Automating Static Code Analysis for Risk Assessment and Quality Assurance of Medical Record Software

by

Harneet Kaur
B. Tech Information Technology, Guru Nanak Dev Engg College, 2013

A Project Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the Department of Computer Science

© Harneet Kaur, 2017
University of Victoria

All rights reserved. This project may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Automating Static Code Analysis for Risk Assessment and Quality Assurance of Medical Record Software

by

Harneet Kaur
B. Tech Information Technology, Guru Nanak Dev Engg College, 2013

Supervisory Committee

Dr. Jens Weber, Supervisor
(Department of Computer Science)

Dr. Neil Ernst, Committee Member
(Department of Computer Science)

ABSTRACT

As a part of agile transformation in the past few years, it has been observed that adopting continuous integration principles in the software delivery lifecycle leads to the improvement of efficiency of development teams over time. This is because unless all the pieces of a software development lifecycle do not work like a well-oiled machine, consistency to optimize the code for product delivery is hard to attain. Therefore, the main aim of this project is to integrate and automate quality assurance and risk management for Medical Record Software, OSCAR EMR using SonarQube as static code analysis tool, such that the process of software delivery is transparent enough for improvement and early stage detection of vulnerabilities or bugs that persist in the code during or before product deployment. SonarQube [1] is an open source platform for continuous inspection of code quality which performs automatic reviews with static analysis of code to detect bugs, code smells and security vulnerabilities on 20+ programming languages. Its support for multiple languages, continuous integration abilities and an effective GUI makes it one of the most prominent tools to be used for the static code analysis of OSCAR over the tools such as PMD or FindBugs, which were used earlier in the process.

The integration process of SonarQube for static code analysis revolves around two phases, setting up SonarQube as a continuous integration tool in Jenkins, setting up an external database with SonarQube on the server. The integration of SonarQube and Jenkins provides an analysis report for the vulnerabilities, bugs, code smells and duplicity found in the source code. In addition to the analysis report, the graphical representation of varying vulnerabilities over the years is generated to compare the trend of security risks over time.

Table of Contents

1. Introduction **1**
1.1 OSCAR-EMR 2
1.2 Continuous Integration 3
1.3 Static Code Analysis 4

2. Problem Statement **5**

3. Workflow **6**

4. Implementing and Integrating SonarQube within Jenkins **7**
4.1 Understanding Gerrit and Vagrant VM 7
4.1.1 Connecting to Server via PuTTY 7
4.1.2 Setting Up Vagrant VM 8
4.1.3 Installing SonarQube and Sonar-Scanner on Guest VM 9
4.1.4 Setting Up Database for SonarQube 10
4.1.5 Initializing and Running SonarQube as Service 11
4.2 Configuring SonarQube and Sonar Scanner with Jenkins Build 12
4.2.1 Configuring SonarQube and Sonar Scanner plugins in Jenkins 12
4.2.2 Creating Jenkins Build integrated with SonarQube 14

5. Accessing and Administrating Analysis Results **20**
5.1 Accessing Analysis Results 20
5.2 Administrating and Authorizing Analysis Results 21
5.2.1 Administrating Analysis Results Using SonarQube Parameters 21
5.2.2 Authorization to Analysis Results 24
5.2.3 Administrating Configuration Settings 25

6. Analysis Results **26**
6.1 Understanding Analysis Results 26

6.2 Filtering Analysis Results 27
6.3 Graphical Representation of Analysis Results 28
6.1 Configuring Jenkins for Cloning Commit Ids 28
6.1 Accessing Graphical Representation of Analysis Results 29

7. Challenges and Solutions 30

8. Conclusion and Future Scope 31
8.1 Conclusion 31
8.2 Future Scope 31

Bibliography 32

List of Figures

Figure 3.1 Workflow and components required for static code analysis....................................6

Figure 4.1 Entering the Host Machine....................................8

Figure 4.2 Initiating guest Vagrant VM from the host machine....................................9

Figure 4.3 Root access to Mysql database 10

Figure 4.4 Configuring the bind-address for MySql....................................11

Figure 4.5 Modifying sonar.properties file for database connection 11

Figure 4.6 Configuring sonar.web.port for SonarQube server 12

Figure 4.7 Jenkins Dashboard....................................12

Figure 4.8 Configuring SonarQube server for Jenkins 13

Figure 4.9 Configuring SonarQube Scanner within Jenkins 13

Figure 4.10 Jenkins Dashboard....................................14

Figure 4.11 Configuring project name and strategy 15

Figure 4.12 Cloning Source Code Management, BitBucket....................................15

Figure 4.13 Scheduling build for the source code 16

Figure 4.14 Configuring build environment....................................17

Figure 4.15 Initiating sonar-scanner for the build 18

Figure 4.16 Specifying post-build actions 18

Figure 4.17 Building a project 19

Figure 5.1 SonarQube login-page....................................20

Figure 5.2 SonarQube Project Dashboard 20

Figure 5.3 Access to the project issues 21

Figure 5.4 Parameters used to Administrate Analysis Results 22

Figure 5.5 Accessing Creation Date 22

Figure 5.6 Accessing the source code for vulnerabilities 23

Figure 5.7 Configuring Authorization and Security Settings 24

Figure 5.8 Administrating global permissions.....24
Figure 5.9 Administrating assignees for vulnerabilities.....25
Figure 6.1 Filtering analysis Results.....27
Figure 6.2 Analysis properties configuration for plotting graphical data.....28
Figure 6.3 Comparing two consecutive analysis.....29
Figure 6.4 Graphical representation of analysis results.....29

Table

Table 2.1 Comparison between CheckStyle, PMD, FindBugs and SonarQube.....5

Acknowledgements

I would like to thank my supervisor, Dr. Jens Weber for providing me an opportunity to work on this project with a lot of flexibility and necessary guidance, wherever required, during the course of project.

I would also like to thank Dr. Raymond Rusk and Tomas Bednar, who helped me throughout and aided the challenges faced with an immense knowledge resource.

'The integral part of success consists of going from failure to failure without the loss of enthusiasm.' – Winston Churchill

Structure of Project Report

This section provides information on the structure of discussed topics in each Chapter of the report:

Chapter 1 gives an introduction about Medical Record Software to be analyzed for quality assurance along with the importance of Continuous Integration and Static Code Analysis. This section also states the reasons for implementing SonarQube as an analysis tool and its service in the field of static code analysis.

Chapter 2 gives an overview of the problem statement, which aims to elaborate the issues related to existing static code analysis tools and necessity of introducing a more advanced tool into the development process of Medical Record Software, OSCAR.

Chapter 3 explains the workflow of Jenkins and SonarQube in addition to listing the reasons for integrating both the tools, instead of using them independently.

Chapter 4 discusses about implementation and integration details of Jenkins and SonarQube on the server. It also explains the customization and configuration details of SonarQube to obtain desired analysis. The main aim of this chapter is to provide a step-by-step user guide to perform static code analysis on OSCAR using SonarQube. This chapter also helps to understand the advanced features of SonarQube once it has been integrated with Jenkins.

Chapter 5 revolves around administrating analysis results generated in Chapter 4 and plotting graphical representation of vulnerabilities and other security risks found in source code of OSCAR varying from the year 2012 till 2017.

Chapter 6 acts as a documentation for the major obstacles faced in the overall process of setting up the project. This section also gives details about the potential networking problems that can occur in future, along with their solutions.

Chapter 7 concludes the implementation and integration of SonarQube and Jenkins to analyze source code of OSCAR along with its future scope. The future scope is further linked to dynamic code analysis that can help development and operations teams to collaborate resulting into a successful DevOps environment.

1 Introduction

OSCAR (Open Source Clinical Application and Resource) [2] is an open source web-based Electronic Medical Record (EMR) system. Initially developed at McMaster University, OSCAR [2] is one of Canada's widely implemented EMRs, suitable for general practitioners, specialists, public health units. It is a suite of Web-based Applications which serves to enhance the health and social service community's ability to provide the highest level of care. OSCAR is one of a few Electronic Medical Record (EMR) systems worldwide that has multilingual support. For example, one can run the program in English while one of the assistants can run it simultaneously in French, and other colleague see it in Spanish [2].

To manage quality and potential security risks in OSCAR's source code, it was necessary to introduce continuous integration along with static code analysis for development environment. In addition to this, one of the vital aspects of this analysis was that the results produced after should provide valuable input to secure the source code. One of the most prominent abilities of continuous integration is that it supports vast variety of static code analysis tools as internal plugins. The most popular and widely used tool for continuous integration and delivery is Jenkins [3] as it allows various plugins to be integrated into development environment to automate analysis process as per the requirements of a project.

A similar workflow was required to manage code security and quality of OSCAR, where a jenkins build integrated with SonarQube analysis tool could be automated over time producing static code analysis results and graphical representation of these vulnerabilities varying over time. While the introduction of SonarQube brings a definite workflow in development environment, it also addresses bugs, coding rules, test coverage, duplications, API documentation, complexities and displays details on SonarQube dashboard.

Therefore, the above features and advantages of SonarQube over the existing tools used for static code analysis necessitated to research more on the tool, integrate and customize it with Jenkins. The project is divided into three prominent phases, where phase 1 represents getting familiar with OSCAR and its features, Jenkins (continuous integration tool) and SonarQube in detail. Phase 2 facilitates information on setting up Jenkins and SonarQube on server, integrating SonarQube with Jenkins and initializing static code analysis from within Jenkins. Phase 3 helps to visualize the results obtained from phase 2 and plotting graphs for vulnerabilities that have shown variation in the source code in past few years.

1.1 OSCAR-EMR

OSCAR (Open Source Clinical Application and Resource) [4] is an open source web-based Electronic Medical Record (EMR) system, which supports all the necessary functions and features to run and administer a clinic efficiently. One of the main features of OSCAR includes its ability to provide multi-lingual support simultaneously for different individuals authorized to use OSCAR. OSCAR is much more than a regular MR and comprises of suite of web applications

which aims to help medical service community's ability to provide the highest level of care and organized patient records. A summary of OSCAR's applications has been listed below:

- **OSCAR Electronic Medical Record (EMR)** has all the functionality of a modern EMR.

- **MyOSCAR** [4] is a patient controlled personal health record that integrates with the OSCAR EMR.

- **MyDrugRef** [4] is a social network for pharmacists and physicians to facilitate knowledge transfer of drug information and is fully integrated with the OSCAR EMR.

- **OSCAR CAISI** [4] is a full-featured Case Management, Bed Management, and Program/Facility Management system

- **OSCAR Resource** [4] is a compilation of free clinical resources which can be accessed through the EMR at the point of care.

Technical Features:

OSCAR is primarily written in Java and JavaServer pages (JSP) served via Apache Tomcat servlet container [5]. The backend storage is managed by MySql database and Hibernate [5] is used as an interface layer between Java and MySql. Currently, OSCAR uses the following Atlassian [6] tools for project development and management.

- **Confluence** [6] is one of the Atlassian tools that holds documentation for OSCAR and allows to view any recent activity.

- **Bitbucket** [6] hosts the source code for current version of OSCAR and dominantly uses Git revision control systems for source code management.

- **JIRA** [6] is an issue-tracking software and provides an interface for bug tracking, issue tracking and project management functions for OSCAR.

1.2 Continuous Integration

Continuous Integration [7] is a software development practice where members of a team integrate their work frequently leading to multiple integrations per day depending on the size of project.

It is, therefore, necessary to verify every integration by an automated build to detect errors, such as dependency issues, unit testing, bugs, vulnerabilities or non-compliant code. This approach leads to significantly reduced integration problems and allows to develop cohesive software rapidly. Continuous integration processes operate at a regular frequency, ideally after every commit, thereby the system is:

- Integrated: All the changes are updated frequently and combined into the project
- Built: The code is compiled into an executable or package
- Tested: Automated test suites are run as per the requirements of project or plugins used
- Archived: The results are versioned and stored for future comparisons
- Deployed: Loaded onto a system where the developers can interact with it

Continuous integration [7] is a practice which can be achieved by integrating Jenkins with development environment, one of the top rated continuous building tools, that enables teams to automate the builds, manage artifacts and deployment processes. The core functionality and flexibility of Jenkins allows it to fit in a variety of environments and can help streamline development process [7]. It is a platform for continuous integration used to build and test software projects written in almost every language, thereby making it easier for developers to integrate changes to the project and for users to obtain a fresh build. Jenkins is a platform that possesses following features:

- It is a Java based continuous build system.
- It runs in a servlet container, such as Glassfish, Tomcat or a separate server space can be created to deploy it [7].
- It is supported by over 400 plugins used for SCM, testing, notifications, reporting, artifact saving, triggers, external integration and many more [7].

1.3 Static Code Analysis

Static code analysis [8] is a method of quality analysis of source code by detecting security threats, bugs and improving code quality. The process provides an understanding of code structure and helps to ensure that the code adheres to industry standards. Static Code Analysis (also known as Source Code Analysis) is usually performed as part of a code review (also known as white-box testing) and is carried out at the implementation phase of Security Development Lifecycle [8] (SDL). The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after the product release.

To exercise a system that continuously integrates source code of the medical software, OSCAR and analyzes the committed code for vulnerabilities, bugs, duplicity and non-compliant code, it is necessary to introduce a tool which is powerful and flexible at the same time. Along with this, it the tool should exist as a plugin in Jenkins to automate and customize static code analysis. There are many open-source static code analysis tools existing to maintain industrial standards of code, such as PMD [9], CheckStyle [9], FindBugs [9] and SonarQube.

Reasons of implementing Static Code Analysis

- **Create a transparent and automated development process** which will potentially result into building code periodically and provide analysis reports on bugs, security threats, code quality and critical faults found in the code.
- **Embed static code analysis as an integral part of continuous integration** in Jenkins, such that whenever a build is processed, it will automatically invoke static code analysis process as integrated in the build.
- **Can identify severe vulnerability and bugs** in early stages of development rather than discovering them after the project release.
- **To match the code quality to dynamically changing industry standards,** for example, with static code analysis, code is frequently reviewed to find problems like repeating the same name for two variables of different scopes or network threats where IP address is hard-coded.
- Buffer overruns, SQL injection, cross-site scripting, information leakage, uninitialized variables and integer overflow errors can be identified and eliminated through automated builds, hence providing a quality and secure development environment for OSCAR [10].
- **Better resource utilization:** Identifying bugs and security issues early in a development cycle will result into lower management costs along with limiting security risks for dynamic code analysis.
- **Code documentation** can be improved by running static analysis and commenting the reasons of issues being declared as closed, false positives or resolved. Code that is not well documented is extremely difficult to work with and consumes unnecessary time.

2 Problem Statement

The existing system uses static code analysis tools like PMD and FindBugs, which help to identify potential bugs and bad code practices respectively. In addition to these tools, Junit testing plugin is also used to validate the result of the code in expected state and expected sequence of events. The results obtained from analysis are displayed on Jenkins server in the form of graphs plotted for a specific build number against the count of issues. While these tools do keep a track of issues identified during analysis, they do not succeed well when it comes to visualization, reporting, logging and managing analysis results. Additionally, the range of detecting vulnerabilities or security risks for these tools is relatively low. While PMD [9] mostly detects potential defects, unsafe and non-optimized code, FindBugs [9] focuses on identifying bugs and performance issues. Therefore, it was necessary to introduce a tool in the continuous integration environment, which combines the functionality of the tools defined above and provides a better platform to visualize and manage analysis results. Irrespective of a lot of open-source tools available for static code

analysis, SonarQube is the most appropriate because it complies with OSCAR's continuous integration environment, development standards and provides flexibility to personalize analysis results. Table 2.1 lists main features and advantages of SonarQube over PMD, FindBugs and CheckStyle for security code assessment.

	CheckStyle	**PMD**	**FindBugs**	**SonarQube**
Purpose	Verifying coding conventions	Detecting bad coding practice	Finding potential bugs	Managing overall quality assurance
Types of Verification [9]	Java based naming conventions, headers, Javadoc comments, method parameters	Dead and duplicated code, overcomplicated expressions in Java	Design flaws, bad practice, multithread correctness	Bugs, duplications, vulnerabilities, code smell, technical debt, overall quality statistics and metrics
Plugins and Integration with Jenkins	CheckStyle plugin available within Jenkins	PMD plugin available within Jenkins	FindBugs plugin available within Jenkins	SonarQube plugin once integrated with Jenkins has an option to further integrate FindBugs, PMD and CheckStyle as sub-plugins on SonarQube server
Custom Rules [9]	408 customizable rules written in Java	234 rules written in Java and limited language support	132 rules written in Java and analyzes Java code only	Customizable 1000+ rules supporting more than 20 languages
Analysis Results	Displayed on Jenkins server with minor flexibility of customization for false positives	Displayed on Jenkins server with no flexibility of customization for false positives	Displayed on Jenkins server with no flexibility of customization for false positives	Displayed on SonarQube server with flexibility to eliminate false positives, assign severity levels, close issues and check compliant code examples
Authorization and Accessibility	Non-private accessibility of results on Jenkins server	Non-private accessibility of results on Jenkins server	Non-private accessibility of results on Jenkins server	Only authorized users can access results by logging into SonarQube server

Table 2.1: Comparison between CheckStyle, PMD, FindBugs and SonarQube

3 Workflow

SonarQube dashboard and Jenkins server can be deployed as two separate components. SonarQube tool is available as a plugin in Jenkins marketplace, which allows to initiate a build from within Jenkins. In addition to this, Jenkins requires the installation of Sonar-Scanner, also existing as a plugin for analysis to take place. SonarQube tool helps to setup a server displaying analysis results, while sonar-scanner analyzes the source code cloned from BitBucket, a source code management tool. The workflow of Jenkins and SonarQube integration is illustrated in the following steps:

1. The code developed in IDE's such as Visual Studio, Eclipse or NetBeans is committed, checked and integrated into a repository, BitBucket in case of OSCAR.
2. Jenkins further makes checkouts of the code from repository and performs automated builds by creating a workspace. The automation of these builds can be monitored, such that a build can be initiated whenever a code is committed to the repository or a specific time frame can be set to initialize the build. In case of OSCAR, the latter option is more convenient and efficient to use.
3. Once the build is initiated, sonar-scanner prepares to analyze the source code detecting for vulnerabilities, bugs, dependencies, code smells, duplicity of code and major security risks. If the build is successful, console output displays a link to SonarQube server, where the results can be accessed on SonarQube dashboard.
4. Vulnerabilities and security risks detected are further assigned to developers, who modify the source code to persist and maintain its quality. This cycle repeats itself until all the vulnerabilities in the source code are eliminated.

A diagrammatical representation of the workflow followed by Jenkins and SonarQube for static code analysis is illustrated below in Figure 3.1.

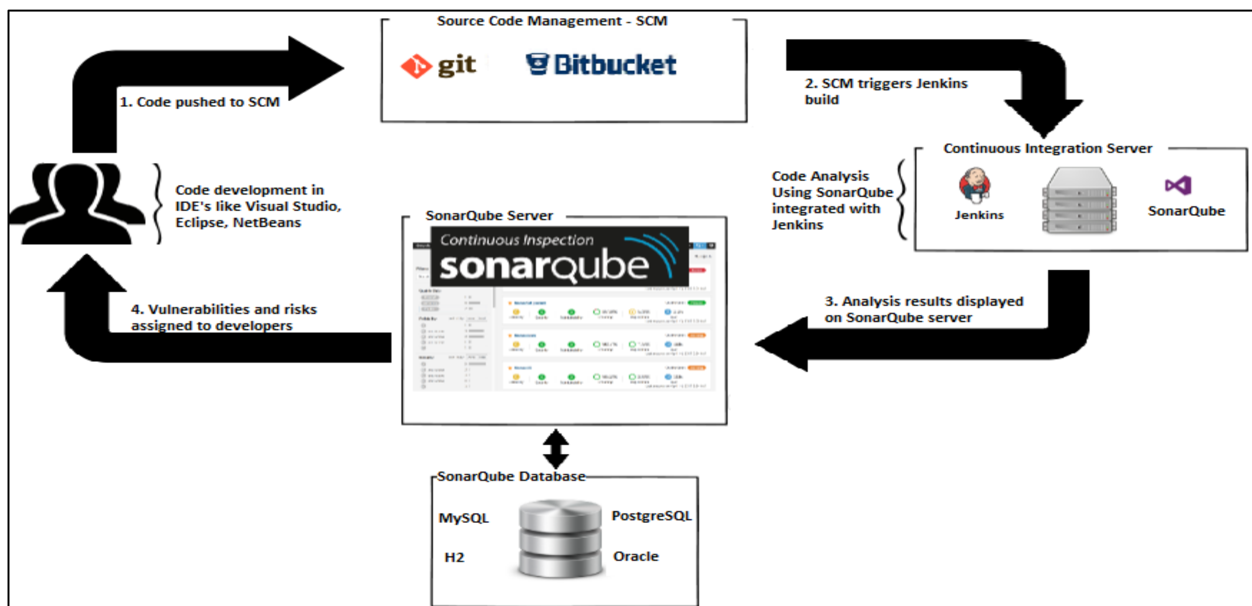


Figure 3.1 Workflow and components required for static code analysis

4 Implementing and Integrating SonarQube with Jenkins

4.1 Understanding Gerrit and Vagrant VM

A server setup is preferred for integrating Jenkins with SonarQube over setting it up on a local machine. The main reason accounts to the fact that server space allows to create a few user names which have access to Jenkins server space, therefore analysis results can be accessed by various users in the access groups from their respective machines.

A separate server space http://gerrit.seng.uvic.ca used for Jenkins is hosted by Apache [5] server running on default port 80. This Apache server is configured to reverse proxy all the requests going to **/gerrit/** to the gerrit http server. Gerrit [11] software used for server setup is a web based code review tool built on top of git version control system that can be bound to Jenkins by configuring it in the guest machine. A welcome page running at http://gerrit.seng.uvic.ca:80 is used to test the correct operation of Apache2 server after installation on Ubuntu host system.

In addition to Gerrit server, a guest Virtual Machine is setup with the help of Vagrant. Vagrant [12] is a utility that helps to manage VM boxes operating inside it via a simple to use command line interface. The VM boxes [13] from Vagrant can be found from website registered under https://app.vagrantup.com/boxes/, which currently lists approximately 322 different VM boxes from RHEL [13], CentOS [13], Arch Linux to ubuntu/xenial64 [13], bento/ubuntu-16.04 [13] and many more. VM box installed is the guest machine to the host Gerrit and is reached via ssh connection established from host to guest.

The sections illustrated below act as a complete guide for working with Gerrit server, Vagrant VM, installing tools on Vagrant VM and integrating them on the server.

4.1.1 Connecting to Server via PuTTY

A connection from local machine to Gerrit server is established using an open-source terminal emulator knows as PuTTY [14] via SSH connection. The server can only be accessed by authorized users and user names, from default port 22. PuTTY supports several networking protocols, such as SSH [14], SCP [14] and Telnet [14] primarily resulting into secure remote connections or secure file transfers. Once the tool is downloaded to the local machine from PuTTY, the steps demonstrated below can be followed to access the host server.

- Install PuTTY and connect to Gerrit server by specifying the destination as username@gerrit.seng.uvic.ca and port as “22” for secure remote connection as shown in Figure 4.1.
- The terminal window prompts for password to connect to host machine, Ubuntu 14.04.5 LTS.
- Once authorization is successful, terminal displays system information of the host machine listing its IP address 142.104.90.77, System Load, Memory Usage and IP addresses of VM boxed associated with the host machine as shown in Figure 4.1.

```
harneet@gerrit: ~
Using username "harneet".
harneet@gerrit.seng.uvic.ca's password:
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-110-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Tue Sep 19 13:36:16 PDT 2017

  System load:  0.01               Users logged in:        0
  Usage of /:   63.9% of 701.94GB  IP address for eth0:     142.104.90.77
  Memory usage: 22%                IP address for vboxnet0: 10.0.2.1
  Swap usage:   0%                 IP address for vboxnet1: 142.104.90.1
  Processes:    174                IP address for vboxnet2: 172.28.128.1

  Graph this data and manage this system at:
    https://landscape.canonical.com/

*** System restart required ***
Last login: Tue Sep 19 13:36:18 2017 from 142.104.232.131
harneet@gerrit:~$
```

Figure 4.1: Entering the Host Machine

4.1.2 Setting up Vagrant VM

Once a connection is established with the host machine, next step is setting up guest machine, which is this case is Vagrant box bento/ubuntu-16.04. Installing Vagrant is a pre-requisite for setting up a VM Box because Vagrant is a utility that helps to manage Virtual Machines effectively and can be installed using the following command [15]:

- sudo apt-get install vagrant

The next step is downloading and installing Vagrant box onto the host machine using commands listed below [15].

- vagrant box add bento/ubuntu-16.04
- vagrant up

vagrant up command primarily imports virtual box image, boots bento/ubuntu-16.04 box and establishes an ssh connection from host machine to guest machine. The installation of Vagrant box onto the host machine also creates a file called "Vagrantfile" inside **vagrant** folder, which is the main configuration file for the host machine to communicate to the guest machine. Figure 4.2 shows that the guest Ubuntu-16.04 has been installed on the host machine and is successfully accepting ssh connection. Booting of the guest VM lists other important features with respect to the host machine, such as type of adapters being used, forwarding ports from the guest to the host and port at which the host establishes a connection with the guest VM.

```
harneet@gerrit:~/vagrant$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Checking if box 'bento/ubuntu-16.04' is up to date...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 8080 (guest) => 8080 (host) (adapter 1)
    default: 8090 (guest) => 8090 (host) (adapter 1)
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key


Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-81-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

23 packages can be updated.
15 updates are security updates.
```

Figure 4.2: Initiating guest Vagrant VM from the host machine

4.1.3 Installing SonarQube and Sonar-Scanner on guest VM

After logging into guest machine, root user of the guest machine is invoked to perform specific functions such as downloading, installing and initiating tools along with setting up their network services. This can be done by using the command:

- sudo su

Now, tools like SonarQube and Sonar-Scanner are installed within Jenkins folder located on the guest VM under the location: **cd /var/lib/jenkins.** The installation can be performed using the commands in the order defined below.

- wget https://sonarsource.bintray.com/Distribution/sonarqube/sonarqube-6.5.zip
- unzip sonarqube-6.5.zip
- wget https://sonarsource.bintray.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-3.0.3.778-linux.zip
- unzip sonar-scanner-cli-3.0.3.778-linux.zip
- ls

Finally, **ls** command can be used to list files stored in folder **cd /var/lib/jenkins**, including sonarqube-6.5 and sonar-scanner-cli-3.0.3.778-linux.

4.1.4 Setting Up Database for SonarQube

While SonarQube server uses the H2 database by default, it clearly states that the database should be used for evaluation purpose only and there is no support for migrating data to a different database engine. Due to this reason, the external database MySql is being used for storing analysis results. After pointing to location **cd /etc,** the following steps are used in the order for successful installation of MySql:

- **sudo apt-get install mysql-server,** which installs both the server and the client
- While mysql-server is depackaging, it prompts a dialogue box for ‘new password’ to authorize MySql “root” user.
- **mysql -u root -p** is used to access the database as a root user for creating and modifying databases.

```
root@vagrant:~# mysql
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
root@vagrant:~# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.19 MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

Figure 4.3: Root access to Mysql database

The next section corresponds to creating a database for SonarQube, once user is logged in as the root inside mysql service. The commands illustrated below help create a database as well as a user named “sonar”.

- CREATE DATABASE sonar CHARACTER SET utf8 COLLATE utf8_general_ci;
- CREATE USER 'sonar' IDENTIFIED BY 'sonar';
- GRANT ALL ON sonar.*TO 'sonar'@'%' IDENTIFIED BY 'sonar';
- GRANT ALL ON sonar.*TO 'sonar'@'localhost' IDENTIFIED BY 'sonar';
- FLUSH PRIVILEGES;
- EXIT

To confirm if the database has been created for SonarQube, the command : **show databases;** will list the names of the existing databases.

It is necessary to configure MySQL in accordance to the IP address of the guest VM to avoid bind-address exception. Therefore, the default bind-address for the database is modified to 10.0.2.15, which is IP address of the guest VM as well. Along with this, the value of variable **max_allowed_packet** is also increased, in case the memory of analysis results produced exceeds

the default value after sonar analysis. The network and variable settings can be configured by navigating to the folder **cd /etc/mysql/mysql.con.f** and modifying the file **mysqld.cnf** as shown in Figure 4.4.

```
[mysqld]
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
datadir         = /var/lib/mysql
log-error       = /var/log/mysql/error.log

# By default we only accept connections from localhost
bind-address    = 10.0.2.15

max_allowed_packet=524288000

# Disabling symbolic-links is recommended to prevent assorted security risks
symbolic-links=0
```

Figure 4.4: Configuring the bind-address for MySQL

4.1.5 Initializing and Running SonarQube as a Service

As explained in Section 4.1.2, the installation location of SonarQube can be found in the folder **cd /var/lib/jenkins/sonarqube-6.4.** The tool itself does not perform static code analysis, instead provides a GUI and server space for the analysis results to be displayed. Before integrating SonarQube with Jenkins, it requires to be initiated as a standalone service following the steps demonstrated below.

- The sonar.properties files located in the folder cd /var/lib/jenkins/sonarqube-6.4/conf is modified for database username, password and URL as shown in Figure 4.5.

```
# DATABASE
#
# IMPORTANT: the embedded H2 database is used by default. It is recommended for tests but not for
# production use. Supported databases are MySQL, Oracle, PostgreSQL and Microsoft SQLServer.

# User credentials.
# Permissions to create tables, indices and triggers must be granted to JDBC user.
# The schema must be created first.
sonar.jdbc.username=sonar
sonar.jdbc.password=sonar

#----- Embedded Database (default)
# H2 embedded database server listening port, defaults to 9092
#sonar.embeddedDatabase.port=9092
#----- MySQL 5.6 or greater
# Only InnoDB storage engine is supported (not myISAM).
# Only the bundled driver is supported. It can not be changed.
sonar.jdbc.url=jdbc:mysql://10.0.2.15:3306/sonar?useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true&use$
```

Figure 4.5: Modifying sonar.properties file for database connection

- The web server settings in the sonar.properties file are configured in accordance to the port which is open in the firewall for the guest and the host machine. Instead of using the default port 9000, port 8090 is used for it is an open port listed in the host and the guest machine firewall.

```
# Binding IP address. For servers with more than one IP address, this property specifies which
# address will be used for listening on the specified ports.
# By default, ports will be used on all IP addresses associated with the server.
#sonar.web.host=10.0.2.15

# Web context. When set, it must start with forward slash (for example /sonarqube).
# The default value is root context (empty value).
#sonar.web.context=
# TCP port for incoming HTTP connections. Default value is 9000.
sonar.web.port=8090
```

Figure 4.6: Configuring sonar.web.port for SonarQube server

- SonarQube start-up file can be found by navigating to the folder **cd /var/lib/jenkins/sonarqube-6.4/bin/linux-x86-64** and the service can be started using **sudo ./sonar.sh start**
- To confirm the status of SonarQube server, the commands **./sonar.sh status** or **netstat -tulnp** can be used. The latter lists process name alongside port number being used.

4.2 Configuring SonarQube and Sonar Scanner within Jenkins Build

The backend process for completing security code review includes configuring Vagrant VM, SonarQube and Sonar-Scanner on the host machine. The default port on which Apache operates is 80 on the server http://gerrit.seng.uvic.ca, jenkins uses port 8080 and is redirected to the login page of Jenkins using the address http://gerrit.seng.uvic.ca:8080/login. The login page further redirects to Jenkins Dashboard which lists vital information about the project, such as its state (success, failure, unstable), duration of build completion, last success or failure.

Figure 4.7: Jenkins Dashboard

4.2.1 Configuring SonarQube and Sonar Scanner plugins in Jenkins

While SonarQube and Sonar Scanner have already been installed on the guest VM, a successful integration with the tools also requires installation of their plugins within Jenkins. Once the plugins are installed, a freestyle project is created which binds all the aspects required for static code analysis from checking out the source code from the repository to analyzing it and displaying the results on SonarQube dashboard. The steps described below are to be followed to configure SonarQube and Sonar Scanner in Jenkins.

- Install the plugin by navigating to Manage Jenkins>Manage Plugins>SonarQube Scanner for Jenkins>Install
- Installation of this plugin creates two sections: one for configuring SonarQube server and the other for sonar-scanner.
- The SonarQube server section is to be configured for the URL where the results are posted after analysis. Meanwhile, this server's URL should be accessible by the guest VM installed on the host gerrit.seng.uvic.ca. This section can be found under the location Manage Jenkins>Configuration and is configured as shown in Figure 4.8.

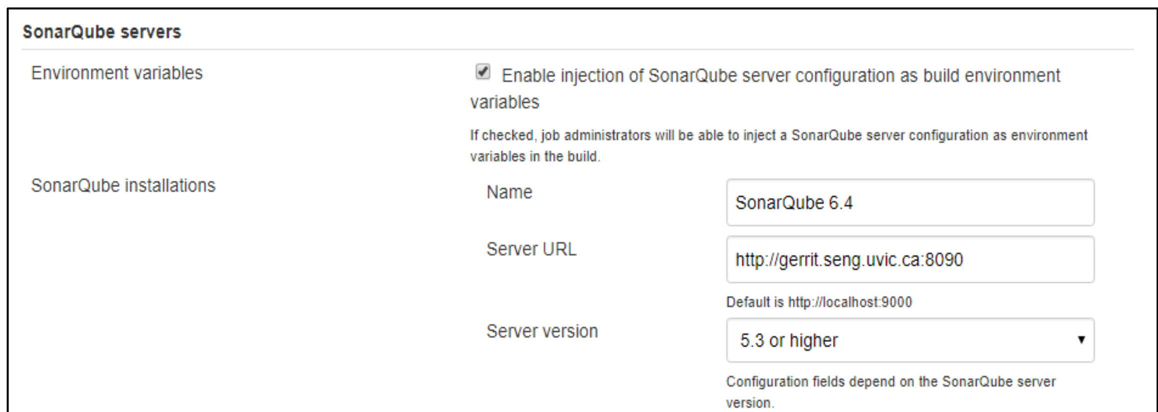


Figure 4.8: Configuring SonarQube server for Jenkins

- Under **Mange Jenkins>Global Tool Configuration**, a section for sonar scanner is configured by specifying the location of sonar-scanner tool on the guest VM.

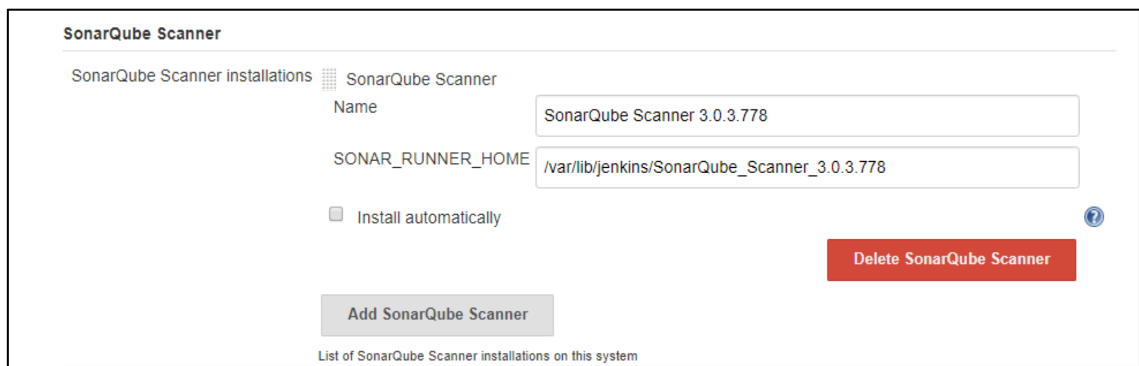


Figure 4.9: Configuring SonarQube Scanner within Jenkins

The plugins installed on Jenkins are now integrated with their counter tools installed on the guest VM, therefore the source code of OSCAR is ready to be reviewed for static code analysis.

4.2.2 Creating Jenkins Build integrated with SonarQube

Installing and configuring SonarQube and Sonar-Scanner plugins serves as a pre-requisite for setting up Jenkins build resulting into static code analysis of OSCAR. These plugins are then initiated and their function is customized within the build as per the requirements of the project. A simple Jenkins build comprises of several blocks, such as General information, Source Code Management, Build Triggers, Build Environment, Build, Post Build Actions, which are executed on the Jenkins Dashboard, in the sequence, as part of the build.

Jenkins Dashboard acts as a home-page for Jenkins tool installed on the guest VM hosted by the server: gerrit.seng.uvi.ca and runs on the port 8080. Jenkins dashboard or home-page consists of several options, which help to create new projects, view build history, list the groups using Jenkins and manage Jenkins configuration, plugins and global tools. It also lists details about project information regarding build's stability, date of last failure or last success and the duration a build takes to succeed or complete as shown in Figure 4.10.

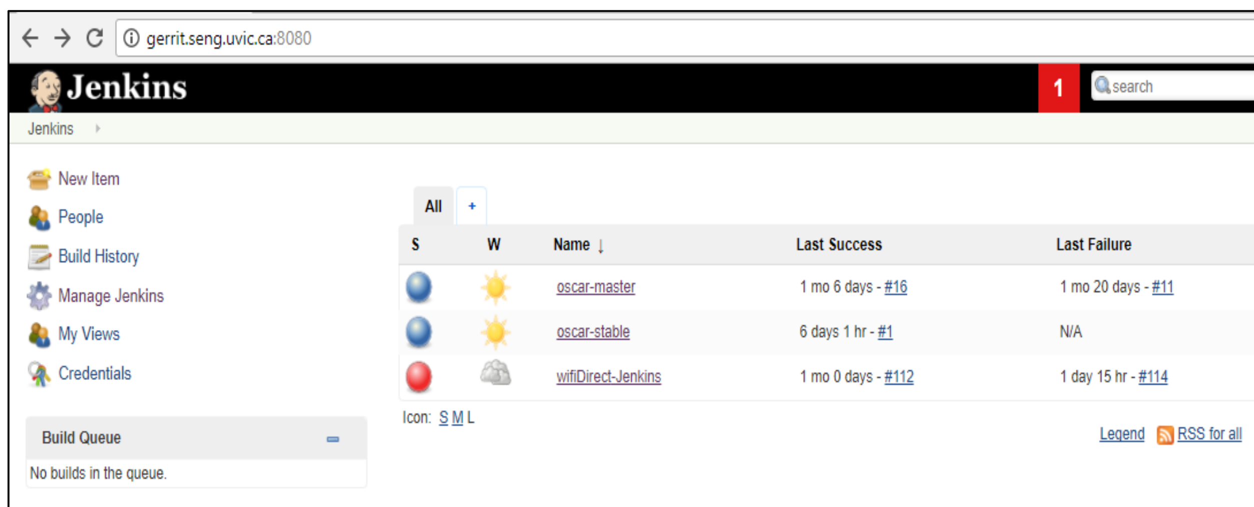


Figure 4.10: Jenkins Dashboard

The sequence of execution and the importance of these steps is explained as below:

1. General:
   As the name suggests, it includes general information about the project from its name, description to defining a strategy to discard old builds and maximum number of builds to keep. As shown in Figure 4.11, in this case, the most recent six builds are kept and the rest are discarded. This number differs as per project's requirements.

Figure 4.11: Configuring project name and strategy

2. Source Code Management:
   This block of build integrates source code from various developers working on the same project into a common repository, such as GitHub, Subversion or BitBucket [16]. The URL of the integrated source code is then cloned and used as a repository URL along with the credentials for the repository. The other prominent feature of SCM block is that it allows to specify branch of the project to be build, in this case */master. To analyze different versions of the project, the branch to build can be specified as a commitId, which checks out a specific commit associated with that commitId.

Figure 4.12: Cloning Source Code Management, BitBucket

3. Build Triggers:
   This section defines how and when a build should be triggered. One of the prominent options present is Poll SCM, which helps to trigger a build on a specific schedule. To allow periodically scheduled tasks to produce even load on the system, the symbol H (for "hash") should be used wherever possible. This field follows the syntax of cron (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace: MINUTE HOUR DOM MONTH DOW [17]
   For example, using 0 0 * * * for a dozen daily jobs will cause a large spike at midnight. In contrast, using H H * * * would still execute each job once a day, but not all at the same time, better using limited resources. The H symbol can be used with a range. For example, H H(0-7) * * * means some time between 12:00 AM (midnight) to 7:59 AM [17]. The H symbol can be thought of as a random value over a range, but it is a hash of the job name, not a random function, so that the value remains stable for any given project [17].
   Also, for this project the schedule of triggering a build will be thrice a month during night time and the syntax used for the schedule is: **H H(0-5) 1,12,25 1-12 ***.

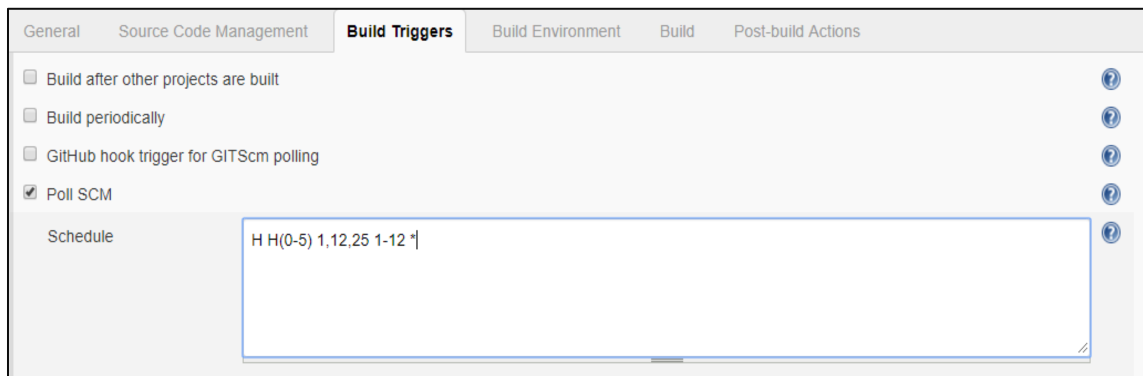


Figure 4.13: Scheduling build for the source code

4. Build Environment:
   While 'Build Triggers' define schedule of the build to run, Build Environment sets up an environment compatible for the build before it even initializes. It helps to customize builds of a project in a way that the workspace can be deleted before the build starts, to avoid memory issues or aborting a build if it gets stuck mid-way.
   For instance, while setting up a build for static code analysis, the environment variable SONAR_SCANNER is added to the build environment, which facilitates the location of sonar scanner tool on the guest VM.

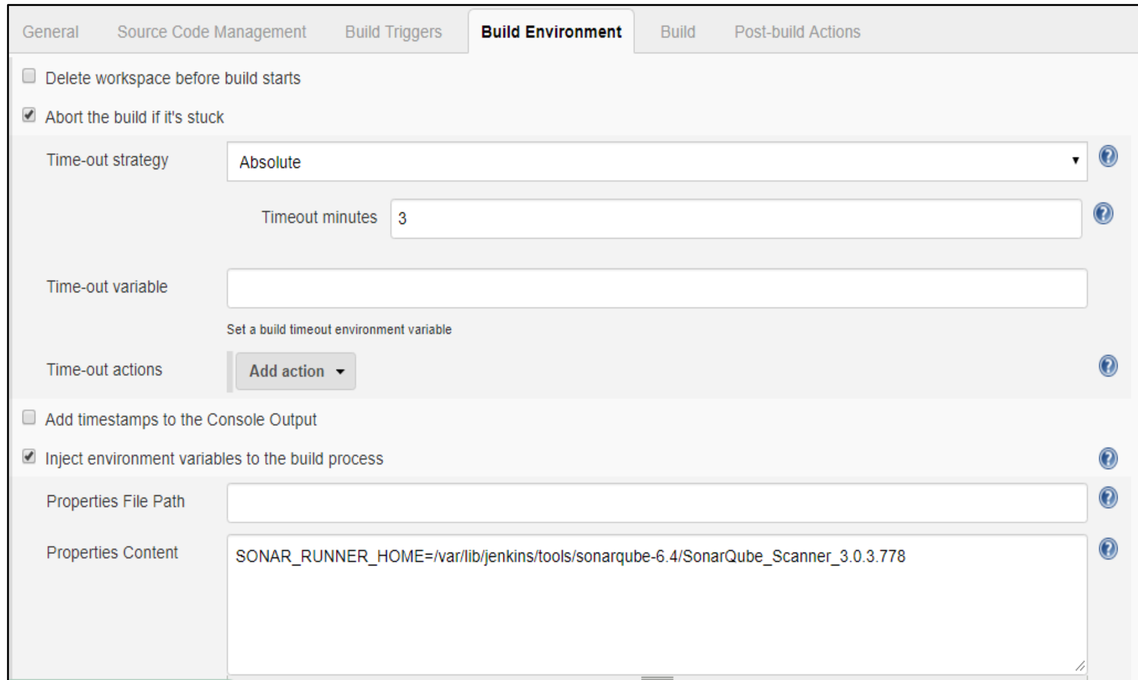


Figure 4.14: Configuring build environment

5. Build:
   Build is the main block which consists of a list of plugins installed, and helps Jenkins to initialize the plugin by configuring it. The build step performs several functions, the most important being invoking sonar-scanner for static code analysis. Sonar-scanner is configured by specifying the tasks to run, JDK version being used, analysis properties and additional arguments. The analysis properties provide detailed information on the project name, project key and project source on the guest VM.
   When sonar-scanner is invoked in this section, it directs to the location where the source code of the project has been placed. Once it gets the hold of the source code, it identifies the language (in this case Java) and performs the analysis based on pre-defined rules for that specific language. The following configuration is the main building block of the analysis. Here, **"Add build step"** is used to invoke sonar-scanner by selecting SonarQube Scanner from the dropdown and configured for analysis properties of a project. Additional argument –X is added to provide debugging while building the project.
   The analysis properties consist of parameters such as:

- **sonar.projectKey** defines a unique key for the project.
- **sonar.projectName** specifies the name of the project, and is recommended to use the same name as in the General section of the build.
- **sonar.projectVersion** lists the version of the project being analyzed.
- **sonar.sources** specify the location of the workspace created for the project oscar-master.
- **sonar.login** defines the default login for the build
- **sonar.password** defines the default password for logging into SonarQube once the build is successful

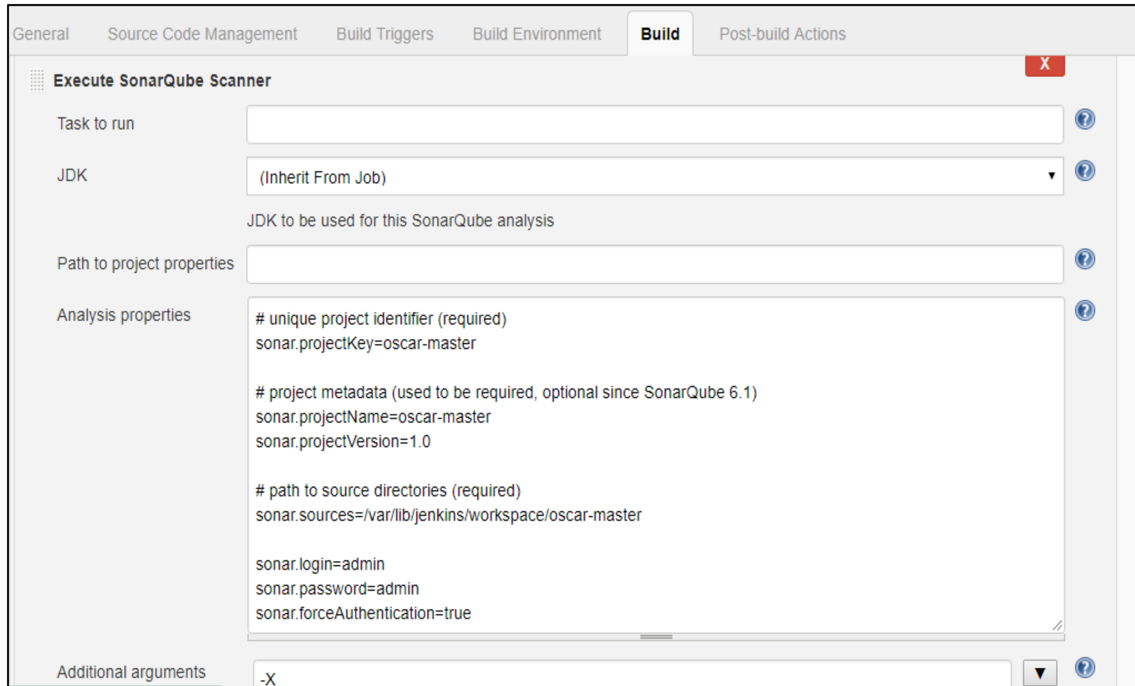


Figure 4.15: Initiating sonar-scanner for the build

6. Post-Build Actions:
   Post Build-Actions perform various functions once the build action has been completed. For instance, if the build is unstable, post-build actions help to notify the project lead or designated person by sending them emails including console output from the build, hence notifying them of the unstable build and name of the person who initiated it.
   In addition to notifying about unstable builds, post-build actions can be used for publishing Junit test results, achieving artifacts, building other projects once the current build is successful and implementing quality gate for the current build.

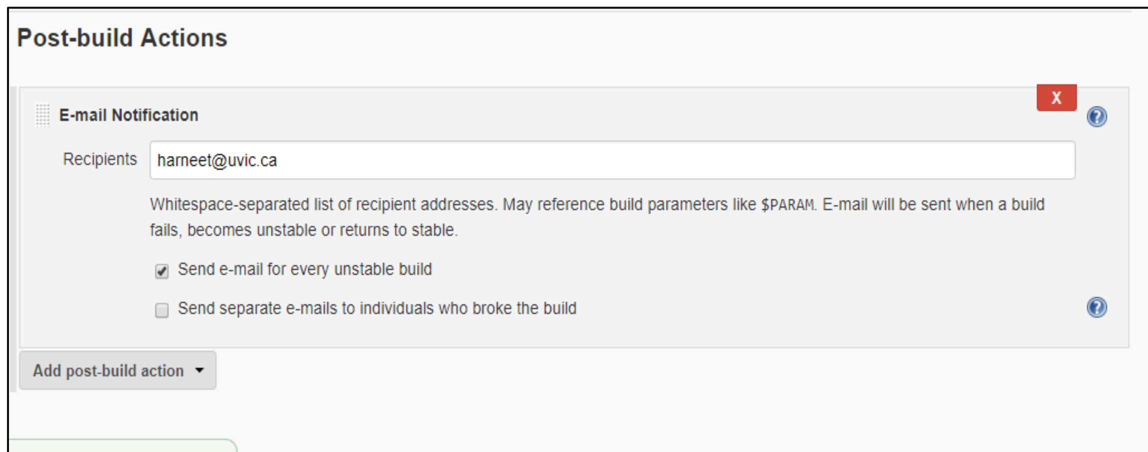


Figure 4.16: Specifying post-build actions

7. Building a Project:
   To initialize and customize a build, the order of steps described as above is to be followed and the project is saved, which navigates to the main dashboard. The dashboard lists features like status of the project, changes made, workspace, build now options and delete project. The build of the project is then initiated by ‘Build Now’ option displayed on the main Project Dashboard.

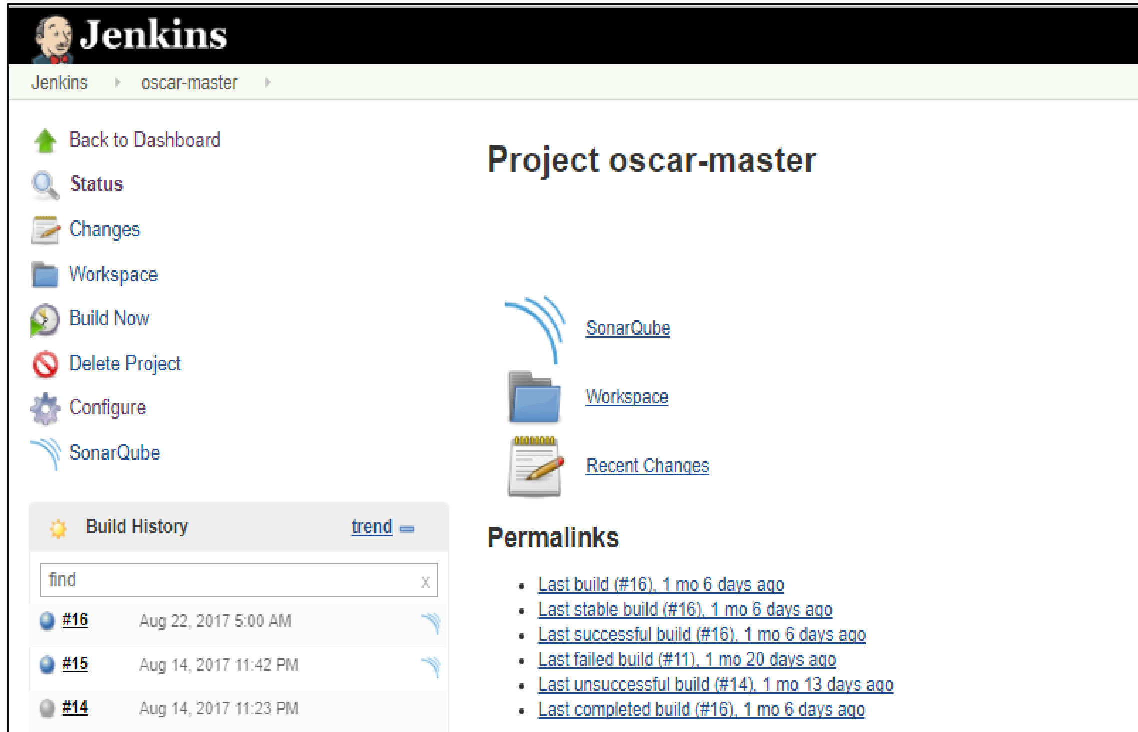
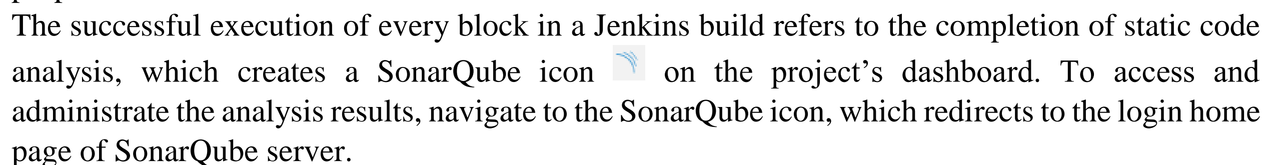


Figure 4.17: Building a project

This option prepares the build to execute all the configuration settings defined for the project in the order, General, Source Code Management, Build Triggers, Build Environment, Build, Post-Build Actions. It may be noted that if one step in the configuration fails to execute, the whole build will fail irrespective of the steps being in sequence.
The build’s processing and proceeding are seen by clicking on the #build number, for example in this case, #16 > Console Output shows a complete log of the build. In case of a failure, the console output states the reason of the build failing, which can be due to unidentified location of sonar-scanner on the guest VM, failure to clone the SCM or incorrect configuration of sonar-analysis properties.
The successful execution of every block in a Jenkins build refers to the completion of static code analysis, which creates a SonarQube icon  on the project’s dashboard. To access and administrate the analysis results, navigate to the SonarQube icon, which redirects to the login home page of SonarQube server.

5 Accessing and Administrating Analysis Results

Accessing and administrating analysis results produced on the SonarQube server is one of the most vital aspects for these facilitate with highly sensitive information about the project. The information proves to be very useful for future reference, if efficiently administered. Once the SonarQube icon redirects to SonarQube server, administrator or authorized users can login to access the analysis results.

Figure 5.1: SonarQube login-page

5.1 Accessing Analysis Results

If the analysis results are accessed for the first time on SonarQube server, the authorization must be done by the administrator only. Once the authorization is successful, it redirects to a Project Dashboard page, listing all the projects which have been analyzed within Jenkins build.

In addition to the listing of the projects, Dashboard displays rating on various rules, decreasing code quality on parameters such as Reliability, Security, Maintainability, Code Coverage and Duplications from A to E as shown in Figure 5.2 [18].

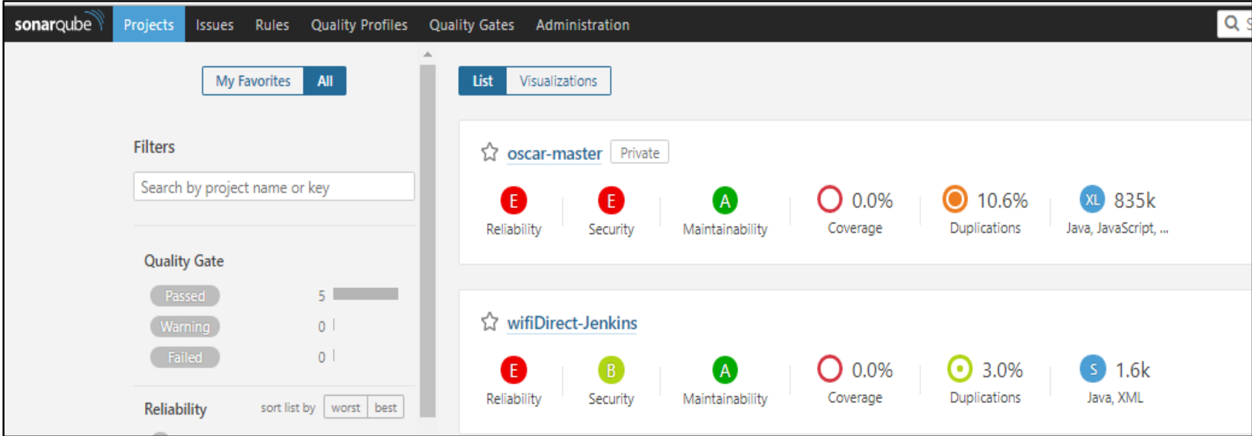


Figure 5.2: SonarQube Project Dashboard

The dashboard represents an outline of analysis results for project **oscar-master and** gives detailed view and count on the number of bugs, vulnerabilities, code smells and other security risks persisting in the source code. In addition to this, details such as number of lines of code and

language being written in are also included on the project information page. For instance, the number of bugs and vulnerabilities shown for the source code of oscar-master is 2.2k and 1.9k, respectively.

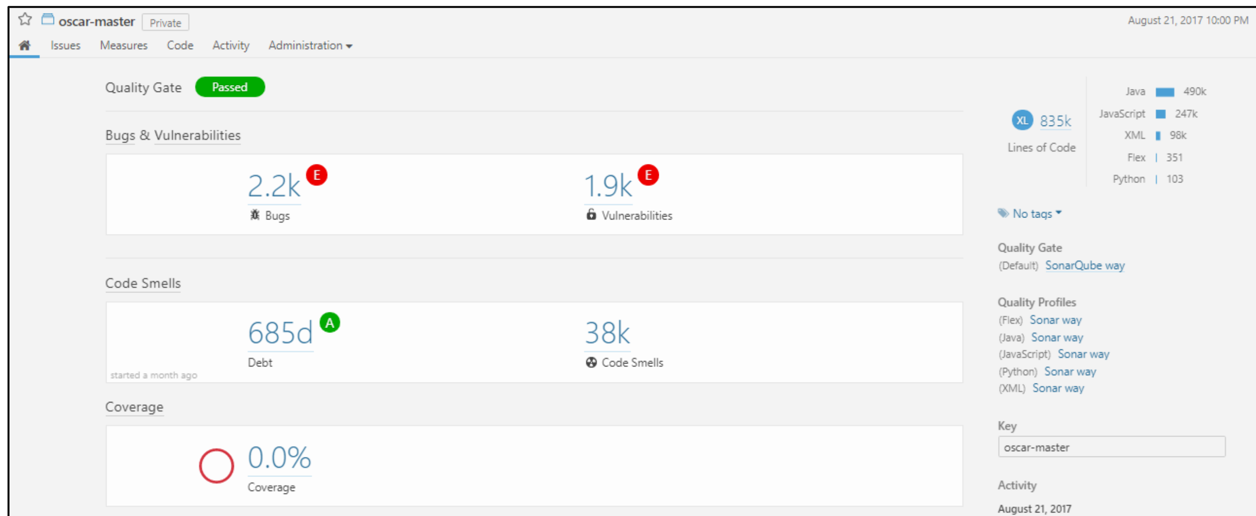


Figure 5.3: Access to the project issues

To access or view all the issues raised as a part of the analysis results, click on the number which displays the category. For example, as shown in Figure 5.3, clicking on 1.9k navigates the user to a more organized format of analysis results, such that the results in the “Vulnerability” category are displayed as per severity level: **Blocker, Critical, Major, Minor** and **Informational**. The next section proves to be helpful while managing and administrating results and anticipating the current state of the project.

5.2 Administrating and Authorizing Analysis Results

It is very important to administrate the results produced by sonar-scanner efficiently, for these results become a vital part for future references as well. SonarQube is a flexible tool, which provides numerous sub-categories to distinguish and differentiate analysis results, thereby making it easier for administrator or users to organize and assign these vulnerabilities to designated professionals. In addition to this, analysis results can be administrated such that different projects are made visible to designated individuals by creating Permission Templates. One of the vital aspects of administrating results is configuring general settings for the project, such as security, database history, SCM setting, webhooks and many more. There are various parameters other than severity level that help to distinguish analysis results as explained in the sections below.

5.2.1 Administrating Analysis Results Through SonarQube Parameters

The parameters are sub-division of the project category and help to organize and administrate the project as per its type, resolution, status, severity level, creation date, tags, rules, file or directory, assignee and author as shown in Figure 5.4.

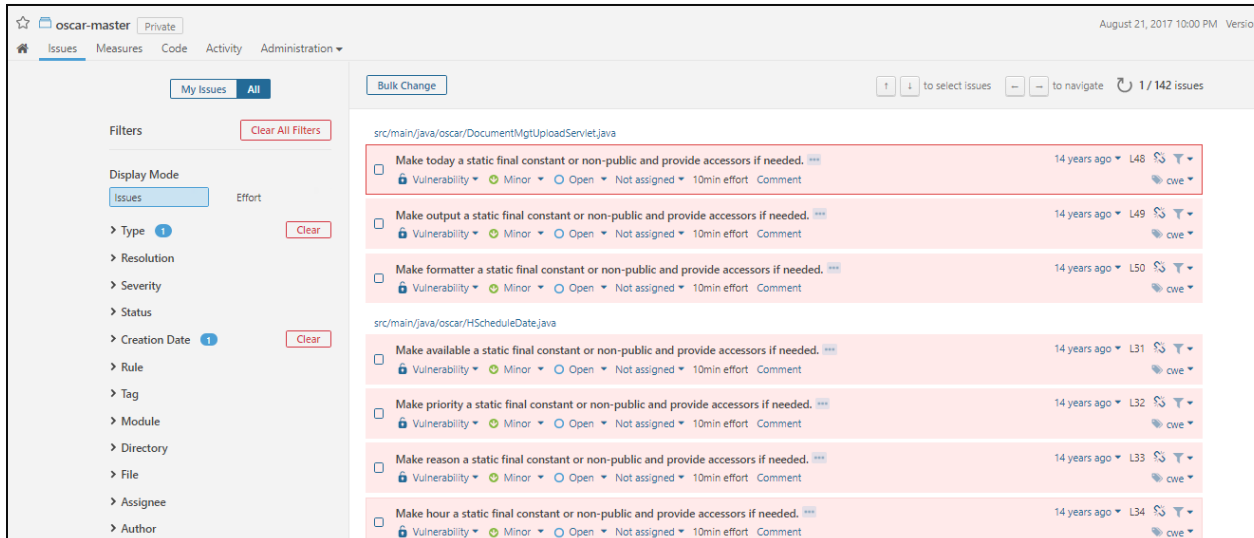


Figure 5.4: Parameters used to Administrate Analysis Results

The following parameters act as sub-divisions and define one category of analysis results, i.e. **Vulnerabilities**. The procedure to administrate and manage issues produced in other categories remain the same.

- **Resolution** gives an estimate number of Unresolved, Resolved, False Positives, Won't Fix and Removed vulnerabilities.
- **Status** gives information about the number of Open, Resolved, Reopened, Closed and Confirmed vulnerabilities.
- **Creation Date** is a very useful parameter while comparing the vulnerabilities from previous years starting from 2003 to 2016. This section creates a bar graph for the vulnerabilities from the beginning of the project and displays the year responsible for the addition of most number of vulnerabilities. Every bar in the graph denotes the number of vulnerabilities produced in one year, for example the bar depicting the range from 31 December 2011 to 30 December 2012 shows the highest number of vulnerabilities having showed up in the source code this year.

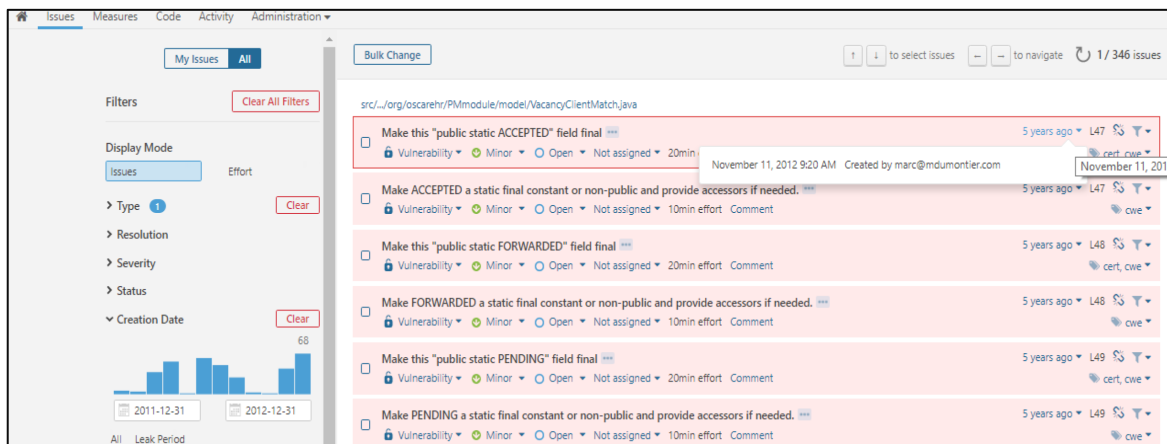


Figure 5.5: Accessing Creation Date

- **Rule** parameter consists of a list of rules against which the displayed vulnerabilities are identified. SonarQube consists of pre-defined rules for every language, which are checked against the source code for detecting issues present in it. The lines of codes which do not justify these rules are marked as a vulnerability, code smell or bug depending upon its category.
- **Tag** categorizes vulnerabilities as per the categories associated with the industry standards, such as cert, owas-a3, error handling, sans-top-25 insecure, cwe and misra [19].
- **Directory** specifies the location of the directory corresponding to the number of vulnerabilities associated with that directory.
- **File** is similar to the directory parameter and specifies the location of the file along with the number of vulnerabilities associated with it.
- **Assignee** lists names of the individuals assigned with the task of resolving vulnerabilities along with the number of vulnerabilities assigned to each assignee.
- **Author** lists names of the individuals who have contributed to source code since the beginning of the project.
- **Language** parameter specifies the languages which have been used to develop the source code of the project, like in this case it is Java, JavaScript and Flex.

The parameters defined above provide an ease of access for vulnerabilities through sub-divisions. Moreover, clicking on an individual vulnerability will redirect to the source code of the project pointing to the line of code and the location of existing vulnerability in the file. Figure 5.6 illustrates the name, severity, status, creation date and tags for a vulnerability.

Figure 5.6: Accessing source code for vulnerabilities

In addition to this, Figure 5.6 gives detailed explanation on non-compliant source code because of which the vulnerability exists and its compliant solution.

5.2.2 Authorization to Analysis Results

Authorization plays a very important role for the managing security and privacy of projects with highly sensitive information. For this purpose, SonarQube has an option to create different Users and User Groups for different projects. The build for sonar analysis initiates on Jenkins server and the projects consisting analysis results are created on SonarQube server. The privacy of the analysis results displayed on SonarQube dashboard is ensured by authorized access to the project names and results. Therefore, in the same organization, one project can be hidden from the team working on the other project and vise-versa. SonarQube comprises of the following options in **Administration>Security tab** which help to maintain the security and privacy of the project and its analysis results.

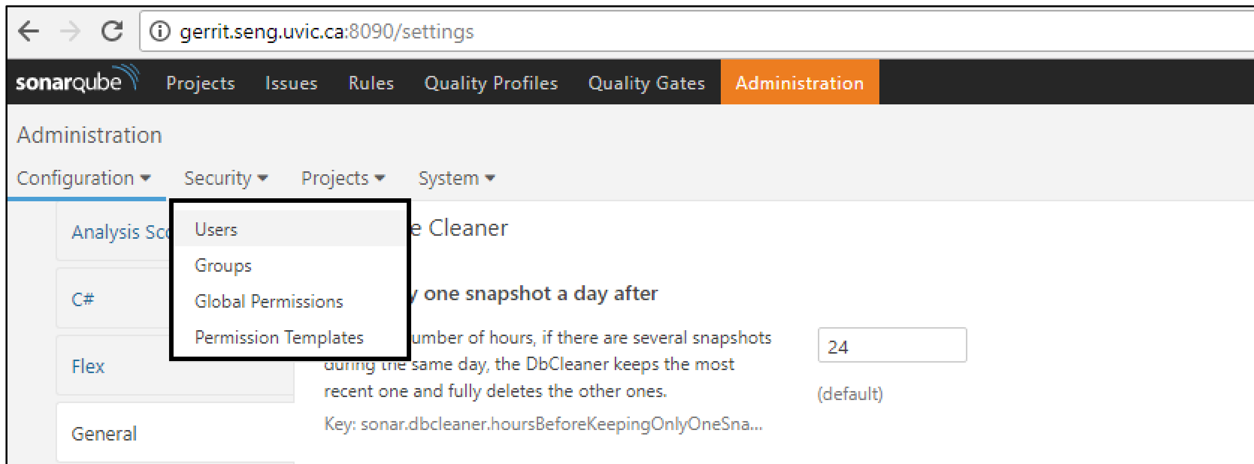


Figure 5.7: Configuring Authorization and Security Settings

- **Users** refer to the authorized individuals that have access to the project's analysis results. These Users further appear in the Assignee list as discussed in Section 5.2.1 to resolve vulnerabilities assigned to them. In addition to the Users, this section helps to create more than one administrator for SonarQube server.
- **Groups** by default exist as sonar-administrators and sonar-users comprising of administrators and regular users respectively.
- **Global Permissions** are the security restrictions applied to all the projects listed on the SonarQube dashboard. These permissions are strictly based on the authority to administer System, Quality Profiles, Quality Gates, Execute Analysis and Create Projects.

Global Permissions

Grant and revoke permissions to make changes at the global level. These permissions include editing quality profiles, sharing dashboards, and performing global system administration.

All | Users | Groups | Search

	Administer System	Administer Quality Profiles	Administer Quality Gates	Execute Analysis	Create Projects
Anyone	☐	☐	☐	☐	☑
sonar-administrators System administrators	☑	☑	☑	☑	☑

Figure 5.8: Administrating global permissions

Global permissions are further categorized for Users and Groups authorized to access project results. It may be noted that a few Users might have more global permissions than the others.

- **Permission Template** defines the privileges granted to various Groups and Users, such as browsing, checking source code of the project and executing analysis. The default permission template is used when no other permission configuration is available, which can be updated as per the authorization requirements of the project.

Authorization settings play a prominent role in managing security and privacy of projects for various project teams by customizing the access for the team members. Also, it provides a platform to assign vulnerabilities and issues to authorized Users as shown in Figure 5.9.

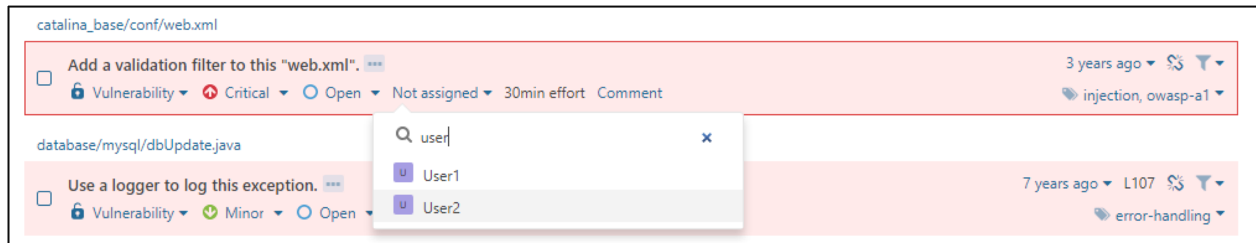


Figure 5.9: Administrating assignees for vulnerabilities

Assignees can further perform various actions on the vulnerabilities assigned to them. The privileges defined in authorization settings allow the assignees to change the status of a vulnerability from Open to Closed, Won't Resolve, Fixed or False positive along with a comment explaining the reasons of the change in status.

5.2.3 Administrating Configuration Settings

The global or general settings for a SonarQube instance are configured by navigating to **Administration>Configuration**. These settings help to configure analysis scope, database history, force user authentication and installing third-party plugins from the Update Center. The project, oscar-master, has a set of requirements that are to be configured from the general settings as described below.

- **Clean Directory/Package History** is set to false for this project as the history of analysis results assists in the representation graphical data depicting variation in vulnerabilities over the past years.
- **Force User Authentication** is set to true under **Configuration>Security** tab and ensures to redirect the user accessing SonarQube server from Jenkins to the login page for authentication.

6 Analysis Results

6.1 Understanding Analysis Results

Analysis results are generated by mapping the rules specified for a language against project's source code. These rules check for coding conventions, bugs, duplications, vulnerabilities, security and technical debt and report the issues as per severity levels blocker, critical, major, minor and informational. The analysis results generated for OSCAR's source code comprise of 2.2k bugs and .19k vulnerabilities from 835k lines of code. The issues addressed by sonar-scanner as per categories are discussed below to have a better understanding of analysis results.

1. Bugs
   As stated, 2.2k bugs are identified from OSCAR's current version, which are sub-categorized as severity level, creation date, resolution, status, language and assignee. As per severity levels, the source code consists of 243 blockers, 50 critical, 1.7k major and 249 minor bugs. It may be noted that the "major" severity level is contributing the most to the list of bugs and primarily identified dead store values [19], NullPointerException [19] and Serialize nature [19] of classes from over 200 directories. In addition to this some of the critical bugs like failure to properly close Connection, Statement, PreparedStatement, FileInputStream and ResultSet were identified as non-compliant code. The creation date parameter also states that the number of bugs are listed highest in the year 2015-2016 followed by the year 2011-2012.

2. Vulnerabilities
   Vulnerabilities have comparatively low rate of existence in OSCAR's source code with a count of 1.9k existing in the current version. In comparison to Bugs, no vulnerability with "major" severity level is detected from the code. Although, the number of blocker and critical vulnerabilities is found to be 59 and 86 respectively. The rest 1.8k vulnerabilities belong to minor severity level. Most of the vulnerabilities identified are related to adding a validation filter to various xml files, using a variable binding mechanism instead of concatenation to construct queries, removal of hard-coded passwords from a few files and reviewing the arguments of "eval" [19] call for validation. Moreover, the year 2011-2012 displays the highest number of vulnerabilities with a count of 346. The identification of these vulnerabilities is based on industrial experience and tags, such as cert, owasp-a3, sans-top-25-insecure and error handling [19].

6.2 Filtration Methodology

Filtering analysis results require a filtration methodology to target and eliminate critical, blocker and major bugs and vulnerabilities from source code. Filtration methodology adopted for addressing such vulnerabilities is defined by applying the filters, such as type of issue, severity, resolution and rule. While SonarQube addresses and categorizes the raised vulnerabilities on several additional levels like author, assignee, tag, module, directory and file but the filters listed above perform crucial roles while identifying and extracting critical and blocker vulnerabilities. This filtration methodology forms a hierarchy of filters where the sub-filters act on the data extracted by their parent filter as defined below.

1. Type: This filter allows to extract the type of issue as "Vulnerability" from the listed issues and forms the parent filter of all the other sub-filters in this hierarchy.
2. Resolution: This filter is important to identify the resolution of vulnerabilities as Unresolved, Resolved, Won't Fix, Fixed and False Positives. The category "Unresolved" associated with the filter is selected which lists all the unresolved and non-addressed issues as vulnerabilities in OSCAR's source code
3. Severity: The open vulnerability issues need to be displayed as per severity level in this step by selecting the severity sub-categories critical and blocker. This filter then targets all the unresolved vulnerabilities with severity levels blocker and critical simultaneously on the listing panel.
4. Rule: This filter lists the rules responsible for identification of unresolved vulnerabilities with critical and blocker severity levels. While addressing such vulnerabilities, this filter helps to minimize the range of resolution and assists the developers to resolve similar type of issues at one time rather than revisiting them and applying the solution again as shown in Figure 6.1.

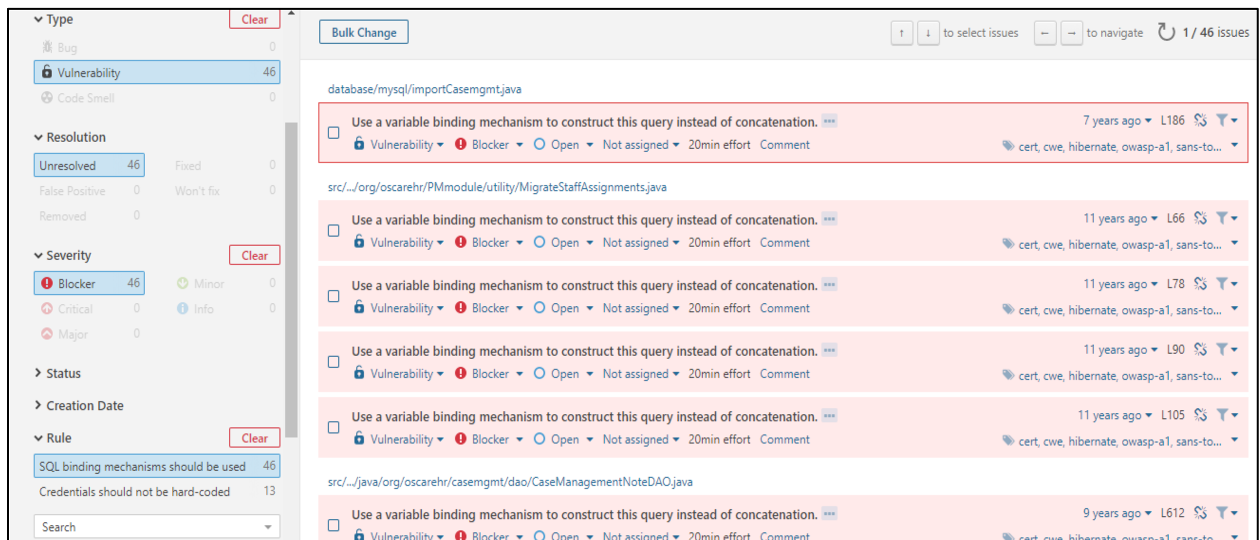


Figure:6.1 Filtering Analysis Results

6.3 Graphical representation of Analysis Results

Graphical representation of vulnerabilities helps to understand the variation in number over the years. For Oscar's source code, it is interesting to see this variation since the adoption of an ISO certified Quality Management System, for the period 2011 to 2016. In Section 5.1.1, the parameter creation date performs a similar function by identifying and organizing the number of vulnerabilities represented in a bar graph.

The difference between creation date parameter and graphical representation is that the latter allows to clone a specific commit id from a specific date and release from BitBucket into Jenkins. BitBucket [7] has a history of all the previous commits and 'git log' [7] on the cloned git repo is used to see the commit message and the date of commit. While commit message helps to understand the type of changes associated with the commit, the date of commit helps to validate the duration period between two cloned commits.

A strategy needs to be established to clone five commit ids from BitBucket as different versions of the project over the course of five years, such that the commits which show huge changes in the project over these years are cloned into Jenkins. For instance, the old Jenkins server consisting of 533 builds of stable (oscar15BetaMaster) is compared with the build 361 from the same site, which was the QA release of Oscar 15, hence providing the variation of change required to represent the analysis results graphically.

6.3.1 Configuring Jenkins for Cloning Commit Ids

The commit ids are first cloned in Jenkins configuration. Instead of building */master branch, specific commit id associated with the release is build. For developing OSCAR's source code, major changes have appeared since the year 2012, hence the variation of security risks is checked during these years to attain a better idea of analysis trend.

As the source code for OSCAR is hosted on BitBucket, commits specifying the major changes or different releases of OSCAR are cloned in Jenkins and analyzed. In addition to this, the analysis properties in the build section are changed as per the version and the date of project as shown in Figure 6.1.

Analysis properties

```
# unique project identifier (required)
sonar.projectKey=oscar-stable

# project metadata (used to be required, optional since SonarQube 6.1)
sonar.projectName=oscar-stable
sonar.projectVersion=4.0

# path to source directories (required)
sonar.sources=/var/lib/jenkins/workspace/oscar-stable

sonar.projectDate=2015-09-11
```

Figure 6.2: Analysis properties configuration for plotting graphical data

6.3.2 Accessing Graphical representation of Analysis Results

The graphical representation of the analysis results produced by SonarQube is accessed on SonarQube server. The project dashboard charts out comparisons between the most recent analysis and the previous analysis by issuing security, reliability and coverage code rating on the new code or analysis. In case the most recent analysis has lower security or reliability rating than the previous one, dashboard displays the Quality gate of the project to be failed as shown in Figure 6.2.

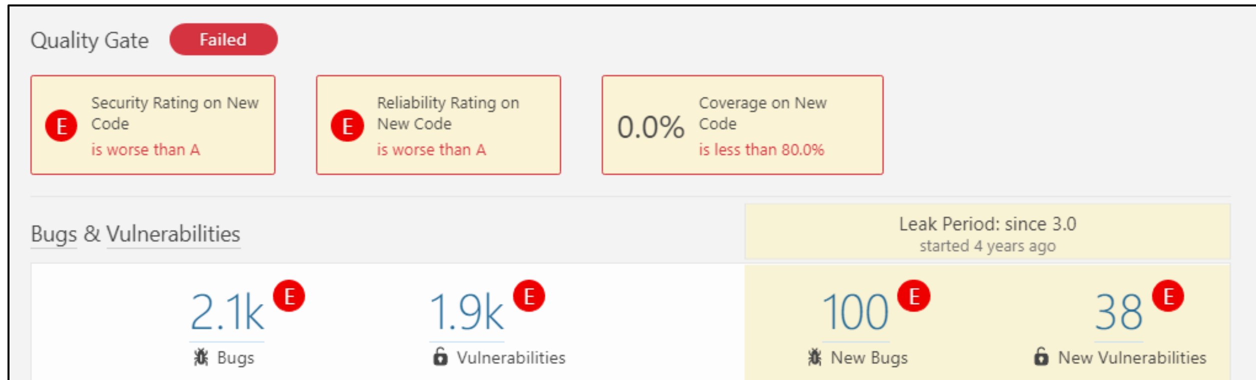


Figure 6.3: Comparing two consecutive analysis

While project dashboard displays security ratings and compares two consecutive analysis, graphical representation of more than two analyses is shown under Measures> Security> History. Figure 6.3 shows such a graphical representation of the vulnerabilities varying in the source code of OSCAR from years 2012 to 2017.

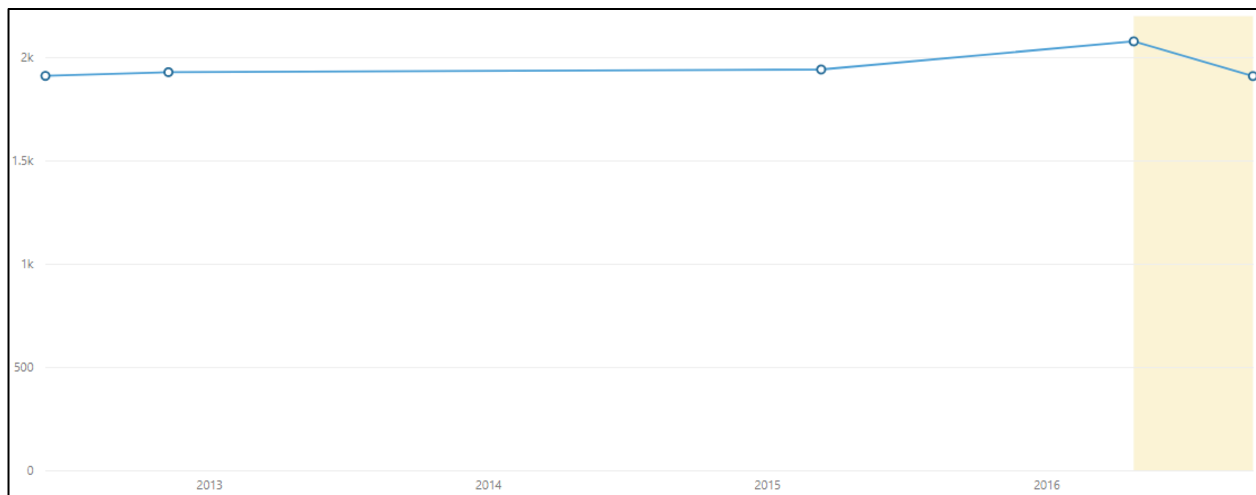


Figure 6.4: Graphical Representation of Analysis Results

In Figure 6.3, the vertical axis denotes the 'number of vulnerabilities' plotted against the 'years raised' on the horizontal axis. It can be clearly seen that there is not much variation in the number of vulnerabilities over the years except the years 2012-13 and 2016-17 show a drop in the count of vulnerabilities.

7 Challenges and Solutions

The integration of Jenkins and SonarQube, where the tools reside on host vagrant VM, is the most challenging pre-requisite for static code analysis. In addition to this, managing networking connections from host machine to guest VM is one of the prominent challenges faced during project setup. As the operation of this project is meant to be on the server instead of local installation, therefore it is vital to analyze the potential access of the guest machine to the host machine and the internet. Some of the challenges faced during the project set-up are describes as follows.

1. The location of the tools on the guest machine should be inside Jenkins folder. This folder is created as a result of setting up Jenkins on the server gerrit.seng.uvic.ca:8080 and consists of project workspace, plugins installed, tools, logs, jobs, users and nodes sub-folders. The location of SonarQube and Sonar-Scanner must be within the folder var/lib/jenkins as the plugins installed on the Jenkins server can only interact with the files inside the Jenkins folder.

2. The networking configuration of guest machine is to be dealt with carefully, otherwise resulting into host machine failing to connect to the guest machine via ssh. Once the guest machine is locked for the connection, it becomes very difficult to retrieve the machine. Therefore, it must have its backup copy on the host machine, which helps to retrieve the machine in case it dismisses the connection from the host. The networking configuration, however, can be modified from outside of guest VM from its Vagrantfile located on the host machine.

3. Vagrantfile consists of numerous options for guest machine configuration including the list of forwarding ports. Creating a forwarded port mapping allows access to a specific port within the machine from a port on the host machine as shown below.

config.vm.network "forwarded_port", guest: 8080, host: 8080

config.vm.network "forwarded_port", guest: 8090, host: 8090

In addition to this, both the ports 8080 (for Jenkins) and 8090 (for SonarQube) must be open in the Firewall settings of gerrit.seng.uvic.ca.

4. In case of the SonarQube server not starting up, there might be a java process already running for the tool which prevents the re-initialization of the server. The status of the SonarQube server is displayed by the command./sonar.sh status. If the status displayed is ‘not running’, then the java processes need to be killed using the following commands:

ps -aux | grep java displays the process id associated with the java processes

kill -9 pid is used to kill the process associated with the process id retrieved from the above command.

The SonarQube server can be started again and validated by using the link http://gerrit.seng.uvic.ca:8080.

8 Conclusion and Future Scope

8.1 Conclusion

The idea behind integrating Jenkins and SonarQube is to setup a system efficient for meaningful static code analysis of huge projects, like Oscar. Static code analysis, if used at the right stage during the development of a project, provides huge benefits of identifying critical vulnerabilities or bugs which may not appear to the surface during or after the project release. In addition to this, code management and documentation are the most vital aspects of a project, which sometimes are pushed to the rear side because of deadlines, therefore integration of these tools ensures to provide a successful user experience. With a tool like SonarQube, it becomes very easy to assign a vulnerability to a team member, eliminate generated false positives and setup a quality gate which fails a build when the quality of the project does not meet specified quality threshold.

The outcome of implementing Jenkins integrated with SonarQube for static code analysis is that it is not only useful for maintaining and assuring the security of one project but the configuration can be used by as many projects without the restriction of language used to develop the project. The main aim was to provide a setup, which is independent of the type of SCM, length of the source code, language written in and efficient to manage and customize analysis results as per the project's requirements.

8.2 Future Scope

The introduction of SonarQube as a static code analysis tool forms a strong foundation for Agile development [20], where code is integrated frequently and production deployment occurs regularly. In such scenarios, SonarQube holds a high potential to improve software quality and delivery standards of any project team or organization. With the help of SonarQube and Jenkins, integrated code is regularly build and analyzed to identify blocker, critical or security threats persisting in the code.

In addition to this, SonarQube can be integrated as a plugin with local IDEs such as Eclipse known as SonarLint. SonarLint [20] is an Eclipse plugin that provides on-the-fly feedback to developers on new bugs and quality issues injected into Java, JavaScript, PHP, Python code. SonarLint offers a fully-integrated user experience in Eclipse-based IDEs. After installing the plugin, issues are reported as Eclipse markers. The Eclipse projects, if required, can be further connected to their alter-existing SonarQube projects using SonarLint analyzers, quality profiles and configuring SonarQube server within Eclipse [20]. Moreover, the introduction of static code analysis for the Oscar's source code unlatches the path for dynamic code analysis. In contrast to the static code analysis, dynamic code analysis relies on the behaviour of the code during execution. Both the static and dynamic code analysis form an integral part of the security review.

Bibliography

[1] *Documentation - SonarQube Documentation - Doc SonarQube*. (n.d.). Retrieved from Docs.sonarqube.org: https://docs.sonarqube.org/display/SONAR/Documentation

[2] *OSCARMcMaster*. (n.d.). Retrieved from SourceForge: https://sourceforge.net/projects/oscarmcmaster/

[3] Pecanac, V. (n.d.). *Top 8 Continuous Integration Tools - Code Maze*. Retrieved from Code Maze: https://www.code-maze.com/top-8-continuous-integration-tools/

[4] *Welcome — Site*. (n.d.). Retrieved from Oscarmanual.org: http://oscarmanual.org/home

[5] Rattan, J. (2013, Oct. 5). *The Architecture of Open Source Applications (Volume 2): OSCAR.* Retrieved from Aosabook.org: http://aosabook.org/en/oscar.html

[6] *Products | Atlassian*. (n.d.). Retrieved from Atlassian: https://www.atlassian.com/software

[7] Fowler, M. (n.d.). *Continuous Integration.* Retrieved from martinfowler.com: https://martinfowler.com/articles/continuousIntegration.html

[8] *Static Code Analysis.* (n.d.). Retrieved from Mathworks.com: https://www.mathworks.com/discovery/static-code-analysis.html

[9] Oktaba, P. (2015, Aug 24). *Checkstyle vs PMD vs Findbugs | Continuous Dev.* Retrieved from Continuousdev.com: http://continuousdev.com/2015/08/checkstyle-vs-pmd-vs-findbugs/

[10] Keary, E. (2008). *The OWASP Code Review Guide 1.1.* Retrieved from: https://www.owasp.org/images/a/a1/OWASP_Code_Review_Guide-_V1_1.doc

[11] *Gerrit Code Review - A Quick Introduction.* (n.d.). Retrieved from Review.openstack.org: https://review.openstack.org/Documentation/intro-quick.html

[12] Palko, T. (2014, Dec 11). *DevOps Technologies: Vagrant.* Retrieved from Insights.sei.cmu.edu: https://insights.sei.cmu.edu/devops/2014/12/devops-technologies-vagrant.html

[13] *Vagrant Cloud by HashiCorp.* (n.d.). Retrieved from Vagrant Cloud by HashiCorp: https://app.vagrantup.com/boxes/search

[14] Tatham S. (Jul 2017). *PuTTY: a free SSH and Telnet client. Retrieved from* https://www.chiark.greenend.org.uk/~sgtatham/putty

[15] *Getting Started - Vagrant by HashiCorp.* (n.d.). Retrieved from Vagrant by HashiCorp: https://www.vagrantup.com/intro/getting-started/boxes.html

[16] *Linking to source code repositories - Atlassian Documentation.* (n.d.). Retrieved from Confluence.atlassian.com: https://confluence.atlassian.com/bamboo/linking-to-source-code-repositories-671089223.html

[17] Avery, R. G. (2016, Sept 15). *Jenkins 101: Getting Started.* Retrieved from Slideshare.net: https://www.slideshare.net/rGeoffrey/jenkins-101-getting-started

[18] *Metric Definitions - SonarQube Documentation - Doc SonarQube.* (n.d.). Retrieved from Docs.sonarqube.org: https://docs.sonarqube.org/display/SONAR/Metric+Definitions

[19] Roongsiriwong, N. (2017, Apr 1). *Top 10 Bad Coding Practices Lead to Security Problems.* Retrieved from Slideshare.net: https://www.slideshare.net/narudomr/top-10-bad-coding-practices-lead-to-security-problems

[20] *SonarLint for Eclipse*. (n.d.). Retrieved from Sonarlint.org: http://www.sonarlint.org/eclipse/index.html