OpenMRS Final Report

Jordan Beichler(jabeichl), Nilay Kapadia(nkapadi), Parin Dawaldi(pddalwad)

Executive Summary:

The following report details an extensive security review of OpenMRS, a collaborative open source project to develop software to support the delivery of health care in developing countries. Our report has been divided up into four different sections. The first section details an ethical hack test plan and execution/vulnerability and dependency analysis for the Open Web Application Security Project's top ten most critical web application security risks.

The second section begins with an investigation of OpenMRS' password policy, detailing password security characteristics discovered in OpenMRS. We then proceed by creating an abuse/misuse case diagram and cases to demonstrate likely attack scenarios that could occur on the OpenMRS system. Along with abuse and misuse cases, we crafted an attack and defense tree for a targeted OpenMRS module and justify our choices with evidence discovered exploring OpenMRS. We finish off this section by looking at the vulnerability history of OpenMRS, in an attempt to learn more about the attack vectors that have caused issues for the platform in the past.

The third section of this report focuses primarily on logging, static analysis, fuzzing, and requirements. The section begins with an investigation into how OpenMRS logs activity for adding, editing, deleting, and viewing sensitive medical data. We then begin our first attempt at discovering current vulnerabilities within OpenMRS, using a static code analysis tool called Fortify. After static analysis, we explore dynamic testing for bugs using a fuzzer, called ZAP, to test for injection, buffer overflows, XSS, and SQL injection attacks. This testing includes client-side bypassing to uncover any validation checks that may only be present on the clients interface. We finish the section by developing our own security requirements for OpenMRS that we believe are essential to preventing many vulnerabilities within the application.

The first half of fourth section focuses on architectural design and usable security principles. We then play protection poker to help develop new functional requirements that could be added to future OpenMRS releases. Lastly we provide an overview of five different security bugs that we found in OpenMRS, with solutions to mitigate the issues and prevent attacks.

Finally, we utilize the details discovered in our investigation to recommend changes to the current OpenMRS system, through bug reports that are intended mitigate our found vulnerabilities. The discovered bugs should quickly be patched to prevent the potential loss, damage, or theft of patient data and medical records. Failure to fix the issues outlined in this report, could be harmful to those relying on OpenMRS to meet their daily health care needs.

I. OWASP Top 10

1.1 Ethical Hack Test Plan and Execution/Vulnerability and Dependency Analysis

Black Box Test Prerequisites:
If anywhere in the document you are asked to return to the OpenMRS login screen or page. Please navigate to http://localhost:8081/openmrs-standalone/login.htm, replacing the port number with your own.

Most of the following tests require that you have a patient input into the system. If this is the first time running the tests, please perform the following: From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select "Inpatient Ward" as the location for the session and hit the "Login" button. Select on the "Register a patient" box. Input Bob as the given name and Smith as the family name then hit enter. Select Male for gender and hit enter. Input 01 for the day, January for the month, and 1990 for the year then hit enter. Enter 123 Western Blvd as the address, Raleigh as the city, NC as the state, U.S.A. as the country, and 27607 as the postal code then hit enter. Enter 555-555-5555 for the phone number and hit enter. Select Parent for the relationship type and enter Dad as the person's name then hit enter. Select Confirm to confirm the submission.

A1 – Injection

1.1.1 SQL Injection on Find Patient Record Page

Detailed Instruction:
From the OpenMRS login page enter ***admin' --*** as the username, **--** as the password and "Inpatient Ward" as the location for the session and hit the "Login" button. This demonstrates a typical SQL injection tautology into the form to attempt to log into OpenMRS without a password.

Expected Result:
The user is given an error message stating "Invalid username/password. Please try again" and they should not be redirected to the OpenMRS home screen.

Actual Result:
Pass: The user is prompted with the message "Invalid username/password. Please try again" and they are not redirected to the OpenMRS home screen.

1.1.2 URL Command Injection

Detailed Instruction:

From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select “Inpatient Ward” as the location for the session and hit the “Login” button. Once we are redirected to the homepage, paste the following url into the browser replacing the port number with your corresponding port: http://localhost:8081/openmrs-standalone/ws/rest/v1/patient?identifier=brian;ls. Appending “;ls” attempts to inject a command if they don’t sanitize the identifier input or leave it open to command injection.

Expected Result:

Only OpenMRS simple Object XML is displayed. There should be no directory listing.

Actual Result:

Pass: Only OpenMRS simple Object XML is displayed. There is no directory listing. OpenMRS avoids command injection by sanitizing input of parameters so that semicolons do not terminate a shell command, which would allow you to type additional commands.

A2 – Broken Authentication

1.1.3 Destroy Sessions after Logging Out

Detailed Instruction:

From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select “Inpatient Ward” as the location for the session and hit the “Login” button. Once we are redirected to the homepage, press the Logout button in the upper right hand corner. After we see the login page, we try to go back to the previous page to stay in the system.

Expected Result:

We should not go back into the system, and we should stay on the login page because the session is already destroyed.

Actual Result:

Pass: We stay on the login page and are not redirected to the previous page. To prevent this attack OpenMRS is managing sessions with cookies and session ids such that when a logout occurs, they user cannot access any pages that require authentication.

1.1.4 Rotate Session ID on Successful Login

Detailed Instruction:
Using Chrome as your browser, right click a window and hit inspect to open up the developer tools window. Click on the Network tab and in the browser window navigate to the login screen for OpenMRS "http://localhost:8081/openmrs-standalone/login.htm" replacing the port number with your own. In the list of network requests select login.htm. Click the cookies tab and write down the JSESSIONID. From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select "Inpatient Ward" as the location for the session and hit the "Login" button. Once again, within the developer tools select login.htm to view the JSESSIONID. Compare the session id before and after logging in.

Expected Result:
The user should be redirected to the OpenMRS homepage and the two session ids should not be equivalent.

Actual Result:
Fail: The session ids before and after authentication are identical. According to the OWASP Top 10, session ids should be rotated after a successful login.

A3 – Sensitive Data Exposure

1.1.5 PII should not be sent in plain text over HTTP

Detail Instruction:
From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select "Inpatient Ward" as the location for the session and hit the "Login" button. Click Find Patient Record. Right click on the page and hit inspect within Google Chrome. Select the network tab. Enter "Bob" into the search box within OpenMRS. Under the network tab on Chrome developer tools a new listing should appear beginning with "patient?identifier=". Click the headers tab and view the request URL.

Expected Result:
This request could potentially be returning PII so we would expect the request url not to use the http protocol without any encryption. A network encryption protocol such as SSL should be used. The request url should not begin with http://.

Actual Result:
Fail: No network encryption protocol is used, as identified by http:// in the request headers.

1.1.6 Encryption of REST API call URL

Detail Instruction:

After starting the openmrs server, copy paste the following URL in the browser "http://localhost:8081/openmrs-standalone/ws/rest/v1/user"(Change port number accordingly). This URL accesses the REST api calls of the application and would pop-up a login window.

Expected Result:

The attacker must not be able to call the API without proper authentication. This ensures that sensitive data cannot be accessed by an attacker using unencrypted API calls.

Actual Result:

Pass: When the URL is copy-pasted, a login password screen pops which requires a username(admin) and password(Admin123). This indicates that the server is secure and cannot be attacked using URLs. This also raises an important issue, an attacker, if has admin access, will be able to see all the patient information as well as their encrypted values(UUID's) in the database. This could be potentially used to reverse engineer the UUID using cryptanalysis tools if the hashing algorithm is weak.

A4 – XML External Entities (XXE)

1.1.7 CVE-2009-0819

This CVE involves a vulnerability in MySQL that allows remote authenticated users to cause a denial of service crash via an XPath expression using a scalar expression as a FilterExpr with ExtractValue() or UpdateXML(), which triggers an assertion failure. To mitigate this issue MySQL should be updated past version 6.0.10 (preferably to the newest version).
Link:https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0819

1.1.8 CVE-2016-3674

Multiple XXE vulnerabilities in the Dom4J, Dom, JDom, JDom2, Sjsxp, StandardStax and Wstx drivers in XStream, before version 1.4.9 could allow remote attackers to read random files using a well crafted XML document. The vulnerability can be removed by updating to xstream-1.4.9-1.fc24.
Link:https://nvd.nist.gov/vuln/detail/CVE-2016-3674

A5 – Broken Access Control

1.1.9 Object Reference URL exploit

Detailed Instruction

Consider the url(http://localhost:8083/openmrs-standalone/coreapps/clinicianfacing/patient.page?patientId=32821723-54cd-476d-8441-b29df9a23d19). Here port number is 8083, please change the port number to the one used by you on your machine. Copy the URL in the browser without logging in as an administrator.

Expected Result
The url should redirect to the login page thus ensuring that the specific patient id cannot be accessed.

Actual Result
Pass: The url will redirect to the login page thus ensuring that the specific patient id cannot be accessed. This ensures safety of information. OpenMRS uses authentication to ensure that unauthenticated users cannot access system components.

1.1.10 Insecure File access through Directory Traversal

Detailed Instruction
From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select "Inpatient Ward" as the location for the session and hit the "Login" button. Select the "Find Patient Record" button on the home screen. Copy the URL in the browser (probably something like this "http://localhost:8083/openmrs-standalone/coreapps/findpatient/findPatient.page?app=coreapps.findPatient")(adjust port numbers accordingly) and change it to ("http://localhost:8083/openmrs-standalone/coreapps/../findpatient/findPatient.page?app=coreapps.findPatient")

Expected Result
The application should display a blank webpage, a 404 error, an access denied error or redirect to the URL again without displaying any form of error stack.

Actual Result
Fail: This generates an error log in the browser window which contains names of different functions used by the application as well as a complete error log. This information is potentially harmful as it can reveal important function names to an attacker.

A6 – Security Misconfiguration

1.1.11 Hide Error Stack Traces from Users

Detailed Instruction
From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select “Inpatient Ward” as the location for the session and hit the “Login” button. Replace the url with “http://localhost:8081/openmrs-standalone/referenceapplication/*.page” and hit enter to navigate to the url. Here the port number is 8081, please change the port number to the one used by you on your machine. This is an attempt to cause an error revealing a stack trace to the user.

Expected Result
A 404 error should be presented to the user.

Actual Result:
Fail: A root error occurs and the user is presented with a stack trace rather than a 404 error. Stack traces should not be presented to the user.

1.1.12 Directory Listing Disabled

Detailed Instruction
From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select “Inpatient Ward” as the location for the session and hit the “Login” button. In an attempt to view if a directory listing attack is possible, enter the URL “http://localhost:8081/openmrs-standalone/bin”. “bin” is a likely folder name containing files in a typical web application.

Expected Result
A 404 error should be presented to the user.

Actual Result:
Pass: A directory listing of bin is not presented to the user. OpenMRS may have mitigated this attack by disabling directory listings on their web server, or bin may have an inaccurate guess for a directory name.

A7 – Cross-site Scripting (XSS)

1.1.13 Cross-site Scripting possibility in name field of patient

Detailed Instruction
From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select “Inpatient Ward” as the location for the session and hit the “Login” button. Navigate to the “Register a patient” button on the main menu. When registering the patient, enter the following string as the name “<script>alert(“1”)</script>”. **(DO NOT COPY PASTE THE**

STRING DIRECTLY, and do not include outside quotes). Enter Johnson as the family name then hit enter. Select Male for gender and hit enter. Input 02 for the day, February for the month, and 1996 for the year then hit enter. Enter 321 Western Blvd as the address, Raleigh as the city, NC as the state, U.S.A. as the country, and 27607 as the postal code then hit enter. Enter 123-456-7890 for the phone number and hit enter. Select Parent for the relationship type and enter Dad as the person's name then hit enter. Select Confirm to confirm the submission. Once the patient is registered return to the OpenMRS home screen by clicking the OpenMRS logo in the top left. Then lookup the patient using the "Find Patient Record" button on the main page and start typing "<script" on the search bar until an entry is returned with a blank name.

Expected Result

The application should not let you enter "<script>alert("1")</script>" as the name and should return an error message to the user. If the application does not return an error when creating the name, when you go to find a patient record "<script>alert("1")</script>" should be displayed as the name and there should not be anything removed from the text.

Actual Result

Fail: The application allows the user to register their name as "<script>alert("1")</script>" and the script tags are removed when searching for the name. The application should not be register a name of the person with such random characters such as "<, (,), "", and should display an error message to the user. This did not run the script but it could potentially generate a false record. Moreover, the blank result indicates that the input was not taken as a complete string and did not display "<script>alert("1")</script>" in the name field, instead it displayed a blank.

1.1.14 Cross-site Scripting crashes search functionality

Detailed Instruction

From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select "Inpatient Ward" as the location for the session. Navigate to the "Register a patient" button on the main menu. When registering the patient, enter the following string as the name "<script>alert(\"1");</script>"**(DO NOT COPY PASTE THE STRING DIRECTLY, and do not include outside quotes)**. Enter Brian as the family name then hit enter. Select Male for gender and hit enter. Input 03 for the day, March for the month, and 1980 for the year then hit enter. Enter 333 Western Blvd as the address, Raleigh as the city, NC as the state, U.S.A. as the country, and 27607 as the postal code then hit enter. Enter 123-123-1230 for the phone number and hit enter. Select Parent for the relationship type and enter Mom as the person's name then hit enter. Select Confirm to confirm the submission. Once the patient is registered return to the OpenMRS home screen by clicking the OpenMRS logo in the top left. Then lookup the patient using the "Find Patient Record" button on the main page. Try searching for "Brian" (this is an existing patient name).

Expected Result
When searching for Brian there should be a user with the first name "<script>alert(\"1");</script>", or the system should have posted an error message to the user when they try and set this value as the name.

Actual Result
Fail: This is a catastrophic failure and potentially a major bug. The search functionality will not work for any patient(it crashes). Starting and stopping the server will not work and a fresh-copy of openmrs standalone would be needed or the name above would have to be explicitly deleted using a sql command. The application should not register such inputs as they may be able to run malicious scripts. When special characters such as '<','>','\','/',etc. are entered, the application should reject the input with an error message.

A8 – Insecure Deserialization

1.1.15 CVE-2017-7525
A flaw in deserialization was discovered in the jackson-databind for version below 2.6.7.1, 2.7.9.1 and 2.8.9 which could allow unauthenticated user to perform code execution by sending malicious input in JSON to the readValue method under ObjectMapper. This was due to improper deserialization methods used in these versions. This vulnerability was identified by NIST as deserialization of Untrusted Data. According to NIST a recent fix (07/18/2018) was applied to the software and is being analysed.
Link:https://nvd.nist.gov/vuln/detail/CVE-2017-7525

1.1.16 CVE-2018-5968
FasterXML jackson-databind, version 28.111 and 2.9.x through 2.9.3 allows unauthenticated remote code execution due to an incomplete fix of CVE-2017-7525 and bypasses a blacklist using two gadgets. As this flaw is directly dependent on CVE-2017-7525, the vulnerability can be resolved only after CVE-2017-7525 has been resolved.
Link:https://nvd.nist.gov/vuln/detail/CVE-2018-5968

A9 – Using Components with Known Vulnerabilities

1.1.17 MySQL Database vulnerability
There was a directory traversal vulnerability (CVE-2004-0407) in MySQL before 3.23.36 which allowed local users to modify random files to gain elevated privileges by creating a DB whose name starts with a '..'. This vulnerability is present in OpenMRS due to the use of an older

version of MySQL DB(3.23.36). This can be easily fixed by updating MySQL to version 3.23.38 or later.
Link to third-party vulnerability: https://nvd.nist.gov/vuln/detail/CVE-2001-0407

1.1.18 MySQL Connector vulnerability

There is a vulnerability in the MySQL Connectors component of Oracle MySQL(5.1.40 or earlier). This vulnerability is difficult to exploit, although in the event of an exploit by an attacker, can cause a takeover of the MySQL Connectors. This vulnerability can be fixed by applying a CPU(Critical Patch Update) by Oracle which was released in April 2017.
Link to third-party vulnerability patch:
http://www.oracle.com/technetwork/security-advisory/cpuapr2017-3236618.html

A10 – Insufficient Logging and Monitoring

1.1.19 Logging Change of Patient Allergy

Detail Instruction:

From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select “Inpatient Ward” as the location for the session. Select “Find Patient Record” and hit the “Login” button. Type “Bob Smith” into the search box and select him from the results. Click the pencil button next to allergies. Click “Add New Allergy”. Click “Codeine” and hit save. Next hit the x in the actions box next to Codeine to delete the allergy. Hit “Yes” to remove the allergy. Return the the OpenMRS homepage. Select “System Administration”. Select “Advanced Administration”. Under Patients click “View Log Entries”. Hit search and look for the most recent logs.

Expected Result:

A log should exist describing the user “admin” who changed the allergies of patient “Bob Smith”.

Actual Result:

Fail: No logs about changing patient information regarding allergies are displayed. Changes to allergies which can affect patient health should be logged to be viewed in the case of doctor misconduct.

1.1.20 Editing Log Files not Allowed

Detail Instruction:

From the OpenMRS homepage enter admin as the username and Admin123 as the password. Select “Inpatient Ward” as the location for the session and hit the “Login” button. Select

“System Administration”. Select “Advanced Administration”. Under Patients click “View Log Entries”. Hit search and to display all of the logs. Search for an option to edit log files.

Expected Result:
No visible option should be present to modify log files.

Actual Result:
Pass: The user is not able to change log files. OpenMRS prevents against log tampering by not allowing users to edit log files. This helps ensure accurate logging.

II. Password, Abuse/misuse cases, Attack trees, Vulnerability History

2.1 Password Policy Strength

1. The minimum allowed password length is 8 characters.
2. There is no maximum password length defined in the application. It usually is constrained by the default limit of the html input field which is 524288 characters. This issue was opened as a bug but hasn’t been resolved yet. The link to it is provided in the reference section below.
3. The password should have one of both upper and lower case characters, moreover, it should have at least one digit and one non-digit.
4. A user can enter a password for a maximum of 7 times. After that he/she would be locked out of their account. This is a default number and can be changed by the administrator of the application.
5. In case a user is locked out of their account, they will have to notify the administrator to change their password. There is no automatic mechanism for this.
6. There is an option which an administrator can specify which, post account creation, allows a user to change their password after their first login. This feature has to be explicitly set while creating a user, by the administrator and is not a default.
7. There is no periodic reuse policy associated with this application although the user has an option to change their password, whenever they wish to, from the login menu.
8. The password cannot match the username of the user.
9. The administrator can also set a custom regular expression which the password has to match before being accepted as a valid entry.

10. If a user is locked out of their account for entering the maximum number of password attempts, they will not be allowed to attempt to login to their account again for a period of 5 minutes.

References:

- [1]The minimum length is specified in the System Administration->Advanced Administration->Settings(under Maintainence)->Security.
- [2]Link to max password issue:https://issues.openmrs.org/browse/TRUNK-5181
- [3,4,6,8,9] Other constraints are specified in the System Administration->Advanced Administration->Settings(under Maintainence)->Security section.
- [5]The "Notify Admin" feature can be seen when forgot password is clicked on the login page.

2.2 Abuse/Misuse Cases

- See Abuse/Misuse case diagram below.

1. *Name*: Make Unnecessary Changes
   Summary: An incompetent admin is unfamiliar with the Manage Accounts module and makes unnecessary changes to privilege levels, account capabilities, and may accidentally add provider or additional user accounts.
   Author: Jordan Beichler
   Date: 10/1/2018
   Basic Path: (step bp0) An user of the OpenMRS system has gained access to the system for a valid reason by another admin of OpenMRS. (step bp1) They have then been granted access to the Manage Accounts Module. (step bp2) The new admin user navigates to the Manage Accounts Module and selects a user. (step bp3) Unfamiliar with what the system capabilities do, they select all of the capabilities for the user and hit save. (step bp4) The updated user logs in and now has access to previously unheld system capabilities, leaving the potential to cause damage.
   Alternative Paths:
   - Ap1: The incompetent admin changes the privilege level from high to full because they don't understand the difference between the two. (changes steps bp3, bp4).
   - Ap2: The incompetent admin adds an additional unnecessary user or provider account to the system. (changes steps bp3, bp4).
   - Ap3: The incompetent admin accidentally removes all capabilities for another admin in the system. The updated admin then logs on and is no longer to perform any functionality within the system. (changes steps bp3, bp4).

   Capture Points:

- *Cp1: All account capability changes are logged and another administrator can undo the incorrect changes (step bp3).*
- *Cp2: IT Security is monitoring the system for suspicious activity and detects numerous unwarranted changes to the system. They can quickly flag the changes and restore to the correct state (step bp3).*
- *Cp3: The admin is not given access to the Manage Accounts module until they are properly trained by another admin (step bp1).*

Extension Points: There are no extension points because "Making Unnecessary Changes" does not have an includes arrow to another element.

Preconditions:

- Pc1: The system has an admin that is capable of granting to admin capabilities for the manage accounts module to another incompetent admin.
- Pc2: The system allows admins with access to the manage accounts module to make changes to user privileges, system capabilities, add/retire/restore user and provider accounts.
- Pc3: The incompetent admin is allowed to login over the network.

Assumptions:

- As1: The operator has not been properly trained on the Manage Accounts module and is unaware how the system works (all paths).
- As2: The operator does not understand the definitions for the basic system capabilities or what access they grant (basic path, ap3).

Worse Case Threat (postcondition): The incompetent admin reduces the privileges for every other account on the system (including themself lastly), leaving no one capable of making changes to the system. Without any admin accounts, OpenMRS is not able to be restored to the correct state because no one has the correct privileges.

Capture Guarantee (postcondition): Cp3, if the incompetent admin is given proper training, they should never accidentally make harmful and unintended changes to the system.

Related Business Rules:

- Br1: Only administrators of OpenMRS are allowed to grant privileges to other users and update system capabilities for different users.

Stakeholders and Threats:

- Sh1: OpenMRS users (doctors, nurses, etc…):
    - Users could lose access to functionality needed to enter patient data, create visits, and perform their daily needed tasks.
    - With unnecessary elevated privileges users could perform actions that have unintended harmful consequences to the system.
    - Users could have their accounts retired, preventing them from performing any work

- Sh2: Hospital using OpenMRS
    - Hospital could be crippled by not being able to view patient records, diagnoses, appointments, etc.
    - Loss of confidence if security issue becomes publicized
    - Loss of revenue from not being able to treat patients
- Sh3: Patients of Hospital using OpenMRS
    - Loss of privacy if unauthorized user accounts are granted access to patient data that they should not have access to.
    - Loss of the ability to receive treatment if doctors cannot view patient information.

Potential Misuser Profile: Highly incompetent employee who is promoted to an admin without any training or guidance.
Scope: Entire health care system and business using the system.
Abstraction Level: Mis-user goal
Precision Level: Focused

2. *Name*: Steal or Leak Account Information
   Summary: A rogue admin may gain access to the Manage accounts section by obtaining/hacking the administrator's credentials and may make unnecessary changes or steal account information.
   Author: Nilay Kapadia
   Date: 10/02/2018
   Basic Path: (step bp0) A rogue entity has gained access to the system for an invalid reason by stealing/hacking another administrator of OpenMRS. (step bp1) They have obtained access to the Manage Accounts Module. (step bp2) The rogue admin navigates to the Manage Accounts module and obtains data about all the users in the system. (step bp3) The rogue admin may create a new user and revoke privileges of other administrators of OpenMRS thus performing a hostile takeover or elevate privileges of other users thereby further aggravating the situation. (step bp4) The rogue admin may then go ahead and release/steal all information associated with the users of the application
   Alternative Paths:
    - Ap1: The rogue admin creates a dummy user with full privileges thereby having full access to the system without anyone noticing.(changes step bp3).
    - Ap2: The rogue admin downloads all the user data in the application and misuses this information thereby destroying privacy of users. (changes step bp4).
    - Ap3: The rogue admin removes all administrators of the OpenMRS system or revokes their privileges thereby completely taking over the entire system.

   Capture Points:

- Cp1: All account capabilities are logged and another administrator can remove the dummy user by cross-checking with his/her own records.
- Cp2: Safeguards and monitoring systems alert the IT team when large amounts of user data is being queried and limits the number of queries per user.
- Cp3: Implementing safeguard mechanisms which avoid a situation where all power/access rests in one user.

Extension Points:There are no extension points because "Steal/Leak Account Information" does not have an includes arrow to another element.

Preconditions:

- Pc1: The system has an admin which is capable/has malicious intent against the organization.
- Pc2: The system allows admins with access to the manage accounts module to access personal user data.
- Pc3: A situation could arise that could lead to presence of a single admin or the admin has power to remove all other admins.

Assumptions:

- As1: A system administrator could have malicious/harmful intent/grudge towards the organisation as a whole
- As2: A system administrator is a vulnerable entity and can be compromised by an attacker.

Worse Case Threat (postcondition): The rogue admin revokes the privileges or worse completely deletes all other admin users leaving no one(except him/herself) capable of making changes to the system. This allows the rogue admin complete access to user data and thus compromises the privacy as well as security of the users and the organization as a whole.

Capture Guarantee (postcondition): Preventing the existence of a singular admin would prevent any form of hostile takeover by a rogue admin and can thus be overridden or completely incapable of taking over the application.

Related Business Rules:

Br1: All system administrators are vetted and proper background checks are performed to prevent any rogue agents.

Stakeholders and Threats:

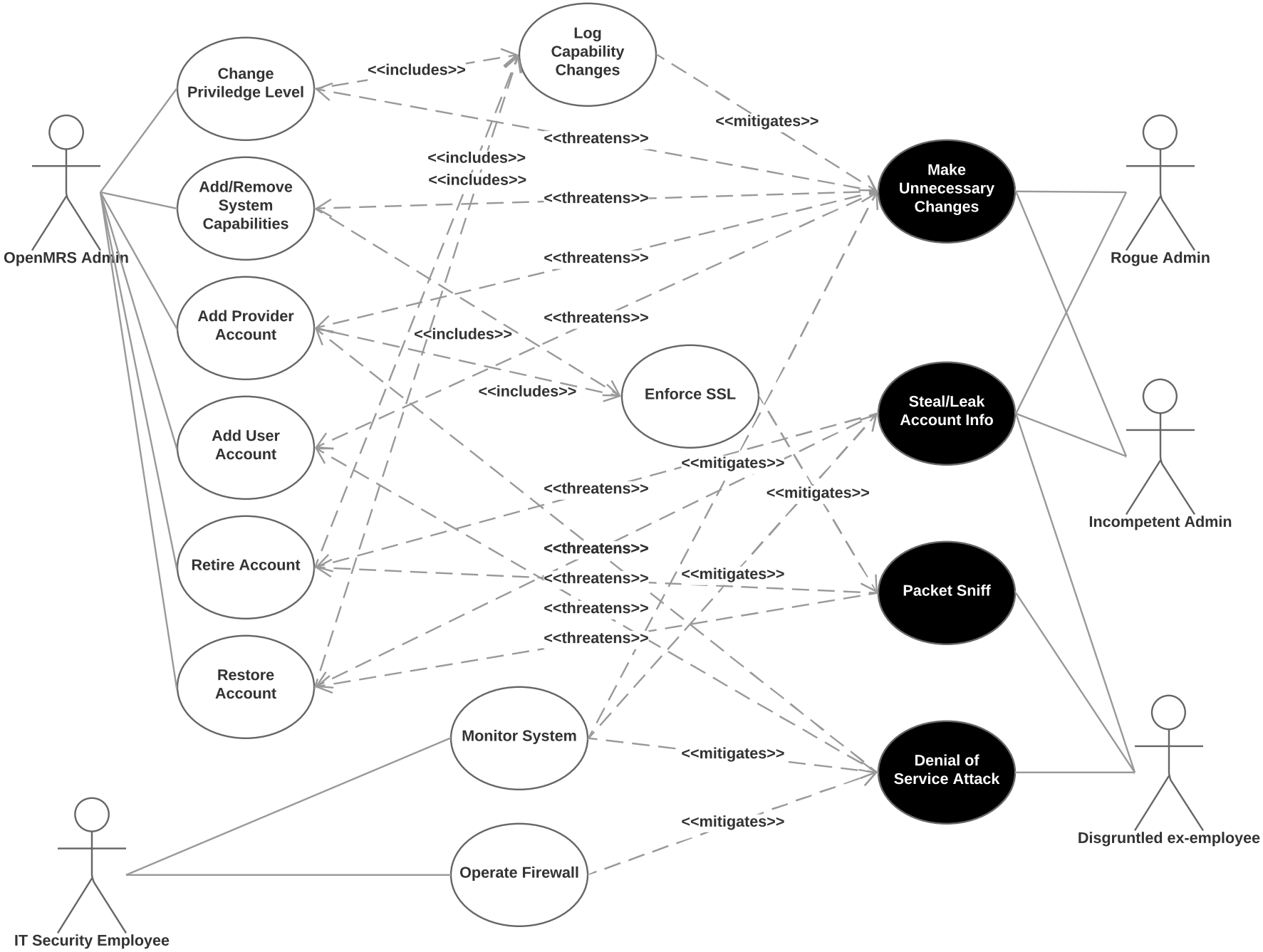
- Sh1: OpenMRS users:
    - ❏ Users could lose access to functionality needed to enter patient data, create visits, and perform their daily needed tasks.
    - ❏ Users could have their accounts retired, preventing them from performing any work
    - ❏ Users could have their private information(addresses, birthdays,SSN's…) leaked or sold on the black market.

- Sh2: Hospital using OpenMRS:
    - ❏ Hospital could be crippled by not being able to view patient records, diagnoses, appointments, etc.
    - ❏ Loss of confidence if security issue becomes publicized.
    - ❏ Loss of revenue from not being able to treat patients.
- Sh3: Patients of the Hospital using OpenMRS:
    - ❏ Loss of privacy if unauthorized user accounts are granted access to patient data that they should not have access to.
    - ❏ Loss of the ability to receive treatment if doctors cannot view patient information.

Potential Misuser Profile: A rogue admin or an attacker masquerading as an admin.
Scope:Entire health care system and business using the system.
Abstraction Level:Abuser goal
Precision Level:Focused

MANAGE ACCOUNTS ABUSE/MISUSE CASE DIAGRAM

jabeichl/nkapadi | November 26, 2018



2.3 Attack and Defense Trees

- See attack and defense trees below.

1. Attack Tree
    a. Abuser's Goal: Delete user data. The abuser would be motivated to destroy all data to cripple the system through a coordinated cyber attack or could be motivated to steal information and sell it in the black market.
    b. The attack tree uses a consolidated cost structure which combines all metrics of an attack tree such as probability, impact, etc. into one metric for ease of understanding. The table below illustrates what cost value corresponds to what value of the metric.
    c.

Cost Metric	0-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80	80-90	90-100
Probability	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	<0.1
Cost	<=$ 10k	$10k -$20 k	$20k -$30 k	$30k -$40 k	$40k -$50 k	$50k -$60 k	$60k -$70 k	$70k -$80 k	$80k -$90 k	>=$ 100k
Impact	<1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10

    d. Justification of each node in the tree:
        - Destroy Physical storage devices or data centers: This is a highly unlikely scenario. Usually, data centers are heavily guarded buildings with multiple redundant copies spread over the globe. The cost associated with this node is 95 because breaking into a data center and destroying data is almost impossible due to high-levels of security and geo-replication and no special tools are required.
        - Masquerade as a legitimate user: This is a highly unlikely scenario as an admin(if not corrupt) can easily identify a fake user from a real one. The cost associated with this node is 90 as spoofing user credentials is very difficult due to strong encryption techniques and anonymity of admins and no special tools are required.
        - Bribe and physical threats: It is a very organization dependent scenario. In case of a larger organization, it is almost impossible to bribe or threaten an

admin due to high security but admins of smaller organizations are highly vulnerable. The cost associated with this node is 75 because it is difficult to identify admins as there is usually more than one admin and their identities are kept anonymous and no special tools are required.

- Obtain password and username by guessing: This is also a very unlikely scenario as admin passwords are usually highly complex and stored in a hashed manner to avoid any compromise in case a database is breached. The only method to obtain the password would be using brute force and would require immense computing resources. The cost associated with this node is 90 because unless a very weak encryption algorithm is used or a weak password is used, it is impossible to crack a password using brute force with current computing power and no special tools are required.
- Get cookies through access to admin's personal machine: There is a small likelihood of this occurring in a real-life scenario. The most obvious case would be if an admin leaves his or her workstation without following security measures such as locking the machine using the software as well as physically(maybe using a kensington lock). The cost associated with this node is 70 because it is difficult to obtain private information from cookies due to them being encrypted as well as they may not be present on a local machine and no special tools are required.
- Use of malware to obtain cookies: This is a very likely scenario in case certain compliance measures are not followed such as restricting certain websites or emails from outside the organizations. In extreme cases, admin's are expected to use air-gapped machines(machines that have been connected to the internet or any network). The cost associated with this node is 55 as it is difficult to develop a malware program that can bypass modern anti-virus protections although payload delivery is extremely easy, this can be done using a flash-drive or a malicious email and special malware programs such as trojans, worms, etc. are required to be placed on the user's machine.
- Snooping on admin's unencrypted network: This is a major flaw which is exploited by many attackers. They use packet sniffing softwares such as Wireshark to listen on any network communication from a source(in this case the admin's network). In case the data is unencrypted, it is extremely easy to obtain access tokens of any login attempt to the main server. The cost associated with this node is 30 because it is very easy to use packet sniffers as they are openly available although finding an unencrypted network is much harder and a packet sniffer such as wireshark is required to obtain packet data.

2. Defense Tree

a. Defender's Goal: The goal of the defender is to protect all forms of attack scenarios, shown in the attack tree. The main aim would be to prevent any form of attack which could result in deletion of all user data.

b. The defense tree uses a consolidated cost structure which combines all metrics of a defense tree such as probability of success and cost into one metric for ease of understanding. The table below illustrates what cost value corresponds to what value of the metric.

c.

Cost Metric	0-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80	80-90	90-100
Probability of success	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	>0.9
Cost	<=$ 10k	$10k -$20 k	$20k -$30 k	$30k -$40 k	$40k -$50 k	$50k -$60 k	$60k -$70 k	$70k -$80 k	$80k -$90 k	>=$ 100k

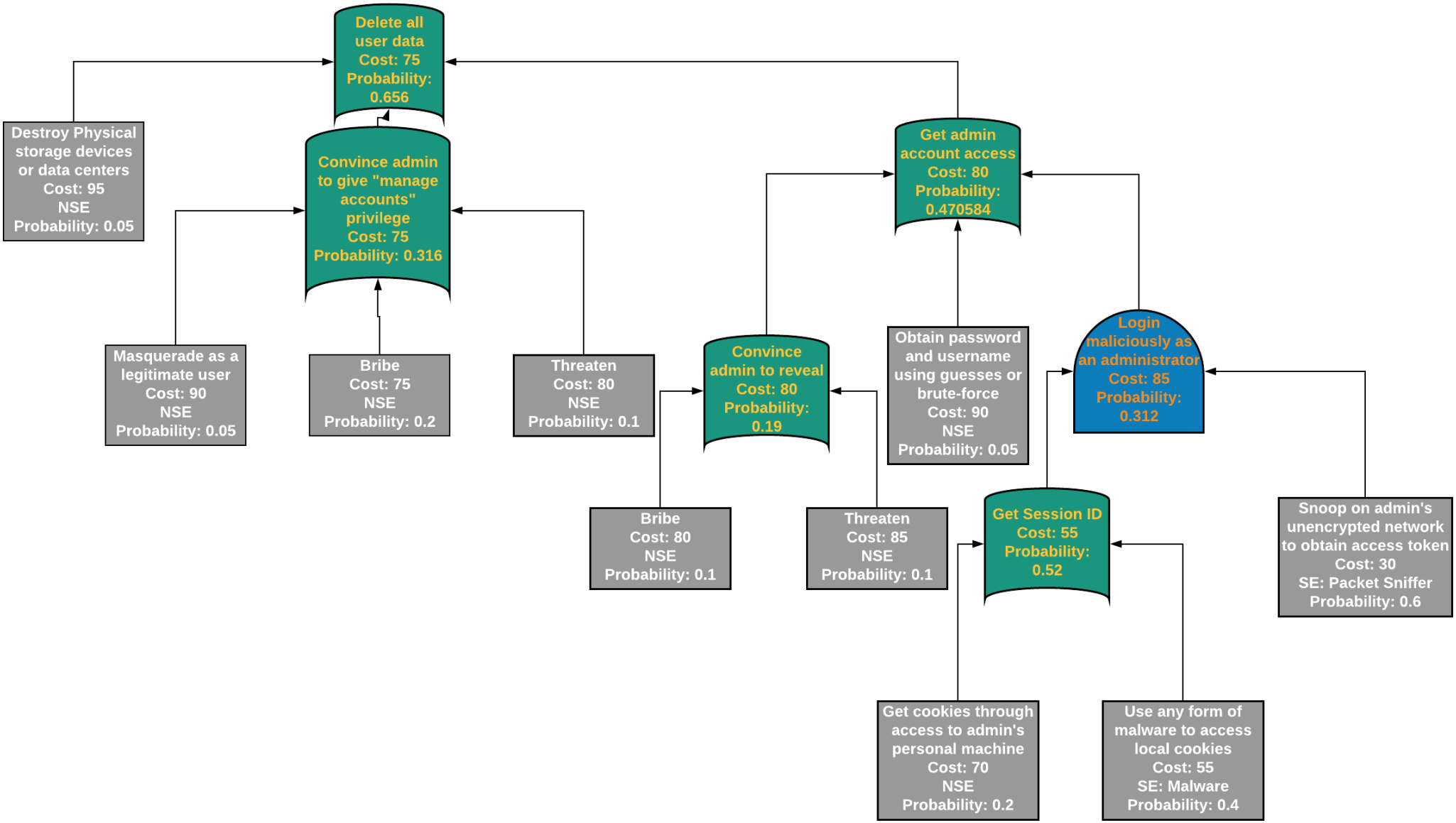
d. Justification of each node in the tree:

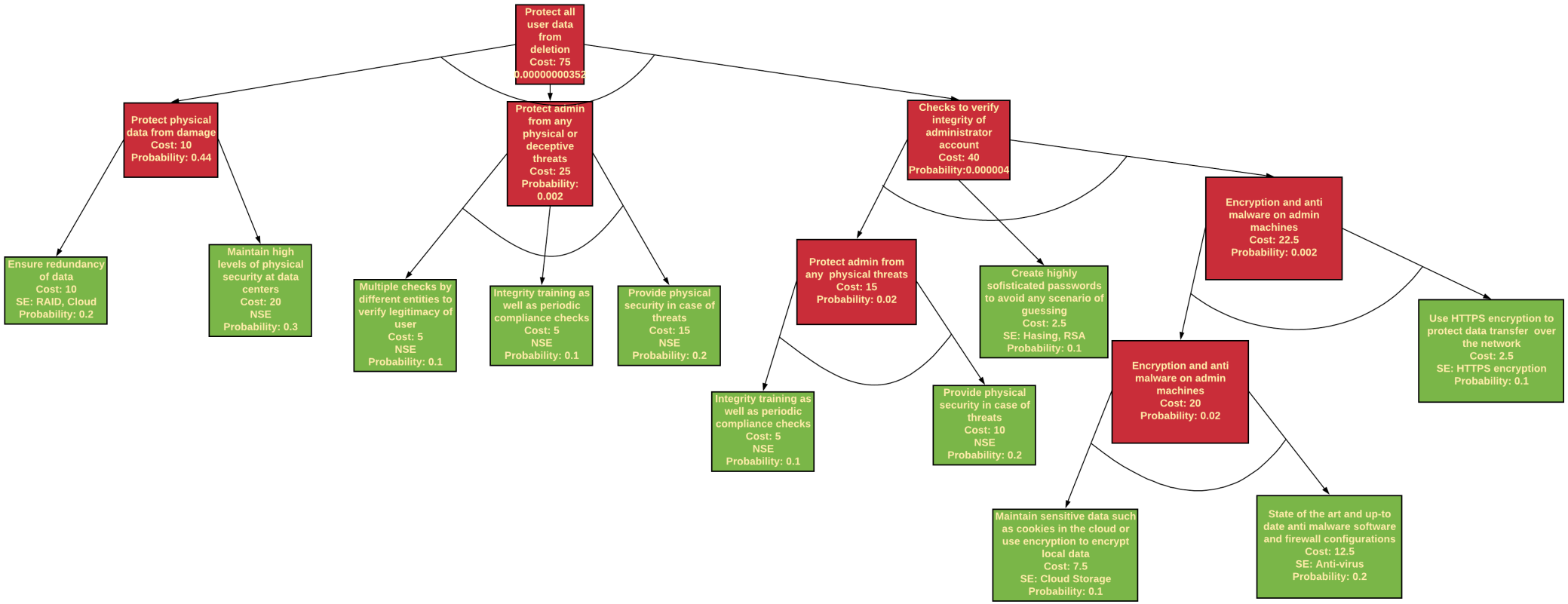
- Ensure redundancy of data: This is a low cost high impact data defense strategy and is widely used all over the world. In case of any natural disaster or in this case a physical attack by the attacker it would ensure total safety, integrity and availability of data. The cost associated with this node is 10 as nowadays, it is extremely easy to ensure redundancy of data using replication and the the special tools required are RAID and Cloud facilities. These, allow recovery during failure and redundancy through replication.
- Maintain high levels of physical security: This is a high cost defense strategy as it would involve large infrastructure costs and it clearly outweighs the benefits it generates. The cost associated with this node is 20 because it is relatively expensive to maintain physical security at data centers and no special tools are required.
- Multiple checks to ensure legitimacy: This is a very low-cost and efficient strategy as it can be easily employed in many ways. The cost associated

with this node is 5 because it is very easy to ensure legitimacy of users nowadays with the advent of blockchain algorithms and no special tools are required.

- Integrity training and periodic checks: This is also a very low-cost and effective strategy as it would avoid any integrity violations by admins(such as bribes) and periodic checks would dissuade admins from engaging in unlawful activity. The cost associated with this node is 5 as it is relatively inexpensive to create training programs to train employees on integrity values and no special tools are required.
- Provide physical security for admins: This is a viable and moderately expensive strategy. The cost associated with this node is 15, this is due to the fact that human security is very difficult to maintain as humans are the weakest link and have many vulnerabilities and no special tools are required.
- Creating highly sophisticated passwords to avoid guessing: This strategy is low-cost and easy to implement due to many hashing algorithms. This clearly dissuades any form of brute-force/guessing attacks. The cost associated with this node is 2.5 as it is very easy to use hashing algorithms as they are openly available and the special tools required are hashing algorithms like RSA.
- Maintain sensitive access data(cookies) in the cloud: This is a low to moderate cost strategy that would prevent any cookies or session ids leaking to the attacker. The cost associated with this node is 7.5 as setting up a cloud infrastructure as well as using hashing algorithms combined have a reasonable cost and the special tools required are cloud storage/encryption to keep sensitive data safe from attackers.
- State of the art and up-to-date anti-malware: This is a moderate to high cost strategy as licenses to these softwares are expensive and they do not always take into account the latest vulnerabilities as they could take time to add patches. The cost associated with this node is 12.5 as current large scale licenses for anti-virus software are considerably expensive and the special tools required would be an anti-virus software to protect devices against malware.
- Use HTTPS encryption to protect data over the network: This is a simple low-cost strategy which has almost become a compliance requirement in many major organizations to avoid any data being leaked due to lack of proper encryption. The cost associated with this node is 2.5 as using HTTPS encryption has become a norm and is relatively inexpensive and

the special tools required would be a simple HTTPS encryption algorithm which uses SSL encryption.


Delete all user data
Cost: 75
Probability: 0.656
Destroy Physical storage devices or data centers
Cost: 95
NSE
Probability: 0.05
Convince admin to give "manage accounts" privilege
Cost: 75
Probability: 0.316
Get admin account access
Cost: 80
Probability: 0.470584
Masquerade as a legitimate user
Cost: 90
NSE
Probability: 0.05
Bribe
Cost: 75
NSE
Probability: 0.2
Threaten
Cost: 80
NSE
Probability: 0.1
Convince admin to reveal
Cost: 80
Probability: 0.19
Obtain password and username using guesses or brute-force
Cost: 90
NSE
Probability: 0.05
Login maliciously as an administrator
Cost: 85
Probability: 0.312
Bribe
Cost: 80
NSE
Probability: 0.1
Threaten
Cost: 85
NSE
Probability: 0.1
Get Session ID
Cost: 55
Probability: 0.52
Snoop on admin's unencrypted network to obtain access token
Cost: 30
SE: Packet Sniffer
Probability: 0.6
Get cookies through access to admin's personal machine
Cost: 70
NSE
Probability: 0.2
Use any form of malware to access local cookies
Cost: 55
SE: Malware
Probability: 0.4


Protect all user data from deletion
Cost: 75
0.00000000352
Protect physical data from damage
Cost: 10
Probability: 0.44
Protect admin from any physical or deceptive threats
Cost: 25
Probability: 0.002
Checks to verify integrity of administrator account
Cost: 40
Probability:0.000004
Ensure redundancy of data
Cost: 10
SE: RAID, Cloud
Probability: 0.2
Maintain high levels of physical security at data centers
Cost: 20
NSE
Probability: 0.3
Multiple checks by different entities to verify legitimacy of user
Cost: 5
NSE
Probability: 0.1
Integrity training as well as periodic compliance checks
Cost: 5
NSE
Probability: 0.1
Provide physical security in case of threats
Cost: 15
NSE
Probability: 0.2
Protect admin from any physical threats
Cost: 15
Probability: 0.02
Create highly sofisticated passwords to avoid any scenario of guessing
Cost: 2.5
SE: Hasing, RSA
Probability: 0.1
Encryption and anti malware on admin machines
Cost: 22.5
Probability: 0.002
Integrity training as well as periodic compliance checks
Cost: 5
NSE
Probability: 0.1
Provide physical security in case of threats
Cost: 10
NSE
Probability: 0.2
Encryption and anti malware on admin machines
Cost: 20
Probability: 0.02
Use HTTPS encryption to protect data transfer over the network
Cost: 2.5
SE: HTTPS encryption
Probability: 0.1
Maintain sensitive data such as cookies in the cloud or use encryption to encrypt local data
Cost: 7.5
SE: Cloud Storage
Probability: 0.1
State of the art and up-to date anti malware software and firewall configurations
Cost: 12.5
SE: Anti-virus
Probability: 0.2

2.4 OpenMRS Vulnerability History

2.4.1 CVE-2017-12796:

a. *Description*: Users were not authenticated by openMRS when XML inputs were deserialized into ReportSchema objects. Remote code execution could be performed with carefully crafted XML payloads that were not serialized correctly.
b. *Discovery*: This vulnerability was likely discovered by sending a malformed XML ReportSchema object to OpenMRS that either returned a stack trace or performed some interesting behavior. The CVE did not explicitly describe how the vulnerability was first discovered though.
c. *Commonly-Occurring*: Deserialization is a less commonly discovered vulnerability. In the case of OpenMRS it was isolated to the ReportSchema object and was not widespread throughout the entire application. Users are also typically authenticated before being allowed to make privileged API requests making it less common as well.
d. *Fix*: To prevent this from happening users performing privileged API calls should first be authenticated. Object inputs being passed from the client should also be sanitized and validated to match the expected objects that are passed back to the user.
e. *Common Format*: This vulnerability follows the format of CWE - 502 which involves the deserialization of untrusted data.

2.4.2 CVE-2014-8073:

f. *Description*: OpenMRS was vulnerable to Cross-site request forgery attacks that allowed hackers to piggyback off of an administrators authentication and make requests involving adding and saving users.
g. *Discovery*: This vulnerability was likely discovered by crafting a malicious url that performed an administrative OpenMRS user action and the URL placed on another non-OpenMRS web page that performed the OpenMRS action when clicked. The CVE did not explicitly describe how the vulnerability was first discovered though.
h. *Commonly-Occurring*: Cross-site request forgery attacks are fairly common and fairly simple to implement if the vulnerability exists. According to the CVE this attack type was restricted to adding in saving users, mentioned in the description.
i. *Fix*: To prevent this from happening, OpenMRS should check the request headers and verify that the request is occuring from the same origin. Another better solution would be to use CSRF tokens which are randomized and known to the

server when making requests, but would be extremely difficult for an attacker to guess. If the token is not correct, the action would not be performed.

j. *Common Format*: This vulnerability follows the format of CWE - 352 which involves cross-site request forgery attacks.

2.4.3 CVE-2014-8072:

k. *Description*: An OpenMRS user can acquire read admin access by simply navigating to to /admin directly. This link is not available within the application to non administrative users and must be typed into the browser directly.

l. *Discovery*: This vulnerability was discovered by a non-admin user accessing admin read privileges by directly navigating to the admin page by appending '/admin'.

m. *Commonly-Occurring*: This attack is very common for systems that do not strictly enforce permissions, privileges, and access control. Within OpenMRS this action was not directed at a single page but for all sub-admin pages, making it fairly common across the application.

n. *Fix*: To prevent this from happening, OpenMRS should validate that any user attempting to access admin controls is an admin user of the system. In this case a request to /admin should be met with a check of the requesting user's permissions.

o. *Common Format*: This vulnerability follows the format of CWE - 264 which involves permissions, privileges, and access controls.

2.4.4 CVE-2014-8071:

p. *Description*: OpenMRS is vulnerable to numerous cross-site scripting vulnerabilities. These include stored and reflected attacks that can occur within many of the fields when registering a patient, adding an allergy, logging in, etc.

q. *Discovery*: This attack was discovered by implementing XSS attacks on the register patient page, allergy page, login page, and other pages throughout OpenMRS. Users crafted special input html scripts that were then either reflected or stored after being input into these vulnerable fields.

r. *Commonly-Occurring*: This attack is an extremely common type of attack and is widespread throughout OpenMRS. The reason this attack is very common is because of the difficulty in preventing all possible XSS attacks. Malicious users can encode symbols, use different case, use different tags, and more to circumvent XSS protection mechanisms.

s. *Fix*: XSS should be prevented using a validation library due to all of the possible different ways to cross site script. A few rules to follow include never inserting untrusted data except in allowed locations, html escape before inserting untrusted

data, attribute escape before inserting untrusted data, javascript escape before inserting untrusted data, css escape before inserting untrusted data, etc.

t. *Common Format*: This vulnerability follows the format of CWE - 264 which involves improper neutralization of input during web page generation (XSS).

2.4.5 CVE-2017-7990:

u. *Description*: OpenMRS is vulnerable to a cross site request forgery attack with XSS where admin access can be gained to insert JavaScript code into a name field when managing reports.

v. *Discovery*: This attack was likely discovered by discovering that CSRF was capable for OpenMRS and chaining that attack with a XSS attack to insert code using JavaScript through a maliciously crafted link. The CVE did not explicitly describe how the vulnerability was first discovered though.

w. *Commonly-Occurring*: This attack is less common because it is chained involving a CSRF attack and XSS attack. However, these attacks on their own are fairly straightforward to implement, commonly occuring, and somewhat difficult to prevent against.

x. *Fix*: XSS should be prevented using a validation library due to all of the possible different ways to cross site script. A few rules to follow include never inserting untrusted data except in allowed locations, html escape before inserting untrusted data, attribute escape before inserting untrusted data, javascript escape before inserting untrusted data, css escape before inserting untrusted data, etc. CSRF tokens should be used which are randomized and known to the server when making requests, but would be extremely difficult for an attacker to guess. If the token is not correct, the action would not be performed.

y. *Common Format*: This vulnerability follows the format of CWE - 264 which involves improper neutralization of input during web page generation (XSS) and CWE - 352 which involves cross-site request forgery attacks.

2.4.6 CVE-2018-16521:

z. *Description*: OpenMRS is vulnerable to an XXE attack in its HTML form entry that leaves it vulnerable to documents containing URIs that point to documents that were not intended to be accessed and documents were therefore incorrectly embedded.

aa. *Discovery*: I was unable to find any information regarding the origins of this attack.

bb. *Commonly-Occurring*: This attack is common for applications that parse XML input and are weakly configured. Since not all applications rely on the parsing of XML, in practice this attack strategy is less commonly implemented.

cc. *Fix*: XXE should be prevented by disabling external entities completely. Most parsers have the ability to disable document type definitions (external entities).
dd. *Common Format*: This vulnerability follows the format of CWE - 611 which involves improper restriction of XML external entity reference (XXE).

CVE References:

- [1,2,3,4]https://www.cvedetails.com/vulnerability-list/vendor_id-14221/product_id-29315/Openmrs-Openmrs.html
- [1] https://www.cvedetails.com/cve/CVE-2017-12796/
- [1] https://cwe.mitre.org/data/definitions/502.html
- [2] https://www.cvedetails.com/cve/CVE-2014-8073/
- [2, 5] https://cwe.mitre.org/data/definitions/352.html
- [3] https://www.cvedetails.com/cve/CVE-2014-8072/
- [3, 5] https://cwe.mitre.org/data/definitions/264.html
- [4] https://www.cvedetails.com/cve/CVE-2014-8071/
- [4] https://cwe.mitre.org/data/definitions/79.html
- [4]https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#RULE_.230_-_Never_Insert_Untrusted_Data_Except_in_Allowed_Locations
- [5,6]https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=openMRS
- [5]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7990
- [6]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16521
- [6]https://cwe.mitre.org/data/definitions/611.html

III. Audit/logging, Static Analysis, Fuzzing, and Requirements

3.1 Audit/Logging Implementation

Module in reference: Patient module

Note: The following test cases are performed by an admin and admin privileges are required to view logs. Please login as admin(admin,Admin123) before executing any of the following test cases.

3.1.1 Add a new patient

TestCaseID=tid_np

TestCase Type=Add/Register a new patient

Steps: Post patient registration, check the logs. The logs can be found in System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should log all the new fields which were added by the patient as well keep all this information private in the log files.

Actual Log Result: The logger correctly logs all the added fields in xml format as well as keep the information for each field private/secret. The log tags the log-entry as INFO.

Logging Adequacy and Test Case Description: In the test case above, the OpenMRS logger adequately logs the required information. The test case aims to highlight the logs generated by OpenMRS on adding/registering a new patient.

Screenshot:

```
INFO - LoggingAdvice.invoke(115) |2018-10-29 08:10:06,430| In method AdministrationService.saveGlobalProperty. Arguments: GlobalProperty=property:
layout.address.format value: <org.openmrs.layout.address.AddressTemplate> <nameMappings class="properties"> <property name="postalCode"
value="Location.postalCode"/> <property name="address2" value="Location.address2"/> <property name="address1" value="Location.address1"/> <property
name="country" value="Location.country"/> <property name="stateProvince" value="Location.stateProvince"/> <property name="cityVillage"
value="Location.cityVillage"/> </nameMappings> <sizeMappings class="properties"> <property name="postalCode" value="10"/> <property name="address2"
value="40"/> <property name="address1" value="40"/> <property name="country" value="10"/> <property name="stateProvince" value="10"/> <property
name="cityVillage" value="10"/> </sizeMappings> <lineByLineFormat> <string>address1</string> <string>address2</string> <string>cityVillage stateProvince country
postalCode</string> </lineByLineFormat> </org.openmrs.layout.address.AddressTemplate>,
```

3.1.2 Search and view patient details:

TestCaseID=tid_snv

TesCase Type=View a patient's record

Steps: Post patient creation. Go to the homepage of the admin view->Go to Find a Patient Record->Click on the most recent record that shows up to display its details->Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should log which user viewed which patients record.

Actual Log Result: The logger only shows that a user admin accessed the method save user. It does not show which patient the admin viewed, which it should log.

Screenshot:

```
INFO - LoggingAdvice.invoke(115) |2018-10-29 08:35:40,439| In method UserService.saveUser. Arguments: User=admin,
INFO - LoggingAdvice.invoke(155) |2018-10-29 08:35:40,442| Exiting method saveUser
```

Logging Adequacy and Test Case Description: In the test case above, the OpenMRS logger inadequately logs the required information. In this test case, the logger is expected to log the user which viewed the patient's record.

3.1.3 Delete a patient

TestCaseID=tid_dp

TestCase Type=Delete a patient's record

Steps: Post patient creation. Go to the homepage of the admin view->Go to Find a Patient Record->Click on the most recent record that shows up to display its details-> Click on delete a patient on the right side panel-> Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should log which user deleted which patients record.

Actual Log Result: The logger logs the required details correctly. It first logs who searched for a patient, it then logs that Patient#108 was voided with reason something.

Screenshot:

```
INFO - LoggingAdvice.invoke(115) |2018-10-29 08:43:09,740| In method UserService.saveUser. Arguments: User=admin,
INFO - LoggingAdvice.invoke(155) |2018-10-29 08:43:09,744| Exiting method saveUser
INFO - LoggingAdvice.invoke(115) |2018-10-29 08:45:51,556| In method PatientService.voidPatient. Arguments: Patient=Patient#108, String=something,
INFO - LoggingAdvice.invoke(155) |2018-10-29 08:45:51,637| Exiting method voidPatient
```

Comments: The reason field should be obfuscated in the logs to avoid any privacy issues.

Logging Adequacy and Test Case Description: The logger adequately logs the required information when deleting a patient's record. It is essential to log which user deleted a patient's data to avoid lack of accountability.

3.1.4 Edit the name of a patient:

TestCaseID=tid_epn

TestCase Type = Edit the name of a patient

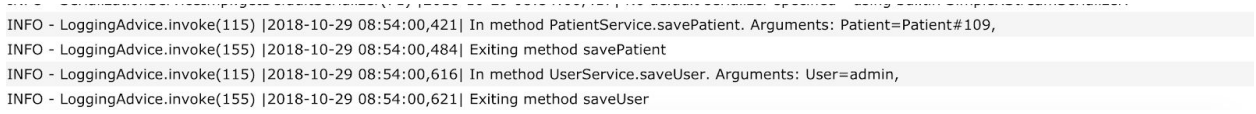
Steps:Post patient creation. Go to the homepage of the admin view->Go to Find a Patient Record->Click on the most recent record that shows up to display its details->Click on the blue

colored “Edit” text and change the name-> Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should log which user edited which patients record as well as which field was changed.

Actual Log Result: The logger logs all the required details except which field was changed. This needs to be recorded to avoid lack of accountability.

Screenshot:



Logging Adequacy and Test Case Description: The logger does not capture the fields that were changed while making the edit. It is important to know which fields were changed to avoid repudiation.

3.1.5 Add a past doctor’s visit by the patient:

TestCaseID=tid_pv

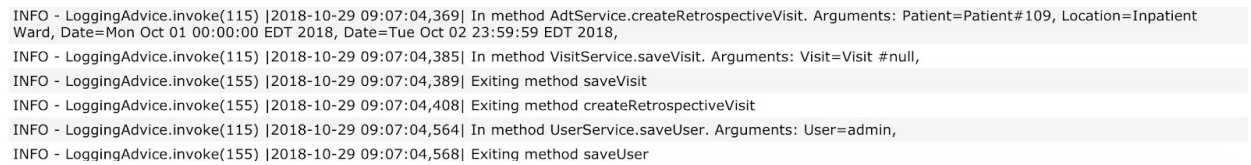
TestCase Type = Add a past doctor’s visit by the patient

Steps:Post patient creation. Go to the homepage of the admin view->Go to Find a Patient Record->Click on the most recent record that shows up to display its details->Click on add a past visit in the panel on right->Select the from and to date of the past visit->Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should log which user added a past visit for which patient as well as the dates of the past visit as well as which part of the hospital the patient visited.

Actual Log Result: The logger correctly logs all the information stated above.

Screenshot:



Logging Adequacy and Test Case Description: The logger adequately logs the required information when adding a past doctor's visit. It is essential to log which user added a past doctor's visit to avoid lack of accountability.

3.1.6 Merge two visits of a given patient:

TestCaseID=tid_mv

TestCase Type = Merge two visits of a given patient

Steps:Post patient creation. Go to the homepage of the admin view->Go to Find a Patient Record->Click on the most recent record that shows up to display its details->Click on Merge Visits-> Select any two visits and merge them->Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should log which user wanted to merge which patients visit as well as the visits that were merged.

Actual Log Result: The logger correctly logs the above expected results. (It is assumed that the Visit information contains details about the patient and is thus not logged).

Screenshot:

```
INFO - LoggingAdvice.invoke(115) |2018-10-29 09:14:17,380| In method VisitService.voidVisit. Arguments: Visit=Visit #509, String=EMR - Merge Patients: merged into visit 510,
INFO - LoggingAdvice.invoke(155) |2018-10-29 09:14:17,393| Exiting method voidVisit
INFO - LoggingAdvice.invoke(115) |2018-10-29 09:14:17,400| In method VisitService.saveVisit. Arguments: Visit=Visit #510,
INFO - LoggingAdvice.invoke(155) |2018-10-29 09:14:17,404| Exiting method saveVisit
```

Logging Adequacy and Test Case Description: The logger adequately logs the required information when merging two patients.

3.1.7 Change Location of Patient:

TestCaseID = tid_loc

TestCase Type = Change Location of Patient

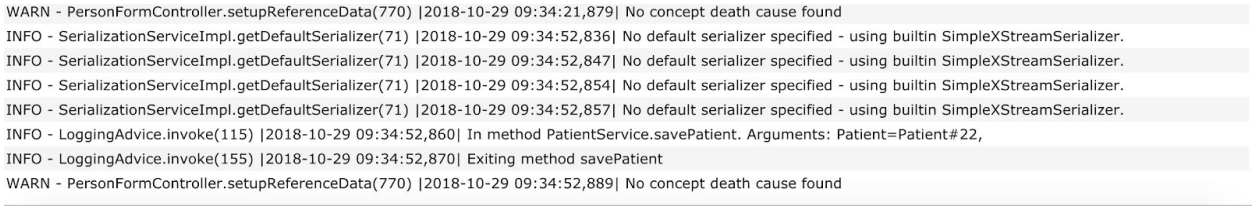
Steps: Post Patient Creation. Go to System Administration on the HomePage-> Go to Advanced Administration->Manage Patients under Patients->Search for a patient and select the corresponding row(say Mary Perez)->Change the location to Amani Hospital->Click on Save

Patient->Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should log which user changed the location of which patient and what was the previous as well as the new location.

Actual Log Result: The logger only logs the user which changed the location and the patient whose location was changed. It is expected of the logger to log the locations in the log for the purposes of keeping the admin accountable in case a patient is transferred to the wrong location in the system.

Screenshot:



Logging Adequacy and Test Case Description: The logger does not log the locations being changed as it is necessary to hold people accountable in case a patient is transferred to a wrong location.

3.1.8 Change properties of Patient Identifier:

TestCaseID=tid_pi

TestCase Type: Change properties of Patient Identifier

Steps:Post Patient Creation. Go to System Administration on the HomePage-> Go to Advanced Administration->Manage Identifier Types->OpenMRSID->Change Uniqueness Behaviour to Unique->Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should record which user changed the OpenMRSID type as well as which property of the ID was changed.

Actual Log Result: The logger only logs changes made by the user("admin") to the OpenMRSID and not the internal changes which were made, in this case uniqueness. The logger

should log this result to avoid accountability issues as well as debugging in case there are issue with the OpenMRSID type.

Screenshot:

```
INFO - LoggingAdvice.invoke(115) |2018-10-29 09:46:12,640| In method PatientService.savePatientIdentifierType. Arguments: PatientIdentifierType=OpenMRS ID,
INFO - LoggingAdvice.invoke(155) |2018-10-29 09:46:12,646| Exiting method savePatientIdentifierType
```

Logging Adequacy and Test Case Description: The logger fails to log the uniqueness field while changing properties of patients. This should be done to avoid accountability issues as well as debugging scenarios in case there is a problem with the OpenMRSID type.

3.1.9 Merge two patient records:

TestCaseID=tid_mpr

TestCase Type: Merge two patient records

Steps: From the Home Screen Go to Data Management->Merge Patient Electronic Records->Search and select the patients using the search by name functionality->Merge the two patients->Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs

Expected Log Result: The logger should record which user initiated the merge as well as which patients were merged.

Actual Log Result: The logger correctly records the above information, moreover it also records the fields which changed during the merge as well as the fields which remained the same.

Screenshot:

```
INFO - SerializationServiceImpl.getDefaultSerializer(71) |2018-10-29 10:02:05,236| No default serializer specified - using builtin SimpleXStreamSerializer.
INFO - SerializationServiceImpl.getDefaultSerializer(71) |2018-10-29 10:02:05,238| No default serializer specified - using builtin SimpleXStreamSerializer.
INFO - LoggingAdvice.invoke(155) |2018-10-29 10:02:05,241| Exiting method savePersonMergeLog
INFO - LoggingAdvice.invoke(115) |2018-10-29 10:02:05,249| In method PatientService.voidPatient. Arguments: Patient=Patient#27, String=Merged with patient #59,
INFO - LoggingAdvice.invoke(155) |2018-10-29 10:02:05,253| Exiting method voidPatient
INFO - LoggingAdvice.invoke(115) |2018-10-29 10:02:05,261| In method PersonService.voidPerson. Arguments: Person=Patient#27, String=The patient corresponding to this person has been voided and Merged with patient #59,
INFO - LoggingAdvice.invoke(155) |2018-10-29 10:02:05,262| Exiting method voidPerson
INFO - LoggingAdvice.invoke(115) |2018-10-29 10:02:05,276| In method PersonService.savePersonMergeLog. Arguments:
PersonMergeLog=PersonMergeLog[hashCode=1b9a9d1e,uuid=645954bb-38aa-4774-bd85-456833f4237f],
INFO - SerializationServiceImpl.getDefaultSerializer(71) |2018-10-29 10:02:05,278| No default serializer specified - using builtin SimpleXStreamSerializer.
INFO - SerializationServiceImpl.getDefaultSerializer(71) |2018-10-29 10:02:05,280| No default serializer specified - using builtin SimpleXStreamSerializer.
INFO - LoggingAdvice.invoke(155) |2018-10-29 10:02:05,283| Exiting method savePersonMergeLog
INFO - LoggingAdvice.invoke(115) |2018-10-29 10:02:05,516| In method UserService.saveUser. Arguments: User=admin,
INFO - LoggingAdvice.invoke(155) |2018-10-29 10:02:05,519| Exiting method saveUser
```

Logging Adequacy and Test Case Description: The logger adequately records the required information for merging two patient records. Additionally, it records the fields which changed as well as the unchanged fields.

3.1.10 Change ID Auto Generation options for patient IDs:

TestCaseID-tid_ag

TestCase Type:Change ID Auto Generation options for patient IDs

Steps:From the Home Screen Go to System Administration->Advanced Administration->AutoGenerationOptions->Select Identifier Type-> Click on Add->Select the Source of AutoGeneration and click save->Check the logs->System Administration->Advanced Administration->Maintenance->View Server Logs.

Expected Result: The logger should log which user added a new auto-generation method and for which ID.

Actual Result: There is no log maintained for these actions by the logger. This clearly indicates a severe problem of lack of logging for this part of the module. In case an attacker changes/deletes the auto generation method for a given type of ID, when a new patient is generated by the system, it would not be able to automatically assign an ID to the patient and thus create major problems in terms of correctness in the database as well as a host of other problems.

Logging Adequacy and Test Case Description: The logger fails to log the user which added the new auto-generation method.

3.2 Static Code Analysis with Fortify

3.2.1 Issue: Path Manipulation(Critical)

This part of the report highlights the issue of path manipulation where an attacker has the ability to manipulate different files inside the directory structure of the application by manipulating the directory paths.

Change Required: The best method to avoid any form of path manipulation is to use a form of indirection. One possible solution to this problem is to reject/sanitize the input from special characters. This prevents the malicious attacker from accessing any special resource as well as

using path commands to view stack traces or worse, internal files. For the case shown in the cross-reference, the attacker could inject their own modified string, possibly containing special characters, as the String in question, is not generated parametrically, it could cause problems. To avoid this, whenever the string, which is the input to the function, is made, should be sanitized of special characters.

Weakness Mitigated: This mitigates the possibility of any form of path manipulation by avoiding the use of path escaping/altering characters, although the issue of using resource names for path manipulation still remains.

Cross-reference:

HL7ServiceImpl.java, line 1165 (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 1165, which allows them to access or modify otherwise protected files.		
Source:	HibernateHL7DAO.java:369 org.hibernate.Criteria.list()		
367	`crit.add(Restrictions.lt("dateCreated", cal.getTime()));`		
368	`}`		
369	`return crit.list();`		
370	`}`		
371			
Sink:	HL7ServiceImpl.java:1165 java.io.File.File()		
1163	`//use the uuid, source id and source key(if present) to generate the file name`		
1164	`File fileToWriteTo = new File(dayDir, hl7InArchive.getUuid()`		
1165	`+ (StringUtils.isBlank(hl7InArchive.getHL7SourceKey()) ? "" : "_" + hl7InArchive.getHL7SourceKey())`		
1166	`+ ".txt");`		
1167			

3.2.2 Issue: Privacy Violation(Critical)

This part of the report highlights the issue of privacy violations. In this case, the username is being printed out in the log debug files. In case the log files fall into the hands of the attacker, they would be able to see private information.

Change Required: When there is a conflict between the demand for privacy and that of security, privacy is always given a priority. To get the best of both worlds, there is a need to clean any private information before it exits the program. The best policy w.r.t. To privacy is to minimize it's exposure. In this case, the log.debug line of the code, writes the username(which is confidential information) to the log without using any tools/methods to hide it. Instead, the program should obfuscate the username while printing the logs.

Weakness Mitigated: The weakness of privacy violation, or private information exposure is greatly mitigated by obfuscating the username with some other value to avoid any form of private information exposure.

Cross-reference:

Context.java, line 293 (Privacy Violation)			
Fortify Priority:	Critical	Folder	Critical
Kingdom:	Security Features		
Abstract:	The method authenticate() in Context.java mishandles confidential information, which can compromise user privacy and is often illegal.		
Source:	BaseContextSensitiveTest.java:425 Read junitpassword()		

```
423                 return;
424               } else
425                 credentials = new String[] { junitusername, junitpassword };
426
427               // try to authenticate to the Context with either the runtime
```

Sink: Context.java:293 org.slf4j.Logger.debug()


```
291               public static void authenticate(String username, String password) throws
                  ContextAuthenticationException {
292                 if (log.isDebugEnabled()) {
293                   log.debug("Authenticating with username: " + username);
294                 }
```

3.2.3 Issue: Command Injection(Critical)

This part of the report highlights the issue of command injection. In this case the application is taking unsanitized input from the user. Without any command blacklists, an attacker could insert commands that could manipulate the inner workings of the application.

Change Required: In this scenario, a malicious user can supply commands instead of a "valid" entry with a motive to run a command on the system. In such cases, there is a need to have checks on the strings entered by the user to avoid any form of malicious commands/input. Moreover, the user should be restricted only to perform certain commands, a user, inherently should not have the ability to execute, say a "sudo" level command, even if the input supplied by the user is not sanitized. This acts as a "double" barrier. On line 207 of the reference, the program tries to execute a command with wd as one of the inputs, wd is an input provided by the user to mitigate this threat, the application must sanitize wd in some form to avoid an attack.

Weakness Mitigated: The solution proposed above, reduces the attackers ability to execute commands using user provided input, thus mitigating the problem.

Cross-reference:

SourceMySqldiffFile.java, line 207 (Command Injection)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method execCmd() in SourceMySqldiffFile.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:	SourceMySqldiffFile.java:76 java.lang.System.getProperty()		

```
74
75                    if (username == null) {
76                     username = System.getProperty(CONNECTION_USERNAME);
77                    }
78                    if (password =****** null) {
```

Sink: SourceMySqldiffFile.java:207 java.lang.Runtime.exec()

```
205                   }
206
207                   Process p = (wd != null) ? Runtime.getRuntime().exec(cmdWithArguments, null, wd) :
Runtime.getRuntime().exec(
208                        cmdWithArguments);
209
```

3.2.4 Issue: Server-Side Template Injection(Critical)

This part of the report highlights the issue of Template Injection. In this case the application is taking unsanitized templates from the user. Without any template blacklists, an attacker could insert malicious templates that could create vulnerabilities in the application.

Change Required: An application should, in practice, never allow any form of user provided templates to prevent an attacker from being able to execute arbitrary code, which is the issue detailed in this scenario. In case they do, any form of input by the user must be sanitized to avoid any malicious input. In this case, on line 60 of the reference, the template used is one provided by the user. Moreover, the scary part is the comment by the programmer which says that he/she doesn't know what is the purpose of this line. To mitigate this, first we need to understand what exactly this line does. Then we should also sanitize the template before using it further in the application to avoid any malicious input.

Weakness Mitigated: The weakness of template-based server side injection can be easily mitigated by the solution provided above.

Cross-reference:

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 on 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 wher
                    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) {
```

3.2.5 Issue: SQL Injection(Critical)

This part of the report highlights the common issue of SQL Injection. In this case, the query is not being created using prepared statements and is thus vulnerable to malicious commands from an attacker.

Change Required: This is a very basic case of SQL injection. Here, the programmer uses a prepared query using a string to execute it. In the case that uuid provided, is tampered with by an attacker, they can easily execute SQL commands and get details about the different tables as well as the data within. To avoid this, the user must use parameterized queries.

Weakness Mitigated: The solution mentioned above mitigates the issue of SQL Injection. If this practise is followed all over the application, any further issues would be resolved.

Cross-reference:

```
159                   PreparedStatement stmt = connection.prepareStatement("SELECT property_value FROM
                    global_property WHERE property = ?");
160                   stmt.setString(1, globalPropertyName);
161                   ResultSet rs = stmt.executeQuery();
162                   if (rs.next()) {
163                    String uuid = rs.getString("property_value");
```

Sink:	MigrateAllergiesChangeSet.java:165 java.sql.Statement.executeQuery()

```
163                    String uuid = rs.getString("property_value");
164
165                    rs = stmt.executeQuery("SELECT concept_id FROM concept WHERE uuid = '" + uuid + "'");
166                    if (rs.next()) {
167                     return rs.getInt("concept_id");
```

3.2.6 Issue: Log Forging(High)

This part of the report highlights the issue of Log Forging. In this case, the variable being written to the log file, being supplied by the user, are not sanitized for malicious inputs.

Change Required: In this scenario a user has the capability to potentially forge a log entry by injecting content into the log. To avoid log forging attacks, we need to seperate log entries based on different events in the system. To capture user-based content, always use sanitized input from the user and do the analysis server-side to avoid obtaining malicious user entries. In the example shown in the reference, the query is directly used as a list without sanitizing it.

Weakness Mitigated: The weakness of, giving the ability to the attacker to add malicious input to the user while the user analyses logs is greatly mitigated by the recommendation mentioned above.

Cross-reference:

Context.java, line 334 (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 334. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log.		
Source:	HibernateUserDAO.java:100 org.hibernate.Query.list()		
98	`query.setString(0, username);`		
99	`query.setString(1, username);`		
100	`List<User> users = query.list();`		
101			
102	`if (users == null || users.isEmpty()) {`		
Sink:	Context.java:334 org.slf4j.Logger.info()		
332	`public static void becomeUser(String systemId) throws ContextAuthenticationException {`		
333	`if (log.isInfoEnabled()) {`		
334	`log.info("systemId: " + systemId);`		
335	`}`		

3.2.7 Issue: Password in Configuration File(High)

This part of the report highlights the issue of Passwords in Configuration files. In this case, the default password to setup a the hibernate connection has not been changed from default as well as it is written in plaintext, which is a very naive mistake.

Change Required: It is always important to never hardcode passwords, instead they should be obfuscated and stored in some other place, maybe using a form of hashing or some other form of encryption. As OpenMRS is an open-source application, it is very obvious for any attacker to figure out the hard-coded passwords and compromise the system. In this case, the hibernate

connection default password is set as "******". To mitigate this, store the password in an external db and use some form of hashing to access it.

Weakness Mitigated: By applying the solution mentioned above, there would be no need to write the password in the source code. This mitigates the weakness of "Password Configuration in File".

Cross-reference:

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

3.2.8 Issue: Password Management, Hardcoded password(High)

This part of the report highlights the issue of hard coded passwords. In this case, the default password of the scheduler is written in plaintext in the file as well as it hasn't been changed in the later part of the code.

Change Required: This is a similar issue to other issues where the password is directly hard-coded in the source-code. This issue as stated previously can be easily resolved by obfuscating the password and storing the obfuscated reference in an external database. In this case, the SCHEDULER_DEFAULT_PASSWORD is hard-coded to the value of "******". To resolve it, obfuscate the password using a hash and store the hashed value in an external db, reference it to check for the password instead of directly hard-coding it.

Weakness Mitigated: This mitigates the weakness of hard-coding the password in the source code.

Cross-reference:

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 */
```

3.2.9 Issue: Denial of Service, Regular Expression(High)

This part of the report highlights the issue of DoS using regex. In this case, the regex being used are not sanitized against a blacklist. An attacker could supply a very large regex which repeatedly sends requests to the server thus simulating a DoS attack.

Change Required: In this case, the regex supplied to the regex pattern compiler uses a variable called regex which is a variable passed by the user. This could cause an attacker to maliciously supply inputs which could consume large amounts of CPU or worse, paralyze the system or in other words, perform a DDoS attack. To avoid this, sanitize the user data for any possible maliciously crafted entries, for example, if the entry for name contains some form of ".." or "\+" this is a malicious entry and could create problems, thus it needs to be sanitized.

Weakness Mitigated: This mitigates the weakness that an attacker could supply malicious information to render the system unusable, thus avoiding a possible DDoS scenario.

Cross-reference:

HibernatePatientDAO.java, line 866 (Denial of Service: Regular Expression)			
Fortify Priority:	High	Folder	High
Kingdom:	Input Validation and Representation		
Abstract:	Untrusted data is passed to the application and used as a regular expression. This can cause the thread to overconsume CPU resources.		
Source:	HibernateAdministrationDAO.java:100 org.hibernate.Criteria.uniqueResult()		
98	`Criteria criteria = sessionFactory.getCurrentSession().createCriteria(GlobalProperty.class);`		
99	`return (GlobalProperty) criteria.add(Restrictions.eq("property", propertyName).ignoreCase())`		
100	`.uniqueResult();`		
101	`} else {`		
102	`return (GlobalProperty) sessionFactory.getCurrentSession().get(GlobalProperty.class, propertyName);`		
Sink:	HibernatePatientDAO.java:866 java.util.regex.Pattern.compile()		
864	`if (Pattern.matches("^\\^.{1}\\*.*$", regex)) {`		
865	`String padding = regex.substring(regex.indexOf("^") + 1, regex.indexOf("*"));`		
866	`Pattern pattern = Pattern.compile("^" + padding + "+");`		
867	`query = pattern.matcher(query).replaceFirst("");`		
868	`}`		

3.2.10 Issue: Key Management, Hard-coded encryption Key(High)

This part of the report highlights the issue of hard coded encryption. In this case, the encryption key specification has not been encrypted and is written in plaintext in the code. This allows the attacker to view the type of encryption being used and thus narrow down on the attack surface by using specific attacks based on the encryption.

Change Required: Similar to passwords, encryption keys should never be hard-coded, instead they should be obfuscated and their reference stored in an external database. Displaying the form of encryption used gives the ability to the attacker to know what kind of attack to use, in the case that the encryption used is a weak one. In this case, the attacker knows that the encryption used is AES. To mitigate this problem, the application should hide this information using a hash and store the hashed value in a database to avoid any exposure of encryption algorithms used.

Weakness Mitigated: This mitigates the weakness of the exposure of sensitive security information to any probable attacker.

Cross-reference:

OpenmrsConstants.java, line 528 (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:528 FieldAccess: ENCRYPTION_KEY_SPEC()		

```
526    public static final String ENCRYPTION_CIPHER_CONFIGURATION = "AES/CBC/PKCS5Padding";
527
528    public static final String ENCRYPTION_KEY_SPEC = "AES";
529
530    public static final String ENCRYPTION_VECTOR_RUNTIME_PROPERTY = "encryption.vector";
```

3.3 Fuzzing with ZAP

Module: Manage Accounts

Preconditions: From the OpenMRS home screen log into the application using the username ‘admin’ and password ‘Admin123’. Also select ‘Inpatient Ward’ as the location for the session. Navigate to ‘System Administration’ and then ‘Manage Accounts’.

3.3.1 Family Name SQL Injection When Adding a New Account

Detailed Instruction: Follow the Preconditions. Select the “Add New Account” button. For this id we are fuzzing the family name using a predefined zap list for sql injection (see fuzzer type). Enter in ‘name’ for family name, ‘given’ for given name, ‘male’ for gender, select the checkbox for ‘add provider account’, set identifier to ‘id’, and provider role to ‘Clerk’. Then hit save. Then within ZAP, view the post request and select to fuzz the family name using the Passive SQL Injection list from the Fuzzer Type.

Fuzzer Type: jbrofuzz -> SQL Injection -> Passive SQL Injection

Expected Result: ZAP should not locate any SQL injection vulnerabilities (whether reflected or stored) when attempting to sql inject the family name field when adding a new account.

Actual Result: The ZAP results show that there are 4 error response messages that could potentially be SQL error messages leading use to believe that this part of the application is vulnerable to SQL Injection. However upon further investigation, we see that these error messages are a result of the a check in the family name field that doesn’t allow more than 50 characters.

Fix: There was not anything to fix because we did not produce find any SQL vulnerabilities. OpenMRS could have prevented SQL Injection in a few different ways. They could have used prepared statements, sanitizing inputs, using white/black lists, etc. In the future we could adjust the fuzzing rules by adding more lists in an attempt to try and hit a specific condition for the database language being used. This would including adding language specific characters. We could also fuzz more fields than just the name.

3.3.2 Identifier Buffer Overflow Injection When Adding a New Account

Detailed Instruction: Follow the Preconditions. Select the "Add New Account" button. For this id we are fuzzing the identifier using a predefined zap list for sql injection (see fuzzer type). Enter in 'name' for family name, 'given' for given name, 'male' for gender, select the checkbox for 'add provider account', set identifier to 'id', and provider role to 'Clerk'. Then hit save. Then within ZAP, view the post request and select to fuzz the identifier using the long strings of a's as the fuzzer type detailed below.

Fuzzer Type: jbrofuzz -> Buffer Overflows -> Long Strings of aaa's

Expected Result: ZAP should not locate any buffer overflow vulnerabilities when attempting to inject long strings into the application field labeled "identifier".

Actual Result: The ZAP results show that there are there were no error messages, issues, or vulnerabilities resulting from the attempted buffer overflow fuzzing inputs. This likely means that the server side is protecting against this type of attack.

Fix: There was not anything to fix because we did not produce find a buffer overflow vulnerabilities. OpenMRS could have prevented buffer overflow vulnerability in a few different ways. On the client side, they limit the amount of characters to the identifier field to 255 by displaying an error message if the value has more than 255 characters. On the server side, they do not return a validation check when there are more than 255 characters, but adding more also does not produce a server side error. In order to improve the fuzzing rules we could attempt to send a very large amount of a's compared to the provided list in jbrofuzz. For example gigabytes or terabytes of a's (although unlikely that this will cause an issue unless the buffer is extremely large).

3.3.3 Given Name XSS Attack When Adding a New Account

Detailed Instruction: Follow the Preconditions. Select the "Add New Account" button. For this id we are fuzzing the given name using a predefined zap list for cross site scripting (see fuzzer type). Enter in 'name' for family name, 'given' for given name, 'male' for gender, select the

checkbox for ‘add provider account’, set identifier to ‘id’, and provider role to ‘Clerk’. Then hit save. Then within ZAP, view the post request and select to fuzz the given name using the long strings of a’s as the fuzzer type detailed below.

Fuzzer Type: jbrofuzz -> XSS -> XSS 101 & XSS 102

Expected Result: ZAP should not locate any XSS vulnerabilities, whether reflected or store,d when attempting to fuzz the ‘given name’ field.

Actual Result: The ZAP results show that there are there were two reflected XSS vulnerabilities. The inputs were <script>alert(string.fromcharcode(88,83,83))</script> and </noscript>
<code onmouseover=a=eval;b=alert;a(b(/h/.source));>MOVE MOUSE OVER THIS AREA</code>. Upon manual inspection, these inputs did not actually trigger an XSS vulnerability, but rather an error message from the server saying the “This value exceeds the maximum length of 50 permitted for this field”. However, it does also print what appears to be a json object to the screen revealing some additional information about the code that it likely should not have. It also prints error messages “This Field Is Required A maximum of -1 character(s) is allowed” which don’t make sense because you can’t have a negative maximum character amount.

Fix: OpenMRS should prevent the json object from being printed to the screen in the case of an error by try catching any input errors that may occur. They also should fix there message about having a maximum character count of -1 which only seems to appear when trying to use an XSS attack. It does not appear when simply entering too many characters. The fix involves input filtering using black/white lists, output filtering using black/white lists to avoid the scripts from being reflected, and using output encoding libraries. For future fuzzing rules I would add more obfuscation of the script tags to attempt to trick a simple blacklisting xss prevention library.

3.3.4 Username Injection Attack When Adding a New Account

Detailed Instruction: Follow the Preconditions. Select the “Add New Account” button. For this id we are fuzzing the identifier using a predefined zap list for sql injection (see fuzzer type). Enter in ‘name’ for family name, ‘given’ for given name, ‘male’ for gender, select the checkbox for ‘add user account’, set username to ‘username’, set privilege level to full, and password to ‘Admin123’. Then hit save. Then within ZAP, view the post request and select to fuzz the identifier using the long strings of a’s as the fuzzer type detailed below.

Fuzzer Type: jbrofuzz -> Injection -> XPath Injection

Expected Result: ZAP should not locate any Injection vulnerabilities when attempting to fuzz the application field labeled “username”.

Actual Result: The ZAP results produces error messages for all of the input types. After manual inspection though, it was determined that these were not injection vulnerabilities but instead involved input validation messages, stating “Username is invalid. It must be between 2 and 50 characters. Only letters, digits, ".", "-", and "_" are allowed.”

Fix: There was not anything to fix because we did not produce any Injection vulnerabilities. It appears that OpenMRS prevents injection vulnerabilities in the username field by whitelisting letter characters, “.”, “-”, and “-”. It appears that all other inputs are blocked making it difficult to perform injection attempts on this input field. These checks are present on both the client and server side of the application. In order to improve the fuzzing rules we could attempt to perform more injection attacks using the characters that are specified as valid and see if we can discover a vulnerability using validly selected characters.

3.4 Client Side Bypassing with ZAP

Module: Manage Global Properties

Preconditions: From the OpenMRS home screen log into the application using the username ‘admin’ and password ‘Admin123’. Also select ‘Inpatient Ward’ as the location for the session. Navigate to ‘System Administration’ and then ‘Manage Global Properties.

3.4.1 Name XSS Attack when Adding New Global Property

Detailed Instruction: Follow the Preconditions. Select the “Add New Global Property” button. Enter “Bob” for the name, “This is the description” for the description and “5” for the value. Hit enter and stop the user input with ZAP. Replace the property field with **<script>alert('xss')</script>** and send the new request to the server.

URL:
http://localhost:8082/openmrs-standalone/adminui/systemadmin/globalproperties/manageGlobalProperties.page#/edit/

Expected Result: An error should be returned stating that the name cannot contain special characters and no XSS popup should occur.

Actual Result: No XSS message is displayed to the user, but the user is not told that the name has invalid characters.

3.4.2 Description Command Injection Attack when Adding New Global Property

Detailed Instruction: Follow the Preconditions. Select the “Add New Global Property” button. Enter “Sally” for the name, “Random comment” for the description and “hello” for the value. Hit enter and stop the user input with ZAP. Replace the description field with **;ls** and send the new request to the server.

URL:
http://localhost:8082/openmrs-standalone/adminui/systemadmin/globalproperties/manageGlobalProperties.page#/edit/

Expected Result: The server should not reply with a directory listing and and error should be returned stating that the name cannot contain special characters.

Actual Result: No directory listing is displayed to the user, but the user is not told that the name has invalid characters.

3.4.3 Value Buffer Overflow Attack when Adding New Global Property

Detailed Instruction: Follow the Preconditions. Select the “Add New Global Property” button. Enter “Craig” for the name, “blah blah” for the description and “1234” for the value. Hit enter and stop the user input with ZAP. Replace the value field with
**aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**
and send the new request to the server.

URL:
http://localhost:8082/openmrs-standalone/adminui/systemadmin/globalproperties/manageGlobalProperties.page#/edit/

Expected Result: The server should not return a segmentation fault error or another error to the user and should also send a response saying that the value has exceeded the maximum number of characters.

Actual Result: No segmentation fault or error is displayed to the user, but also did not notify that the value has exceeded the maximum number of characters (may not have reached the limit).

3.4.4 Name SQL Injection Attack when Adding New Global Property

Detailed Instruction: Follow the Preconditions. Select the "Add New Global Property" button. Enter "Kevin" for the name, "CSC 515" for the description and "apple" for the value. Hit enter and stop the user input with ZAP. Replace the property field with **' --** and send the new request to the server.

URL:
http://localhost:8082/openmrs-standalone/adminui/systemadmin/globalproperties/manageGlobalProperties.page#/edit/

Expected Result: The server should not reply with a SQL error and should notify the user that invalid characters have been entered for the name.

Actual Result: No SQL error is displayed to the user, but they are not told that the name has invalid characters. The input data is saved implying that the query was valid.

3.4.5 Value XML Injection Attack when Adding New Global Property

Detailed Instruction: Follow the Preconditions. Select the "Add New Global Property" button. Enter "Kevin" for the name, "CSC 515" for the description and "apple" for the value. Hit enter and stop the user input with ZAP. Replace the property field with **<xml id=i><x><c><![cdata[<img src=\"javas]]><![cdata[cript:alert('xss');\">]]></c></x></xml>** and send the new request to the server.

URL:
http://localhost:8082/openmrs-standalone/adminui/systemadmin/globalproperties/manageGlobalProperties.page#/edit/

Expected Result: The replaced XML code should be stored as the value and no errors, stack traces, or popups should occur as a result of the parsed XML.

Actual Result: The replaced XML code should was stored as the value and no errors, stack traces, or popups occurred.

3.5 OpenMRS Security Requirements

Identifier	Security Requirements
1. Least Privilege	**The ability to access and view the logs should be provided only to the Super-Admin and the Admin to preserve the integrity of the information contained in the logs. Moreover, Users should be able to only view and edit their own data and not tamper with others' data.**
2. Fail-Safe Defaults	**Adequate Fail-Safe defaults should be implemented. Wherein, the "Advanced Admin" controls link and tab should be hidden by default and should not be visible to any random user except for the verified and trusted Admin or a Super Admin.**
3. Complete Mediation	**When a User wants to delete his account or he has forgotten his password, it becomes necessary for the software to ask for the adequate and necessary credentials from the User so that it is validating every access to the required functionalities and also ensuring the access.**
4. Separation	**When an admin or a User wants to upgrade his privileges or make changes to the privilege, a multi-factor authentication mechanism should be prompted by the application so as to ensure that it's not just a single step procedure requiring the user to enter a code from Google Authenticator.**
5. Minimize Trust	**When a user attempts to log into OpenMRS, the system should check that the username does not use any special characters and should take use of prepared statements when cross referencing the given username and password with the existing ones stored in the database. If special characters are used in the username, an error should be returned to the user stating "Invalid**

	username/password combination". (It is assumed that no special characters can be used in the username).
6. Economy of Mechanism	**The only way to authenticate to OpenMRS should be through the user login screen. If a user attempts to navigate to any page and is not logged in, they will be redirected to the login screen. Additionally, the authentication checks from this screen should require the User credentials to be verified through a single method that would compare hashed and salted pre-existing credentials. This way we reduce the complexity of authentication to OpenMRS.**
7. Minimize Common Mechanism	**The amount of common functionalities between a user, an admin, a Super-Admin and a nurse should be kept as minimal as possible. For example, a nurse should be able to see columns relevant to her and not all the column even if private and irrelevant to her of the user or patient in question.** **This can be tested by logging in as a nurse and then trying to view Super Admin functionalities. In this case she shouldn't be allowed to do so and there should be a method to raise a message notification flag warning of this activity to the nurse and the admin.**
8. Layering	**The OpenMRS application is only as strong as its weakest link, which in our case is the Patients or the Admins which are the symbolism of the Human Factor coming into play. They are weakest because they are susceptible to insider threats and social engineering attacks. This can be avoided by ensuring security of them. The other way is the attacker targeting the endpoints of the servers whilst they are communicating, these can be obviated and tested by using secure communication protocols. Whenever someone tries to insert a malicious cryptographic code, that IP should be blocked and an error flag should be raised to the super-admin**
9. Abstraction	**The system should provide one API interface for encrypting data before being stored in the system. This will allow the encryption to be abstracted away and the method can be updated to use whatever the most advanced and latest cryptographic algorithm is without affecting components of other layers.**
10. Design for Iteration	**Patients should contain patient ids that are used as the reference/primary keys when relating to information in a database such as health records. This will allow for low coupling and the ability to easily change patient structures and health data**

	structures without having to rewrite the database architecture.
11. Least Astonishment	**For this one, there should be minimal obscurity in the assigned functionality of the various users, and the nurse, admin and super-admin should be able to use their functionalities in a clear and a simple manner. There shouldn't be a form or a functionality for example that should ask for too many strenuous details from a nurse and so on.**

IV. Design/Security Principles, Protection Poker, and Bug Fixes

4.1 Architectural Design Principle Violations

Violation 1: Separation - Software should not grant access to a resource, or take a security-relevant action, based on a single condition.
Test Case ID: Requiring extra authentication when granting admin privileges

Detailed Instruction:

1. Navigate to the OpenMRS login screen: http://localhost:8081/openmrs-standalone/login.htm (change port as needed).
2. From the OpenMRS login screen enter admin as the username and Admin123 as the password.
3. Select "Inpatient Ward" as the location for the session and hit the "Login" button.
4. Click System Administration
5. Click Manage Accounts
6. Click on the edit icon next to John Smith. (It is assumed you have the OpenMRS standalone version with default patients already entered.)
7. In the user account details box, select the pencil edit icon on the right side of the screen.
8. Set privilege level to full and click the box labeled "Administers System".
9. Click the save button.

Expected Results: OpenMRS should prompt the user to input a 2FA 6 digit code with the option for the code to be sent to them by their account email, or synced google authenticator app. The user selects their option, receives the code (by email or app), inputs the correct code and the the privilege change is updated. If the code is incorrect the user is presented with an error message about an invalid code.

Actual Results: The updated privileges are automatically saved and given to the user without a need for a second authentication factor. This mechanism for changing privileges violates separation by granting a security-relevant action based on a single condition. If you are granting someone administrative privileges, you should have to provide a second factor. This would prevent scenarios where and admin forgets to lock their computer and an employee tries to change their privilege level.

Violation 2: Complete Mediation: Software should validate every access to object to ensure that the access is allowed.
Test Case ID: Navigating to unauthorized page through direct link

Detailed Instructions:

1. Navigate to the OpenMRS login screen: http://localhost:8081/openmrs-standalone/login.htm (change port as needed).
2. From the OpenMRS login screen enter "clerk" as the username and "Clerk123" as the password.
3. Select "Inpatient Ward" as the location for the session and hit the "Login" button.
4. Navigate to the following url (be sure to change the port number to match your port): http://localhost:8082/openmrs-standalone/adminui/metadata/configureMetadata.page

Expected Result: OpenMRS should fail to load this page for a clerk because they are not authorized to configure metadata from within the admin UI. Only admins should be able to view this page.

Actual Results: Fail - The configure metadata pages loads and allows the clerk to view the page. Based on complete mediation, the software should validate every access to a page, which in this case it does not because only admins should be allowed to view this page. The application should not assume that the user will only click links on their page and needs to handle the case of a user pasting in a direct link.

Violation 3: Minimize Trust: Software should check all inputs and the results of all security-relevant actions.
Test Case ID: Ensure script tags are not used in patient name inputs

Detailed Instruction:

1. From the OpenMRS login screen enter admin as the username and Admin123 as the password.
2. Select "Inpatient Ward" as the location for the session and hit the "Login" button.

3. Navigate to the “Register a patient” button on the main menu.
4. When registering the patient, enter “<script>alert(“1”)</script>” (Do not copy paste the string directly and do not include outside quotes).
5. Enter Johnson as the family name then hit enter.
6. Select Male for the gender and hit enter.
7. Input 02 for the day, February for the month and 1996 as the year and hit enter.
8. Enter 321 Western Blvd as the address, Raleigh as the city, NC as the state, U.S.A. as the country, and 27607 as the postal code then hit enter.
9. Enter 123-456-7890 for the phone number and hit enter.
10. Select Parent for the relationship type and enter Dad as the person’s name, then hit enter twice.
11. Select Confirm to confirm the submission.

Expected Result: The application should not let you enter “<script>alert(“1”)</script>” as the name and should return an error message to the user saying “Invalid characters used in name field, please use only use letters, apostrophes, and hyphens”. OpenMRS should validate this input to be sure that potential XSS attacks cannot occur on the system, when there are no need for these extra special characters in a person’s name.

Actual Result: Fail - The application allows the user to register their name as “<script>alert(“1”)</script>” and does not display an error message to the user. The application should not be register a name of the person with such random characters such as ”<, (,), “”, and should display an error message to the user. Failing to check all inputs into the system could cause security issues, such as cross site scripting attacks in this case.

4.2 Usable Security Principle Violations

Violation 1: Expected Ability
Test Case ID: ea

Test Case Assumptions: The assumption is that the user ‘sysadmin’ has lower privilege(Organizational: System Administrator) than the ‘admin’ user which has the highest privilege level(System Developer, Provider) as the user ‘sysadmin’ is not able to do all the actions that ‘admin’ can do.

Test Case Steps:
1. Login as admin with username as admin and password as Admin123
2. Go to System Administration->Advanced Administration->Manage Users(In the users module).

3. Click on the search button to reveal all the users in the system.
4. Click on 6-7. The user with the username sysadmin.
5. Change the password of this user and note it down. Then save.
6. Logout and Login as sysadmin with the new password.
7. Go to System Administration->Advanced Administration->Manage Users(In the users module).
8. Click on the search button to reveal all the users in the system.
9. Select the user with the name admin.
10. Change the password of this user.

Expected Result
The user ‘sysadmin’ should not be able to access and change permissions of the user ‘admin’ which has a higher privilege level.

Actual Result
The user ‘sysadmin’ is able to access and change permissions of the user ‘admin’ which has a higher privilege level. This is a clear violation of expected ability.

Conclusion
The above test clearly indicates that a user with a lower level of privilege is able to change the privilege level of a user with a comparatively higher privilege level and demonstrates a serious flaw in the ‘Expected Ability’ principle which states that “The interface must not generate the impression that it is possible to do something that cannot actually be done”. This is most unexpected and can have serious consequences.

Possible Fixes
Besides a serious design overhaul, other short-term fixes could be setting a condition where a user with the same privilege levels cannot access each others information as well as more importantly users with a lower privilege cannot change access information of a user with a higher privilege.

Violation 2: Revocability

Test Case ID: rv

Test Case Assumptions: The assumption is that the user ‘sysadmin’ has lower privilege(Organizational: System Administrator) than the ‘admin’ user which has the highest privilege level(System Developer, Provider) as the user ‘sysadmin’ is not able to do all the actions that ‘admin’ can do.

Initial Test Case Context: A user with a higher privilege level, when assigns a privilege level that is lower than the user itself, should be able to revoke this privilege whenever necessary. In this case, the user 'admin' gives a user 'sysadmin', 'Organizational: System Administrator' privilege which is a lower level privilege than the admin itself. It is assumed that this has already been done in the way that a user 'sysadmin' already exists in the OpenMRS standalone version.

Test Case Steps:

1. Login as admin with username as admin and password as Admin123
2. Go to System Administration->Advanced Administration->Manage Users(In the users module).
3. Click on the search button to reveal all the users in the system.
4. Click on 6-7. The user with the username sysadmin.
5. Change the password of this user and note it down. Then save.
6. Logout and Login as sysadmin with the new password.
7. Go to System Administration->Advanced Administration->Manage Users(In the users module).
8. Click on the search button to reveal all the users in the system.
9. Select the user with the name admin.
10. Deselect 'System Provider' and 'System Developer' .
11. Save the user.
12. Do another blank search and select user sysadmin.
13. Select 'System Provider' and 'System Developer'.
14. Save the user

Expected Result

A user with a higher privilege should be able to revoke his/her decision regarding the privileges of other users.

Actual Result

A user with a higher privilege is not able to revoke his/her decision regarding the privileges of other users as the other user removed the privileges of the admin.

Conclusion

Once the steps are completed, a privilege which was provided by the user with the highest privilege cannot be revoked as the user granting the privilege has his/her privilege level revoked thereby revoking their ability to revoke in the first place. Thus, this is a clear violation of the principle of revocability.

Possible fixes
Besides a major design overhaul, a short-term fix would be that users with the same privilege level cannot provide or revoke access of users as well as more importantly users with a lower-level cannot change their privilege to a higher level. Moreover, the user/users with the highest privilege have the right to revoke all possible actions keeping in mind the overlap between users with the highest privilege.

Violation 3: Path of least Resistance

Test Case ID: pr

Test Case Steps:
0. Start with a fresh installation of openmrs in case you changed admins in the cases above.
1. Login as admin with username as admin and password as Admin123.
2. Go to System Administration->Advanced Administration->Find Patients to merge
3. Select 'Gender' and click search.(In case the search doesn't directly give results, try selecting birthdate as well and then click search. Then deselect Birthdate and search again).
4. Checkbox the first user and click continue.
5. A new window within the tab should open with details to merge.
6. Enter 'John' in the search box and click on the first identifier which comes up.
7. Now you should be able to see LogOut, Home, etc. options inside the merge patient window.(If not visible try scrolling up a bit).
8. Now click logout. It should take you to the login screen, inside the merge patients window.
9. Now close the merge window using the small 'x' sign on the top right.
10. You will not be logged out from the admin user.(although by clicking any option, you would be logged out). Do not click on any buttons on the page.
11. Click the back button on the browser, you should be able to see all the pages the 'admin' visited even after he/she has logged out.

Expected Result
Once the merger of two patients is complete the user should be able to close the window without logging out.

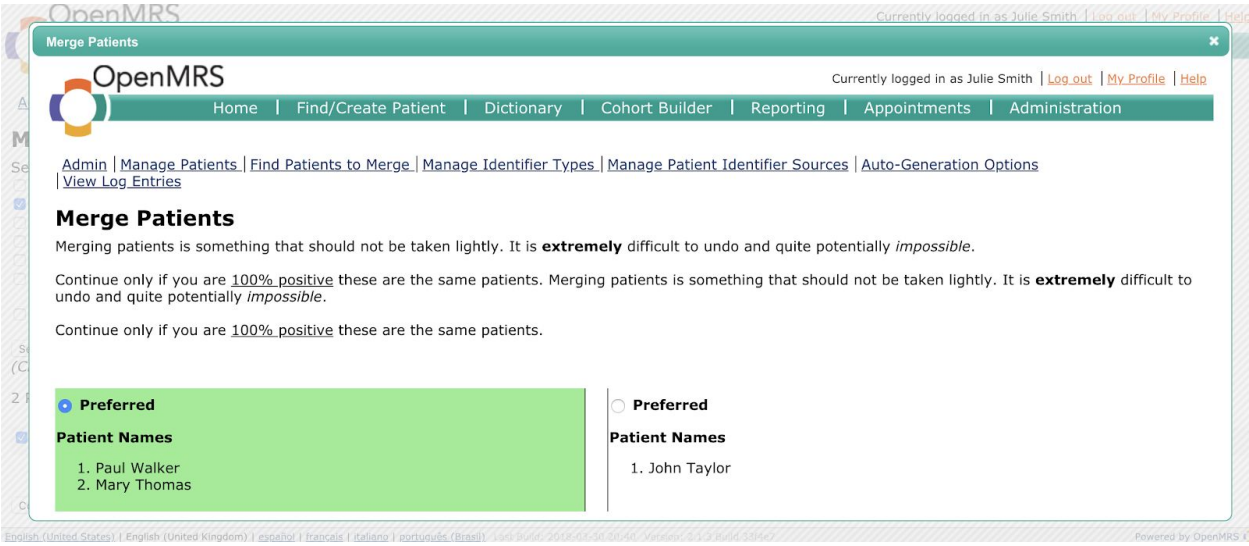
Actual Result
The user sees the header page of the application (which contains the links to home/, logout, etc.) within the merge window. This may cause the user to use these links and this causes the user to logout from the application although closing the window would revert the user back to the same page.

Conclusion

The above test case shows many serious design flaws in the system which are of grave concern. One of the flaws violates the design principle of 'Path of least resistance' because any inexperienced admin would logout from the LogOut button in merge window as the background is blurred and he/she would find it pretty natural to use that button instead of clicking on the 'x' on the top-right as it is the path of least resistance but this bypasses security. As the description of the design flaw states 'The natural and easiest way to do any task should also be the secure way' but the above test case clearly shows that it isn't the case.

Possible Fixes

Besides a major design-overhaul, another possible fix is, when a user clicks on one of the search results, instead of using a link which redirects to another page causing the visibility of the homescreen 'headers' it only displays the searched information within an embedded div element without the headers. This would require major changes to how search works within the merge module.



4.3 Protection Poker

Below we have documented 5 new features that we have decided to implement for OpenMRS.

4.3.1 Pharmacist Account - This is a functionality that has not been implemented in the sense, a hospital generally has Pharmacist who provides the medications, shows what medications he has and his job of keeping maintaining the records of the stocks of medicines. It should have various fields in the database such as Pharmacist ID,

Pharmacist Qualifications, OrdersOfMedicine, PrescriptionList, MedicalStocks, PharmacistPassword.

4.3.2 Patient Account - A Legitimate Patient account doesn't exist which a patient can use to sign in, update his medical history, schedule appointments and so on. This is a very important functionality that should be added. Because everyone has the crud functionality attributed to them. Various features in the db should be PatientPassword, PatientDetails, PatientCondition and so on.

4.3.3 Ambulance Online Services - This is also an important functionality that should be added which would provide the ailing patients the availability for booking transit to the hospital during emergencies. It should have its functionalities in the db as AmbulanceNumber, AvailableAmbulances, AmbulanceSize.

4.3.4 Medication Logs - This a feature that should be present so that whenever any medication is used, prescribed or the amount of them is changed can be logged. The various users who have access to this are the nurses, doctors, pharmacist and patients who can add the quantity and names of it. It should have various fields of it in the db including MedicineType, QuantityOfMedicine, AvailableStocks, and so on.

4.3.5 Room Logs - This is also a functionality that should be added to the OpenMRS for assigning various rooms for various purposes to various people during various times of the day. Even the availability and the number of rooms can be logged by the admins or the other staff to keep track. Various fields can be TotalNumberOfRooms, RoomType (Doctors Cabin, ICU, Waiting Room, Emergency Rooms, and so on), AvailableRooms, TimingsOccupied.

An additional possible Feature:

4.3.6 Treatment History - This is to keep track possible of the patient's past medical conditions and the level and the number of treatments that has gone and under which doctors, also the history of operations done by which doctors and so on. Possible fields are MedicalCondition, DoctorConcerned, NumberOfTreatments, etc.

Protection Poker Table

ADDITIONAL FUNCTIONAL	EASE POINTS	VALUE POINTS	SECURITY RISKS	RANKING

REQUIREMENTS				
Pharmacist Account	30	60	1800	2
Patient Account	20	100	2000	1
Ambulance Online Services	50	15	750	6
Medication Logs	60	25	1500	3
Room Logs	70	15	850	5
Treatment History	25	50	1000	4

Reactions and META Approach:

Patient Account - Ranks No. 1, as it is the most important element of the whole system and the most important Victim that would be the most likely to incur the effects of attacks, as expected. This is the account that all of the hackers would be going after, so this should definitely be mitigated in case of attacks. Some common attacks can be SQL Injection and XML attacks where in the WhiteListing and Sanitizing of the inputs is a good way to mitigate them.

Pharmacist Account - Ranks No. 2, in terms of Security Risks as it is also vulnerable to be exploited by attackers for getting records of patient data and other sensitive data regarding medicinal prescriptions and so on. So, it is no surprise that it is the second most important record to be going after by the hacker, even though the ease points lie a bit on the “Hard to attack” side. Even in this case the hackers would apply XSS attacks and Security Misconfiguration Attacks which can be mitigated by Application hardening and Sanitizing Inputs.

Medication Logs - Ranks No. 3 which is a good position in terms of Security Risk because it is surely a case of concern if someone were to start tampering with the medical logs which have important patient data linked to them in the form of prescriptions or someone altering the data of the amount of available medicines which makes it difficult to access them in times of need. Of course, the vulnerability applicable to this scenario is the “Insufficient Logging and Monitoring” which can be mitigated by ensuring secure logs are generated which can be easily consumed by

log management solutions. Even so, the data of medicines is not of that grave importance which can lead to a limited window of acceptance if the attack is of a minimal nature.

Treatment History - Ranks No. 4 which is a surprise that it ranks below the Medication Logs as it is a much more richer and an important source of a patient's previous medical information as compared to the medication logs. A hacker would be likely to attack this db more than the medication logs one since this is more patient-mapped and more informative, employing attacks like Sensitive Data Exposure and Injection which should be mitigated with Data Encryption.

Room Logs - Ranks No. 5 in the Security Risk table which is expected as this is a do away feature and one that is not of high importance as the above ones in case of tampering. For instance, if someone just tampered all the rooms data, then it would still be possible to maintain a manual record of rooms available by manual inspection of the states of them. The primary functionality of this feature was just to ease assistance in the online maintenance of the overall functionality of the application and thus cover breadth. An attacker is less likely to mess this db up even if he does so, he is likely to use Insufficient Logging and Monitoring and as long as the attacks don't delete the whole db let's say, they can be acceptable.

Ambulance Online Services - This is ranking no. 6 which is again a slight surprise as it is a rather helpful and a necessary feature that would not be incurring such a low security risk. It should have gotten somewhere midway between the ranks, but this current ranking can be justified for it by saying that a hacker would not be concerned with this as much as he would be with the first 3 functional requirements as he would not gain much by lets say DDOSing all the db related to Ambulances. Only harm would come is when in the time of emergency if the ambulances db is unavailable, then it becomes challenging to ascertain the availability of resources. Relatively easy to the hacking as this will not be having as stringent checks at the top ones and the vulnerabilities such as Injection and Broken Access Control will get the hackers through which can be mitigated by Limiting the number of inputs and the allowable IP addresses.

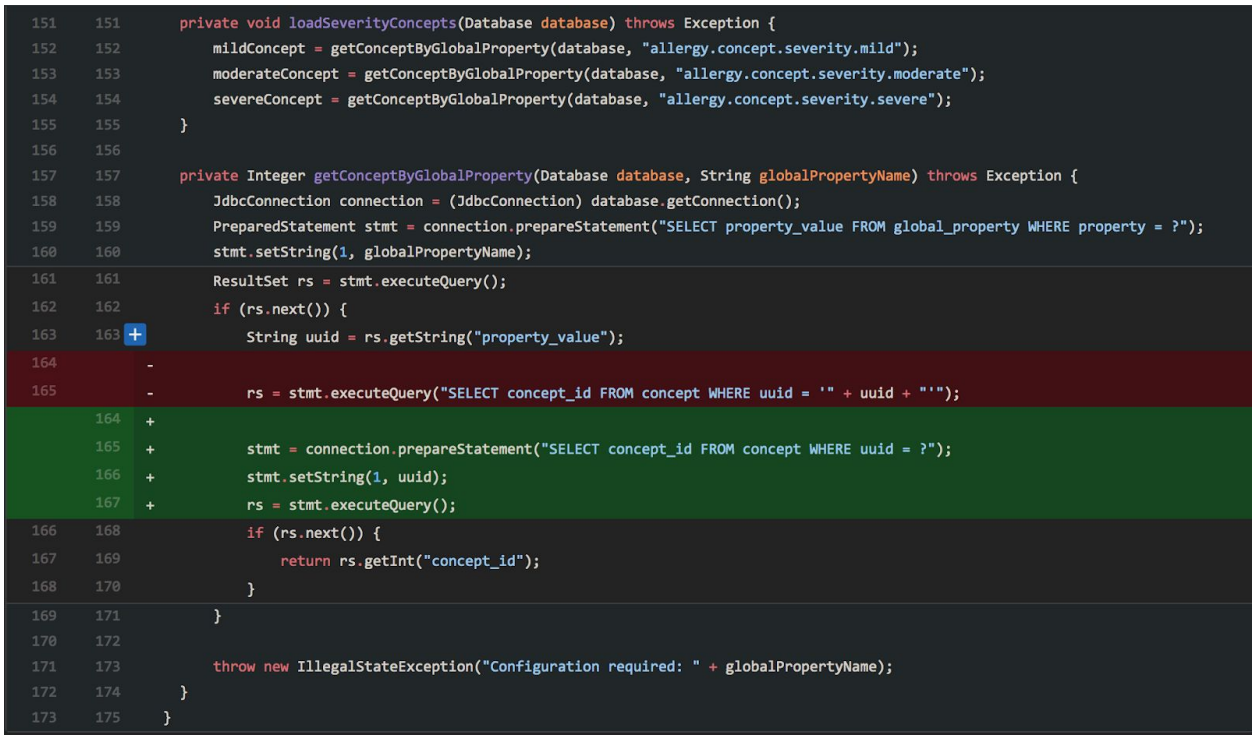
4.4 OpenMRS Bug Fixes

4.4.1 Bug Fix 1: Change SQL Injection Vulnerability to Use a Prepared Statement

Description/Fix: For this bug, the issue involved an SQL injection vulnerability where the uuid could be tampered with to craft arbitrary SQL queries. The original code, found with Fortify, shows that the uuid is appended to the SQL query as a string. The uuid could contain malicious SQL code that escapes the statement and creates its own query potentially leaking data to the attacker or allowing them to modify the database. Our fix for this code was to place the query

inside a prepared statement. This prevents the user from escaping the query and creating their own because the specific value is being looked up, not an arbitrary string query.

Reference: This bug was found using static analysis with Fortify and corresponds to issue 5 from our Project 3 part 2.



```
151  151      private void loadSeverityConcepts(Database database) throws Exception {
152  152          mildConcept = getConceptByGlobalProperty(database, "allergy.concept.severity.mild");
153  153          moderateConcept = getConceptByGlobalProperty(database, "allergy.concept.severity.moderate");
154  154          severeConcept = getConceptByGlobalProperty(database, "allergy.concept.severity.severe");
155  155      }
156  156
157  157      private Integer getConceptByGlobalProperty(Database database, String globalPropertyName) throws Exception {
158  158          JdbcConnection connection = (JdbcConnection) database.getConnection();
159  159          PreparedStatement stmt = connection.prepareStatement("SELECT property_value FROM global_property WHERE property = ?");
160  160          stmt.setString(1, globalPropertyName);
161  161          ResultSet rs = stmt.executeQuery();
162  162          if (rs.next()) {
163  163  +           String uuid = rs.getString("property_value");
164       -
165       -           rs = stmt.executeQuery("SELECT concept_id FROM concept WHERE uuid = '" + uuid + "'");
     164  +
     165  +           stmt = connection.prepareStatement("SELECT concept_id FROM concept WHERE uuid = ?");
     166  +           stmt.setString(1, uuid);
     167  +           rs = stmt.executeQuery();
166  168              if (rs.next()) {
167  169                  return rs.getInt("concept_id");
168  170              }
169  171          }
170  172
171  173          throw new IllegalStateException("Configuration required: " + globalPropertyName);
172  174      }
173  175  }
```

4.4.2 Bug Fix 2: Heap Inspection Vulnerability

Description/Fix: The bug in question is the use of a String type for storing a password to pass to a function. String data type variable are not cleared from memory immediately, they are cleared only when the JVM runs a garbage collector. In the case of a crash, there is a memory dump. Attackers may use the this ‘heap’, a data structure used to store memory dump information, to obtain the values of the string variables as they are not being cleared by the JVM garbage collector. A solution to this would be the use of byte/character arrays and clearing the arrays once their use has been depleted. Strings are more vulnerable as they are immutable whereas character arrays are not immutable.

```
String pass;
{


    int length = rand.nextInt(4) + 8;
    char[] password = new char[length];
    for (int x = 0; x < length; x++) {
        int randDecimalAsciiVal = rand.nextInt(93) + 33;
        password[x] = (char) randDecimalAsciiVal;
    }
    pass = new String(password);
}
us.createUser(user, pass);


us.createUser(user, String.valueOf(password));
Arrays.fill(password,'');
```

Reference: This bug was found using static analysis with Fortify. The bug description is located at page number 27 of the attached Fortify report that was performed for the previous part of the project.

4.4.3 Bug Fix 3: Change SQL Injection Vulnerability to Use a Prepared Statement

Description/Fix: For this bug, the issue involved an SQL injection vulnerability where the name could be tampered with to craft arbitrary SQL queries. The original code, shows that the name is appended to the SQL query as a string, this string is then passed to a function getInt which uses the string directly without setting the preparedstatement. The name could contain malicious SQL code that escapes the statement and creates its own query potentially leaking data to the attacker or allowing them to modify the database. Our fix for this code was to place the query inside a prepared statement. The variable ‘stmt’ has multiple internal dependencies which are being used in other functions where the argument to the function is a string, these have also been changed to keep the code error free.

```
...
                for (Map.Entry<String, String[]> e : names.entrySet()) {
                        String locale = e.getKey();
                        for (String name : e.getValue()) {
                                Integer ret = getInt(connection, "select concept_id from
concept_name where name = '" + name
                                                + "' and locale like '" + locale + "%'");
        PreparedStatement stmt = connection.prepareStatement("select concept_id from
concept_name where name = ?");
        stmt.setString(1,name);
                                Integer ret = getInt(connection, stmt);
                                if (ret != null) {
                                        return ret;
                                }
...
         * @return integer resulting from the execution of the sql statement
         * @throws CustomChangeException
         */
        private Integer getInt(JdbcConnection connection, String sql) throws
CustomChangeException {
        private Integer getInt(JdbcConnection connection, PreparedStatement stmt) throws
CustomChangeException {
                Statement stmt = null;
                try {
                        stmt = connection.createStatement();
                        ResultSet rs = stmt.executeQuery(sql);

                        ResultSet rs = stmt.executeQuery(stmt);
                        Integer result = null;

                        if (rs.next()) {
                                result = rs.getInt(1);
                        } else {
                                // this is okay, we just return null in this case
                                log.debug("Query returned no results: " + sql);
                                log.debug("Query returned no results: " + stmt.toString());
                        }

                        if (rs.next()) {
                                log.warn("Query returned multiple results when we expected just
one: " + sql);
                                log.warn("Query returned multiple results when we expected just
one: " + stmt.toString());
                        }

                        return result;
...
```

Reference: We came across this piece of code while browsing the repository. The file in question is BooleanConceptChangeSet.java.

4.4.4 Bug Fix 4: Lack of proper checks for special characters in username

Description/Fix: This is more of an functionality enhancement rather than a fix. The username received by the function is only being checked if its null or has length 0, although it is not being checked for special characters such as "#$<>@%^&*", etc. A potential attacker could supply a malicious string which could potentially harm the application in many ways from revealing user information to planting malicious XSS scripts. We fixed this issue by writing a small check before passing the username forward in the program flow.

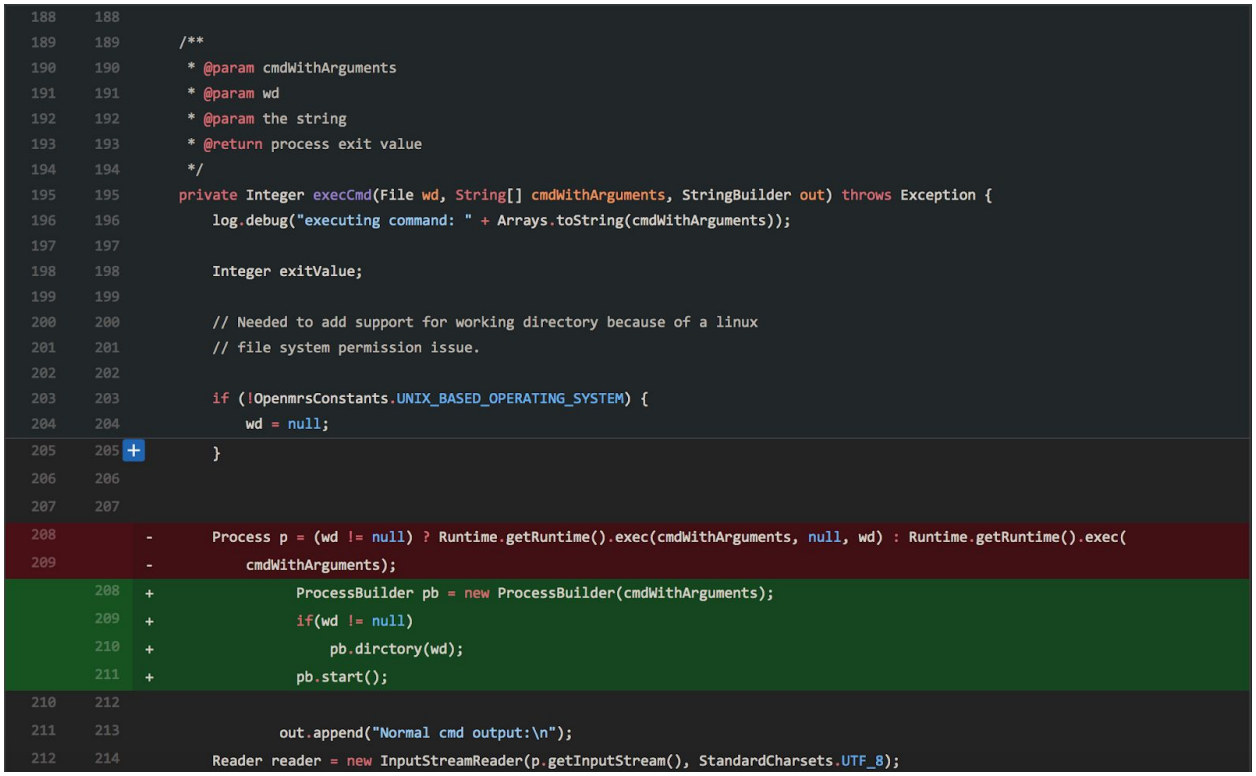
```
...
                        if (username == null || username.length() == 0) {
                                throw new IllegalArgumentException("each <user /> element must
define a user_name attribute");
                        }
                        Pattern p = Pattern.compile("[^a-z0-9]",Pattern.CASE_INSENSITIVE);
                        Matcher m = p.matcher(username);
                        boolean check = m.find();
                        if(check){
                                throw new IllegalArgumentException("each <user /> element must
not contain special characters");
                        }
                        if (us.getUserByUsername(username) != null) {
                                continue;
                        }
...
```

Reference: We came across this piece of code while browsing the repository. The file in question is MigrationHelper.java. We found that you could input arbitrary characters for the username to cause potential SQL injections in project part 1.

4.4.5 Bug Fix 5: Fix Command Injection Vulnerability with Process Builder

Description/Fix: For this bug, the issue involved a command injection vulnerability where the arguments after the first argument in the cmdWithArguments array could be tampered with to perform arbitrary shell commands and pipes. The user could insert untrusted data into the argument array after the first to their desired input. The original code, found with Fortify, shows that this method is vulnerable due to the use of exec without checking the user input. Our fix for this code was to place command arguments into a process builder which only treats follow up array indices as command arguments to the initiated process at the beginning of the command array. This prevents the user from performing a command injection attack by attempting to escape the command.

Reference: This bug was found using static analysis with Fortify and corresponds to issue 5 from our Project 3 part 3.



```
188 188
189 189     /**
190 190      * @param cmdWithArguments
191 191      * @param wd
192 192      * @param the string
193 193      * @return process exit value
194 194      */
195 195     private Integer execCmd(File wd, String[] cmdWithArguments, StringBuilder out) throws Exception {
196 196         log.debug("executing command: " + Arrays.toString(cmdWithArguments));
197 197
198 198         Integer exitValue;
199 199
200 200         // Needed to add support for working directory because of a linux
201 201         // file system permission issue.
202 202
203 203         if (!OpenmrsConstants.UNIX_BASED_OPERATING_SYSTEM) {
204 204             wd = null;
205 205 +       }
206 206
207 207
208     -       Process p = (wd != null) ? Runtime.getRuntime().exec(cmdWithArguments, null, wd) : Runtime.getRuntime().exec(
209     -           cmdWithArguments);
    208 +               ProcessBuilder pb = new ProcessBuilder(cmdWithArguments);
    209 +               if(wd != null)
    210 +                   pb.dirctory(wd);
    211 +               pb.start();
210 212
211 213                 out.append("Normal cmd output:\n");
212 214         Reader reader = new InputStreamReader(p.getInputStream(), StandardCharsets.UTF_8);
```

V. Mock Bug Report

Bug Fix 1: SQL Injection Vulnerability

5.1.1 Vulnerability Name: SQL Injection

5.1.2 Business Impact: Failure to fix this vulnerability could drastically affect OpenMRS as this vulnerability can be easily exploited by a malicious user. In case a malicious user exploits this vulnerability, they can interfere with the different concepts of the database which define the concept dictionary. As certain tables define their properties based on these concepts, it could create a cascade effect over the entire database corrupting different rows and columns of the database thereby rendering the application completely useless. This can have long term effects even if the database is replicated because the data definition itself is being modified.

5.1.3 Affected component: One of the major component affected by this vulnerability being exploited is the database itself although there are other minor components that are affected as well.

5.1.4 Description: The code utilizes the use of a simple string to query data using variables instead of using prepared statements. This creates the possibility of a SQL Injection attack. The attacker may modify the name of the variable, in this case concept_id or name, such that instead of getting the concept_id, the query returns some garbage value or data from other tables. Moreover, an attacker could modify the query in such a way that the name of the concept or concept_id are changed or deleted entirely. In the worst case, an attacker may modify the query statement in such a way that it drops the table entirely.

5.1.5 Result: Each of the scenarios mentioned above can have severe consequences:

1. When the attacker modifies the name of the variable such as concept_id or name it could return a garbage value, this could be taken as input by some other variable and this variable with the garbage value could possibly apply it to other parts of the database. This creates garbage values in the tables.

2. When the attacker stores this data in a variable and prints it to console, they could possibly modify the query in such a way that it returns private/confidential information from the database after execution and print it. This could expose personal health records as well as personal information of all the users of the application.

3. When the attacker modifies the query to an “insert” or “update” query they could change the concept_id numbers, names, etc. of existing rows. This could corrupt other parts of the database that have key dependencies with the concept table.

4. In the worst case, the attacker may modify and execute the query in such a way that they drop the entire table. This could wipe off the concept table completely. This could cause the entire application to shut down immediately as all patient records, visits, appointments, etc. are dependent on the concept table.

5.1.6 Mitigation: The mitigation of this bug involves using a prepared statement instead of a string with appending variables. This would prevent the user from printing out individual variable names as well as avoid any other form of SQL Injection. This can be achieved by applying the fix show in the screenshot below:

```
...
                for (Map.Entry<String, String[]> e : names.entrySet()) {
                        String locale = e.getKey();
                        for (String name : e.getValue()) {
                                Integer ret = getInt(connection, "select concept_id from
concept_name where name = '" + name
                                                + "' and locale like '" + locale + "%'");
        PreparedStatement stmt = connection.prepareStatement("select concept_id from
concept_name where name = ?");
        stmt.setString(1,name);
                                Integer ret = getInt(connection, stmt);
                                if (ret != null) {
                                        return ret;
                                }
...
         * @return integer resulting from the execution of the sql statement
         * @throws CustomChangeException
         */
        private Integer getInt(JdbcConnection connection, String sql) throws
CustomChangeException {
        private Integer getInt(JdbcConnection connection, PreparedStatement stmt) throws
CustomChangeException {
                Statement stmt = null;
                try {
                        stmt = connection.createStatement();
                        ResultSet rs = stmt.executeQuery(sql);

                        ResultSet rs = stmt.executeQuery(stmt);
                        Integer result = null;

                        if (rs.next()) {
                                result = rs.getInt(1);
                        } else {
                                // this is okay, we just return null in this case
                                log.debug("Query returned no results: " + sql);
                                log.debug("Query returned no results: " + stmt.toString());
                        }

                        if (rs.next()) {
                                log.warn("Query returned multiple results when we expected just
one: " + sql);
                                log.warn("Query returned multiple results when we expected just
one: " + stmt.toString());
                        }

                        return result;
...
```

This modification has to done in the BooleanConceptChangeSet.java file. This fix changes the string based query to a prepared statement and fixes all the variable dependencies associated with the changes made to the main SQL query statement such as passing it to another function.

Bug Fix 2: Fix Command Injection Vulnerability with Process Builder

5.2.1 Vulnerability Name: SQL Injection and Injection

5.2.2 Business Impact: This vulnerability if not taken care of, could drastically affect OpenMRS as this vulnerability can be easily exploited by a malicious user. THe attacker can insert malicious data into the argument to exploit out the data use it to his own ends for deletion, duplication and selling to a third party and so on. Targeted malicious arguments could also delete the entire database making the application to an absolute standstill and thus incurring considerable losses and expenses even if simply to recover the data or perform any immediate emergency actions.

5.2.3 Affected component: The affected component in this case is the database itself as well as the software on a whole to because the injection is taking place in the arguments of the method definition itself, so this results in a heavy loss and a difficulty in narrowing down the source of the bug.

5.2.4 Description: The problem of the bug narrows down to a command injection vulnerability where the arguments after the first argument in the "cmdWithArguments" array which can be changed and replaced with malicious queries to perform arbitrary shell commands and pipes. The malicious attacker could insert untrusted data into the argument array after the first argument to obtain his desired input and thus perform unwanted CRUD operations on the data.

5.2.5 Result: The results of the action going through can be catastrophic. This is because, since the method is an integral part of the code, it is the easiest to hack into and toughest to find at this specific a level such as the method argument. Also, once a malicious query has been entered by the attacker, he can wipe out the database, at a smaller level, he can perform DDOS attacks or perform print commands and give arguments so as to display all the sensitive information. As a matter of fact, the original code, found with Fortify, shows that this method is vulnerable due to the use of 'exec' without checking the user input.

5.2.6 Mitigation: The mitigation for the following code was performed by placing command arguments into a process builder which only treated the follow up array indices as command arguments to the initiated process at the beginning of the command array. This fix prevented any malicious user from performing a command injection attack by attempting to escape the command. This is the suitable way to prevent this kind of an attack and proves to be an effective strategy as pointed out below by the screenshot.

```
188 188
189 189       /**
190 190        * @param cmdWithArguments
191 191        * @param wd
192 192        * @param the string
193 193        * @return process exit value
194 194        */
195 195       private Integer execCmd(File wd, String[] cmdWithArguments, StringBuilder out) throws Exception {
196 196           log.debug("executing command: " + Arrays.toString(cmdWithArguments));
197 197
198 198           Integer exitValue;
199 199
200 200           // Needed to add support for working directory because of a linux
201 201           // file system permission issue.
202 202
203 203           if (!OpenmrsConstants.UNIX_BASED_OPERATING_SYSTEM) {
204 204               wd = null;
205 205 +         }
206 206
207 207
208     -         Process p = (wd != null) ? Runtime.getRuntime().exec(cmdWithArguments, null, wd) : Runtime.getRuntime().exec(
209     -             cmdWithArguments);
    208 +                 ProcessBuilder pb = new ProcessBuilder(cmdWithArguments);
    209 +                 if(wd != null)
    210 +                     pb.dirctory(wd);
    211 +                 pb.start();
210 212
211 213                   out.append("Normal cmd output:\n");
212 214           Reader reader = new InputStreamReader(p.getInputStream(), StandardCharsets.UTF_8);
```

Bug Fix 3: Change SQL Injection Vulnerability to Use a Prepared Statement

5.3.1 Vulnerability Name: SQL Injection for malformed uuid

5.3.2 Business Impact: Failure to fix this vulnerability could drastically affect OpenMRS as this vulnerability can be easily exploited by a malicious user. Exploitation of this vulnerability could result in theft, modification, and deletion of patient data and medical records. Well crafted sql queries could also delete the entire database rendering the OpenMRS application completely unusable. Health care institutions using OpenMRS could no longer rely on records to be accurate, and patients may be prevented from receiving treatment without reliable records and a system to track hospital visits and patient information.

5.3.3 Affected component: One of the major components affected by this vulnerability being exploited is the database itself which allows the database to be modified, have information leaked, and allow information to be deleted. The database touches all aspects of the OpenMRS application and having a method to change information stored within the database could potentially affect any part of the application.

5.3.4 Description: For this bug, the issue involved an SQL injection vulnerability where the uuid could be tampered with to craft arbitrary SQL queries. The original code, found with Fortify, shows that the uuid is appended to the SQL query as a string. The uuid could contain malicious SQL code that escapes the statement and creates its own query potentially leaking data to the attacker or allowing them to modify the database. For example a malicious user may be able to set their uuid to *" or "1" = "1* which could allow an attacker to view the concept id for all users stored in the table.

5.3.5 Result: SQL injections can have a drastic impact on the functionality, reliability and usability of OpenMRS. Concatenating string queries with a uuid that can be modified by the user allows a malicious user to delete database information, steal other patient information, and even make changes to other patient information. The attacker essentially has free reign to make any database changes to perform malicious actions within OpenMRS. Since OpenMRS is open source, this is especially dangerous because the attacker would not have to guess table names and fields, instead they could just search for it in the code with little effort. In a worse case scenario an attacker could remove all tables from the database preventing OpenMRS from functioning as intended.

5.3.6 Mitigation: The mitigation of this bug involves using a prepared statement instead of a string with appending variables. Prepared statements use parameterized queries that prevent a user from escaping the sql and creating their own unique query with a maliciously crafted uuid. This can be achieved by applying the fix show in the screenshot below by replacing the string concatenation with a prepared statement:

```
151 151     private void loadSeverityConcepts(Database database) throws Exception {
152 152         mildConcept = getConceptByGlobalProperty(database, "allergy.concept.severity.mild");
153 153         moderateConcept = getConceptByGlobalProperty(database, "allergy.concept.severity.moderate");
154 154         severeConcept = getConceptByGlobalProperty(database, "allergy.concept.severity.severe");
155 155     }
156 156
157 157     private Integer getConceptByGlobalProperty(Database database, String globalPropertyName) throws Exception {
158 158         JdbcConnection connection = (JdbcConnection) database.getConnection();
159 159         PreparedStatement stmt = connection.prepareStatement("SELECT property_value FROM global_property WHERE property = ?");
160 160         stmt.setString(1, globalPropertyName);
161 161         ResultSet rs = stmt.executeQuery();
162 162         if (rs.next()) {
163 163             String uuid = rs.getString("property_value");
164     -
165     -           rs = stmt.executeQuery("SELECT concept_id FROM concept WHERE uuid = '" + uuid + "'");
    164 +
    165 +           stmt = connection.prepareStatement("SELECT concept_id FROM concept WHERE uuid = ?");
    166 +           stmt.setString(1, uuid);
    167 +           rs = stmt.executeQuery();
166 168             if (rs.next()) {
167 169                 return rs.getInt("concept_id");
168 170             }
169 171         }
170 172
171 173         throw new IllegalStateException("Configuration required: " + globalPropertyName);
172 174     }
173 175 }
```