.example.com/new_user">
<input type="hidden" name="username" value="hacker">
<input type="hidden" name="user_passwd" value="hacked">
</form>
<script>
document.usr_form.submit();
</script>
```

If an administrator for example.com visits the malicious page while she has an active session on the site, she will unwittingly create an account for the attacker. This is a CSRF attack. It is possible because the application does not have a way to determine the provenance of the request. Any request could be a legitimate action chosen by the user or a faked action set up by an attacker. The attacker does not get to see the Web page that the bogus request generates, so the attack technique is only useful for requests that alter the state of the application.

Applications that pass the session identifier in the URL rather than as a cookie do not have CSRF problems because there is no way for the attacker to access the session identifier and include it as part of the bogus request.

CSRF is entry number five on the 2007 OWASP Top 10 list.

Recommendations:

Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, like this:

```
RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");
```

```
body = addToPost(body, new_username);
body = addToPost(body, new_passwd);
body = addToPost(body, request_id);
rb.sendRequest(body, new NewAccountCallback(callback));
```

Then the back-end logic can validate the request identifier before processing the rest of the form data. When possible, the request identifier should be unique to each server request rather than shared across every request for a particular session. As with session identifiers, the harder it is for an attacker to guess the request identifier, the harder it is to conduct a successful CSRF attack. The token should not be easily guessed and it should be protected in the same way that session tokens are protected, such as using SSLv3.

Additional mitigation techniques include:

Framework protection: Most modern web application frameworks embed CSRF protection and they will automatically include and verify CSRF tokens.

Use a Challenge-Response control: Forcing the customer to respond to a challenge sent by the server is a strong defense against CSRF. Some of the challenges that can be used for this purpose are: CAPTCHAs, password re-authentication and one-time tokens.

Check HTTP Referer/Origin headers: An attacker won't be able to spoof these headers while performing a CSRF attack. This makes these headers a useful method to prevent CSRF attacks.

Double-submit Session Cookie: Sending the session ID Cookie as a hidden form value in addition to the actual session ID Cookie is a good protection against CSRF attacks. The server will check both values and make sure they are identical before processing the rest of the form data. If an attacker submits a form in behalf of a user, he won't be able to modify the session ID cookie value as per the same-origin-policy.

Limit Session Lifetime: When accessing protected resources using a CSRF attack, the attack will only be valid as long as the session ID sent as part of the attack is still valid on the server. Limiting the Session lifetime will reduce the probability of a successful attack.

The techniques described here can be defeated with XSS attacks. Effective CSRF mitigation includes XSS mitigation techniques.

Tips:

1. SCA flags all HTML forms and XMLHttpRequest objects that might perform a POST operation. The auditor must determine if each form could be valuable to an attacker as a CSRF target and whether or not an appropriate mitigation technique is in place.

managepharmacy.jsp, line 35 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The form post at managepharmacy.jsp line 35 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	managepharmacy.jsp:35 null()		

```
33
34  <div id="pharmacyManagement">
35  <form:form id="pharmacyFilterForm"  name="pharmacyFilterForm">
36
```

Analysis:	Not an Issue		
Comments:	*VHADURButtS 2018-11-07 2:24 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.	

loginErrorLayout.jsp, line 23 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The form post at loginErrorLayout.jsp line 23 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	loginErrorLayout.jsp:23 null()		

```
21     <div id="content">
22    <br/>
23    <form action="${index}" method="post" >
24
25    <div align="center">
```

Analysis:	Not an Issue

Comments:	*VHAHAMWandlJ 2017-09-22 3:02 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.	

addpharmacy.jsp, line 11 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The form post at addpharmacy.jsp line 11 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	addpharmacy.jsp:11 null()		

```
9   <div id="AddPharm">
10  <h1 class="titleBar"><span class="headerText">Add Pharmacy</span></h1>
11  <form:form id="pharmacyAddForm" name="pharmacyAddForm" commandName="pharmacyAddForm"
    method="POST" action="${pageContext.request.contextPath}/inb-
    erx/managePharm/addNewPharmacy">
12  <div id="warningMessage" class="warning" aria-describedby="Informational message or
    warning message section" tabindex="0">
13  <p id="errorTitle">
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2017-09-22 3:02 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.	

manageusers.jsp, line 10 (Cross-Site Request Forgery)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The form post at manageusers.jsp line 10 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	manageusers.jsp:10 null()		

```
8   <h2 class="titleBar"><span> User Management</span>
9   </h2>
10  <form:form name="manageUsersForm" id ="manageUsersForm"
    modelAttribute="userManagementModel" method="POST"
    action="${pageContext.request.contextPath}/inb-erx/manageUsers/saveUserData">
11      <c:if test="${not empty data.errorMessage }">
12    <div id="errorMessages" class="error" aria-describedby="message section" tabindex="0">
```

Analysis:	Not an Issue		
Comments:	*VHADURButtS 2018-11-07 2:24 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.	

editpharmacy.jsp, line 12 (Cross-Site Request Forgery)

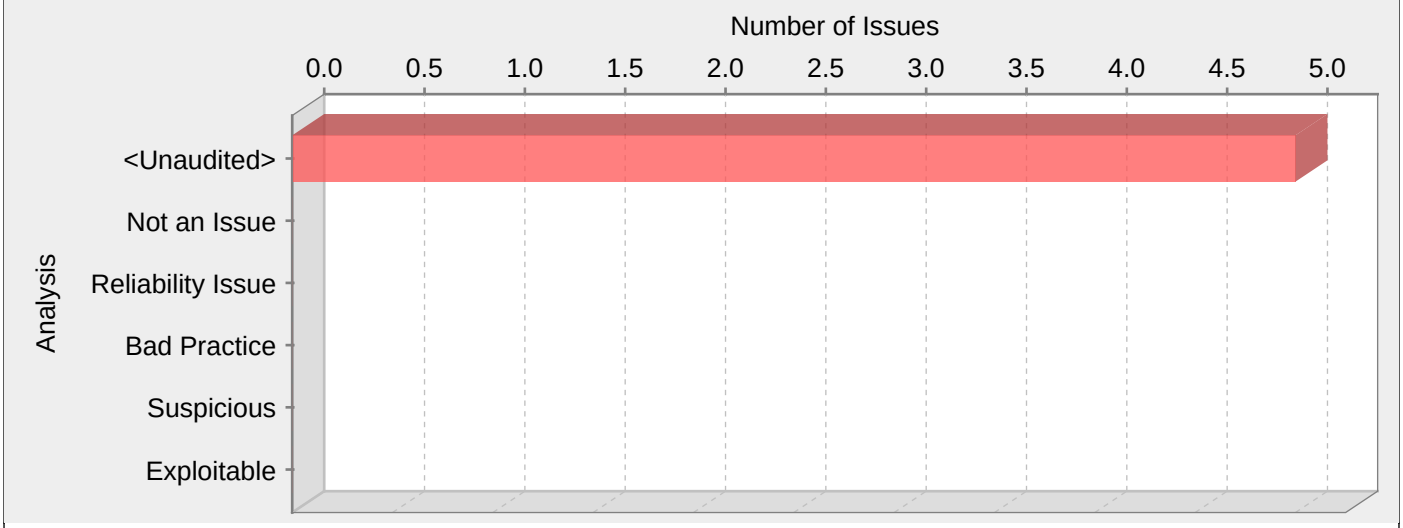
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The form post at editpharmacy.jsp line 12 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.		
Sink:	editpharmacy.jsp:12 null()		

```
10
11  <h1 class="titleBar"><span class="headerText">Edit Pharmacy</span></h1>
12  <form:form id="pharmacyEditForm" name="pharmacyEditForm"
    commandName="pharmacyEditForm" method="POST"
    action="${pageContext.request.contextPath}/inb-erx/managePharm/updatePharmacy">
13  <div id="warningMessage" class="warning" aria-describedby="message section"
    tabindex="0">
14  <p id="errorTitle">
```

Analysis:	Not an Issue

Comments:	*VHAHAMWandlJ 2017-09-22 3:02 PM*	Spring MVC Framework is in use and providing protection. Spring is automatically embedding/include CSRF tokens and verifying CSRF tokens.

Category: Password Management (5 Issues)



Abstract:

The method setupTLS() in EESummary_Client.java uses a plain text password on line 164. Storing a password in plain text can result in a system compromise.

Explanation:

Password management issues occur when a password is stored in plain text in an application's properties or configuration file.

Example 1: The following code reads a password from a properties file and uses the password to connect to a database.

```
...
Properties prop = new Properties();
prop.load(new FileInputStream("config.properties"));
String password = prop.getProperty("password");
DriverManager.getConnection(url, usr, password);
...
```

This code will run successfully, but anyone who has access to config.properties can read the value of password. Any devious employee with access to this information can use it to break into the system.

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

Example 2: The code below reads username and password from an Android WebView store and uses them to setup authentication for viewing protected pages.

```
...
webview.setWebViewClient(new WebViewClient() {
public void onReceivedHttpAuthRequest(WebView view,
HttpAuthHandler handler, String host, String realm) {
String[] credentials = view.getHttpAuthUsernamePassword(host, realm);
String username = credentials[0];
String password = credentials[1];
handler.proceed(username, password);
}
});
...
```

By default, WebView credentials are stored in plain text and are not hashed. So if a user has a rooted device (or uses an emulator), she is able to read stored passwords for given sites.

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. At the very least, passwords should be hashed before being stored.

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. 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 code below 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, WebKit has to be re-compiled with the sqlcipher.so library.

Tips:

1. The Fortify Secure Coding Rulepacks identify password management issues by looking for functions that are known to take passwords as arguments. If a password is provided from outside the program and is used without passing through an identified de-obfuscation routine, then SCA flags a password management issue.

To audit a password management issue, trace through the program starting from where the password enters the system and ending where it is used. Look for code that performs de-obfuscation. If no such code is present, then this issue has not been mitigated. If the password passes through a de-obfuscation function, verify that the algorithm used to protect the password is sufficiently robust.

After you are convinced that the password is adequately protected, write a custom passthrough rule for the de-obfuscation routine that indicates that the password is protected with obfuscation. If you include this rule in future analyses of the application, passwords that pass through the identified de-obfuscation routine will no longer trigger password management vulnerabilities.

2. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. 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.

OutboundNCPDPMessageServiceImpl.java, line 253 (Password Management)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method setupTLS() in OutboundNCPDPMessageServiceImpl.java uses a plain text password on line 253. Storing a password in plain text can result in a system compromise.		

Source: OutboundNCPDPMessageServiceImpl.java:125 java.util.Properties.load()

```
123
124                 if (null != inputStream) {
125                  properties.load(inputStream);
126                 }
127                }
```

Sink: OutboundNCPDPMessageServiceImpl.java:253 java.security.KeyStore.load()

```
251               try {
252                fileIn1 = new FileInputStream(file);
253                keyStore.load(fileIn1, keyPasswd.toCharArray());
254                KeyManager[] myKeyManagers = getKeyManagers(keyStore, keyPasswd);
255                tlsCP.setKeyManagers(myKeyManagers);
```

MVIClient.java, line 749 (Password Management)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	The method setupTLS() in MVIClient.java uses a plain text password on line 749. Storing a password in plain text can result in a system compromise.		

Source: MVIClient.java:164 java.util.Properties.load()

```
162                 .getResourceAsStream(WSCLIENTS_PROPERTIES_FILE);
163               if (null != inputStream) {
164                properties.load(inputStream);
165               }
166              } finally {
```

Sink: MVIClient.java:749 java.security.KeyStore.load()

```
747              try {
748              fileIn1 = new FileInputStream(file);
749              keyStore.load(fileIn1, keyPasswd.toCharArray());
750              KeyManager[] myKeyManagers = getKeyManagers(keyStore, keyPasswd);
751              tlsCP.setKeyManagers(myKeyManagers);
```

MVIClient.java, line 763 (Password Management)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	The method setupTLS() in MVIClient.java uses a plain text password on line 763. Storing a password in plain text can result in a system compromise.		

Source: MVIClient.java:164 java.util.Properties.load()

```
162                 .getResourceAsStream(WSCLIENTS_PROPERTIES_FILE);
163               if (null != inputStream) {
164                properties.load(inputStream);
165               }
166              } finally {
```

Sink: MVIClient.java:763 java.security.KeyStore.load()

```
761              fileIn2 = new FileInputStream(file);
762              KeyStore trustStore = KeyStore.getInstance(KEYSTORE_FILE_TYPE);
763              trustStore.load(fileIn2, keyPasswd.toCharArray());
764              TrustManager[] myTrustStoreKeyManagers = getTrustManagers(trustStore);
765              tlsCP.setTrustManagers(myTrustStoreKeyManagers);
```

OutboundNCPDPMessageServiceImpl.java, line 267 (Password Management)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		
Abstract:	The method setupTLS() in OutboundNCPDPMessageServiceImpl.java uses a plain text password on line 267. Storing a password in plain text can result in a system compromise.		

Source: OutboundNCPDPMessageServiceImpl.java:125 java.util.Properties.load()

```
123
124                 if (null != inputStream) {
125                  properties.load(inputStream);
126                 }
127                }
```

Sink: OutboundNCPDPMessageServiceImpl.java:267 java.security.KeyStore.load()

```
265               fileIn2 = new FileInputStream(file);
266               KeyStore trustStore = KeyStore.getInstance(KEYSTORE_FILE_TYPE);
267               trustStore.load(fileIn2, keyPasswd.toCharArray());
268               TrustManager[] myTrustStoreKeyManagers = getTrustManagers(trustStore);
269               tlsCP.setTrustManagers(myTrustStoreKeyManagers);
```

EESummary_Client.java, line 164 (Password Management)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Security Features		

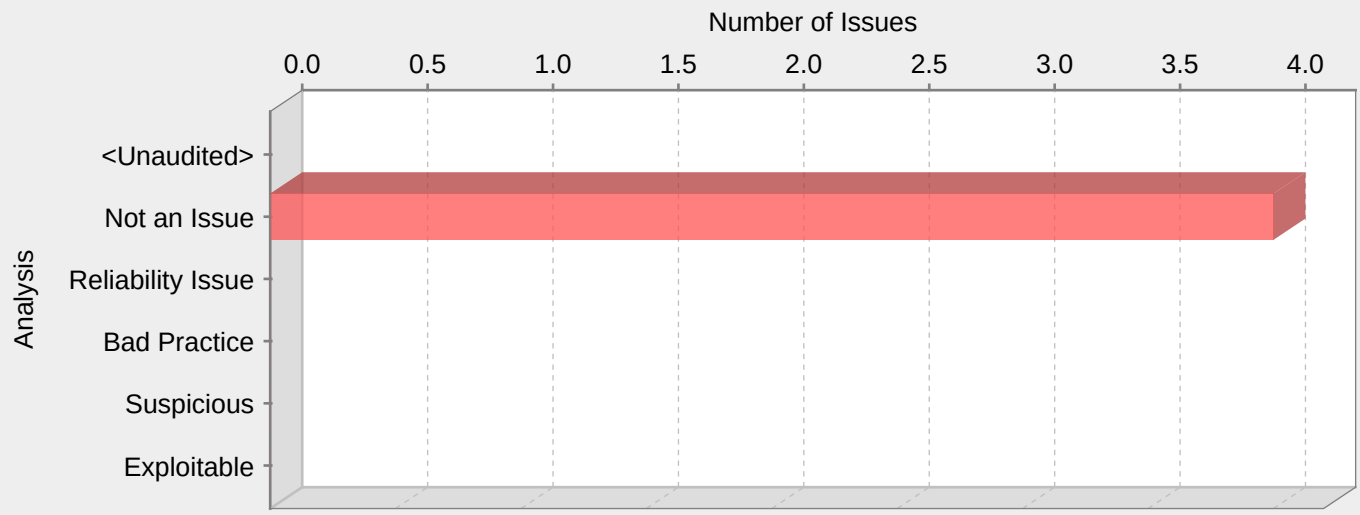
Abstract:	The method setupTLS() in EESummary_Client.java uses a plain text password on line 164. Storing a password in plain text can result in a system compromise.
Source:	EESummary_Client.java:69 java.util.Properties.load()

```
67    inputStream =
this.getClass().getClassLoader().getResourceAsStream(WSCLIENTS_PROPERTIES_FILE);
68    if(null != inputStream){
69     properties.load(inputStream);
70    } else
71    {
```

Sink:	EESummary_Client.java:164 java.security.KeyStore.load()

```
162   fileIn2 = new FileInputStream(file);
163   KeyStore trustStore = KeyStore.getInstance(KEYSTORE_FILE_TYPE);
164   trustStore.load(fileIn2, keyPasswd.toCharArray());
165   TrustManager[] myTrustStoreKeyManagers = getTrustManagers(trustStore);
166   tlsCP.setTrustManagers(myTrustStoreKeyManagers);
```

Category: Code Correctness: Misleading Method Signature (4 Issues)



Abstract:

The declaration of equals() in InboundNcpdpMsgUserType looks like an effort to override a common Java method, but it does not have the intended effect.

Explanation:

This method's name is similar to a common Java method name, but it is either spelled incorrectly or the argument list causes it to not override the intended method.

Example 1: The following method is meant to override Object.equals():

```
public boolean equals(Object obj1, Object obj2) {
...
}
```

But since Object.equals() only takes a single argument, the method above is never called.

Recommendations:

Carefully check to make sure the method does what was intended.

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

```
public boolean equals(Object obj) {
...
}
```

OutboundNcpdpMsgUserType.java, line 57 (Code Correctness: Misleading Method Signature)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The declaration of hashCode() in OutboundNcpdpMsgUserType looks like an effort to override a common Java method, but it does not have the intended effect.		
Sink:	OutboundNcpdpMsgUserType.java:57 Function: hashCode()		

```
55
56      @Override
57      public int hashCode(Object x) throws HibernateException {
58       return (x != null) ? x.hashCode() : 0;
59      }
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2017-09-26 9:42 AM*	The method does not override Object.hashCode. It overides gov.va.med.pharmacy.persistence.model.usertypes.InboundNcpdpMsgUserType.hashCode and has the desired effect.	

InboundNcpdpMsgUserType.java, line 51 (Code Correctness: Misleading Method Signature)

Fortify Priority:	Low	**Folder**	Low

Kingdom:	Code Quality		
Abstract:	The declaration of hashCode() in InboundNcpdpMsgUserType looks like an effort to override a common Java method, but it does not have the intended effect.		
Sink:	InboundNcpdpMsgUserType.java:51 Function: hashCode()		

```
49
50                  @Override
51                  public int hashCode(Object x) throws HibernateException {
52
53                   return (x != null) ? x.hashCode() : 0;
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2017-09-26 9:42 AM*	The method does not override Object.hashCode. It overides gov.va.med.pharmacy.persistence.model.usertypes.InboundNcpdpMsgUserType.hashCode and has the desired effect.	

OutboundNcpdpMsgUserType.java, line 52 (Code Correctness: Misleading Method Signature)

Fortify Priority:	Low	Folder	Low
Kingdom:	Code Quality		
Abstract:	The declaration of equals() in OutboundNcpdpMsgUserType looks like an effort to override a common Java method, but it does not have the intended effect.		
Sink:	OutboundNcpdpMsgUserType.java:52 Function: equals()		

```
50
51                  @Override
52                  public boolean equals(Object x,  Object y) throws HibernateException {
53                   return (x != null) && x.equals(y);
54                  }
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2017-09-26 9:41 AM*	The method does not override Object.equals. It overides gov.va.med.pharmacy.persistence.model.usertypes.InboundNcpdpMsgUserType.equals and has the desired effect.	

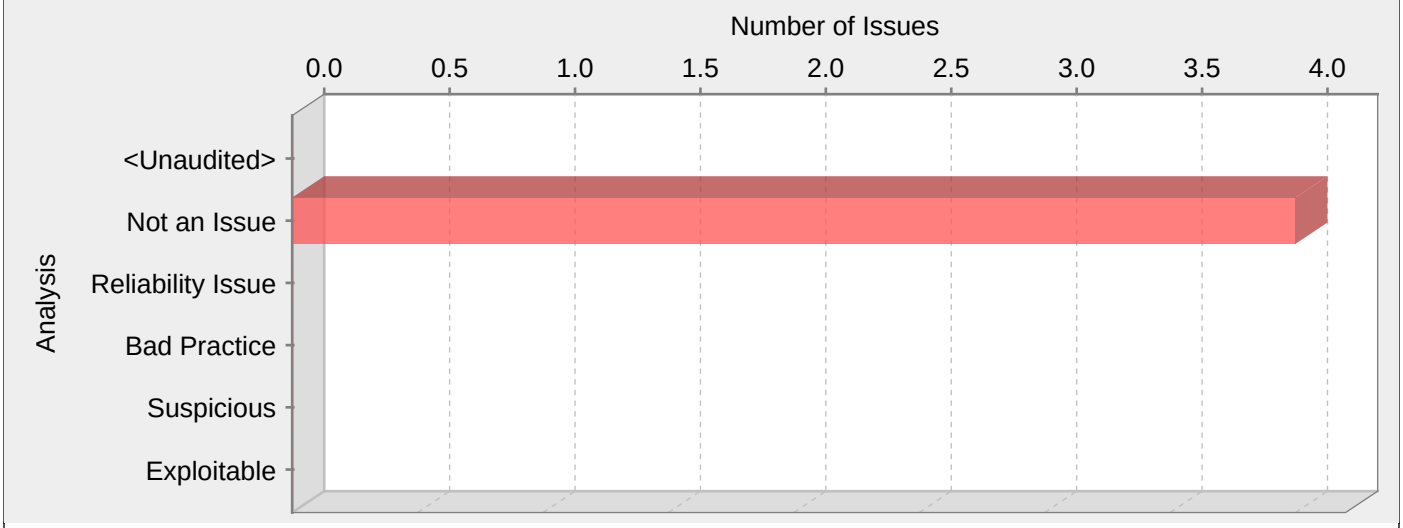
InboundNcpdpMsgUserType.java, line 44 (Code Correctness: Misleading Method Signature)

Fortify Priority:	Low	Folder	Low
Kingdom:	Code Quality		
Abstract:	The declaration of equals() in InboundNcpdpMsgUserType looks like an effort to override a common Java method, but it does not have the intended effect.		
Sink:	InboundNcpdpMsgUserType.java:44 Function: equals()		

```
42
43                  @Override
44                  public boolean equals(Object x, Object y) throws HibernateException {
45
46                   return (x != null) && x.equals(y);
```

Analysis:	Not an Issue		
Comments:	*VHAHAMWandlJ 2017-09-26 9:41 AM*	The method does not override Object.equals. It overides gov.va.med.pharmacy.persistence.model.usertypes.InboundNcpdpMsgUserType.equals and has the desired effect.	

Category: Password Management: Hardcoded Password (4 Issues)



Abstract:

Hardcoded passwords may compromise system security in a way that cannot be easily remedied.

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 has shipped, 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 the example above:

```
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 code below 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, passwords should be hashed before being stored.

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. 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 code below 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, WebKit has to be re-compiled with the sqlcipher.so library.

Tips:

1. The Fortify Java Annotations FortifyPassword and FortifyNotPassword can be used 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.

ClientPasswordCallback.java, line 18 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords may compromise system security in a way that cannot be easily remedied.		
Sink:	ClientPasswordCallback.java:18 FieldAccess: E_AND_E_PASSWD()		

```
16
17    private static final String WSCLIENTS_PROPERTIES_FILE =
"gov.va.med.pharmacy.inboundeRx.properties";
18    private static final String E_AND_E_PASSWD = "eAnde.password";
19    private static final String E_AND_E_USERNAME = "eAnde.username";
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-05-10 11:28 AM*	The references are not hardcoded passwords, but System Property names that refer to passwords

OutboundNCPDPMessageServiceImpl.java, line 77 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords may compromise system security in a way that cannot be easily remedied.		
Sink:	OutboundNCPDPMessageServiceImpl.java:77 FieldAccess: KEYSTORE_PASSWD_PROPERTY()		

```
75    private static final String KEYSTORE_FILE_TYPE = "JKS";
76    private static final String KEYSTORE_FILE_NAME_PROPERTY = "keystore.filename";
77    private static final String KEYSTORE_PASSWD_PROPERTY = "keystore.password";
78
79    @Autowired
```

Analysis:	Not an Issue

Comments:	*VHAHAMWandlJ 2017-05-10 11:28 AM*	The references are not hardcoded passwords, but System Property names that refer to passwords	

MVIClient.java, line 97 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords may compromise system security in a way that cannot be easily remedied.		
Sink:	MVIClient.java:97 FieldAccess: KEYSTORE_PASSWD_PROPERTY()		

```
95  private static final String KEYSTORE_FILE_TYPE = "JKS";
96  private static final String KEYSTORE_FILE_NAME_PROPERTY = "keystore.filename";
97  private static final String KEYSTORE_PASSWD_PROPERTY = "keystore.password";
98  private static final String LEGAL_NAME = "Legal Name";
99  private static final String L_CHARACTER = "L";
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-05-10 11:28 AM*	The references are not hardcoded passwords, but System Property names that refer to passwords

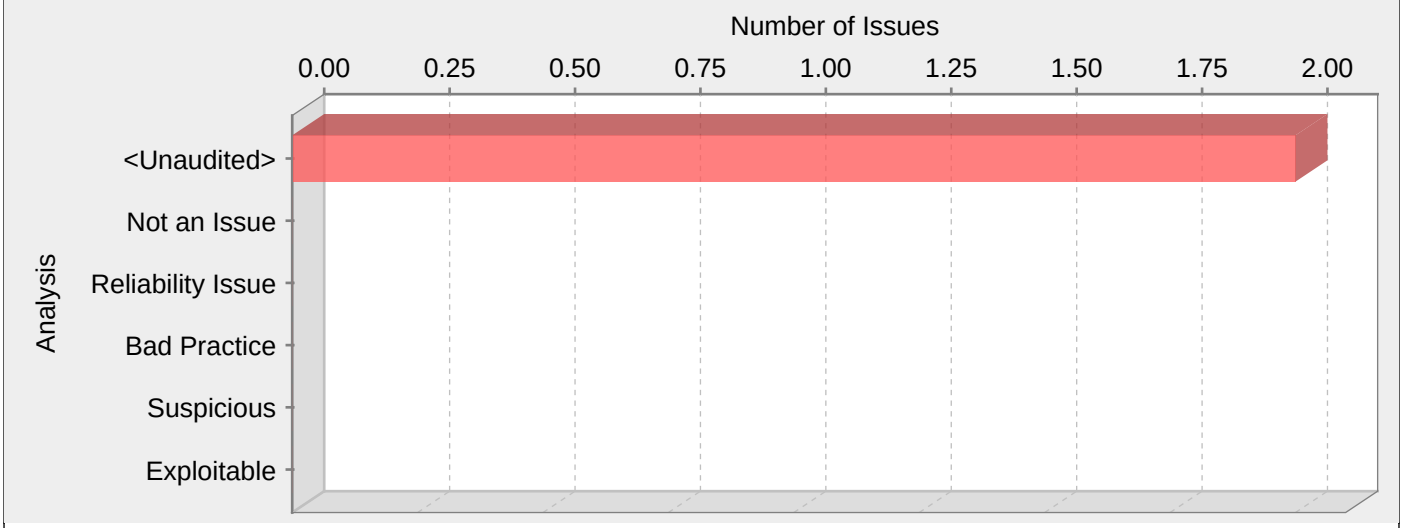
EESummary_Client.java, line 55 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords may compromise system security in a way that cannot be easily remedied.		
Sink:	EESummary_Client.java:55 FieldAccess: KEYSTORE_PASSWD_PROPERTY()		

```
53  private static final String KEYSTORE_FILE_TYPE = "JKS";
54  private static final String KEYSTORE_FILE_NAME_PROPERTY = "keystore.filename";
55  private static final String KEYSTORE_PASSWD_PROPERTY = "keystore.password";
56
57      public getEESummaryResponse returnResponse(String key) throws java.lang.Exception
{
```

Analysis:	Not an Issue	
Comments:	*VHAHAMWandlJ 2017-09-14 4:55 PM*	The references are not hardcoded passwords, but System Property names that refer to passwords

Category: Access Control: Database (2 Issues)



Abstract:

Without proper access control, the method getPharmacyStationIdsByVisn() in PharmacyDaoImpl.java can execute a SQL statement on line 224 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 may 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 for performing validation of user input. Struts and Spring MVC are among them. 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:

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 requiring that the current authenticated user have specific access to the invoice.

```
...
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});
...
```

PharmacyDaoImpl.java, line 224 (Access Control: Database)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		

Abstract: Without proper access control, the method getPharmacyStationIdsByVisn() in PharmacyDaoImpl.java can execute a SQL statement on line 224 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

Source: PharmacyManagementController.java:837 getStationIdsForSelect(1)

```
835  @CacheControl(policy = {CachePolicy.NO_CACHE})
836  @ResponseBody
837  public List<StationIdSelectModel> getStationIdsForSelect(HttpServletRequest request,
@RequestParam("visn") String visn)
838     throws JsonParseException, JsonMappingException, IOException {
839
```

Sink: PharmacyDaoImpl.java:224 org.hibernate.Criteria.add()

```
222       if(!"All".equalsIgnoreCase(visn)){
223
224        criteria.add(Restrictions.eq("visn", Long.valueOf(visn)));
225
226              try
```

PharmacyDaoImpl.java, line 260 (Access Control: Database)

Fortify Priority:	Low	**Folder**	Low

Kingdom:	Security Features
Abstract:	Without proper access control, the method getSelectedStationIdsByVisn() in PharmacyDaoImpl.java can execute a SQL statement on line 260 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

Source: PharmacyManagementController.java:837 getStationIdsForSelect(1)

```
835  @CacheControl(policy = {CachePolicy.NO_CACHE})
836  @ResponseBody
837  public List<StationIdSelectModel> getStationIdsForSelect(HttpServletRequest request,
@RequestParam("visn") String visn)
838     throws JsonParseException, JsonMappingException, IOException {
839
```

Sink: PharmacyDaoImpl.java:260 org.hibernate.Criteria.add()

```
258      if(!"All".equalsIgnoreCase(visn)){
259
260         criteria.add(Restrictions.eq("visn", Long.valueOf(visn)));
261
262         criteria.add(buildStringInCriterion("vaStationId", userStationIds));
```

Category: Denial of Service (1 Issues)

Abstract:

The call to readLine() at VistaInboundRpc.java line 80 might allow an attacker to crash the program or otherwise make it unavailable to legitimate users.

Explanation:

Attackers may be able to deny service to legitimate users by flooding the application with requests, but flooding attacks can often be defused at the network layer. More problematic are bugs that allow an attacker to overload the application using a small number of requests. Such bugs allow the attacker to specify the quantity of system resources their requests will consume or the duration for which they will use them.

Example 1: The following code allows a user to specify the amount of time for which a thread will sleep. By specifying a large number, an attacker may tie up the thread indefinitely. With a small number of requests, the attacker may deplete the application's thread pool.

```
int usrSleepTime = Integer.parseInt(usrInput);
Thread.sleep(usrSleepTime);
```

Example 2: The following code reads a String from a zip file. Because it uses the readLine() method, it will read an unbounded amount of input. An attacker may take advantage of this code to cause an OutOfMemoryException or to consume a large amount of memory so that the program spends more time performing garbage collection or runs out of memory during some subsequent operation.

```
InputStream zipInput = zipFile.getInputStream(zipEntry);
Reader zipReader = new InputStreamReader(zipInput);
BufferedReader br = new BufferedReader(zipReader);
String line = br.readLine();
```

Recommendations:

Validate user input to ensure that it will not cause inappropriate resource utilization.

Example 3: The following code allows a user to specify the amount of time for which a thread will sleep just as in Example 1, but only if the value is within reasonable bounds.

```
int usrSleepTime = Integer.parseInt(usrInput);
if (usrSleepTime >= SLEEP_MIN &&
usrSleepTime <= SLEEP_MAX) {
Thread.sleep(usrSleepTime);
} else {
throw new Exception("Invalid sleep duration");
}
}
```

Example 4: The following code reads a String from a zip file just as in Example 2, but the maximum string length it will read is MAX_STR_LEN characters.

```
InputStream zipInput = zipFile.getInputStream(zipEntry);
Reader zipReader = new InputStreamReader(zipInput);
BufferedReader br = new BufferedReader(zipReader);
```

```
StringBuffer sb = new StringBuffer();
int intC;
while ((intC = br.read()) != -1) {
char c = (char) intC;
if (c == '\n') {
break;
}
if (sb.length() >= MAX_STR_LEN) {
throw new Exception("input too long");
}
sb.append(c);
}
String line = sb.toString();
```

Tips:

1. Denial of service can happen even if the quantity of system resources that will be consumed or the duration for which they will be used is not controlled by an attacker, or at least not directly. Instead, a programmer might choose unsafe constant values for specifying these parameters. The Fortify Secure Coding Rulepacks will report such cases as potential Denial of Services vulnerabilities.

VistaInboundRpc.java, line 80 (Denial of Service)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The call to readLine() at VistaInboundRpc.java line 80 might allow an attacker to crash the program or otherwise make it unavailable to legitimate users.		
Sink:	VistaInboundRpc.java:80 readLine()		

```
78                      i = 0;
79                      sb2.append("<SIG>");
80                       while(null != (line = br.readLine())) {
81
82                        if (line.trim().equalsIgnoreCase("<StructuredSIG>")){
```

Category: J2EE Misconfiguration: Excessive Session Timeout (1 Issues)

Abstract:

An overly long session timeout gives attackers more time to potentially compromise user accounts.

Explanation:

The longer a session stays open, the larger the window of opportunity an attacker has to compromise user accounts. While a session remains active, an attacker may be able to brute-force a user's password, crack a user's wireless encryption key, or commandeer a session from an open browser. Longer session timeouts can also prevent memory from being released and eventually result in a denial of service if a sufficiently large number of sessions are created.

Example 1: If the session timeout is zero or less than zero, the session never expires. The following example shows a session timeout set to -1, which will cause the session to remain active indefinitely.

<session-config>
<session-timeout>-1</session-timeout>
</session-config>

The <session-timeout> tag defines the default session timeout interval for all sessions in the web application. If the <session-timeout> tag is missing, it is left to the container to set the default timeout.

This category was derived from the Cigital Java Rulepack. http://www.cigital.com/

Recommendations:

Set a session timeout that is 30 minutes or less, which both allows users to interact with the application over a period of time and provides a reasonable bound for the window of attack.

Example 2: The following example sets the session timeout to 20 minutes.

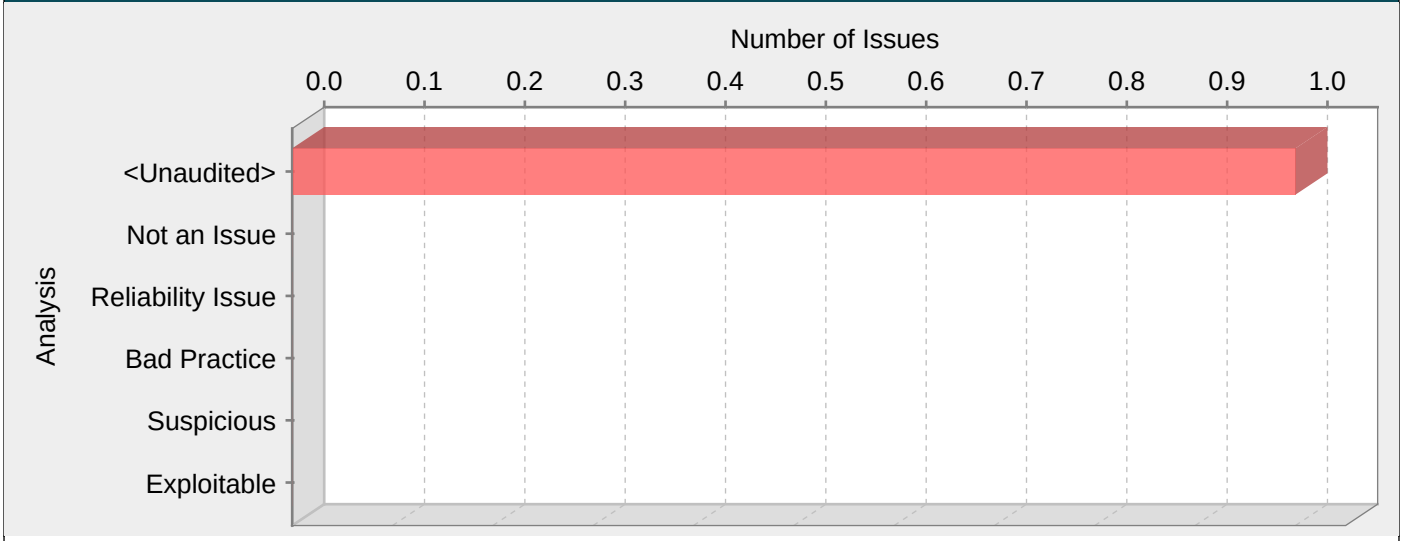
<session-config>
<session-timeout>20</session-timeout>
</session-config>

web.xml, line 103 (J2EE Misconfiguration: Excessive Session Timeout)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Environment		
Abstract:	An overly long session timeout gives attackers more time to potentially compromise user accounts.		
Sink:	web.xml:103 null()		

```
101
102                    <!--  Let javascript timeout on the UI first which is 30 minutes. -->
103                  <session-config>
104                  <!-- Defect 767939 set now in SessionExtensionController
105                   <session-timeout>31</session-timeout>
```

Category: Poor Error Handling: Throw Inside Finally (1 Issues)



Abstract:

Using a throw statement inside a finally block breaks the logical progression through the try-catch-finally.

Explanation:

In Java, finally blocks are always executed after their corresponding try-catch blocks and are often used to free allocated resources, such as file handles or database cursors. Throwing an exception in a finally block can bypass critical cleanup code since normal program execution will be disrupted.

Example 1: In the following code, the call to stmt.close() is bypassed when the FileNotFoundException is thrown.

```
public void processTransaction(Connection conn) throws FileNotFoundException
{
FileInputStream fis = null;
Statement stmt = null;
try
{
stmt = conn.createStatement();
fis = new FileInputStream("badFile.txt");
...
}
catch (FileNotFoundException fe)
{
log("File not found.");
}
catch (SQLException se)
{
//handle error
}
finally
{
if (fis == null)
{
throw new FileNotFoundException();
}
if (stmt != null)
{
try
{
stmt.close();
}
catch (SQLException e)
```

```
{
log(e);
}
}
}
}
```

This category is from the Cigital Java Rulepack. http://www.cigital.com/

Recommendations:

Never throw exceptions from within finally blocks. If you must re-throw an exception, do it inside a catch block so as not to interrupt the normal execution of the finally block.

Example 2: The following code re-throws the FileNotFoundException in the catch block.

```
public void processTransaction(Connection conn) throws FileNotFoundException
{
FileInputStream fis = null;
Statement stmt = null;
try
{
stmt = conn.createStatement();
fis = new FileInputStream("badFile.txt");
...
}
catch (FileNotFoundException fe)
{
log("File not found.");
throw fe;
}
catch (SQLException se)
{
//handle error
}
finally
{
if (fis != null)
{
try
{
fis.close();
}
catch (IOException ie)
{
log(ie);
}
}
if (stmt != null)
{
try
{
stmt.close();
}
catch (SQLException e)
{
log(e);
}
}
```

```
}
}
```

VistaLinkConnectionUtility.java, line 160 (Poor Error Handling: Throw Inside Finally)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	Using a throw statement inside a finally block breaks the logical progression through the try-catch-finally.		
Sink:	VistaLinkConnectionUtility.java:160 FinallyBlock()		

```
158                 //LOG.error("VistALink error while retrieving connection.", e);
159               throw new ResourceException(e);
160             }finally {
161               if (ic != null){
162                 ic.close();
```

Category: SQL Injection (1 Issues)

Abstract:

On line 1378 of NcpdpMessagesDaoImpl.java, the method searchMessages() invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Explanation:

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

In this case, Fortify Static Code Analyzer could not determine that the source of the data is trusted.

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

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

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

The query intends to execute the following code:

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

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

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

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

```
SELECT * FROM items;
```

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

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

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

DELETE FROM items;

--'
```

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

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

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

DELETE FROM items;

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

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

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

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

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

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

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

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

Recommendations:

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

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

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

And here is an Android equivalent:

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

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

Tips:

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

2. Data is untrustworthy if it originates from public non-final string fields of a class. These types of fields may be modified by an unknown source.

3. Fortify RTA adds protection against this category.

NcpdpMessagesDaoImpl.java, line 1378 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 1378 of NcpdpMessagesDaoImpl.java, the method searchMessages() invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		

Sink: NcpdpMessagesDaoImpl.java:1378 query()

```
1376    //LOG.info("Retrieving NCPDP message details.");
1377    //System.out.println("Retrieving NCPDP message details.");
1378      ncpdpMsgList = jdbcTemplate.query(sql,new NcpdpMsgListRowMapper(),messageId,
relatesToId, vaStationId, patientLastName, patientFirstName, prescriberLastName,
prescriberFirstName, prescriberDEA, prescriberNpi, prescribedDrug, inboundNcpdpMsgId);
1379    } catch (DataAccessException e) {
```

```
1380                    //ncpdpMsg.setDataError(e.getMessage());
```