Fortify Security Report Oct 24, 2017 npathak2

**Fortify Security Report**

Executive Summary Issues Overview

On Oct 24, 2017, a source code review was performed over the openmrs-api code base. 1,511 files, 52,043 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 356 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order High 282 Critical 74

Recommendations and Conclusions

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

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 2 of 40

**Fortify Security Report**

Code location: C:/eclipse/git/openmrs-core/api Number of Files: 1511 Lines of Code: 52043 Build Label: openmrs-api

Scan time: 34:36 SCA Engine version: 17.10.0156 Machine Name: win10ent Username running scan: npathak2

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface: Command Line Arguments: org.openmrs.test.MigrateDataSet.main

Environment Variables: java.lang.System.getenv

File System: java.io.FileInputStream.FileInputStream java.io.FileReader.FileReader java.util.jar.JarFile.entries java.util.jar.JarFile.getEntry

GUI Form: javax.swing.JPasswordField.getPassword javax.swing.JPasswordField.getText javax.swing.text.JTextComponent.getText

Private Information: null.null.null java.security.Provider.getProperty

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 3 of 40

Project Summary Code Base Summary

Scan Information

Results Certification

Attack Surface

**Fortify Security Report**

java.util.Properties.getProperty javax.crypto.KeyGenerator.generateKey javax.swing.JPasswordField.getPassword javax.swing.JPasswordField.getText org.openmrs.util.Security.decrypt

Java Properties: java.lang.System.getProperties java.lang.System.getProperty java.util.Properties.load org.springframework.orm.hibernate4.LocalSessionFactoryBean.getHibernateProperties

Stream: java.io.BufferedReader.read java.io.FileInputStream.read java.io.FilterInputStream.read java.io.InputStream.read java.io.Reader.read

System Information: null.null.null ca.uhn.hl7v2.HL7Exception.getMessage java.awt.HeadlessException.getMessage java.lang.ClassLoader.getResource java.lang.ClassLoader.getResources java.lang.Runtime.freeMemory java.lang.Runtime.maxMemory java.lang.Runtime.totalMemory java.lang.Throwable.getMessage java.net.InetAddress.getLocalHost java.net.URISyntaxException.getMessage java.net.URLClassLoader.findResource java.net.URLClassLoader.findResources java.rmi.RemoteException.getMessage java.util.regex.PatternSyntaxException.getMessage javax.xml.transform.TransformerFactoryConfigurationError.getMessage liquibase.exception.MigrationFailedException.getMessage org.apache.commons.lang.exception.ExceptionUtils.getStackTrace org.apache.commons.lang.exception.ExceptionUtils.getThrowables org.apache.velocity.exception.MethodInvocationException.getMessage org.apache.velocity.runtime.parser.ParseException.getMessage org.hibernate.QueryException.getMessage org.openmrs.module.ModuleClassLoader.findResource org.openmrs.module.ModuleClassLoader.findResources org.openmrs.util.OpenmrsClassLoader.findResource org.openmrs.util.OpenmrsClassLoader.findResources org.openmrs.util.OpenmrsClassLoader.getResources org.springframework.core.NestedRuntimeException.getMessage org.springframework.core.env.MissingRequiredPropertiesException.getMessage org.springframework.validation.BindException.getMessage org.xml.sax.SAXException.getMessage

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 4 of 40

**Fortify Security Report**

Filter Set Summary

Current Enabled Filter Set: Quick View

Filter Set Details:

Folder Filters: If [fortify priority order] contains critical Then set folder to Critical If [fortify priority order] contains high Then set folder to High If [fortify priority order] contains medium Then set folder to Medium If [fortify priority order] contains low Then set folder to Low Visibility Filters: If impact is not in range [2.5, 5.0] Then hide issue If likelihood is not in range (1.0, 5.0] Then hide issue

Audit Guide Summary

Audit guide not enabled

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 5 of 40

**Fortify Security Report**

The scan found 356 issues.

s i s y l a n A

Abstract: The method becomeUser() in Context.java writes unvalidated user input to the log on line 328. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log. Explanation: Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.

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

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

manually on an as-needed basis or

Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application render the file with unusable input that by includes corrupting appropriate the format characters. of the file If or the injecting log file unexpected is processed characters. automatically, A more the subtle attacker attack may might

be able to

involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate commands another into the party log file in the and commission take advantage of a malicious of a vulnerability act [1]. in In the the log worst processing case, an attacker utility [2].

may inject code or other

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

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

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

INFO: Failed to parse val=twenty-one

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 6 of 40

Results Outline Overall number of results

Vulnerability Examples by Category Category: Log Forging (165 Issues)

Number of Issues 0 25 50 75 100 125 150

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

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

INFO: Failed to parse val=twenty-one

INFO: User logged out=badguy

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

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

Example 2: The following code adapts Example 1 to the Android platform.

... String val = this.getIntent().getExtras().getString("val"); try { int value = Integer.parseInt(); } catch (NumberFormatException nfe) { Log.e(TAG, "Failed to parse val = " + val); } ... Recommendations: Prevent log forging attacks with indirection: create a set of legitimate log entries that correspond to different events that must be logged and only log entries from this set. To capture dynamic content, such as users logging out of the system, always use server- controlled values rather than user-supplied data. This ensures that the input provided by the user is never used directly in a log entry.

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

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

And here is an Android equivalent:

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

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 7 of 40

**Fortify Security Report**

In some situations this approach is impractical because the set of legitimate log entries is too large or complicated. In these situations, developers often fall back on blacklisting. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, a list of unsafe characters can quickly become incomplete or outdated. A better approach is to create a whitelist of characters that are allowed to appear in log entries and accept input composed exclusively of characters in the approved set. The most critical character in most log forging attacks is the '\n' (newline) character, which should never appear on a log entry whitelist. Tips: 1. Many logging operations are created only for the purpose of debugging a program during development and testing. In our experience, debugging will be enabled, either accidentally or purposefully, in production at some point. Do not excuse log forging vulnerabilities simply because a programmer says "I don't have any plans to turn that on in production".

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

Context.java, line 328 (Log Forging) Fortify Priority: High Folder High Kingdom: Input Validation and Representation Abstract: The method becomeUser() in Context.java writes unvalidated user input to the log on

line 328. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log. Source: HibernateUserDAO.java:101 org.hibernate.Query.list() 99 query.setString(0, username); 100 query.setString(1, username); 101 List<User> users = query.list(); 102 103 if (users == null || users.isEmpty()) { Sink: Context.java:328 org.slf4j.Logger.info() 326 public static void becomeUser(String systemId) throws ContextAuthenticationException { 327 if (log.isInfoEnabled()) { 328 log.info("systemId: " + systemId); 329 }

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 8 of 40

**Fortify Security Report**

s

Abstract: The is often method illegal.

APIException() in APIException.java mishandles confidential information, which can compromise user privacy and

Explanation: Privacy violations occur when:

1. Private user information enters the program.

2. The data is written to an external location, such as the console, file system, or network.

Example 1: The following code contains a logging statement that tracks the contents of records added to a database by storing them associated in a log with file. the Among account.

other values that are stored, the getPassword() function returns the user-supplied plaintext password

pass = getPassword(); ... dbmsLog.println(id+":"+pass+":"+type+":"+tstamp);

The code in the example above logs a plaintext password to the file system. Although many developers trust the file system as a safe storage location for data, it should not be trusted implicitly, particularly when privacy is a concern.

Privacy is one of the biggest concerns in the mobile world for a couple of reasons. One of them is a much higher chance of device is applications loss. The that other are has downloaded to do with from inter-process various sources communication and run alongside between each mobile other applications. on the same The device. essence The of likelihood mobile platforms

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 be part of inter-process they include communication in messages addressed between to mobile other applications.

applications running on the device. Sensitive information should never

Example 2: The code below reads username and password for a given site from an Android WebView store and broadcasts them to all the registered receivers.

... 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]; Intent i = new Intent(); i.setAction("SEND\_CREDENTIALS"); i.putExtra("username", username); i.putExtra("password", password); view.getContext().sendBroadcast(i); } }); ...

i s y l a n A

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 9 of 40

Category: Privacy Violation (71 Issues)

Number of Issues 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

There are several problems with this example. First of all, by default, WebView credentials are stored in plaintext 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. Second, plaintext credentials are broadcast to all the registered receivers, which means that any receiver registered to listen to intents with the SEND\_CREDENTIALS action will receive the message. The broadcast is not even protected with a permission to limit the number of recipients, even though in this case, we do not recommend using permissions as a fix.

Private data can enter a program in a variety of ways:

- Directly from the user in the form of a password or personal information

- Accessed from a database or other data store by the application

- Indirectly from a partner or other third party

Typically, in the context of the mobile world, this private information would include (along with passwords, SSNs and other general personal information):

- Location

- Cell phone number

- Serial numbers and device IDs

- Network Operator information

- Voicemail information

Sometimes data that is not labeled as private can have a privacy implication in a different context. For example, student identification numbers are usually not considered private because there is no explicit and publicly-available mapping to an individual student's personal information. However, if a school generates identification numbers based on student social security numbers, then the identification numbers should be considered private.

Security and privacy concerns often seem to compete with each other. From a security perspective, you should record all important operations so that any anomalous activity can later be identified. However, when private data is involved, this practice can create risk.

Although there are many ways in which private data can be handled unsafely, a common risk stems from misplaced trust. Programmers often trust the operating environment in which a program runs, and therefore believe that it is acceptable to store private information on the file system, in the registry, or in other locally-controlled resources. However, even if access to certain resources is restricted, this does not guarantee that the individuals who do have access can be trusted. For example, in 2004, an unscrupulous employee at AOL sold approximately 92 million private customer e-mail addresses to a spammer marketing an offshore gambling web site [1].

In response to such high-profile exploits, the collection and management of private data is becoming increasingly regulated. Depending on its location, the type of business it conducts, and the nature of any private data it handles, an organization may be required to comply with one or more of the following federal and state regulations:

- Safe Harbor Privacy Framework [3]

- Gramm-Leach Bliley Act (GLBA) [4]

- Health Insurance Portability and Accountability Act (HIPAA) [5]

- California SB-1386 [6]

Despite these regulations, privacy violations continue to occur with alarming frequency. Recommendations: When security and privacy demands clash, privacy should usually be given the higher priority. To accomplish this and still maintain required security information, cleanse any private information before it exits the program.

To enforce good privacy management, develop and strictly adhere to internal privacy guidelines. The guidelines should specifically describe how an application should handle private data. If your organization is regulated by federal or state law, ensure that your privacy guidelines are sufficiently strenuous to meet the legal requirements. Even if your organization is not regulated, you must protect private information or risk losing customer confidence.

The best policy with respect to private data is to minimize its exposure. Applications, processes, and employees should not be granted access to any private data unless the access is required for the tasks that they are to perform. Just as the principle of least privilege dictates that no operation should be performed with more than the necessary privileges, access to private data should be restricted to the smallest possible group.

For mobile applications, make sure they never communicate any sensitive data to other applications running on the device. When private data needs to be stored, it should always be encrypted. For Android, as well as any other platform that uses SQLite database, a good option is SQLCipher -- an extension to 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;

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 10 of 40

**Fortify Security Report**

... 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.

Example 4: The code below reads username and password for a given site from an Android WebView store and instead of broadcasting them to all the registered receivers, it only broadcasts internally so that the broadcast can only be seen by other parts of the same app.

... 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]; Intent i = new Intent(); i.setAction("SEND\_CREDENTIALS"); i.putExtra("username", username); i.putExtra("password", password); LocalBroadcastManager.getInstance(view.getContext()).sendBroadcast(i); } }); ... Tips: 1. As part of any thorough audit for privacy violations, ensure that custom rules have been written to identify all sources of private or otherwise sensitive information entering the program. Most sources of private data cannot be identified automatically. Without custom rules, your check for privacy violations is likely to be substantially incomplete.

2. The Fortify Java Annotations FortifyPassword, FortifyNotPassword, FortifyPrivate and FortifyNotPrivate can be used to indicate which fields and variables represent passwords and private data.

3. 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 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.

APIException.java, line 37 (Privacy Violation) Fortify Priority: High Folder High Kingdom: Security Features Abstract: The method APIException() in APIException.java mishandles confidential

information, which can compromise user privacy and is often illegal. Source: Context.java:862 Read password() 860 properties.put("connection.url", url); 861 properties.put("connection.username", username); 862 properties.put("connection.password", password); 863 setRuntimeProperties(properties); Sink: APIException.java:37 java.lang.RuntimeException.RuntimeException() 35 \*/

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 11 of 40

**Fortify Security Report**

36 public APIException(String message) { 37 super(message); 38 } 39

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 12 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: Hardcoded passwords may compromise system security in a way that cannot be easily remedied. Explanation: It the is password, never a good it also idea makes to hardcode fixing the a password. problem extremely Not only does difficult. hardcoding Once the a password code is in allow production, all of the the project's password developers cannot be

to view

changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to 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. Once 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 application access they to this can information use the javap could -c command use it to break to access into the the system. disassembled Even worse, code, which if attackers will contain have access the values to the of bytecode the passwords for the

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 world, password management is even trickier, considering a much higher 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 the plaintext very least, anywhere passwords on the should system be hashed allows before anyone being with stored.

sufficient permissions to read and potentially misuse the password. At

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 13 of 40

Category: Password Management: Hardcoded Password (24 Issues)

Number of Issues 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0 22.5 25.0

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

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

For Android, as well as any other platform that uses SQLite database, a good option is SQLCipher -- an extension to 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. When identifying 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.

SchedulerConstants.java, line 22 (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: SchedulerConstants.java:22 FieldAccess: SCHEDULER\_DEFAULT\_PASSWORD() 20 public static String SCHEDULER\_DEFAULT\_USERNAME = "admin"; 21 22 public static String SCHEDULER\_DEFAULT\_PASSWORD =\*\*\*\*\*\* 23 24 /\*\* The default 'from' address for emails send by the schedule \*/

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 14 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: Storing a plaintext password in a configuration file may result in a system compromise. Explanation: Storing resource. a plaintext Developers password sometimes in a believe configuration that they file cannot allows defend anyone the who application can read from the file someone access to who the has password-protected

access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plaintext. Recommendations: A starts. password If that should approach never is impractical, be stored in a plaintext. less secure Instead, but often the password adequate solution should be is entered to obfuscate by an the administrator password and when scatter the system the de- obfuscation material around the system so that an attacker has to obtain and correctly combine multiple system resources to decipher the password.

Some Server third-party 4.x uses a simple products XOR claim encryption the ability algorithm to manage for passwords obfuscating in values, a more secure but be skeptical way. For about example, such WebSphere facilities. WebSphere Application

and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. For a secure solution the only viable option is a proprietary one. Tips: 1. by Fortify verifying Static that Code the flagged Analyzer entry searches is used configuration as a password files and for that common the password names entry used contains for password plaintext.

properties. Audit these issues

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

hibernate.default.properties, line 5 (Password Management: Password in Configuration File) Fortify Priority: High Folder High Kingdom: Environment Abstract: Storing a plaintext password in a configuration file may result in a system

compromise. Sink: hibernate.default.properties:5 hibernate.connection.password() 3 # Connection Properties --> 4 hibernate.connection.username=test 5 hibernate.connection.password=\*\*\*\*\*\* 6 hibernate.connection.driver\_class=com.mysql.jdbc.Driver 7 hibernate.connection.url=jdbc:mysql://localhost:3306/openmrs?autoReconnect=true

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 15 of 40

Category: Password Management: Password in Configuration File (21 Issues)

Number of Issues 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

s i s y l a n A

Abstract: Untrusted resources.

data is passed to the application and used as a regular expression. This can cause the thread to over-consume CPU

Explanation: There is a vulnerability in implementations of regular expression evaluators and related methods that can cause the thread to hang when evaluating repeating and alternating overlapping of nested and repeated regex groups. This defect can be used to execute a Denial of Service (DoS) attack. Example:

(e+)+ ([a-zA-Z]+)\* (e|ee)+

There vulnerable are no to known this attack.

regular expression implementations which are immune to this vulnerability. All platforms and languages are

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

HibernatePatientDAO.java, line 772 (Denial of Service: Regular Expression) Fortify Priority: High Folder High Kingdom: Input Validation and Representation Abstract: Untrusted data is passed to the application and used as a regular expression. This can

cause the thread to over-consume CPU resources. Source: HibernateAdministrationDAO.java:103 org.hibernate.Session.get() 101 return gp; 102 } else { 103 return (GlobalProperty) sessionFactory.getCurrentSession().get(GlobalProperty.class,

propertyName); 104 } 105 } Sink: HibernatePatientDAO.java:772 java.util.regex.Pattern.compile() 770 if (Pattern.matches("^\\^.{1}\\\*.\*$", regex)) { 771 String padding = regex.substring(regex.indexOf("^") + 1, regex.indexOf("\*")); 772 Pattern pattern = Pattern.compile("^" + padding + "+"); 773 query = pattern.matcher(query).replaceFirst(""); 774 }

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 16 of 40

Category: Denial of Service: Regular Expression (20 Issues)

Number of Issues 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

s i s y l a n A

Abstract: Attackers access or modify are able otherwise to control protected the file system files.

path argument to File() at HL7ServiceImpl.java line 1163, which allows them to

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

1. An attacker is able to specify a path used in an operation on the file system.

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

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

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

the

one of its own configuration files.

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

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

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

Some attack think themself? that in However, the mobile keep world, in mind classic that vulnerabilities, the essence of mobile such as platforms path manipulation, is applications do not that make are sense downloaded -- why would from various

the user

sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application communication.

is high, which necessitates expanding the attack surface of mobile applications to include inter-process

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

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

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 17 of 40

Category: Path Manipulation (15 Issues)

Number of Issues 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a whitelist of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set. Tips: 1. If the program is performing input validation, satisfy yourself that the validation is correct, and use the Fortify Custom Rules Editor to create a cleanse rule for the validation routine.

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

3. 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 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.

HL7ServiceImpl.java, line 1163 (Path Manipulation) Fortify Priority: Critical Folder Critical Kingdom: Input Validation and Representation Abstract: Attackers are able to control the file system path argument to File() at

HL7ServiceImpl.java line 1163, which allows them to access or modify otherwise protected files. Source: HibernateHL7DAO.java:373 org.hibernate.Criteria.list() 371 crit.add(Restrictions.lt("dateCreated", cal.getTime())); 372 } 373 return crit.list(); 374 } 375 Sink: HL7ServiceImpl.java:1163 java.io.File.File() 1161 //use the uuid, source id and source key(if present) to generate the file name 1162 File fileToWriteTo = new File(dayDir, hl7InArchive.getUuid() 1163 + (StringUtils.isBlank(hl7InArchive.getHL7SourceKey()) ? "" : "\_" +

hl7InArchive.getHL7SourceKey()) 1164 + ".txt"); 1165

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 18 of 40

**Fortify Security Report**

s

Abstract: The user class could ServiceContext see another user's is a data.

singleton, so the member field applicationContext is shared between users. The result is that one

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

A common result of this misunderstanding is that developers use Servlet member fields in such a way that one user may inadvertently condition.

see another user's data. In other words, storing user data in Servlet member fields introduces a data access race

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

public class GuestBook extends HttpServlet {

String name;

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

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

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

Thereby showing the first user the second user's name. Recommendations: Do not use Servlet member fields for anything but constants. (i.e. make all member fields static final).

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

of

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

public class GuestBook extends HttpServlet {

protected void doPost (HttpServletRequest req, HttpServletResponse res) { GBRequestHandler handler = new GBRequestHandler(); handler.handle(req, res); }

i s y l a n A

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 19 of 40

Category: Race Condition: Singleton Member Field (15 Issues)

Number of Issues 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

}

public class GBRequestHandler {

String name;

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

}

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

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

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

public class GuestBook extends HttpServlet {

String name;

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

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

public class GuestBook extends HttpServlet {

String name;

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

ServiceContext.java, line 896 (Race Condition: Singleton Member Field) Fortify Priority: High Folder High Kingdom: Time and State Abstract: The class ServiceContext is a singleton, so the member field applicationContext is

shared between users. The result is that one user could see another user's data. Sink: ServiceContext.java:896 AssignmentStatement() 894 @Override 895 public void setApplicationContext(ApplicationContext applicationContext) { 896 this.applicationContext = applicationContext; 897 } 898

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 20 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: The controlled file SimpleXStreamSerializer.java XML documents at run-time deserializes can allow attackers unvalidated to execute XML input malicious using arbitrary XStream code on line on the 107. server.

Deserializing user-

Explanation: XStream library provides the developer with an easy way to transmit objects, serializing them to XML documents. But XStream can by default deserialize dynamic proxies allowing an attacker to run arbitrary Java code on the server when the proxy's InvocationHandler is invoked.

Example 1: The following Java code shows an instance of XStream processing untrusted input.

XStream xstream = new XStream(); String body = IOUtils.toString(request.getInputStream(), "UTF-8"); Contact expl = (Contact) xstream.fromXML(body);

Example the Windows 2: The calculator.

following XML document will instantiate a ProcessBuilder object and will invoke its static start() method to run

<dynamic-proxy> <interface>com.company.model.Contact</interface> <handler class="java.beans.EventHandler"> <target class="java.lang.ProcessBuilder"> <command><string>/Applications/Calculator.app/Contents/MacOS/Calculator</string></command> </target> <action>start</action> </handler> </dynamic-proxy> Recommendations: XStream attackers implicitly to run arbitrary prevents commands. the deserialization In addition, of starting known with bad classes XStream such 1.4.7, as java.beans.EventHandler it is possible to define permissions that can be for used types. by

These permissions can be used to explicitly allow or deny the types that will be deserialized and so it is not possible to inject unexpected limit the danger types of into arbitrary an object command graph. execution. Any application Always that use deserializes the whitelist data approach from an (allowed external types) source since should many use this classes feature can be to

used to achieve remote code execution and to bypass blacklists.

Example 3: The following Java code shows an instance of XStream securely processing untrusted input by defining the allowed types.

XStream xstream = new XStream(); // clear out existing permissions and set own ones xstream.addPermission(NoPermissionType.NONE); // allow some basics xstream.addPermission(NullPermission.NULL); xstream.addPermission(PrimitiveTypePermission.PRIMITIVES); xstream.allowTypeHierarchy(Collection.class);

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 21 of 40

Category: Dynamic Code Evaluation: Unsafe XStream Deserialization (10 Issues)

Number of Issues 0 1 2 3 4 5 6 7 8 9 10

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

// allow any type from the same package xstream.allowTypesByWildcard(new String[] { Contact.class.getPackage().getName()+".\*" }); String body = IOUtils.toString(request.getInputStream(), "UTF-8"); Contact expl = (Contact) xstream.fromXML(body);

Note that any class allowed in the whitelist should be audited to make sure it is safe to deserialize.

If using XStream as the marshalling solution for a Spring project, use the "converters" property to set up a chain of custom converters starting with the org.springframework.oxm.xstream.CatchAllConverter converter as shown in the example below:

<bean id="marshallingHttpMessageConverter" class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter"> <property name="marshaller" ref="xstreamMarshaller"/> <property name="unmarshaller" ref="xstreamMarshaller"/> </bean>

<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller"> <property name="aliases"> <props> <prop key="contact">org.company.converters.Contact</prop> </props> </property> <property name="converters"> <list> <bean class="org.springframework.oxm.xstream.CatchAllConverter"/> <bean class="org.company.converters.ContactConverter"/> </list> </property> </bean>

SimpleXStreamSerializer.java, line 107 (Dynamic Code Evaluation: Unsafe XStream Deserialization) Fortify Priority: High Folder High Kingdom: Input Validation and Representation Abstract: The file SimpleXStreamSerializer.java deserializes unvalidated XML input using

XStream on line 107. Deserializing user-controlled XML documents at run-time can allow attackers to execute malicious arbitrary code on the server. Source: HibernateSerializedObjectDAO.java:96 org.hibernate.Criteria.uniqueResult() 94 Criteria c =

sessionFactory.getCurrentSession().createCriteria(SerializedObject.class); 95 c.add(Restrictions.eq("uuid", uuid)); 96 ret = (SerializedObject) c.uniqueResult(); 97 } 98 return ret; Sink: SimpleXStreamSerializer.java:107

com.thoughtworks.xstream.XStream.fromXML() 105 106 try { 107 return (T) xstream.fromXML(serializedObject); 108 } 109 catch (XStreamException e) {

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 22 of 40

**Fortify Security Report**

s

Abstract: The the program method execMysqlCmd() to execute malicious in MigrateDataSet.java commands on behalf calls of an exec() attacker.

with a command built from untrusted data. This call can cause

Explanation: Command injection vulnerabilities take two forms:

- An attacker can change the command that the program executes: the attacker explicitly controls what the command is.

- An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means.

In this case we are primarily concerned with the first scenario, the possibility that an attacker may be able to control the command that is executed. Command injection vulnerabilities of this type occur when:

1. Data enters the application from an untrusted source.

2. The data is used as or as part of a string representing a command that is executed by the application.

3. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

Example 1: The following code from a system utility uses the system property APPHOME to determine the directory in which it is installed and then executes an initialization script based on a relative path from the specified directory.

... String home = System.getProperty("APPHOME"); String cmd = home + INITCMD; java.lang.Runtime.getRuntime().exec(cmd); ...

The code in Example 1 allows an attacker to execute arbitrary commands with the elevated privilege of the application by modifying the system property APPHOME to point to a different path containing a malicious version of INITCMD. Because the program APPHOME, does then not validate they can the fool value the application read from the into environment, running malicious if an attacker code and can take control control the of value the system.

of the system property

Example 2: The following code is from an administrative web application designed to allow users to kick off a backup of an Oracle database using a batch-file wrapper around the rman utility and then run a cleanup.bat script to delete some temporary files. access The to the script database rmanDB.bat is restricted, accepts the a single application command runs the line backup parameter, as a which privileged specifies user.

the type of backup to perform. Because

... String btype = request.getParameter("backuptype"); String cmd = new String("cmd.exe /K \"c:\\util\\rmanDB.bat "+btype+"&&c:\\util\\cleanup.bat\"") System.Runtime.getRuntime().exec(cmd); ...

i s y l a n A

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 23 of 40

Category: Command Injection (4 Issues)

Number of Issues 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

The problem here is that the program does not do any validation on the backuptype parameter read from the user. Typically the Runtime.exec() function will not execute multiple commands, but in this case the program first runs the cmd.exe shell in order to run multiple commands with a single call to Runtime.exec(). Once the shell is invoked, it will allow for the execution of multiple commands separated by two ampersands. If an attacker passes a string of the form "&& del c:\\dbms\\\*.\*", then the application will execute this command along with the others specified by the program. Because of the nature of the application, it runs with the privileges necessary to interact with the database, which means whatever command the attacker injects will run with those privileges as well.

Example 3: The following code is from a web application that allows users access to an interface through which they can update their password on the system. Part of the process for updating passwords in certain network environments is to run a make command in the /var/yp directory, the code for which is shown below.

... System.Runtime.getRuntime().exec("make"); ...

The problem here is that the program does not specify an absolute path for make and fails to clean its environment prior to executing the call to Runtime.exec(). If an attacker can modify the $PATH variable to point to a malicious binary called make and cause the program to be executed in their environment, then the malicious binary will be loaded instead of the one intended. Because of the nature of the application, it runs with the privileges necessary to perform system operations, which means the attacker's make will now be run with these privileges, possibly giving the attacker complete control of the system.

Some think that in the mobile world, classic vulnerabilities, such as command injection, do not make sense -- why would a user attack him or herself? 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 4: The following code reads commands to be executed from an Android intent.

... String[] cmds = this.getIntent().getStringArrayExtra("commands"); Process p = Runtime.getRuntime().exec("su"); DataOutputStream os = new DataOutputStream(p.getOutputStream()); for (String cmd : cmds) { os.writeBytes(cmd+"\n"); } os.writeBytes("exit\n"); os.flush(); ...

On a rooted device, a malicious application can force a victim application to execute arbitrary commands with super user privileges. Recommendations: Do not allow users to have direct control over the commands executed by the program. In cases where user input must affect the command to be run, use the input only to make a selection from a predetermined set of safe commands. If the input appears to be malicious, the value passed to the command execution function should either default to some safe selection from this set or the program should decline to execute any command at all.

In cases where user input must be used as an argument to a command executed by the program, this approach often becomes impractical because the set of legitimate argument values is too large or too hard to keep track of. Developers often fall back on blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. Any list of unsafe characters is likely to be incomplete and will be heavily dependent on the system where the commands are executed. A better approach is to create a whitelist of characters that are allowed to appear in the input and accept input composed exclusively of characters in the approved set.

An attacker may indirectly control commands executed by a program by modifying the environment in which they are executed. The environment should not be trusted and precautions should be taken to prevent an attacker from using some manipulation of the environment to perform an attack. Whenever possible, commands should be controlled by the application and executed using an absolute path. In cases where the path is not known at compile time, such as for cross-platform applications, an absolute path should be constructed from trusted values during execution. Command values and paths read from configuration files or the environment should be sanity-checked against a set of invariants that define valid values.

Other checks can sometimes be performed to detect if these sources may have been tampered with. For example, if a configuration file is world-writable, the program might refuse to run. In cases where information about the binary to be executed is known in advance, the program may perform checks to verify the identity of the binary. If a binary should always be owned by a particular user or have a particular set of access permissions assigned to it, these properties can be verified programmatically before the binary is executed.

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 24 of 40

**Fortify Security Report**

Although it may be impossible to completely protect a program from an imaginative attacker bent on controlling the commands the program executes, be sure to apply the principle of least privilege wherever the program executes an external command: do not hold privileges that are not essential to the execution of the command. Tips: 1. 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 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.

2. Fortify RTA adds protection against this category.

MigrateDataSet.java, line 187 (Command Injection) Fortify Priority: Critical Folder Critical Kingdom: Input Validation and Representation Abstract: The method execMysqlCmd() in MigrateDataSet.java calls exec() with a command

built from untrusted data. This call can cause the program to execute malicious commands on behalf of an attacker. Source: BaseContextSensitiveTest.java:533

javax.swing.text.JTextComponent.getText() 531 // response of 2 is the cancel button, response of -1 is the little red 532 // X in the top right 533 return (response == 2 || response == -1 ? null : new String[] {

usernameField.getText(), 534 String.valueOf(passwordField.getPassword()) }); 535 } Sink: MigrateDataSet.java:187 java.lang.Runtime.exec() 185 // Could not create lcab.tmp file in default working directory 186 // (jmiranda). 187 Process p = (wd != null) ? Runtime.getRuntime().exec(cmds, null, wd) :

Runtime.getRuntime().exec(cmds); 188 189 // get the stdout

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 25 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: Hardcoded encryption keys may compromise system security in a way that cannot be easily remedied. Explanation: It and is makes never a fixing good the idea problem to hardcode extremely an encryption difficult. key Once because the code it allows is in production, all of the project's the encryption developers key to cannot view be the changed encryption without key,

patching the software. If the account that is protected by the encryption key is compromised, the owners of the system will be forced to choose between security and availability.

Example 1: The following code uses a hardcoded encryption key:

... private static final String encryptionKey = "lakdsljkalkjlksdfkl"; byte[] keyBytes = encryptionKey.getBytes(); SecretKeySpec key = new SecretKeySpec(keyBytes, "AES"); Cipher encryptCipher = Cipher.getInstance("AES"); encryptCipher.init(Cipher.ENCRYPT\_MODE, key); ...

Anyone change the who encryption has access key to unless the code the will program have access is patched. to the An encryption employee key. with Once access the to application this information has shipped, could use there it is to no break way into to

the system. Even worse, if attackers had access to the executable for the application, they could extract the encryption key value. Recommendations: Encryption keys should never be hardcoded and should be obfuscated and managed in an external source. Storing encryption keys key.

in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the encryption

OpenmrsConstants.java, line 524 (Key Management: Hardcoded Encryption Key) Fortify Priority: High Folder High Kingdom: Security Features Abstract: Hardcoded encryption keys may compromise system security in a way that cannot be

easily remedied. Sink: OpenmrsConstants.java:524 FieldAccess: ENCRYPTION\_KEY\_SPEC() 522 public static final String ENCRYPTION\_CIPHER\_CONFIGURATION = "AES/CBC/PKCS5Padding"; 523 524 public static final String ENCRYPTION\_KEY\_SPEC = "AES"; 525 526 public static final String ENCRYPTION\_VECTOR\_RUNTIME\_PROPERTY = "encryption.vector";

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 26 of 40

Category: Key Management: Hardcoded Encryption Key (3 Issues)

Number of Issues 0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

s i s y l a n A

Abstract: The the data method from importUsers() memory.

in MigrationHelper.java stores sensitive data in a String object, making it impossible to reliably purge

Explanation: Sensitive data (such as passwords, social security numbers, credit card numbers etc) stored in memory can be leaked if memory is not cleared after use. Often, Strings are used store sensitive data, however, since String objects are immutable, removing the value unless of the a String JVM is from low memory on memory, can so only there be done is no by guarantee the JVM as garbage to when collector. garbage collection The garbage will collector take place. is not In the required event to of run an application crash, a memory dump of the application might reveal sensitive data.

Example 1: The following code converts a password from a character array to a String.

private JPasswordField pf; ... final char[] password = pf.getPassword(); ... String passwordAsString = new String(password);

This category was derived from the Cigital Java Rulepack. http://www.cigital.com/ Recommendations: Always be sure to clear sensitive data when it is no longer needed. Instead of storing sensitive data in immutable objects like Strings, use byte arrays or character arrays that can be programmatically cleared.

Example 2: The following code clears memory after a password is used.

private JPasswordField pf; ... final char[] password = pf.getPassword(); // use the password ... // erase when finished Arrays.fill(password, ' '); Tips: 1. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Spring MVC are the among issues reported them. To by highlight Fortify Static the unvalidated Code Analyzer sources by of lowering input, the their Fortify probability Secure of Coding exploit Rulepacks and providing dynamically pointers re-prioritize

to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. available the To further Data Validation assist the project Fortify template user with that the groups auditing the process, issues into the Fortify folders Software based on Security the validation Research mechanism group makes

applied to their source of input.

MigrationHelper.java, line 153 (Privacy Violation: Heap Inspection) Fortify Priority: High Folder High Kingdom: Security Features

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 27 of 40

Category: Privacy Violation: Heap Inspection (3 Issues)

Number of Issues 0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

Abstract: The method importUsers() in MigrationHelper.java stores sensitive data in a String

object, making it impossible to reliably purge the data from memory. Source: MigrationHelper.java:153 Read password() 151 password[x] = (char) randDecimalAsciiVal; 152 } 153 pass = new String(password); 154 } 155 us.createUser(user, pass); Sink: MigrationHelper.java:153 java.lang.String.String() 151 password[x] = (char) randDecimalAsciiVal; 152 } 153 pass = new String(password); 154 } 155 us.createUser(user, pass);

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 28 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: Deserializing application logic, user-controlled and/or lead object to denial streams of service.

at runtime can allow attackers to execute arbitrary code on the server, abuse

Explanation: Java serialization turns object graphs into byte streams that contain the objects themselves and the necessary metadata to reconstruct them from the byte stream. Developers can create custom code to aid in the process of deserializing Java objects, where place during they can objects replace reconstruction, the deserialized before objects the objects with different are returned objects, to the or proxies. application The and customized cast into deserialization expected types. process By the takes time developers try to enforce an expected type, code may have already been executed.

Custom deserialization routines are defined in the serializable classes which need to be present in the runtime classpath and cannot environment. be injected Unfortunately, by the attacker common so the third exploitability party classes of or these even attacks JDK classes depends can on be the abused classes to available exhaust in JVM the resources, application

deploy malicious files, or run arbitrary code.

Example 1: An application deserializing untrusted object streams can lead to application compromise.

InputStream is = request.getInputStream(); ObjectInputStream ois = new ObjectInputStream(is); MyObject obj = (MyObject) ois.readObject(); Recommendations: If being possible, deserialized, do not deserialize the look-ahead untrusted deserialization data without pattern validating should the be contents used.

of the object stream. In order to validate classes

The object stream will first contain the class description metadata and then the serialized bytes of their member fields. The Java serialization process allows developers to read the class description and decide whether to proceed with the deserialization of the object the resolveClass(ObjectStreamClass or abort it. In order to do so, it desc) is necessary method to where subclass class java.io.ObjectInputStream validation and verification and should provide take a place.

custom implementation of

There are existing implementations of the look-ahead pattern that can be easily used, such as the Apache Commons IO (org.apache.commons.io.serialization.ValidatingObjectInputStream). Always use a strict whitelist approach to only deserialize expected Also, keep types. in mind A blacklist that although approach some is classes not recommended to achieve code since execution attackers are may publicly use many known, available there gadgets may be to others bypass that the are

blacklist.

unknown or undisclosed, so a whitelist approach will always be preferred. Any class allowed in the whitelist should be audited to make sure it is safe to deserialize.

When recommendation deserialization is not takes useful place since in library, it is beyond or framework the developer's (e.g. when control. using In those JMX, cases, RMI, you JMS, may HTTP want Invokers) to make the sure above

that these protocols meet the following requirements:

- Not exposed publicly. - Use authentication. - Use integrity checks. - Use encryption.

In addition, Fortify Runtime provides security controls to be enforced every time the application performs a deserialization from an ObjectInputStream, protecting both application code but also library and framework code from this type of attack. Tips:

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 29 of 40

Category: Dynamic Code Evaluation: Unsafe Deserialization (2 Issues)

Number of Issues 0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

1. Due to existing flaw in ObjectInputStream implementation and the difficulties of blacklisting basic classes that may be used to perform denial of service (DoS) attacks, this issue will be reported even if a look-ahead ObjectInputStream is implemented but its severity will be lowered to Medium.

JavaSerializationTest.java, line 48 (Dynamic Code Evaluation: Unsafe Deserialization) Fortify Priority: High Folder High Kingdom: Input Validation and Representation Abstract: Deserializing user-controlled object streams at runtime can allow attackers to execute

arbitrary code on the server, abuse application logic, and/or lead to denial of service. Sink: JavaSerializationTest.java:48 FunctionCall: deserialize() 46 47 byte[] serialized = SerializationUtils.serialize(originalPerson); 48 Person copyPerson = (Person) SerializationUtils.deserialize(serialized); 49 50 assertThat(copyPerson.getGender(), is(originalPerson.getGender()));

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 30 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: The resource function URI. openConnection() An attacker may leverage on line 720 this initiates vulnerability a network to send connection a request to on a third-party behalf of the system application using user-controlled server since the data request for

will originate from the application server internal IP. Explanation: A Server-Side Request Forgery occurs when an attacker may influence a network connection made by the application server. The network bypass network connection controls will originate and scan or from attack the internal application resources server that internal are not IP and otherwise an attacker exposed.

will be able to use this connection to

Example: In the following example, an attacker will be able to control the URL the server is connecting to.

String url = request.getParameter("url"); CloseableHttpClient httpclient = HttpClients.createDefault(); HttpGet httpGet = new HttpGet(url); CloseableHttpResponse response1 = httpclient.execute(httpGet);

The ability of the attacker to hijack the network connection will depend on the specific part of the URI that he can control and on libraries from http used or https to stablish like:

the connection. For example, controlling the URI scheme will let the attacker use protocols different

- up:// - ldap:// - jar:// - gopher:// - mailto:// - ssh2:// - telnet:// - expect://

An attacker will be able to leverage this hijacked network connection to perform the following attacks:

- Port Scanning of intranet resources. - Bypass firewalls. - Attack vulnerable programs running on the application server or on the Intranet. - Attack internal/external web applications using Injection attacks or CSRF. - Access local files using file:// scheme. - On Windows systems, file:// scheme and UNC paths can allow an attacker to scan and access internal shares. - Perform a DNS cache poisoning attack. Recommendations: Do destination. not establish If user network data is connections necessary to based build on the user-controlled destination URI, data use and a level ensure of that indirection: the request create is being a list of sent legitimate to the expected

resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 31 of 40

Category: Server-Side Request Forgery (1 Issues)

Number of Issues 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

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

Also, if required, make sure that the user input is only used to specify a resource on the target system but that the URI scheme, host, and port is controlled by the application. This way the damage that an attacker is able to do will be significantly reduced.

ModuleUtil.java, line 720 (Server-Side Request Forgery) Fortify Priority: High Folder High Kingdom: Input Validation and Representation Abstract: The function openConnection() on line 720 initiates a network connection to a third-

party system using user-controlled data for resource URI. An attacker may leverage this vulnerability to send a request on behalf of the application server since the request will originate from the application server internal IP. Source: ModuleUtil.java:707 java.net.URLConnection.getHeaderField() 705 if (stat == 300 || stat == 301 || stat == 302 || stat == 303 || stat == 305 || stat ==

307) { 706 URL base = http.getURL(); 707 String loc = http.getHeaderField("Location"); 708 URL target = null; 709 if (loc != null) { Sink: ModuleUtil.java:720 java.net.URL.openConnection() 718 } 719 redir = true; 720 c = target.openConnection(); 721 redirects++; 722 }

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 32 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: The template, call to allowing evaluate() attackers in VelocityMessagePreparator.java to access the template context on and line in 60 some evaluates cases inject user-controlled and run arbitrary data as code a template in the engine's

application server. Explanation: Template engines are used to render content using dynamic data. This context data is normally controlled by the user and formatted be used in by templates the template in order to generate to render web dynamic pages, content, emails by and processing the like. Template the context engines data with allow code powerful constructs language such expressions as conditionals,

to

loops, etc. If an attacker is able to control the template to be rendered, they will be able to inject expressions that will expose context data or even run arbitrary commands on the server.

Example 1: The example below shows how a template is retrieved from an HTTP request and rendered.

// Set up the context data VelocityContext context = new VelocityContext(); context.put( "name", user.name );

// Load the template String template = getUserTemplateFromRequestBody(request); RuntimeServices runtimeServices = RuntimeSingleton.getRuntimeServices(); StringReader reader = new StringReader(template); SimpleNode node = runtimeServices.parse(reader, "myTemplate"); template = new Template(); template.setRuntimeServices(runtimeServices); template.setData(node); template.initDocument();

// Render the template with the context data StringWriter sw = new StringWriter(); template.merge( context, sw );

The example above uses Velocity as the template engine. For that engine, an attacker could submit the following template to run arbitrary commands on the server:

$name.getClass().forName("java.lang.Runtime").getRuntime().exec(<COMMAND>) Recommendations: Whenever possible do not allow users to provide templates. If user-provided templates are necessary, perform careful input validation to prevent malicious code from being injected in the template.

VelocityMessagePreparator.java, line 60 (Server-Side Template Injection) Fortify Priority: Critical Folder Critical Kingdom: Input Validation and Representation

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 33 of 40

Category: Server-Side Template Injection (1 Issues)

Number of Issues 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

Abstract: The call to evaluate() in VelocityMessagePreparator.java on line 60 evaluates user-

controlled data as a template engine's template, allowing attackers to access the template context and in some cases inject and run arbitrary code in the application server. Source: HibernateTemplateDAO.java:60 org.hibernate.Query.list() 58 log.info("Get template " + name); 59 return sessionFactory.getCurrentSession().createQuery("from Template as template where

template.name = ?") 60 .setString(0, name).list(); 61 } 62 Sink: VelocityMessagePreparator.java:60

org.apache.velocity.app.VelocityEngine.evaluate() 58 try { 59 engine.evaluate(context, writer, "template", // I have no idea what this is used for 60 template.getTemplate()); 61 } 62 catch (Exception e) {

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 34 of 40

**Fortify Security Report**

s i s y l a n A

Abstract: On input line coming 164 of from MigrateAllergiesChangeSet.java, an untrusted source. This call the could method allow getConceptByGlobalProperty() an attacker to modify the statement's invokes meaning a SQL or query to execute

built using

arbitrary SQL commands. Explanation: SQL injection errors occur when:

1. Data enters a program from an untrusted source.

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

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

matching a specified

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 that this code intends to execute follows:

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

However, query only because behaves the correctly query is if constructed itemName does dynamically not contain by concatenating a single-quote a character. constant base If an query attacker string with and the a user user name input wiley

string, the

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.

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 35 of 40

Category: SQL Injection (1 Issues)

Number of Issues 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

<Unaudited>

Not an Issue

Reliability Issue

Bad Practice

Suspicious

Exploitable

**Fortify Security Report**

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

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

DELETE FROM items;

--'

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

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

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

DELETE FROM items;

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

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

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

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

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from 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:

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 36 of 40

**Fortify Security Report**

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

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

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

And here is an Android equivalent:

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

More complicated scenarios, often found in report generation code, require that user input affect the structure of the SQL statement, for instance by adding a dynamic constraint in the WHERE clause. Do not use this requirement to justify concatenating user input to create a query string. Prevent SQL injection attacks where user input must affect command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values. Tips: 1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form parameterized statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

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

3. Fortify RTA adds protection against this category.

MigrateAllergiesChangeSet.java, line 164 (SQL Injection) Fortify Priority: Critical Folder Critical Kingdom: Input Validation and Representation Abstract: On line 164 of MigrateAllergiesChangeSet.java, the method

getConceptByGlobalProperty() invokes a SQL query built using input coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands. Source: MigrateAllergiesChangeSet.java:160 java.sql.Statement.executeQuery() 158 JdbcConnection connection = (JdbcConnection) database.getConnection(); 159 Statement stmt = connection.createStatement();

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 37 of 40

**Fortify Security Report**

160 ResultSet rs = stmt.executeQuery("SELECT property\_value FROM global\_property WHERE

property = '" + globalPropertyName + "'"); 161 if (rs.next()) { 162 String uuid = rs.getString("property\_value"); Sink: MigrateAllergiesChangeSet.java:164 java.sql.Statement.executeQuery() 162 String uuid = rs.getString("property\_value"); 163 164 rs = stmt.executeQuery("SELECT concept\_id FROM concept WHERE uuid = '" + uuid + "'"); 165 if (rs.next()) { 166 return rs.getInt("concept\_id");

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 38 of 40

**Fortify Security Report**

Issue Count by Category Issues by Category Log Forging 165 Privacy Violation 71 Password Management: Hardcoded Password 24 Password Management: Password in Configuration File 21 Denial of Service: Regular Expression 20 Path Manipulation 15 Race Condition: Singleton Member Field 15 Dynamic Code Evaluation: Unsafe XStream Deserialization 10 Command Injection 4 Key Management: Hardcoded Encryption Key 3 Privacy Violation: Heap Inspection 3 Dynamic Code Evaluation: Unsafe Deserialization 2 Server-Side Request Forgery 1 Server-Side Template Injection 1 SQL Injection 1

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 39 of 40

**Fortify Security Report**

Copyright 2017 Hewlett Packard Enterprise Development, L.P. Page 40 of 40

Issue Breakdown by Analysis Issues by Analysis

<none>

<none>: (356, 100%)