Enhancing Open-Source Project Security: A Comprehensive Source Code Analysis and Vulnerability Assessment

Shristi Suman
srssuman@ucdavis.edu

Shreya Gundu
sgundu@ucdavis.edu

Kriti Kriti
kkriti@ucdavis.edu

Fall 2023

Abstract

Open-source software is widely used but can contain security vulnerabilities that put users at risk. This paper performs an in-depth security analysis of OpenMRS, an open-source medical records system, using static and dynamic analysis tools to discover vulnerabilities. We utilize SonarQube, Codacy, and Semgrep to thoroughly analyze the OpenMRS codebase. Numerous vulnerabilities are discovered, including SQL injection, Path Traversal, Hardcoded Sensitive Value Exposure, and Insecure Session Management. These can enable cyberattacks resulting in data breaches, service disruption, compliance violations, and other issues. We outline the specifics of vulnerabilities discovered and provide comprehensive remediation guidance tailored to OpenMRS for addressing each vulnerability type. A comparative analysis of the capabilities of each analysis tool is also provided as additional open-source projects evaluate advanced solutions for identifying software flaws early. Through an end-to-end assessment, this paper aims to spur open-source developers towards proactively embedding security in their design and development lifecycle. Strengthening open-source application security is imperative as vulnerabilities undermine trust in these widely used platforms. Our methodology and remediation guidance provides an effective template for other projects seeking to enhance security posture.

Keywords: Static Analysis Tools, Security Vulnerabilities, Open Source Project, Code Analysis, Software Security, Comparative Analysis

1 Introduction

Open-source projects have emerged as a vital catalyst for innovation, fostering collaboration, and democratizing technology across various disciplines in today's dynamic software landscape. These projects have paved the way for the development of diverse and dynamic software solutions across various sectors, underscoring their pivotal significance in the contemporary digital realm alongside their budget-friendly nature.

Nevertheless, despite open-source projects' widespread adoption, there continues to be concerns about their security, mostly because of potential flaws that might seriously jeopardize their integrity and reliability. Vulnerabilities in open-source can lead to data breaches, service disruptions, security incidents and cascading effect on business operations. Consequently, the imperative to conduct meticulous source code analysis and vulnerability assessments has gained prominence, serving as a crucial means to identify and address potential security gaps and fortify the overarching security infrastructure of open-source software.

Against the backdrop of persistent cybersecurity challenges, the proactive identification and mitigation of vulnerabilities hold a critical position in the operational strategies of organizations. Initiatives such as the Common Vulnerabilities and Exposures (CVE) program and the Common Weakness Enumeration (CWE) framework have underscored the critical role of proactive security measures in preempting emerging software weaknesses.

To embrace a proactive security approach, the seamless integration of rigorous security testing and analysis within the fabric of the software development process is imperative. This project's objective is to contribute to the ongoing efforts in bolstering the security of open-source projects by conducting an in-depth analysis of selected open-source software. By leveraging advanced analysis techniques, the project endeavors to identify and evaluate potential vulnerabilities within the source code of these projects, empowering decision-makers in the open-source community and ultimately striving to promote a more secure and resilient open-source ecosystem.

2 Related Work

In the realm of static code analysis for security vulnerabilities, key insights are drawn from several pivotal research papers.[1] "Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?" explores the impact of analyst expertise on vulnerability detection, emphasizing the critical role experience plays in enhancing analysis accuracy. The paper [3] "A Survey of Static Code Analysis Methods for Security Vulnerabilities Detection" provides a comprehensive overview of various analysis methods, categorizing their strengths and limitations, offering a holistic perspective on the landscape of static code analysis for security. Additionally, [4] "Analysis of the Tools for Static Code Analysis" provides a valuable assessment of static analysis tools, aiding organizations in making informed decisions when selecting tools that suit their needs.

Furthermore, the paper [9]"Detecting Security Vulnerabilities with Vulnerability Nets" introduces innovative Vulnerability Nets as a novel approach to security vulnerability detection, expanding the horizons of static code analysis. This technique presents a unique perspective, employing the concept of nets to identify and mitigate security threats in code, thereby contributing to the evolution of static code analysis.

These research papers collectively form the basis for our exploration of static code analysis within the context of software security, encompassing expertise's role, diverse analysis methods, tool assessment, and innovative techniques like Vulnerability Nets.

3 Methodology - Projects and Tools

Enhancing the security posture of open-source ecosystems requires proactive identification of vulnerabilities. This project aims to uncover software weaknesses in selected open-source project through a comprehensive static analysis of its codebase.

3.1 Open Source Project : OpenMRS-Core

We have chosen OpenMRS-Core as the focal point for our project on enhancing open-source project security through comprehensive source code analysis and vulnerability assessment.

OpenMRS-Core [5] is a patient-centric medical record system platform designed to provide healthcare providers with a customizable, free, and open-source electronic medical record (EMR) platform. OpenMRS, structured with key directories like 'api,' 'tools,' 'web,' and 'webapp,' relies on Java, Maven, and Git for a streamlined build process, yielding a deployable WAR file. Docker support enhances deployment efficiency. Emphasizing comprehensive documentation, the project prioritizes developer support and contributions. Its dynamic communication bus and YAML for configuration highlight robust architectural strengths in adaptability and scalability for managing electronic health records. OpenMRS's modular architecture encourages flexibility, customization through deployable modules, and active community involvement.

The choice of OpenMRS-Core aligns with our commitment to contributing to a project that plays a vital role in advancing health care accessibility and quality, particularly in regions with limited resources. By securing the foundation of OpenMRS-Core, we aim to fortify the confidentiality and integrity of sensitive patient information, ensuring that the platform continues to empower healthcare professionals to deliver optimal care in diverse and challenging settings.

Website: OpenMRS

3.2 Static Analysis Tools

In the pursuit of fortifying software security, this project harnesses the formidable capabilities of robust static analyzers, notably SonarQube, Semgrep, and Codacy.

Deployed to scrutinize the project's source code comprehensively, these automated tools function as vigilant guardians, meticulously uncovering potential security flaws embedded in every line.Aligned with the Common Weakness Enumeration (CWE) catalog, the analyzers identify vulnerabilities ranging from input validation lapses and SQL injections to hardcoded secrets and resource management oversights. This exhaustive scanning process ensures broad coverage across diverse languages and architectures, unveiling risks that might otherwise remain concealed. Each finding is meticulously documented, providing developers with crucial context regarding the affected areas and empowering them to take prompt and informed mitigation actions. The overarching objective is to conduct relentless static analysis, arming project developers with actionable insights that pave the way for the development of resilient security fixes and the unwavering preservation of software integrity.This research aims to advance open source security through methodical, automated vulnerability discovery and responsible disclosure.

3.2.1 Semgrep

Semgrep, [7] a swift and powerful code scanning tool, is recognized for its speed, open-source nature, and versatility in static analysis, making it a cornerstone in our vulnerability analysis strategy. Its exceptional speed and precision ensure meticulous code scanning without compromising efficiency, employing a rule-based approach that simplifies rule creation and eliminates the need for complex abstract syntax trees. Operating locally, Semgrep respects privacy concerns by analyzing code without uploads. The Semgrep ecosystem includes components like Semgrep Cloud Platform, Semgrep Code, Semgrep Supply Chain, and Semgrep Secrets, each tailored to address specific security aspects. For instance, Semgrep Code supports over 30 languages, ensuring comprehensive coverage across our diverse codebase.

Our decision to integrate Semgrep is a strategic choice, driven by its technical prowess, speed, precision, compatibility with our varied codebase, and commitment to privacy. This integration not only enhances vulnerability detection capabilities but also aligns with our proactive security measures, emphasizing Semgrep's rule-based, language-agnostic approach and developer-friendly features, making it an ideal choice for empowering vulnerability analysis and ensuring software integrity.

3.2.2 Codacy

Codacy [2], a robust code analysis tool, plays a pivotal role in our dedication to software quality and security, automatically identifying and addressing code issues, vulnerabilities, and maintainability concerns. Integral to our development pipeline, Codacy excels in streamlining code quality assessment through its automated code review process. Seamlessly integrating with our version control systems, it analyzes every commit and pull request, ensuring adherence to industry best practices and coding standards. Codacy's detailed feedback accelerates our code review process and provides developers with valuable insights into potential improvements and areas of concern.

Our decision to incorporate Codacy is rooted in its effectiveness in enforcing coding standards, catching issues early in the development lifecycle, and promoting a culture of continuous improvement, ultimately enhancing our capacity to deliver reliable, secure, and high-quality software to our users.

3.2.3 SonarQube

SonarQube,[8] an essential tool recognized for extensive code quality and security analysis capabilities, has garnered prominence in our project. This platform is renowned for meticulous scanning, vulnerability identification, and bug detection. SonarQube's detailed reporting system excels in uncovering critical security threats, including SQL injection and cross-site scripting (XSS) vulnerabilities. The platform also stands out for detecting code smells and areas impacting maintainability and readability. We are considering integrating SonarQube into our analysis pipeline to fortify our codebase, align with industry-standard security practices, and proactively shield against potential breaches.

Our decision is driven by SonarQube's proven ability to provide comprehensive insights, making it a strategic choice for enhancing our code's overall quality, readability, and ease of maintenance. Integrating SonarQube aligns with our objective of bolstering code quality, fortifying security measures, and adhering to industry best practices, ultimately resulting in a more robust, secure, and maintainable codebase.

4 Experimental Results

4.1 Semgrep results

These are a few of the vulnerabilities that we detected from Semgrep.

- **XML External Entity (XXE) vulnerability:**
  XML External Entity (XXE) vulnerability is a type of security issue that occurs when an application processes XML input from untrusted sources in an insecure manner. It allows attackers to interfere with the XML parsing process by injecting external entities, which can lead to various exploits.

  In simpler terms, when an application parses XML data, it might reference entities defined externally in the XML document or in an external entity declaration. Attackers can abuse this feature by injecting malicious code or references to external entities that can retrieve sensitive data, perform denial-of-service attacks, or execute arbitrary code on the server.

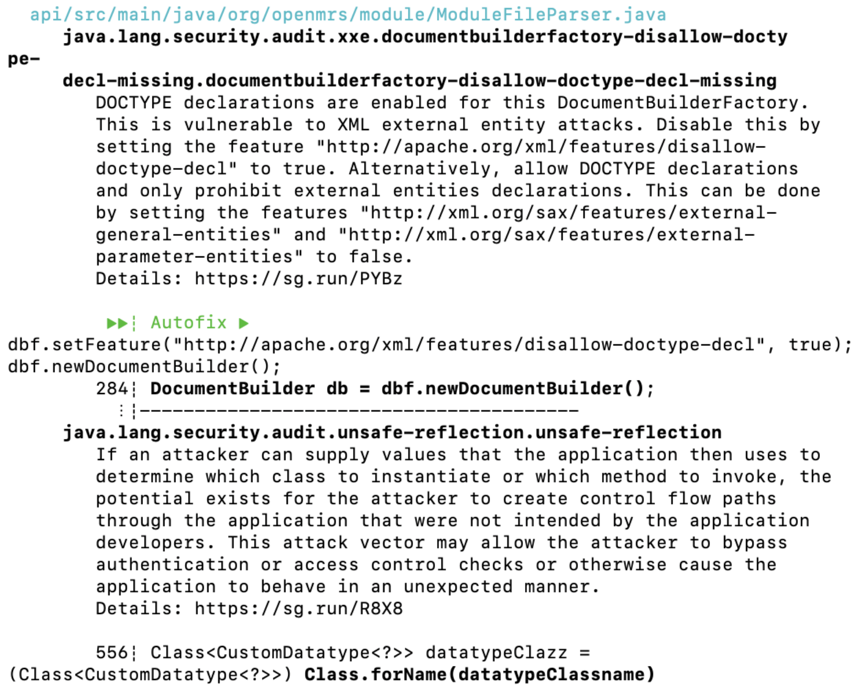
  XXE vulnerabilities should be considered serious due to their potential to compromise data integrity, confidentiality, and even the system's availability. Mitigating these vulnerabilities promptly and employing best practices for secure XML processing is crucial to minimize their impact.

  These are the types of XXE vulnerabilities that we found:

  1. **documentbuilderfactory**:
     In OpenMRS, we found a total of 16 instances of this type, which are spread over documentbuilderfactory-disallow-doctype-decl-missing, owasp.java.xxe.javax.xml.pa

transformerfactory-dtds-not-disabled, documentbuilderfactory-xxe-parameter-entity, and documentbuilderfactory-xxe.

The findings showed that the application processed XML using ‘DocumentBuilder‘. However, if this is used in an insecure context without proper input validation or security measures, it could potentially be vulnerable to XML External Entity (XXE) attacks.



```
    api/src/main/java/org/openmrs/module/ModuleFileParser.java
       java.lang.security.audit.xxe.documentbuilderfactory-disallow-docty
pe-
       decl-missing.documentbuilderfactory-disallow-doctype-decl-missing
          DOCTYPE declarations are enabled for this DocumentBuilderFactory.
          This is vulnerable to XML external entity attacks. Disable this by
          setting the feature "http://apache.org/xml/features/disallow-
          doctype-decl" to true. Alternatively, allow DOCTYPE declarations
          and only prohibit external entities declarations. This can be done
          by setting the features "http://xml.org/sax/features/external-
          general-entities" and "http://xml.org/sax/features/external-
          parameter-entities" to false.
          Details: https://sg.run/PYBz

           ▶▶┆ Autofix ▶
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
dbf.newDocumentBuilder();
          284┆ DocumentBuilder db = dbf.newDocumentBuilder();
            ⋮┆----------------------------------------
       java.lang.security.audit.unsafe-reflection.unsafe-reflection
          If an attacker can supply values that the application then uses to
          determine which class to instantiate or which method to invoke, the
          potential exists for the attacker to create control flow paths
          through the application that were not intended by the application
          developers. This attack vector may allow the attacker to bypass
          authentication or access control checks or otherwise cause the
          application to behave in an unexpected manner.
          Details: https://sg.run/R8X8

          556┆ Class<CustomDatatype<?>> datatypeClazz =
(Class<CustomDatatype<?>>) Class.forName(datatypeClassname)
```

Mitigation Strategies:

(a) **Disallow External Entity Resolution:**

  - **DocumentBuilderFactory Configuration:**
    - ∗ Use **’dbf.setFeature(”http://javax.xml.XMLConstants/feature/secure-processing”, true)’**:
      This feature ensures the secure processing mode, which disables XML external entities by default, enhancing security.
    - ∗ Apply **’dbf.setExpandEntityReferences(false)’**:
      Disabling entity expansion prevents the parser from replacing entity references with their values, mitigating XXE risks.

(b) **Input Validation Whitelisting:**

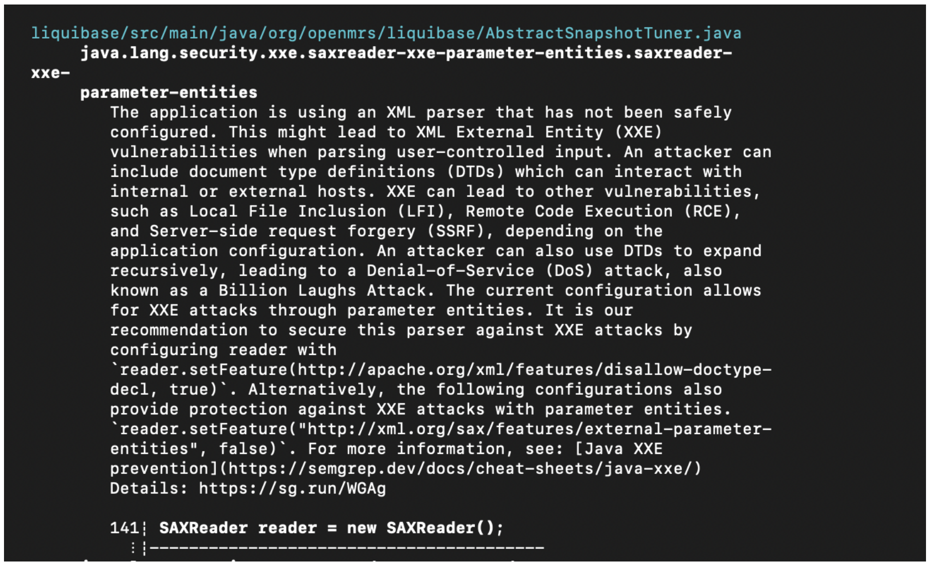
  - **Validate Input XML:** Check that input XML conforms to a predefined schema or structure, ensuring it contains only expected content.
  - **Whitelisting Entities/Elements:** Specify a whitelist of allowed entities or elements within the XML document, restricting what the parser can process.

(c) **Utilize Specific Parsers:**
   - **Use 'SAXParser':** This parser inherently restricts entity expansion and is known for its inherent protection against XXE vulnerabilities.

(d) **Least Privilege Access:** Execute XML processing with the least privilege access:
   - Limit the permissions granted for XML processing to minimize potential damage if a vulnerability is exploited.

2. **saxreader-xxe:**
In OpenMRS, we found a total of 4 instances of this type, which are spread over saxreader-xxe and saxreader-xxe-parameter-entities.

The vulnerability is associated with the usage of SAXReader, particularly in the context of XML External Entity (XXE) attacks. XXE vulnerabilities arise when SAXReader processes XML content that includes external entity references. These external entities can be manipulated by an attacker to access local or sensitive files, execute arbitrary code, or perform other malicious actions. The vulnerability might occur if the XML parsing process isn't properly configured to prevent the resolution of external entities, which could allow an attacker to exploit this functionality.



```
liquibase/src/main/java/org/openmrs/liquibase/AbstractSnapshotTuner.java
     java.lang.security.xxe.saxreader-xxe-parameter-entities.saxreader-
xxe-
     parameter-entities
        The application is using an XML parser that has not been safely
        configured. This might lead to XML External Entity (XXE)
        vulnerabilities when parsing user-controlled input. An attacker can
        include document type definitions (DTDs) which can interact with
        internal or external hosts. XXE can lead to other vulnerabilities,
        such as Local File Inclusion (LFI), Remote Code Execution (RCE),
        and Server-side request forgery (SSRF), depending on the
        application configuration. An attacker can also use DTDs to expand
        recursively, leading to a Denial-of-Service (DoS) attack, also
        known as a Billion Laughs Attack. The current configuration allows
        for XXE attacks through parameter entities. It is our
        recommendation to secure this parser against XXE attacks by
        configuring reader with
        `reader.setFeature(http://apache.org/xml/features/disallow-doctype-
        decl, true)`. Alternatively, the following configurations also
        provide protection against XXE attacks with parameter entities.
        `reader.setFeature("http://xml.org/sax/features/external-parameter-
        entities", false)`. For more information, see: [Java XXE
        prevention](https://semgrep.dev/docs/cheat-sheets/java-xxe/)
        Details: https://sg.run/WGAg

        141┆ SAXReader reader = new SAXReader();
          ⋮┆----------------------------------------
```

The following mitigation techniques can be used by the Codebase administrator to remove XXE vulnerabilities from the code.

Mitigation Strategies:

(a) **Configuration Hardening:**
'SAXReader' Configuration:
   - Use **'saxReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true)'**:

Disallows the declaration of DTDs, preventing entity expansion based on DTDs.

  - Apply **'saxReader.setFeature("http://xml.org/sax/features/external-general-entities", false)'**:
    Blocks external general entities, reducing the risk of XXE attacks.

(b) **Input Sanitization**:
Sanitize and Validate XML Input:
  - Thoroughly inspect and validate incoming XML to remove or block any external entity references.
  - Apply strict validation to block parameter entities and external DTDs.

(c) **Parser Configuration**:
Configure Parsers to Disallow External Entities:
Explicitly set parser configurations to disallow the use of external entities during XML parsing.

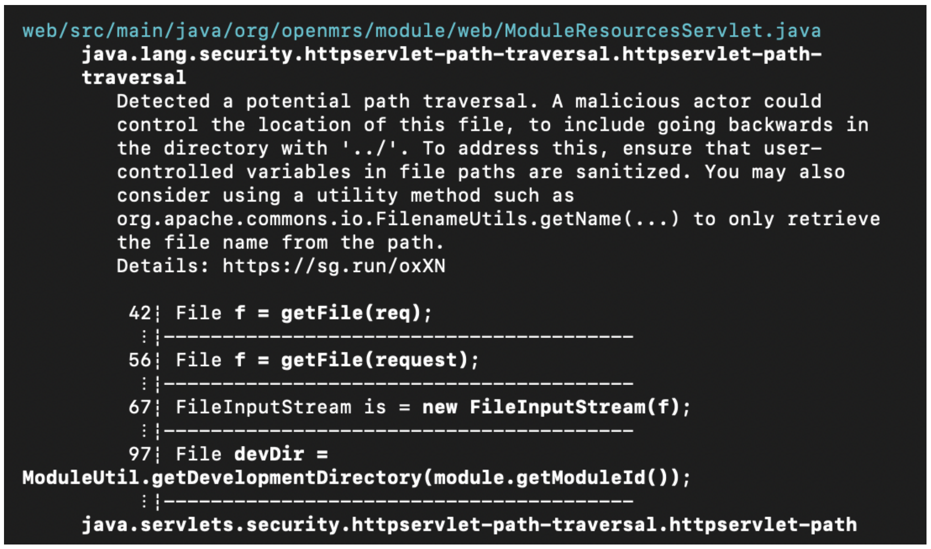
(d) **Secure Parser Usage**:
Employ Parsers with Built-in XXE Protection:
Choose XML processing libraries or parsers explicitly designed to prevent XXE attacks, prioritizing security features.

(e) **Patch Updates**:
Regularly Update XML Processing Libraries:
Keep XML processing libraries up-to-date to incorporate security patches, fixing known vulnerabilities.

The detailed mitigation steps encompass various technical configurations, coding practices, and proactive measures to mitigate XXE vulnerabilities, aiming to strengthen the application's security posture against potential XML-based attacks.

- **Path Traversal:**
  A path traversal attack (also known as directory traversal) aims to access files and directories that are stored outside the web root folder. By manipulating variables that reference files with "dot-dot-slash (../)" sequences and its variations or by using absolute file paths, it may be possible to access arbitrary files and directories stored on the file system including application source code or configuration and critical system files. It should be noted that access to files is limited by system operational access control (such as in the case of locked or in-use files on the Microsoft Windows operating system). This attack is also known as "dot-dot-slash", "directory traversal", "directory climbing" and "backtracking". These vulnerabilities should be addressed with priority due to their potential for data exposure and unauthorized access.[6]

  In OpenMRS, we found a total of 9 instances of httpservlet-path-traversal.



```
web/src/main/java/org/openmrs/module/web/ModuleResourcesServlet.java
     java.lang.security.httpservlet-path-traversal.httpservlet-path-
     traversal
        Detected a potential path traversal. A malicious actor could
        control the location of this file, to include going backwards in
        the directory with '../'. To address this, ensure that user-
        controlled variables in file paths are sanitized. You may also
        consider using a utility method such as
        org.apache.commons.io.FilenameUtils.getName(...) to only retrieve
        the file name from the path.
        Details: https://sg.run/oxXN

         42┆ File f = getFile(req);
          ⋮┆----------------------------------------
         56┆ File f = getFile(request);
          ⋮┆----------------------------------------
         67┆ FileInputStream is = new FileInputStream(f);
          ⋮┆----------------------------------------
         97┆ File devDir =
ModuleUtil.getDevelopmentDirectory(module.getModuleId());
          ⋮┆----------------------------------------
     java.servlets.security.httpservlet-path-traversal.httpservlet-path
```

The findings showed that the application builds a file path from user input or potentially untrusted data. An attacker can manipulate the file path that the application uses to access files leading to path traversal vulnerability. Without proper user input validation and sanitization of file paths, sensitive files such as configuration or user data can be accessed, potentially creating or overwriting files.

The following mitigation techniques can be used by the Codebase administrator to remove Path Traversal from the code.

Mitigation Strategies:

1. **Input Sanitization & Validation**:
   Validate and Sanitize User Input:
   - Perform strict validation and sanitization of user-supplied data used to construct file paths.
   - Filter input to prevent the inclusion of path traversal characters such as '../' or any variations thereof.
2. **Use Whitelisting and Allowed Paths**:
   Implement Whitelists:
   - Define whitelists of allowed file paths or directories that the application can access.
   - Enforce access restrictions to only permitted file paths.
3. **File Path Construction**:
   Avoid Direct User Input in File Paths:
   - Refrain from directly concatenating or constructing file paths using untrusted user input.
   - Use predefined, validated references or predefined paths instead of dynamic user inputs to build file paths.

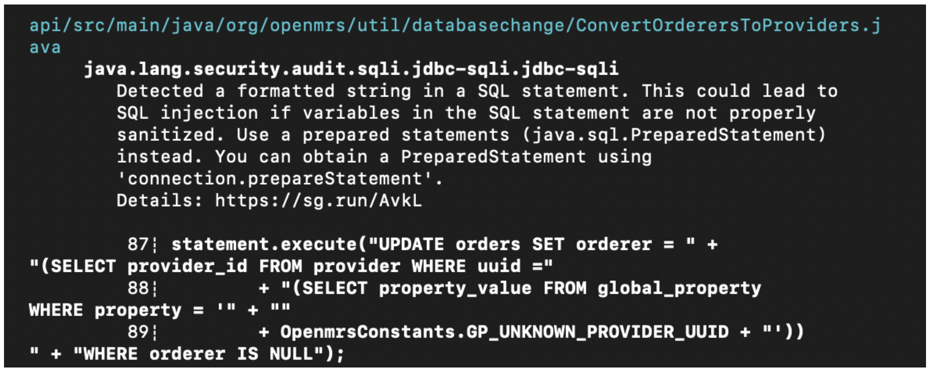
4. **Use Safe APIs or Libraries**:
   Leverage Safe File Access APIs:
   - Utilize file access APIs or libraries that prevent path traversal vulnerabilities.
   - Choose APIs that manage file operations securely, handling path validations and restrictions.
5. **Access Controls Permissions**:
   Implement Strong Access Controls:
   - Ensure robust access controls are in place, restricting access only to authorized files and directories.
   - Apply principle of least privilege: grant only necessary access rights to files and directories.
6. **File System Hardening**:
   Restrict File System Access:
   - Configure the file system to prevent unauthorized access, considering file and directory permissions.
   - Harden the server environment to minimize the impact of potential path traversal attacks.

Addressing Path Traversal vulnerabilities requires a multi-layered approach, focusing on input validation, secure file handling practices, access controls, and continuous monitoring. These measures collectively mitigate the risks associated with unauthorized file access and data exposure through path traversal attacks.

- **SQL Injection:**
  SQL Injection occurs when untrusted data is inserted into SQL queries without proper validation or sanitization. This could allow an attacker to send a valid SQL statement by manipulating the semantics of the initial query in the application. This will enable attackers to modify SQL queries to bypass authentication, retrieve sensitive information, or even delete or modify database records.

  SQL Injection is considered a critical security issue as it directly impacts the confidentiality, integrity, and availability of data within a system.

  In OpenMRS, we found a total of 7 instances of jdbc-sqli. JDBC-SQLI implies that SQL queries are being constructed or executed in a way that allows an attacker to manipulate or inject SQL commands for Java-based applications.



```
api/src/main/java/org/openmrs/util/databasechange/ConvertOrderersToProviders.j
ava
     java.lang.security.audit.sqli.jdbc-sqli.jdbc-sqli
        Detected a formatted string in a SQL statement. This could lead to
        SQL injection if variables in the SQL statement are not properly
        sanitized. Use a prepared statements (java.sql.PreparedStatement)
        instead. You can obtain a PreparedStatement using
        'connection.prepareStatement'.
        Details: https://sg.run/AvkL

             statement.execute("UPDATE orders SET orderer = " +
"(SELECT provider_id FROM provider WHERE uuid ="
                     + "(SELECT property_value FROM global_property
WHERE property = '" + ""
                     + OpenmrsConstants.GP_UNKNOWN_PROVIDER_UUID + "'))
" + "WHERE orderer IS NULL");
```

The analyzer detected a formatted string in a SQL statement. In those code snippets, the application is being used to dynamically construct SQL queries using concatenated strings or explicitly mentioned query parameters together with input keywords. These open up the avenue for multiple attacks leading to unauthorized access, manipulation, or even deletion of data.

The following mitigation techniques can be used by the Codebase administrator to remove SQL Injection from the code.

Mitigation Strategies:

1. **Parameterized Queries and Prepared Statements**:
   Use Parameterized Queries:
   - Employ parameterized queries or prepared statements provided by database APIs.
   - Bind parameters to SQL queries rather than concatenating user input directly into SQL statements.
2. **Input Validation and Sanitization**:
   Validate and Sanitize User Input:
   - Validate and sanitize input data to ensure it adheres to expected formats and doesn't contain malicious SQL commands.
   - Use input validation to reject or neutralize unexpected SQL characters or queries.
3. **Least Privilege Principle**:
   Limit Database Access Privileges:
   - Implement the principle of least privilege, ensuring database user accounts have only necessary permissions.
   - Restrict database access to perform only required operations, preventing unauthorized queries.
4. **Escaping Special Characters**:
   Escape User Input:
   - Escape or neutralize special characters such as quotes ('), semicolons (;), and dashes (-) that can alter query logic.

    - Use built-in escaping functions or libraries provided by the database or programming language.

5. **Whitelist Allowed Characters**:
   Implement Whitelisting:
    - Define whitelists of allowed characters or patterns for user input, validating against these patterns before using in SQL queries.
    - Restrict input to known safe characters, rejecting unexpected or unsafe input.
6. **ORMs and Safe Libraries**:
   Use Object-Relational Mapping (ORM) Tools:
    - Employ ORM frameworks that automatically handle parameter binding and escaping, reducing the risk of SQL injection vulnerabilities.
    - Choose libraries or frameworks that enforce secure query building practices.

Addressing SQL Injection vulnerabilities necessitates a proactive approach involving stringent input validation, secure query building practices, and continuous monitoring to mitigate the risks associated with unauthorized SQL command injections.

4.2 Codacy results

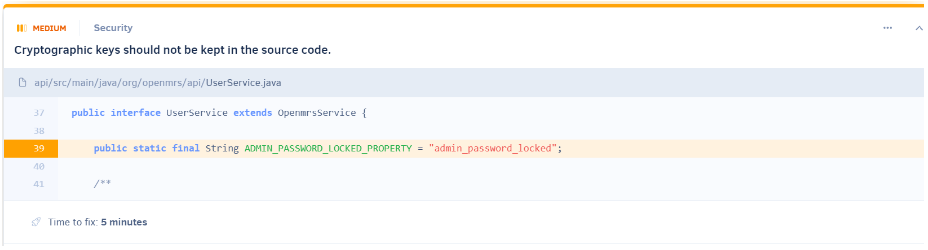
Codacy detected multiple vulnerabilities apart from the ones already listed above here are a few of the other vulnerabilities that we detected from Codacy.

- **Hardcoded Sensitive Values Exposure:**

  The hard code key suspicious value vulnerability signifies a critical security flaw where cryptographic keys, passwords, or other confidential information are directly embedded or hardcoded within the Java source code. This practice poses a substantial risk to the security of an application or system. By including sensitive data within the source code, it becomes easily accessible to anyone who has access to the codebase, including unauthorized individuals or attackers. Consequently, this significantly amplifies the potential for unauthorized access, data breaches, or system compromise, especially in scenarios where the code is shared openly in environments like open source projects or within an enterprise setting.

  Adopting secure storage practices for sensitive information instead of hardcoding keys or passwords in the source code, developers should utilize separate configuration files, environment variables, or dedicated key management systems (keystores) to securely store and manage this sensitive data. Employing these secure storage mechanisms helps in safeguarding the confidentiality of cryptographic keys and passwords, limiting access only to authorized personnel or systems, and reducing the risk of unauthorized exposure or compromise.

In OpenMRS, we found a total of 91 instances of java-password-rule-HardcodeKeySuspiciousVa
vulnerability.



Cryptographic keys should not be kept in the source code.

api/src/main/java/org/openmrs/api/UserService.java

```
public interface UserService extends OpenmrsService {

    public static final String ADMIN_PASSWORD_LOCKED_PROPERTY = "admin_password_locked";

    /**
```

Time to fix: **5 minutes**

The following mitigation techniques can be used by the Codebase administrator to remove Hardcoded Sensitive Values Exposure from the code.

Mitigation Strategies:

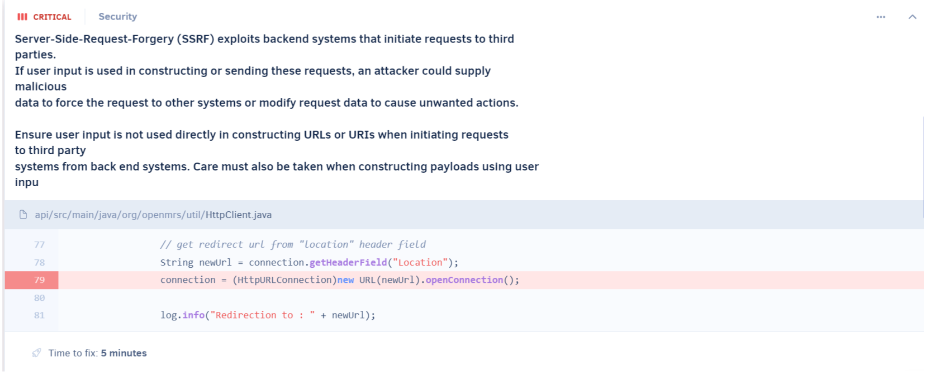
1. **Secure Storage Practices**:
   Avoid Hardcoding Sensitive Information:
   - Refrain from embedding cryptographic keys, passwords, or other confidential data directly into the source code.
   - Avoid storing sensitive information in plain text within the codebase.
2. **Use Separate Configuration Files**:
   Utilize External Configuration Files:
   - Store sensitive information in separate configuration files, external properties files, or JSON files.
   - These files should be securely managed and accessible only to authorized personnel or systems.
3. **Environment Variables**:
   Employ Environment Variables:
   - Leverage environment variables to store sensitive information like passwords or API keys.
   - This approach ensures data security and separation from the codebase.
4. **Dedicated Key Management Systems (Keystores)**:
   Use Key Management Systems:
   - Employ dedicated key management systems (e.g., keystrokes) to securely store and manage cryptographic keys, credentials, or other sensitive data.
   - These systems offer encryption, access control, and auditing functionalities for sensitive information.
5. **Encryption and Hashing**:
   Encrypt Sensitive Data:
   - Encrypt sensitive information before storage, using strong encryption algorithms.

  - Hash passwords or sensitive values using strong, salted hashing techniques before storing or transmitting them.

6. **Secure Access Controls**:
   Limit Access to Sensitive Information:
   - Implement strict access controls and permissions to restrict access to sensitive data to authorized users or services only.
   - Employ role-based access control (RBAC) to define and enforce access policies.

By adopting these secure storage mechanisms and best practices, developers can protect sensitive information, reduce the risk of unauthorized exposure or compromise, and maintain the confidentiality of cryptographic keys, passwords, and other confidential data within their applications or systems.

- **Server-Side Request Forgery (SSRF) Vulnerability:**
  This vulnerability pertains to Server-Side Request Forgery (SSRF) in Java applications. SSRF occurs when a backend system initiates HTTP requests to third-party resources based on user-provided input, allowing attackers to manipulate these requests maliciously. Exploiting SSRF, attackers can trick the server into making unintended requests to internal systems, retrieve sensitive data, or perform actions on behalf of the server.



Server-Side-Request-Forgery (SSRF) exploits backend systems that initiate requests to third
parties.
If user input is used in constructing or sending these requests, an attacker could supply
malicious
data to force the request to other systems or modify request data to cause unwanted actions.

Ensure user input is not used directly in constructing URLs or URIs when initiating requests
to third party
systems from back end systems. Care must also be taken when constructing payloads using user
inpu

api/src/main/java/org/openmrs/util/HttpClient.java

```java
// get redirect url from "location" header field
String newUrl = connection.getHeaderField("Location");
connection = (HttpURLConnection)new URL(newUrl).openConnection();

log.info("Redirection to : " + newUrl);
```

Time to fix: 5 minutes

The vulnerability is exposed in the code snippet in image, it's essential to implement rigorous input validation and careful handling of user-supplied data. Specifically, in this scenario, before utilizing the retrieved URL to establish a new connection, it is imperative to thoroughly validate the URL and ensure that it points to trusted and intended destinations. This involves validating the URL structure, verifying the protocol, and potentially restricting access to specific domains or IP ranges based on a predefined whitelist. By implementing stringent validation measures and employing a secure HTTP client capable of blocking certain IP ranges or unsafe protocols, developers can significantly reduce the risk of SSRF attacks and enhance the overall security of the application.

In OpenMRS, we found a total of 3 instances of java-ssrf-rule-SSRF vulnerability.

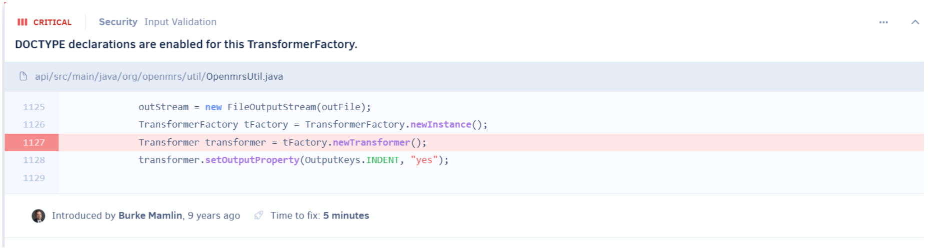
The following mitigation techniques can be used by the Codebase administrator to remove Server-Side Request Forgery (SSRF) Vulnerability from the code.

Mitigation Strategies:

1. **Rigorous Input Validation**:
   Thoroughly Validate User-Supplied URLs:
   - Implement stringent input validation to verify the structure and integrity of user-provided URLs.
   - Validate URL syntax, check for expected protocols (e.g., HTTP, HTTPS), and ensure URLs point to trusted, intended destinations.
2. **Restrictive Access Controls**:
   Restrict Access to Specific Domains or IP Ranges:
   - Implement restrictions based on a predefined whitelist of allowed domains or IP ranges.
   - Deny access to untrusted or internal network resources by restricting outgoing requests to predefined safe destinations.
3. **Secure HTTP Client Configuration**:
   Utilize Secure HTTP Client Libraries:
   - Employ secure HTTP client libraries or frameworks capable of blocking certain IP ranges or disallowed protocols.
   - Configure HTTP clients to enforce security measures such as restricting redirections or disallowing access to local/internal resources.
4. **Protocol and Destination Verification**:
   Verify URL Protocols and Destinations:
   - Ensure URLs use trusted protocols (e.g., HTTP(S)) and are intended for legitimate, authorized destinations.
   - Validate URL paths to prevent access to internal resources or sensitive endpoints.
5. **Content Inspection and Whitelisting**:
   Inspect and Whitelist Allowed Content:
   - Perform content inspection to verify the legitimacy of fetched data before processing or forwarding responses.
   - Maintain a whitelist of allowed content types or expected responses, rejecting unexpected or potentially harmful content.
6. **Logging and Monitoring**:
   Implement Comprehensive Logging:
   - Log outgoing requests made by the backend system, especially those initiated based on user-supplied input.
   - Monitor and review logs to detect any suspicious or unauthorized outgoing requests or unusual patterns.

By implementing these comprehensive measures, developers can significantly reduce the risk of SSRF attacks, prevent unauthorized access to internal resources, and enhance the overall security posture of the Java application.

- **TransformerFactory DTDs Not Disabled Vulnerability:**
  The "transformerfactory-dtds-not-disabled" vulnerability refers to a security risk associated with XML processing when using TransformerFactory in Java without disabling Document Type Definitions (DTDs). In XML processing, DTDs can pose a security threat known as XML External Entity (XXE) attacks when not properly disabled. XXE attacks occur when an attacker injects malicious XML entities, leading to potential data exfiltration, server-side request forgery (SSRF), or other exploits.



CRITICAL | Security Input Validation

DOCTYPE declarations are enabled for this TransformerFactory.

api/src/main/java/org/openmrs/util/OpenmrsUtil.java

```
		outStream = new FileOutputStream(outFile);
		TransformerFactory tFactory = TransformerFactory.newInstance();
		Transformer transformer = tFactory.newTransformer();
		transformer.setOutputProperty(OutputKeys.INDENT, "yes");
```

Introduced by Burke Mamlin, 9 years ago   Time to fix: 5 minutes

  In the provided code snippet, the use of TransformerFactory and Transformer classes indicates XML transformation functionality. However, the absence of explicit measures to disable DTDs suggests a potential vulnerability. When TransformerFactory is created without explicitly disabling DTDs, it might leave the application susceptible to XXE attacks if the input XML contains external entities that an attacker could exploit. To mitigate the "transformerfactory-dtds-not-disabled" vulnerability, it's crucial to disable DTD processing explicitly when using TransformerFactory. This can be achieved by setting certain features or properties to prevent the resolution and processing of external entities.

  In OpenMRS, we found a total of 4 instances of the "transformerfactory-dtds-not-disabled" vulnerability.

  The following mitigation techniques can be used by the Codebase administrator to remove TransformerFactory DTDs Not Disabled Vulnerability from the code.

  Mitigation Strategies:

  1. **Explicitly Disable DTD Processing**:
     Disable DTD Resolution in TransformerFactory:
     – Configure the TransformerFactory explicitly to disable Document Type Definitions (DTDs).
     – Set features or properties to prevent the resolution and processing of external entities within XML.
  2. **Use Secure XML Processing Settings**:
     Enforce Secure Processing Settings:

  - Set secure processing features ('FEATURE_SECURE_PROCESSING') in the TransformerFactory to disable DTDs by default.
  - Use 'setFeature(XMLConstants.FEATURE_DISALLOW_DOCTYPE_DECL, true)' to disallow DTD declarations.

3. **Validate and Sanitize XML Input**:
   Thoroughly Validate Input XML:
   - Perform strict validation of incoming XML data to ensure it adheres to predefined schemas and does not contain external entities.
   - Sanitize XML inputs to remove or block any external entity references.
4. **Apply Least Privilege Principle**:
   Limit XML Processing Permissions:
   - Execute XML processing with the least privilege access necessary to mitigate potential XXE attacks.
5. **Regular Updates and Patch Management**:
   Keep XML Processing Libraries Updated:
   - Regularly update XML processing libraries and frameworks to benefit from security patches and fixes that address XXE vulnerabilities.
6. **Static Code Analysis and Audits**:
   Conduct Code Reviews and Analysis:
   - Perform static code analysis and thorough code reviews to identify instances of TransformerFactory usage without disabled DTDs.
   - Audit the codebase to ensure compliance with secure XML processing practices.
7. **Automated Testing for XXE Vulnerabilities**:
   Conduct Automated Vulnerability Testing:
   - Use automated tools specialized in detecting XXE vulnerabilities to scan and identify potential weaknesses in XML processing.

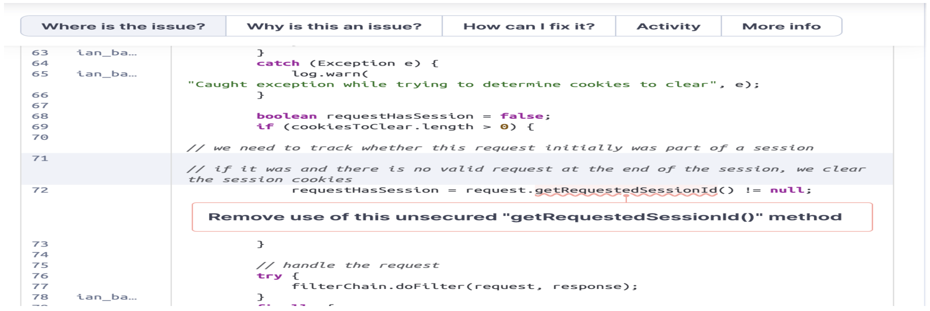
By following these measures and explicitly disabling DTDs in TransformerFactory settings, developers can significantly reduce the risk of the TransformerFactory DTDs Not Disabled Vulnerability, mitigating the potential for XXE attacks and enhancing the security of XML processing within Java applications.

4.3 SonarQube results

SonarQube's analysis has corroborated vulnerabilities identified by Codacy and Semgrep, providing a consistent validation of common issues. In addition, SonarQube has brought to light distinct vulnerabilities not previously reported by the other tools, showcasing its unique contribution to a thorough and comprehensive assessment.

- **Session Management Vulnerability:**
  Session management vulnerability arises when an application inadequately handles user sessions, potentially leading to unauthorized access, data breaches, or

denial of service. Common issues include insecure session IDs, inadequate expiration policies, and susceptibility to session hijacking.



```
        }
        catch (Exception e) {
            log.warn(
"Caught exception while trying to determine cookies to clear", e);
        }

        boolean requestHasSession = false;
        if (cookiesToClear.length > 0) {

// we need to track whether this request initially was part of a session

// if it was and there is no valid request at the end of the session, we clear
the session cookies
            requestHasSession = request.getRequestedSessionId() != null;
```

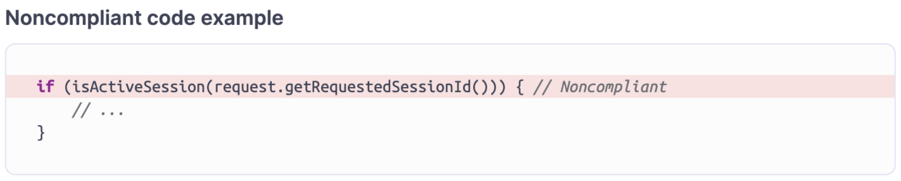
Remove use of this unsecured "getRequestedSessionId()" method

```
        }

        // handle the request
        try {
            filterChain.doFilter(request, response);
        }
```

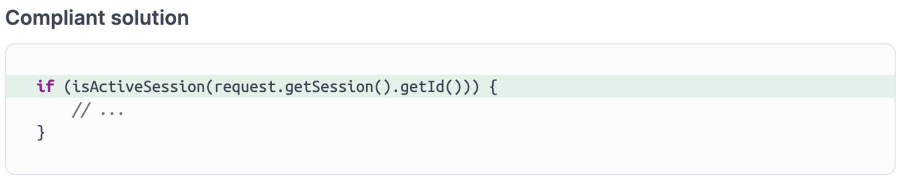
In OpenMRS, we found this issue to be prevalent in the method HttpServletRequest.getRequestedSessionId(), which returns the session ID specified by the client. The client can transmit this ID through a cookie or a URL parameter. The concern arises from the end user's ability to manually update the session ID in an HTTP request. This introduces a potential security risk as attackers could manipulate the session ID, leading to authentication bypass, unauthorized access, and even session hijacking. If exploited, this vulnerability may enable attackers to perform actions on behalf of other users, access sensitive information, or compromise the integrity of user sessions within the application.

The severity of the impact is contingent on factors such as the sensitivity of the data involved, the overall application architecture, and the effectiveness of existing mitigation measures. It underscores the importance of robust session management and stringent security practices to mitigate the potential risks associated with session ID manipulation.



Noncompliant code example

```
if (isActiveSession(request.getRequestedSessionId())) { // Noncompliant
    // ...
}
```



Compliant solution

```
if (isActiveSession(request.getSession().getId())) {
    // ...
}
```

For the present scenario, 'HttpServletRequest.getRequestedSessionId()' is fetching a session ID, which is subsequently employed to check the status of the provided session. Since this value is supplied by a user, its validity is not assured.

In contrast, the compliant illustration utilizes the server's session ID to ascertain the session's activity. Furthermore, if the user's request lacks a valid ID, 'getSession()' generates a new session, enhancing security by relying on server-generated, validated session IDs.

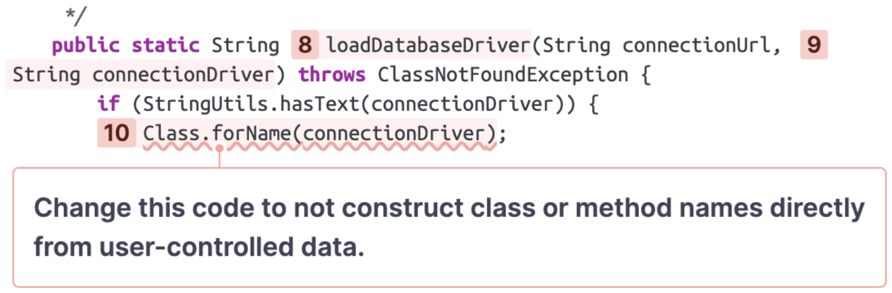
Some of the mitigation techniques that can be used by the Codebase administrator to remove Session Management Vulnerability from the code are as follows:

Mitigation Strategies:

1. **Secure Session ID Generation**:
   - Ensure that session IDs are cryptographically strong, unpredictable, and resistant to brute-force attacks.
   - Utilize a secure random number generator and avoid predictable patterns.
2. **Session Expiration Policies**:
   - Implement session timeout mechanisms to automatically invalidate sessions after a defined period of inactivity. This reduces the window of opportunity for attackers to exploit session-related vulnerabilities.
3. **Session Token Protection:**:
   - Employ secure channels (HTTPS) to encrypt session tokens during transmission, preventing interception by attackers.
   - Avoid transmitting session IDs in URLs as they can be exposed in various logs.
4. **Token Regeneration on Authentication:**:
   - After user authentication, generate a new session token to prevent session fixation attacks, ensuring a unique token for each session.
5. **User Authentication Controls:**:
   - Enforce secure authentication methods to mandate user authentication before initiating a session, mitigating risks of unauthorized access and session hijacking.
6. **Session Revocation:**:
   - Establish procedures to invalidate sessions promptly upon suspected compromise or user logout, preventing unauthorized access even if a session token is compromised.

- **Dynamic Construction of Class/Method Names - Reflection Injection:**
  Reflection injections in a web application occur when it uses data from a user or a third-party service to inspect, load, or invoke a component by name, making it vulnerable to remote code execution attacks. Attackers craft strings involving symbols like class methods to manipulate the reflection logic, potentially leading to the execution of arbitrary code on the server.

  The impact of such vulnerabilities is substantial, ranging from modifying data structures and escalating privileges in application-specific attacks, to compromising the entire application by executing malicious code. In extreme cases, attackers may download sensitive server data, install malware, or escalate privileges to attack other servers, depending on the organization's security measures. This threat is particularly critical if the organization lacks a robust Disaster Recovery Plan, allowing attackers to exploit vulnerabilities in services like Docker, Kubernetes clusters, cloud services, network firewalls, and operating system access control.



```
    */
   public static String 8 loadDatabaseDriver(String connectionUrl,  9
String connectionDriver) throws ClassNotFoundException {
       if (StringUtils.hasText(connectionDriver)) {
       10 Class.forName(connectionDriver);
```

Change this code to not construct class or method names directly from user-controlled data.

The above snippet shows the vulnerability identified in OpenMRS where Class.forName method is used to dynamically load a database driver class based on the user-controlled data provided through the connectionDriver parameter. This pattern poses a risk of reflection injection, allowing an attacker to manipulate the class name and potentially execute arbitrary code on the server.

Some of the mitigation techniques that can be used by the Codebase administrator to remove Reflection Injection vulnerability from the code are as follows:

Mitigation Strategies:

1. **Predefined List of Acceptable Drivers:**:
   - Implement a controlled approach to class loading by maintaining a predefined list of accepted database drivers.
   - Ensure validation of user-supplied connectionDriver against approved drivers to mitigate the risk of arbitrary class loading from user input.
2. **Whitelist Approach**:
   - Enforce a whitelist strategy by explicitly defining authorized database drivers and validate user-input against this predefined list to enhance security.
3. **Input Validation**:
   - Ensure user input is validated and sanitized to conform to expected formats, preventing the injection of malicious values and enhancing security within the class loading process.

5 Comparison of selected analytical tools

Some general insights into the kind of results and outcomes these tools produced when analyzing a codebase like OpenMRS are:

SonarQube Results::

- SonarQube generated comprehensive reports detailing various aspects of code quality, security vulnerabilities, bugs, and potential issues within the codebase.
- Results included severity levels for identified issues, providing insights into critical problems that need immediate attention.
- It categorized issues into different types (e.g., code smells, vulnerabilities, bugs) and offers explanations for each issue along with recommendations for resolution.
- SonarQube provided historical data on code quality trends, allowing to track improvements or degradation over time.

Semgrep Results::

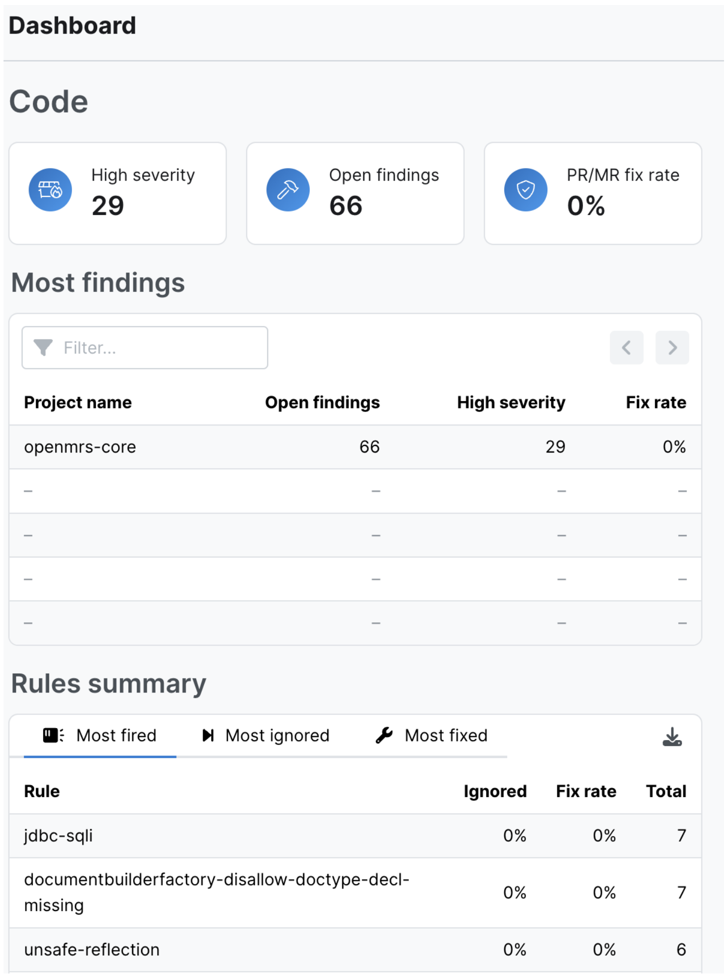
- Semgrep focuses on pattern-based detection and provides results highlighting specific instances of code patterns associated with security vulnerabilities, code smells, or potential bugs.
- Results were more focused and immediate, providing quick feedback based on defined patterns.
- It generated actionable findings directly related to the written rules or patterns, which helps to address issues efficiently.

Codacy Results::

- Codacy delivered reports that summarize code quality, highlighting issues related to security vulnerabilities, code duplication, and coding style violations.
- Results were presented in a dashboard format, offering an overview of the codebase's health and areas needing improvement.Codacy's reports categorize issues by severity, allowing easier prioritization and addressing critical problems first.
- Codacy was the slowest to give results out of all the static analysis tools used.But It offers suggestions or recommendations for code improvements based on the detected issues.

Each of these tools possesses its own strengths and weaknesses. Nonetheless, they excel in identifying distinct vulnerabilities and present their outputs in diverse formats. A comparative screenshot displays diverse vulnerabilities across SonarQube, Semgrep, and Codacy, showcasing their distinct approaches in vulnerability visualization on a comprehensive dashboard for easy comparison.

Semgrep



Dashboard

Code

High severity	Open findings	PR/MR fix rate
29	66	0%

Most findings

Filter...

Project name	Open findings	High severity	Fix rate
openmrs-core	66	29	0%
–	–	–	–
–	–	–	–
–	–	–	–
–	–	–	–

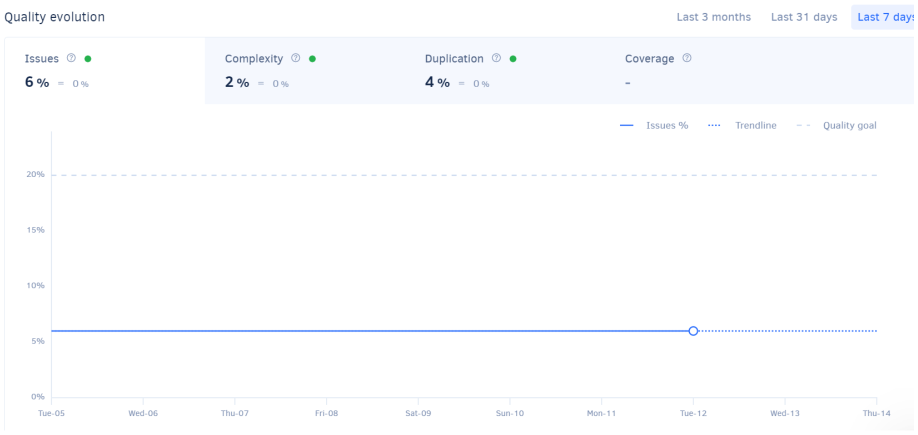
Rules summary

Most fired · Most ignored · Most fixed

Rule	Ignored	Fix rate	Total
jdbc-sqli	0%	0%	7
documentbuilderfactory-disallow-doctype-decl-missing	0%	0%	7
unsafe-reflection	0%	0%	6

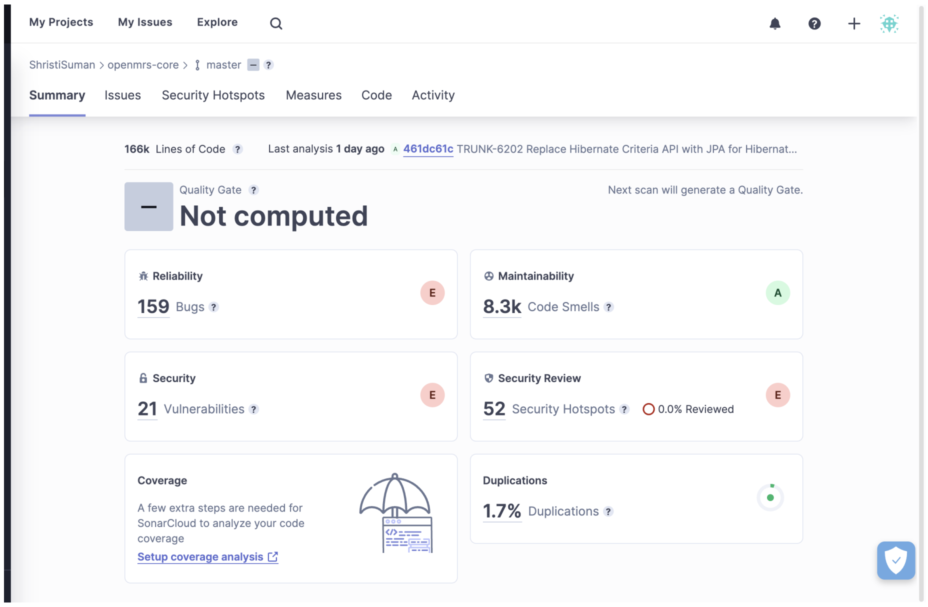
Semgrep's result dashboard displaying a comprehensive overview of actionable insights to code vulnerabilities and potential risks by indicating high severity risks.

Codacy



Codacy's result dashboard providing a user-friendly interface, offering a clear and detailed presentation of code quality metrics, including vulnerabilities, empowering efficient code review and enhancement processes.

SonarQube



166k Lines of Code

Last analysis 1 day ago 461dc61c TRUNK-6202 Replace Hibernate Criteria API with JPA for Hibernat...

Quality Gate

Not computed

Next scan will generate a Quality Gate.

Reliability: 159 Bugs — E

Maintainability: 8.3k Code Smells — A

Security: 21 Vulnerabilities — E

Security Review: 52 Security Hotspots, 0.0% Reviewed — E

Coverage: A few extra steps are needed for SonarCloud to analyze your code coverage. Setup coverage analysis

Duplications: 1.7% Duplications

SonarQube's result dashboard delivering an extensive analysis of code vulnerabilities, providing in-depth metrics and actionable insights to enhance code quality and security measures effectively.

6 Conclusion

Utilizing static analysis tools proves instrumental in identifying and mitigating security vulnerabilities. Our project exposed us to the significance of these tools in uncovering security issues early in the software development cycle. Following thorough research and analysis of various static analysis tools, we opted for SonarQube ,Semgrep and Codacy, all of which are open-source. As we applied these tools to OpenMRS, an open-source project we selected, each one revealed distinct results, highlighting diverse vulnerabilities. This comparative analysis allowed us to comprehend the disparities between the tools and evaluate their respective outputs. Ultimately, this project provided us with invaluable insights into not only employing static analysis tools effectively for bolstering system security but also into understanding their functionality and troubleshooting intricacies.

References

[1] Dejan Baca et al. "Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?" In: *2009 International Conference on Availability, Reliability and Security*. 2009, pp. 804–810. DOI: `10.1109/ARES.2009.163`.

[2] *Codacy Documentation*. `https://docs.codacy.com/`. Accessed: December 13, 2023.

[3] Melina Kulenovic and Dzenana Donko. "A survey of static code analysis methods for security vulnerabilities detection". In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2014, pp. 1381–1386. DOI: `10.1109/MIPRO.2014.6859783`.

[4] Danilo Nikolić et al. "Analysis of the Tools for Static Code Analysis". In: *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*. 2021, pp. 1–6. DOI: `10.1109/INFOTEH51037.2021.9400688`.

[5] *OpenMRS Core GitHub Repository*. `https://github.com/openmrs/openmrs-core#software-development-kit/`. Accessed: December 13, 2023.

[6] *Path Traversal*. `https://owasp.org/www-community/attacks/Path_Traversal`. Accessed: December 13, 2023.

[7] *Semgrep Documentation*. `https://semgrep.dev/docs/`. Accessed: December 13, 2023.

[8] *SonarQube Documentation*. `https://docs.sonarsource.com/sonarqube/8.9/`. Accessed: December 13, 2023.

[9] Pingyan Wang et al. "Detecting Security Vulnerabilities with Vulnerability Nets". In: *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. 2022, pp. 375–383. DOI: `10.1109/QRS-C57518.2022.00062`.