Lab 2: Static Application Security Testing Report

Kollin Labowski

CYBE 366

May 1, 2021

Abstract

This lab assignment was intended to be an opportunity to experience the process of Static Application Security Testing (SAST) firsthand. For the assignment, a SAST tool called SonarQube was chosen, and this tool was used to scan a piece of vulnerable software. The software to be tested was selected by performing a search of different software for ones that exhibited vulnerabilities discussed in class, and the software that was selected was a web application which was programmed for another class. After scanning the software and determining the vulnerabilities it contained, the next step was to patch the vulnerabilities using techniques that were discussed in class. Once the vulnerabilities have been patched, the scanning tool must be run again to show that the vulnerabilities have in fact been fixed. The type of vulnerability that was fixed in this lab was a SQL injection vulnerability, and this was done by using prepared statements rather than inserting inputs directly into SQL statements.

Introduction

Static Application Security Testing (SAST) is the process of determining potential vulnerabilities in a software by review the source code of the software. This is a static process, meaning that it occurs before the code is run, or at compile time rather than runtime. SAST is a very important step in the process of developing secure software, as it can help to prevent malicious hackers from exploiting simple vulnerabilities in the coding. Because software often is made with large amounts of code, it is not usually very easy or efficient to perform SAST manually, at least not all of it. Luckily, there exist tools that automate the SAST process by scanning the code to look for known vulnerabilities. Unfortunately, the scanning tool may not find every single potential vulnerability within a software, however it should find most of them, particularly those which are the most dangerous.

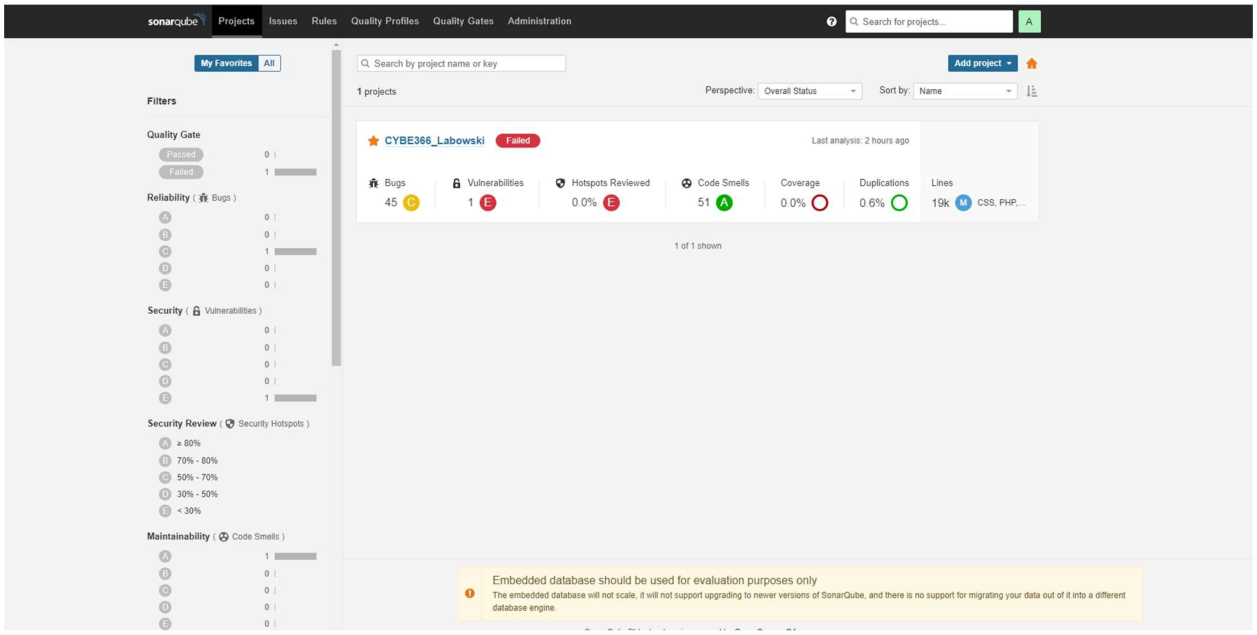
For this lab, a SAST tool will be selected to use for scanning a piece of vulnerable software. The vulnerable software should be chosen such that it contains at least one vulnerability that was discussed in class. These vulnerabilities include buffer overflows, SQL injections, and cross-site scripting (XSS) attacks most notably. Once the software has been selected, the tool chosen for performing the scan of the code will be used to determine the vulnerabilities present within the software. Based on the output of the SAST tool, the present vulnerabilities relating to class topics should be patched, such that when the tool is run on the code again, the vulnerabilities no longer appear. Ideally, by the end of the lab, a more secure version of the software should have been produced.

This lab was conducted on a Windows 10 machine. This machine is a local Windows 10 machine, not a virtual machine. Code was accessed, run, and changed using Visual Studio Code.

Procedures and Results

Task 1: SAST Tool Selection

The first task assigned for the lab was to select a SAST tool to use on the vulnerable software. The tool suggested for use in this lab is called SonarQube, and this software was downloaded to use in the next task. SonarQube is an open-source platform that was made for ensuring quality and security during the software development process. SonarQube is a tool made for more than just SAST, as it will also test for bugs and "smells" within the code, which are both helpful to be aware of when developing software. SonarQube is compatible with Java, JavaScript, C#, TypeScript, Kotlin, Ruby, Go, Scala, Flex, Python, PHP, HTML, CSS, XML, and VB.NET. It is important that the selected software is written in one or more of these programming languages, otherwise SonarQube will be ineffective. SonarQube organizes its findings from scanning the code into bugs, code smells, vulnerabilities, and security hotspots. The vulnerabilities and security hotspots sections are of particular interest for SAST, as they deal with the actual security of the software. After running SonarQube on the selected software, the focus should be on the output in these sections. The software works by running it in the Command Prompt from the top-level directory of the code to scan. More details on how to specifically run the software will be discussed during task 3. See **Figure 1** below to see an example of the interface for the SonarQube tool.



Embedded database should be used for evaluation purposes only

The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.

Figure 1: The Projects page of the SonarQube testing tool

Task 2: Vulnerable Software Selection

The next task to be completed for the lab assignment was to select a software to user SonarQube on. The source code was tested for several different pieces of software. Piazza, Blackboard, Roompact, StarRez, and a few others were tested on various versions for known vulnerabilities, however none of them returned any vulnerabilities covered in class, and in fact, most returned no significant vulnerabilities at all. Eventually, the scanner was run on the source code for a web application created in a previous class (CS 230). The source code was written mostly in PHP, HTML, CSS, and JavaScript, all of which are languages supported by SonarQube. The web application itself was made to act as a collection of local restaurants in the Morgantown area, with individual pages for different restaurants. The main landing page, index.php, for the application can be seen in **Figure 2**.

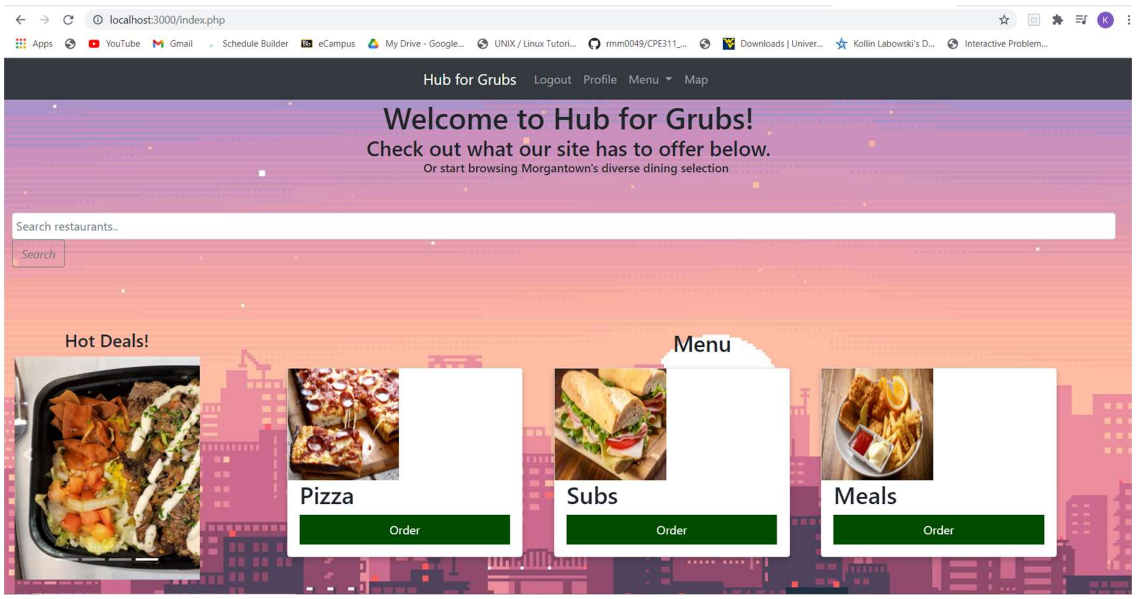


Figure 2: The page index.php from the web application tested for the assignment

Several pieces of information for the application were stored in a SQL database. Among the information that was stored in the database was a table of users for the application, which was used to verify user information so they could log in to their account. The login page for the web application, login.php, can be seen in **Figure 3**, and the login information is checked against the database in another file, login-helper.php, however, no vulnerabilities appear in this file.

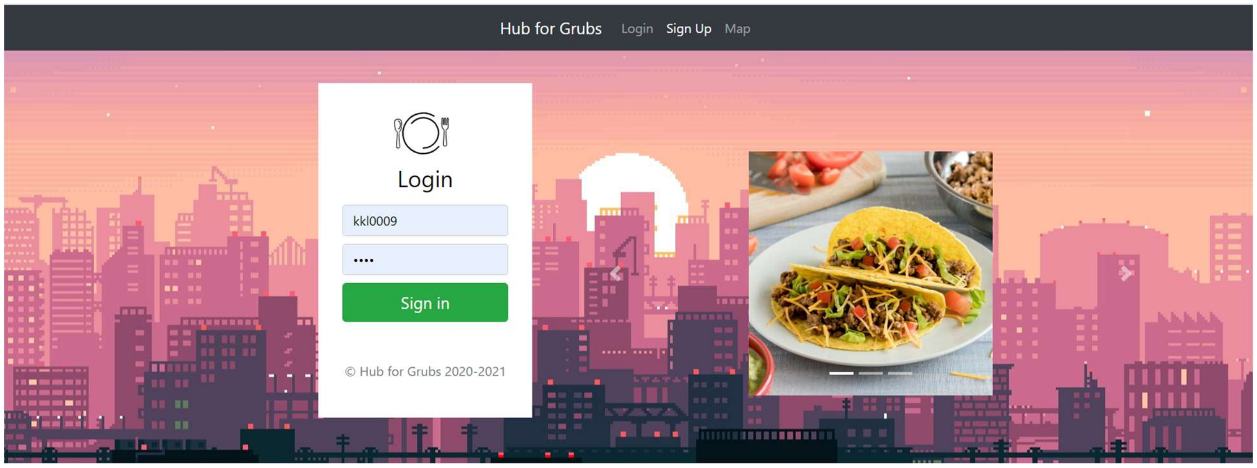


Figure 3: The login page for the web application that is tested

Closely related to the login page was the signup page, signup.php. This is the page where a new account for the application can be made. This account will be added to the table of users in the file signup-helper.php, which works with this page. This system does not include any SQL vulnerabilities either, but it is one of the main ways the application interacts with the database. The signup page can be seen in **Figure 4**.

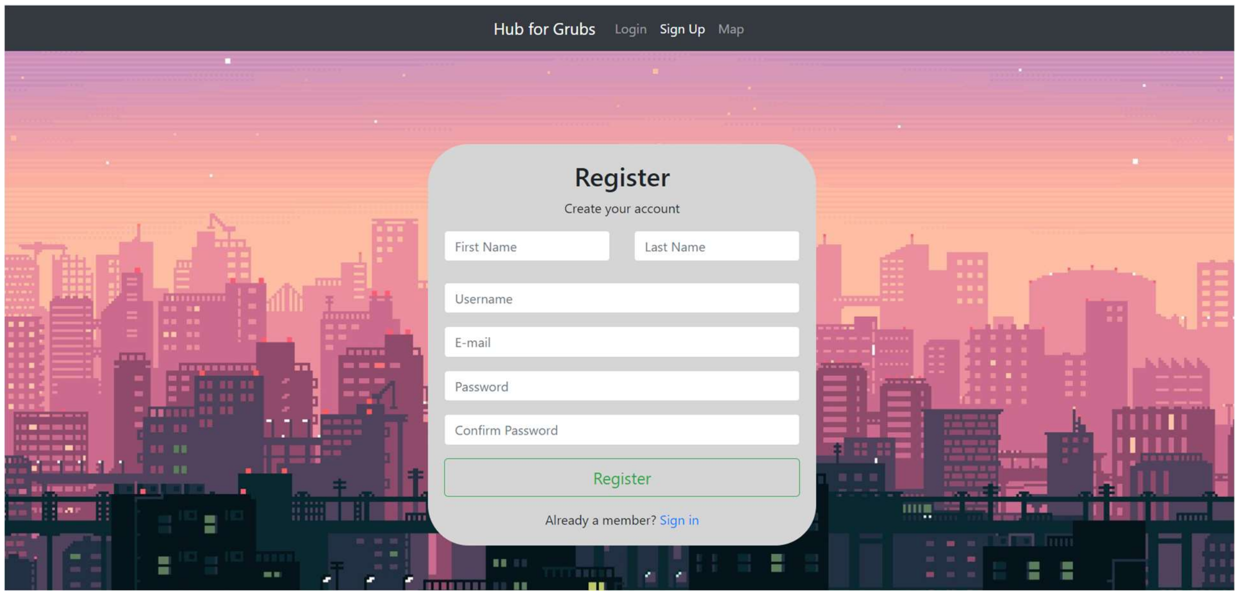


Figure 4: The signup page for the web application

The full SQL database that was used can be examined or edited to get a better idea of how it interacts with the application. This is done by viewing the database after logging in to PHPMyAdmin using credentials that were assigned when the application was created. The tables in the database are listed in **Figure 5**, including the table of particular interest for this assignment, the reviews table.

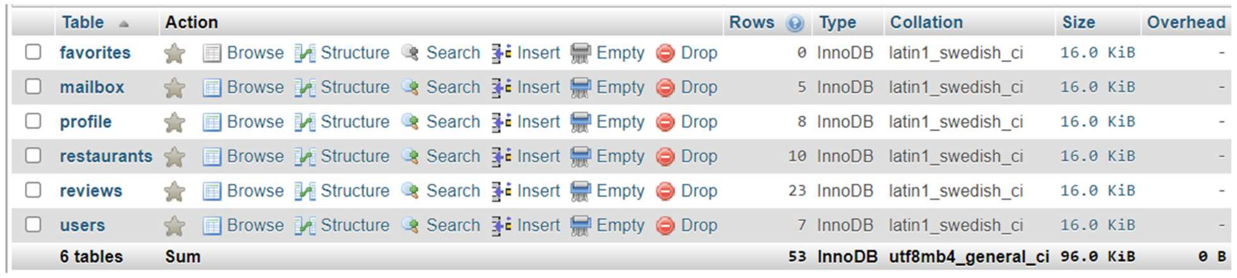


Table	Action	Rows	Type	Collation	Size	Overhead
favorites	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	16.0 KiB	-
mailbox	Browse Structure Search Insert Empty Drop	5	InnoDB	latin1_swedish_ci	16.0 KiB	-
profile	Browse Structure Search Insert Empty Drop	8	InnoDB	latin1_swedish_ci	16.0 KiB	-
restaurants	Browse Structure Search Insert Empty Drop	10	InnoDB	latin1_swedish_ci	16.0 KiB	-
reviews	Browse Structure Search Insert Empty Drop	23	InnoDB	latin1_swedish_ci	16.0 KiB	-
users	Browse Structure Search Insert Empty Drop	7	InnoDB	latin1_swedish_ci	16.0 KiB	-
6 tables	Sum	53	InnoDB	utf8mb4_general_ci	96.0 KiB	0 B

Figure 5: The tables of the database for the application, as viewed in PHPMyAdmin

The reviews table is accessed from the files that work with the review page, restpage.php. To navigate to a restaurant's review page, one can select "Browse Restaurants" from the dropdown menu in the header. This will navigate to the list of restaurants, with the main file for this page being restaurants.php. Clicking on any of the restaurants will open the review page for that restaurant. The search bar can be used to find the review page for a particular restaurant. The list of restaurants can be seen in **Figure 6**.

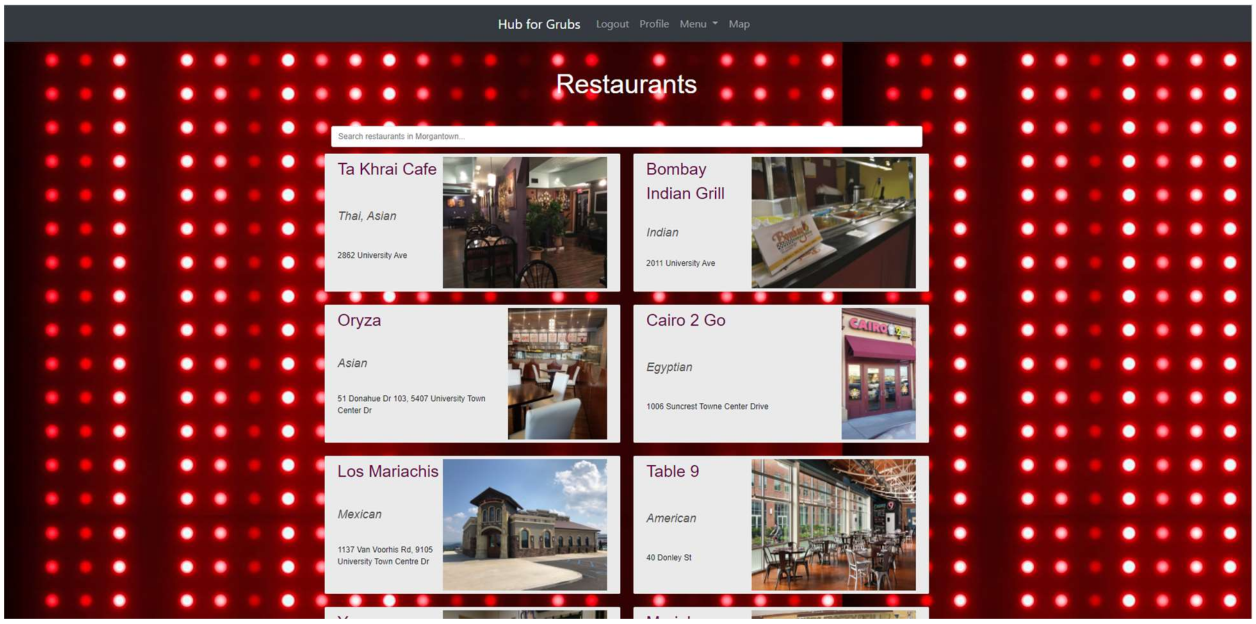
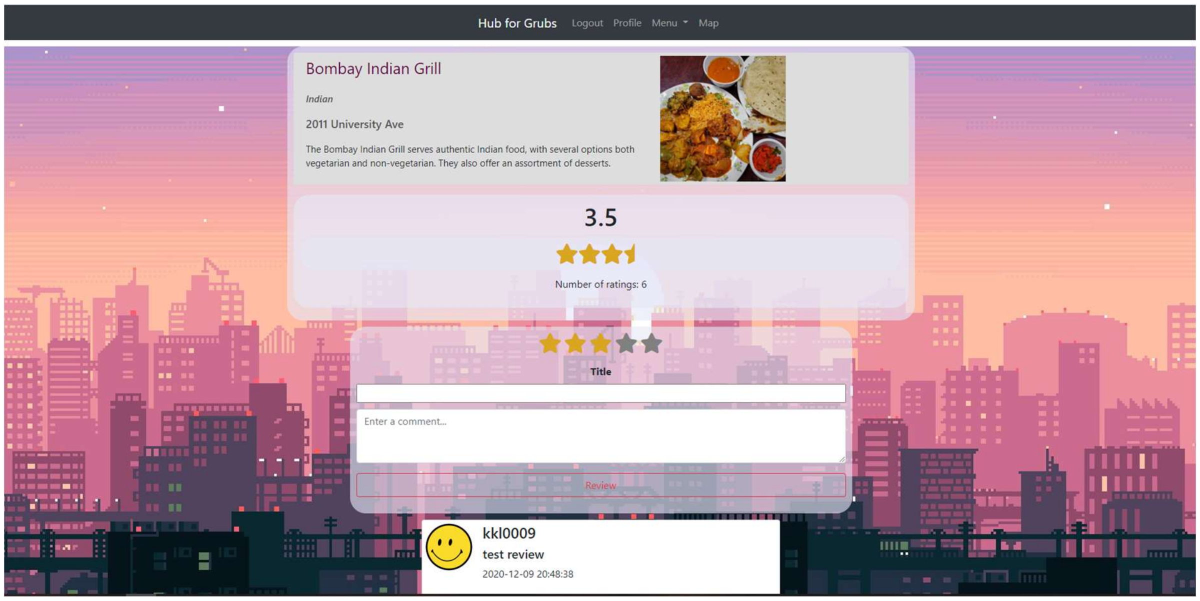


Figure 6: The list of restaurants that can be viewed on the website

After selecting a restaurant from this menu, the review page for that restaurant will be opened. The review page includes a brief description of the restaurant, and more notably, a list of user reviews and a section to leave a review. There are two files in addition to restpage.php that deal with displaying reviews and parsing ratings respectively. Those files are display-reviews.php and get-ratings.php. Because these files must read information from the SQL database, they each include the use of mysqli commands. These two files contain the SQL injection errors which will be the focus of the SAST. The main review page must also access the database to post a new review, however prepared statements were used here to eliminate the threat of SQL injections. See **Figure 7** to see the review page for the restaurant "Bombay Indian Grill".



Bombay Indian Grill

Indian

2011 University Ave

The Bombay Indian Grill serves authentic Indian food, with several options both vegetarian and non-vegetarian. They also offer an assortment of desserts.

3.5

Number of ratings: 6

kkl0009

test review

2020-12-09 20:48:38

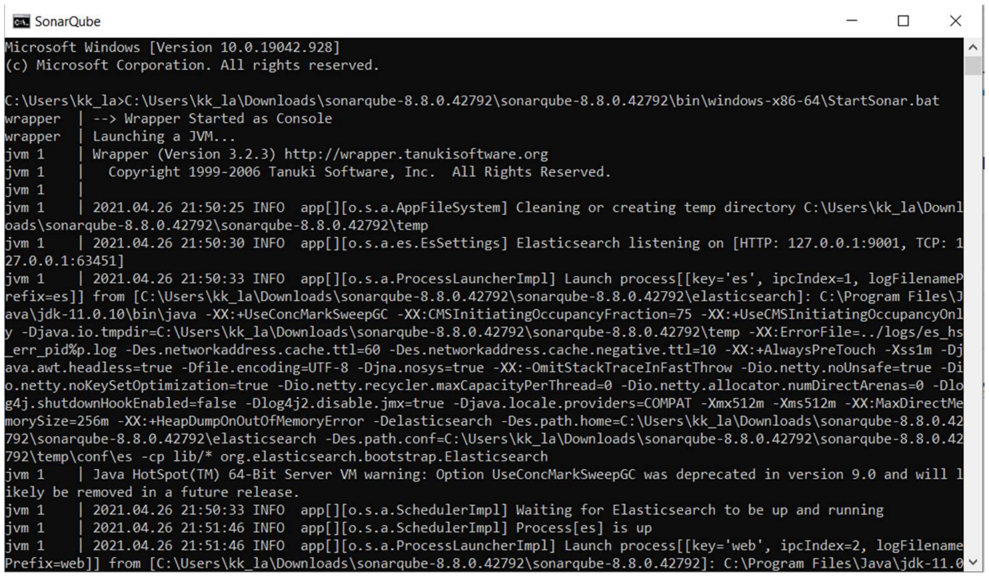
Figure 7: An example review page on the web application, which uses two vulnerable files

The specific vulnerability will be discussed after the SonarQube tests are run on file structure of the software. The vulnerabilities are both SQL injection vulnerabilities caused by the insertion of user input directly into a SQL query. This is dangerous because a user could input a

SQL query to collect more information on the system, or even manipulate parts of the database they should not have access to. This corresponds to CWE-89, “Improper Neutralization of Special Elements used in a SQL Command (‘SQL Injection’)”. The actual vulnerable code will be explained during the completion of task 4, and it will be fixed using prepared MySQL prepared statements.

Task 3: Vulnerability Scanning

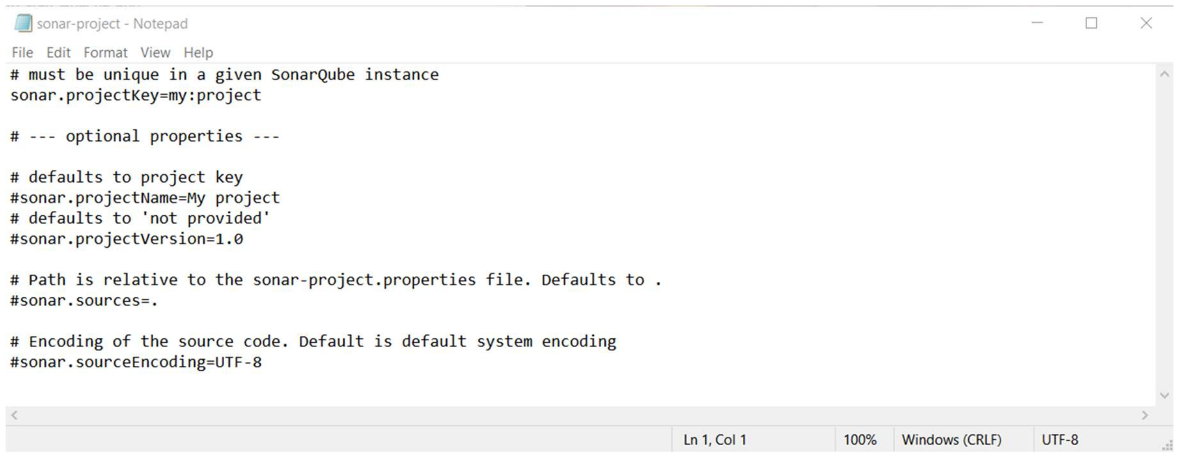
Running the SonarQube testing software can be accomplished by following along the steps on the SonarQube website. First, the zip folder on the website containing the SonarQube console should be downloaded. Because this lab was completed on a Windows machine, the Windows version of the terminal was accessed in the bin folder within the zip file. This terminal output some information before displaying “SonarQube is up” about a minute later. Notably JDK 11 had to be installed for this initialization process to work correctly. A portion of the output from the SonarQube terminal can be seen in **Figure 8**.



```
Microsoft Windows [Version 10.0.19042.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\kk_la>C:\Users\kk_la\Downloads\sonarqube-8.8.0.42792\sonarqube-8.8.0.42792\bin\windows-x86-64\StartSonar.bat
wrapper  | --> Wrapper Started as Console
wrapper  | Launching a JVM...
jvm 1    | Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
jvm 1    |   Copyright 1999-2006 Tanuki Software, Inc.  All Rights Reserved.
jvm 1    |
jvm 1    | 2021.04.26 21:50:25 INFO  app[][o.s.a.AppFileSystem] Cleaning or creating temp directory C:\Users\kk_la\Downl
oads\sonarqube-8.8.0.42792\sonarqube-8.8.0.42792\temp
jvm 1    | 2021.04.26 21:50:30 INFO  app[][o.s.a.es.EsSettings] Elasticsearch listening on [HTTP: 127.0.0.1:9001, TCP: 1
27.0.0.1:63451]
jvm 1    | 2021.04.26 21:50:33 INFO  app[][o.s.a.ProcessLauncherImpl] Launch process[[key='es', ipcIndex=1, logFilenameP
refix=es]] from [C:\Users\kk_la\Downloads\sonarqube-8.8.0.42792\sonarqube-8.8.0.42792\elasticsearch]: C:\Program Files\J
ava\jdk-11.0.10\bin\java -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnl
y -Djava.io.tmpdir=C:\Users\kk_la\Downloads\sonarqube-8.8.0.42792\sonarqube-8.8.0.42792\temp -XX:ErrorFile=../logs/es_hs
_err_pid%p.log -Des.networkaddress.cache.ttl=60 -Des.networkaddress.cache.negative.ttl=10 -XX:+AlwaysPreTouch -Xss1m -Dj
ava.awt.headless=true -Dfile.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -Dio.netty.noUnsafe=true -Di
o.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPerThread=0 -Dio.netty.allocator.numDirectArenas=0 -Dlo
g4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=COMPAT -Xmx512m -Xms512m -XX:MaxDirectMe
morySize=256m -XX:+HeapDumpOnOutOfMemoryError -Delasticsearch -Des.path.home=C:\Users\kk_la\Downloads\sonarqube-8.8.0.42
792\sonarqube-8.8.0.42792\elasticsearch -Des.path.conf=C:\Users\kk_la\Downloads\sonarqube-8.8.0.42792\sonarqube-8.8.0.42
792\temp\conf\es -cp lib/* org.elasticsearch.bootstrap.Elasticsearch
jvm 1    | Java HotSpot(TM) 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in version 9.0 and will l
ikely be removed in a future release.
jvm 1    | 2021.04.26 21:50:33 INFO  app[][o.s.a.SchedulerImpl] Waiting for Elasticsearch to be up and running
jvm 1    | 2021.04.26 21:51:46 INFO  app[][o.s.a.SchedulerImpl] Process[es] is up
jvm 1    | 2021.04.26 21:51:46 INFO  app[][o.s.a.ProcessLauncherImpl] Launch process[[key='web', ipcIndex=2, logFilename
Prefix=web]] from [C:\Users\kk_la\Downloads\sonarqube-8.8.0.42792\sonarqube-8.8.0.42792]: C:\Program Files\Java\jdk-11.0
```

Figure 8: The output of the SonarQube terminal that started up the tool

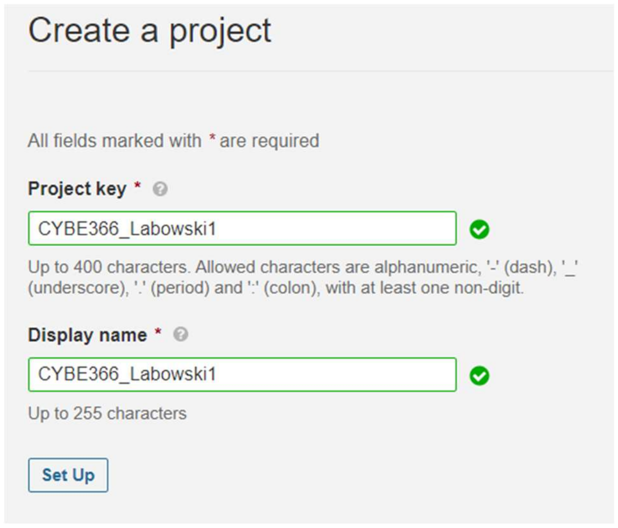
According to the tutorial on the SonarQube website, a new file called sonar-project.properties had to be placed in the top-level directory of the project to test. For the Windows system this was being tested on, this directory was C:\Users\kk_la\College Stuff\FALL 2020\CS 230\WVU_CS230_2020.08_Group06-master. The contents of this PROPERTIES folder, which came from the SonarQube website, can be seen in **Figure 9**.



```
# must be unique in a given SonarQube instance
sonar.projectKey=my:project

# --- optional properties ---

# defaults to project key
#sonar.projectName=My project
# defaults to 'not provided'
#sonar.projectVersion=1.0

# Path is relative to the sonar-project.properties file. Defaults to .
#sonar.sources=.

# Encoding of the source code. Default is default system encoding
#sonar.sourceEncoding=UTF-8
```

Figure 9: The contents of the PROPERTIES file located in the top-level directory of the software's file structure

Because the SonarQube terminal had finished setting up the tool, the interface for the tool could be navigated by visiting localhost:9000 in a web browser. To create a new project in the SonarQube interface, the Add Project option was selected. A prompt was then displayed with a field "Project key" to fill in, so that the project could be distinguished from others. This prompt can be seen in **Figure 10**.


Create a project

All fields marked with * are required

Project key *
CYBE366_Labowski1

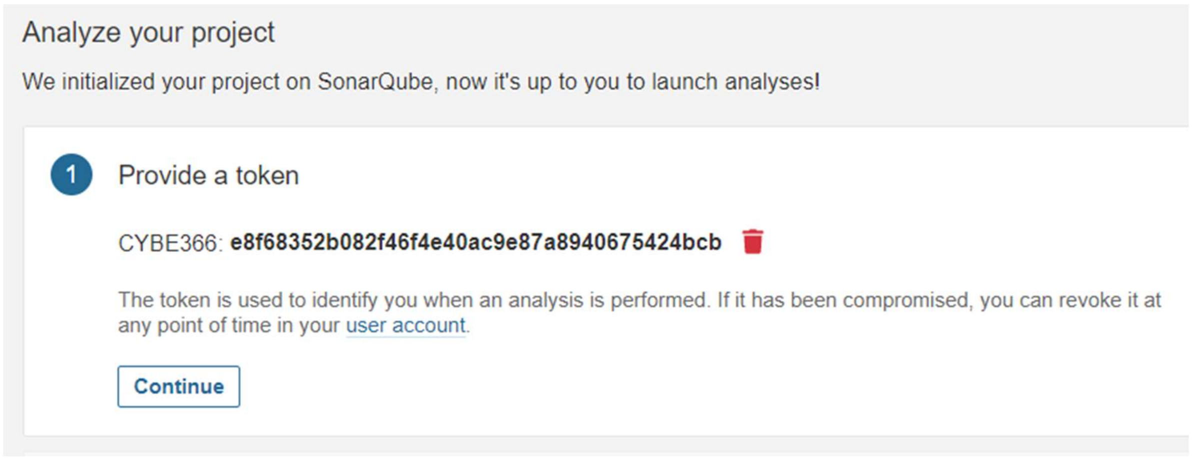
Up to 400 characters. Allowed characters are alphanumeric, '-' (dash), '_' (underscore), '.' (period) and ':' (colon), with at least one non-digit.

Display name *
CYBE366_Labowski1

Up to 255 characters

Figure 10: The prompt to create a project key and display name when creating a new project

Next, a security token was created to ensure the security of the test results. The value "CYBE366" was entered, and a token was generated from this string. See **Figure 11** for the security token that was generated.


Analyze your project

We initialized your project on SonarQube, now it's up to you to launch analyses!

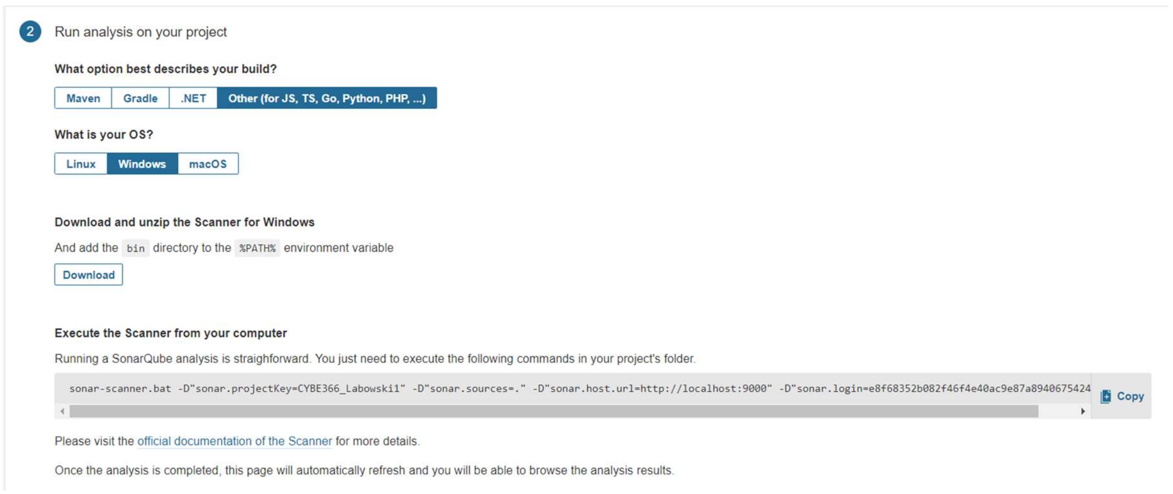
1 Provide a token

CYBE366: **e8f68352b082f46f4e40ac9e87a8940675424bcb**

The token is used to identify you when an analysis is performed. If it has been compromised, you can revoke it at any point of time in your user account.

Figure 11: The security token generation page in the project setup for SonarQube

Next in the project setup process, a few more questions were asked. The build was described as an “Other” build, as this choice encompasses JavaScript, PHP, and other languages, which were used to program the web application. The operating system was selected to be Windows, as the tool was being run on a Windows machine. Then another zip file was provided, this time including the actual scanning tool for the project. At the bottom of the window, a command was provided to run in a Command Prompt window at the top-level directory of the project to test. The full window can be seen in **Figure 12**.



2 Run analysis on your project

What option best describes your build?

Maven | Gradle | .NET | [✓ Other (for JS, TS, Go, Python, PHP, ...)]

What is your OS?

Linux | [✓ Windows] | macOS

Download and unzip the Scanner for Windows

And add the `bin` directory to the `%PATH%` environment variable

Download

Execute the Scanner from your computer

Running a SonarQube analysis is straighforward. You just need to execute the following commands in your project's folder.

```
sonar-scanner.bat -D"sonar.projectKey=CYBE366_Labowski1" -D"sonar.sources=." -D"sonar.host.url=http://localhost:9000" -D"sonar.login=e8f68352b082f46f4e40ac9e87a8940675424
```

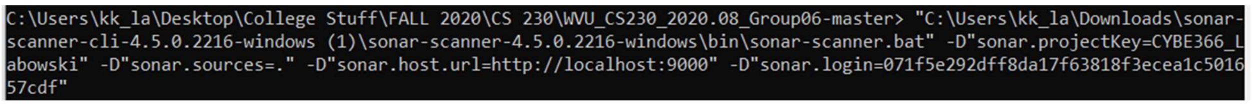
Copy

Please visit the official documentation of the Scanner for more details.

Once the analysis is completed, this page will automatically refresh and you will be able to browse the analysis results.

Figure 12: A window with options to select, and the command used to run the scanning tool

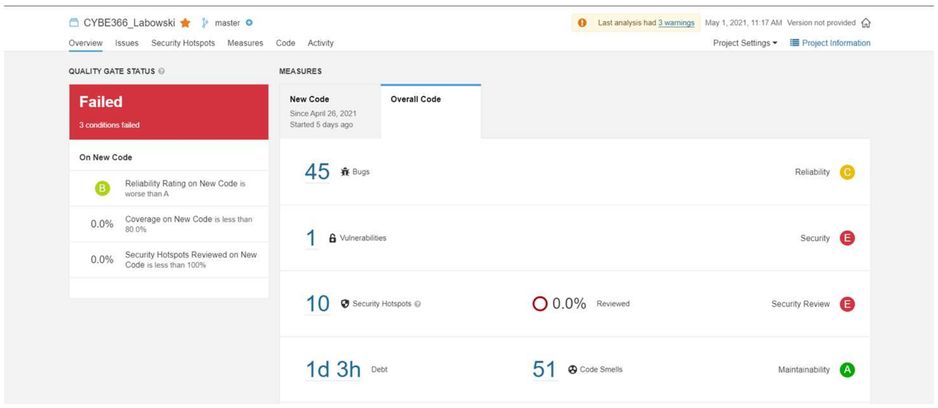
Following the instructions, a new Command Prompt window was opened, and the top-level directory of the project was navigated to using the cd command. The command was then run in the terminal as seen in **Figure 13**. It tested all relevant files within the directory, which took about a minute to complete.



```
C:\Users\kk_la\Desktop\College Stuff\FALL 2020\CS 230\WVU_CS230_2020.08_Group06-master> "C:\Users\kk_la\Downloads\sonar-
scanner-cli-4.5.0.2216-windows (1)\sonar-scanner-4.5.0.2216-windows\bin\sonar-scanner.bat" -D"sonar.projectKey=CYBE366_L
abowski" -D"sonar.sources=." -D"sonar.host.url=http://localhost:9000" -D"sonar.login=071f5e292dff8da17f63818f3ecea1c5016
57cdf"
```

Figure 13: Running the SonarQube tool from a Command Prompt window

Returning to the interface for the SonarQube tool after it was run showed output for the code. Notably, the tool found 45 bugs, 1 security vulnerability, 10 security hotspots, and 51 code smells. According to SonarQube, the web application failed, particularly because of its security scores being E. The overview of the output can be seen in **Figure 14**.

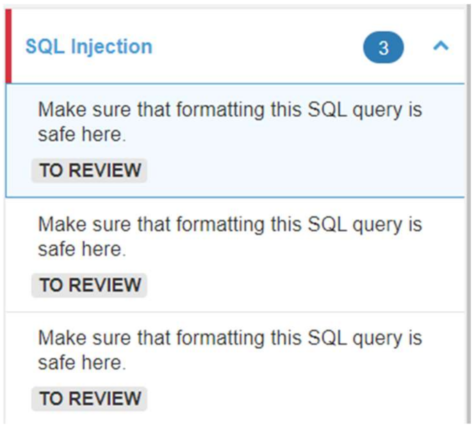

CYBE366_Labowski master
Last analysis had 3 warnings May 1, 2021, 11:17 AM Version not provided
Overview Issues Security Hotspots Measures Code Activity
Project Settings Project Information

QUALITY GATE STATUS
Failed
3 conditions failed
On New Code
B Reliability Rating on New Code is worse than A
0.0% Coverage on New Code is less than 80.0%
0.0% Security Hotspots Reviewed on New Code is less than 100%

MEASURES
New Code Since April 26, 2021 Started 5 days ago
Overall Code
45 Bugs Reliability C
1 Vulnerabilities Security E
10 Security Hotspots 0.0% Reviewed Security Review E
1d 3h Debt 51 Code Smells Maintainability A

Figure 14: The overview page of the test that was completed with SonarQube

After reviewing each section, the security hotspot section was found to be where the SQL injection vulnerabilities were found. The software found a total of 3 SQL injection vulnerabilities, 1 in display-reviews.php and 2 in get-ratings.php. **Figure 15** shows that there were 3 SQL injection vulnerabilities found.


SQL Injection 3

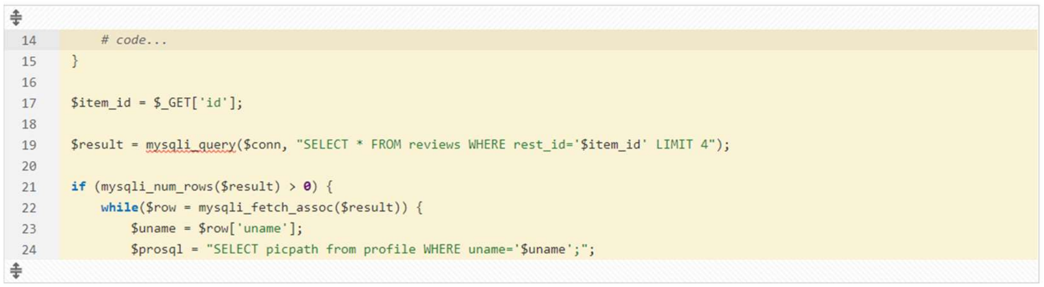
Make sure that formatting this SQL query is safe here.
TO REVIEW

Make sure that formatting this SQL query is safe here.
TO REVIEW

Make sure that formatting this SQL query is safe here.
TO REVIEW

Figure 15: The SQL injections found in the Security Hotspots section

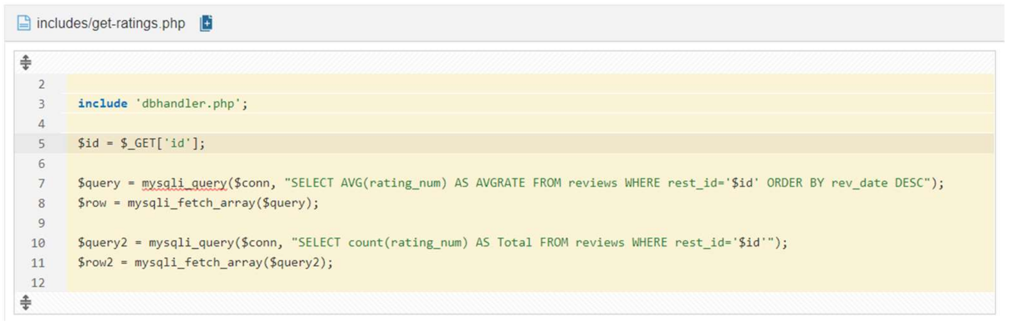
The section of code within display-reviews.php that was found to have the SQL injection vulnerability can be seen in **Figure 16**. This section of code was used to read in 4 of the reviews for the restaurant with the current ID. Each restaurant stored in the database was given a unique ID so that its reviews could be stored and displayed only for that restaurant. Later in the file there is PHP code to write the reviews to the screen on the web application.



```
    # code...
}

$item_id = $_GET['id'];

$result = mysqli_query($conn, "SELECT * FROM reviews WHERE rest_id='$item_id' LIMIT 4");

if (mysqli_num_rows($result) > 0) {
    while($row = mysqli_fetch_assoc($result)) {
        $uname = $row['uname'];
        $prosql = "SELECT picpath from profile WHERE uname='$uname';";
```

Figure 16: The vulnerable code found within display-reviews.php

The next section of code displayed in the Security Hotspots section was in the get-ratings.php file. This section of code was used to find the average rating of the restaurant with a given ID number as well as the total number of ratings for that restaurant. These values would be used later in the same PHP file to determine the rating to display on the restaurant's review page, and it would display it using stars to show the results. The section of vulnerable code can be seen in **Figure 17.**



includes/get-ratings.php

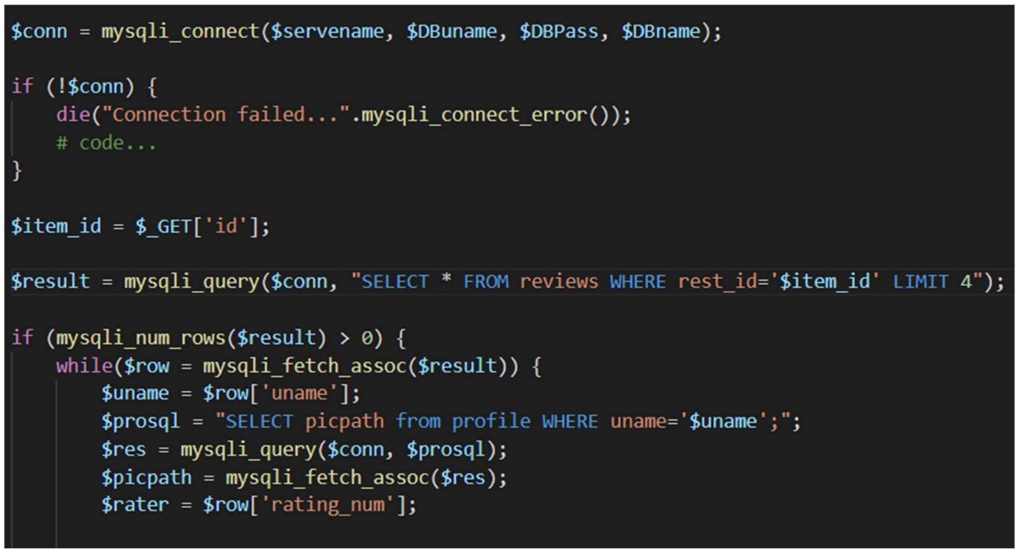
```

include 'dbhandler.php';

$id = $_GET['id'];

$query = mysqli_query($conn, "SELECT AVG(rating_num) AS AVGRATE FROM reviews WHERE rest_id='$id' ORDER BY rev_date DESC");
$row = mysqli_fetch_array($query);

$query2 = mysqli_query($conn, "SELECT count(rating_num) AS Total FROM reviews WHERE rest_id='$id'");
$row2 = mysqli_fetch_array($query2);

```

Figure 17: The vulnerable code found within get-ratings.php

As mentioned previously, the vulnerabilities in each of the code snippets above are vulnerabilities because the restaurant ID is being directly inserted into the SQL database without being sanitized. This means that if a user were able to input a value here, they would be able to insert additional SQL code to gain access to information within the database that they should not have access to. The way to resolve these vulnerabilities is to use prepared statements, which will be shown in the next task.

Task 4: Patching

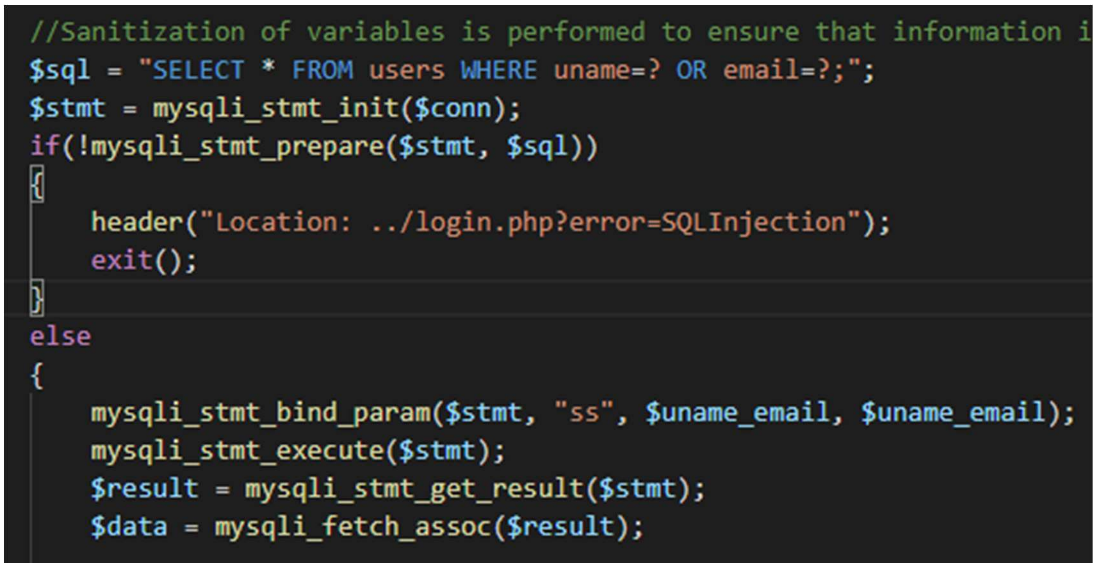
To get a closer look at the code, and to eventually edit it, the code was opened using Visual Studio Code. The first file to focus on was the display-reviews.php file. **Figure 18** shows the code in this file before making any changes to it.



```
$conn = mysqli_connect($servename, $DBuname, $DBPass, $DBname);

if (!$conn) {
    die("Connection failed...".mysqli_connect_error());
    # code...
}

$item_id = $_GET['id'];

$result = mysqli_query($conn, "SELECT * FROM reviews WHERE rest_id='$item_id' LIMIT 4");

if (mysqli_num_rows($result) > 0) {
    while($row = mysqli_fetch_assoc($result)) {
        $uname = $row['uname'];
        $prosql = "SELECT picpath from profile WHERE uname='$uname';";
        $res = mysqli_query($conn, $prosql);
        $picpath = mysqli_fetch_assoc($res);
        $rater = $row['rating_num'];
```

Figure 18: The original code within display-review.php

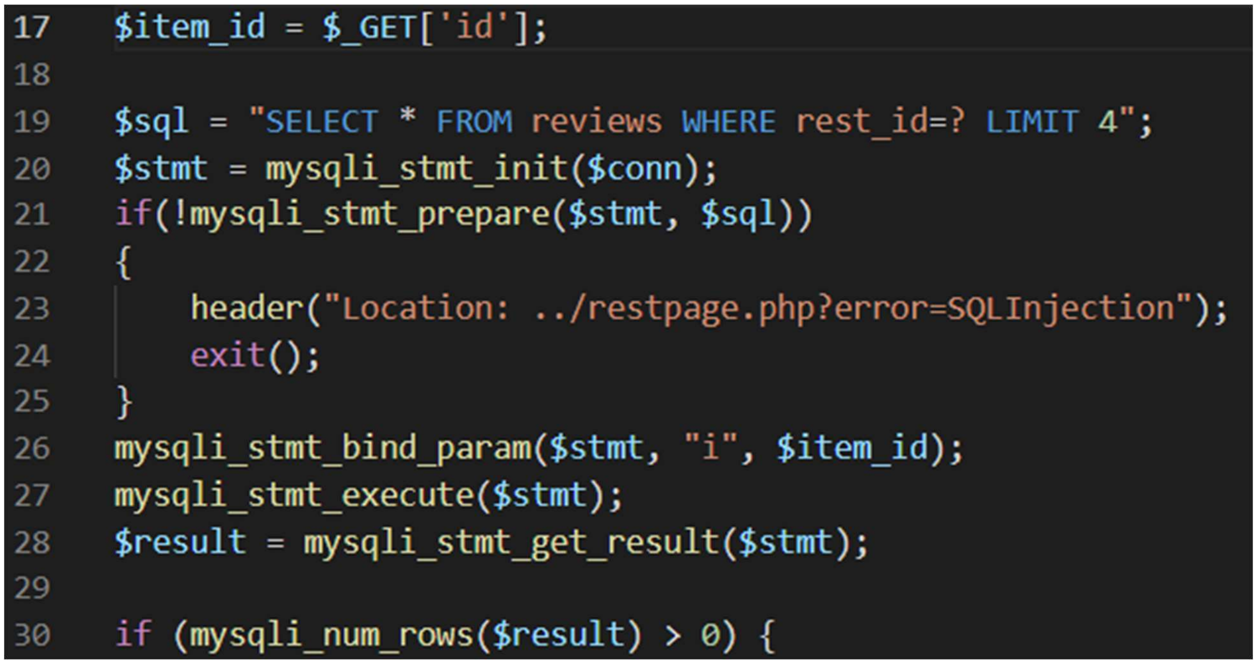
It is known that MySQL prepared statements are needed to resolve the SQL injection vulnerability, and luckily, another section of code within the software already had prepared statements implemented. The file containing the already safe code was login-helper.php, and it was used to determine whether input login information exists within the database. The code for this section can be seen in **Figure 19**.



```
//Sanitization of variables is performed to ensure that information i
$sql = "SELECT * FROM users WHERE uname=? OR email=?;";
$stmt = mysqli_stmt_init($conn);
if(!mysqli_stmt_prepare($stmt, $sql))
{
    header("Location: ../login.php?error=SQLInjection");
    exit();
}
else
{
    mysqli_stmt_bind_param($stmt, "ss", $uname_email, $uname_email);
    mysqli_stmt_execute($stmt);
    $result = mysqli_stmt_get_result($stmt);
    $data = mysqli_fetch_assoc($result);
```

Figure 19: The code for the login-helper.php file, including prepared statements

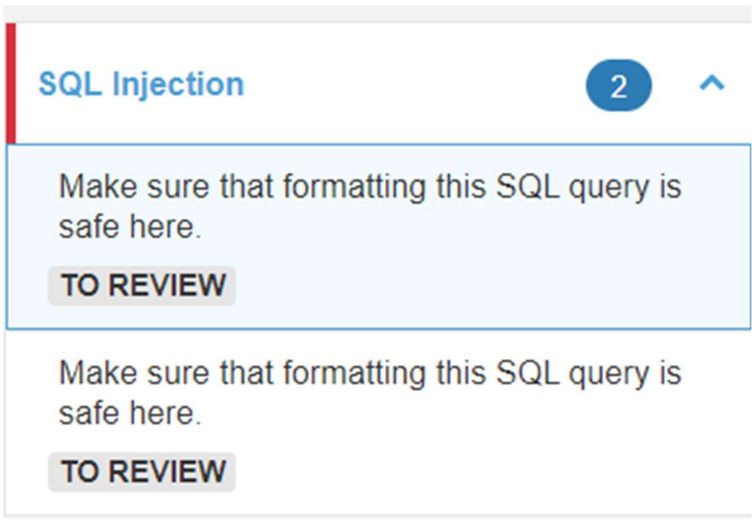
To make the SQL queries from the display-review.php file safe, the prepared statements within the login-helper.php file were referenced. Rather than just using the mysqli_query function as in the original display-reviews.php file, the new version of the file included several lines that accomplish the same while ensuring that input is sanitized. First, the $item_id variable which was directly inserted into the SQL query at first was replaced with a question mark (?). Then, a connection to the database is established, and the mysqli_stmt_prepare function is run, which essentially checks to see if the input was normal user input or an attempted SQL injection. In the case that a SQL injection was detected, an error message will be displayed, and nothing will happen. Otherwise, the function mysqli_stmt_bind_param is used to bind the parameter $item_id to the question mark in the SQL statement safely. A string with the letter 'i' here is used to indicate that there is a single, integer type value to bind. After the parameter has been bind successfully, the function mysqli_stmt_execute function is used to execute the SQL code with the parameter that was input. Finally, the $result variable was set to an evaluation of the function mysqli_stmt_get_result, because this variable is used later in the file. The full changes with prepared statements can be seen in **Figure 20**.



```
$item_id = $_GET['id'];

$sql = "SELECT * FROM reviews WHERE rest_id=? LIMIT 4";
$stmt = mysqli_stmt_init($conn);
if(!mysqli_stmt_prepare($stmt, $sql))
{
    header("Location: ../restpage.php?error=SQLInjection");
    exit();
}
mysqli_stmt_bind_param($stmt, "i", $item_id);
mysqli_stmt_execute($stmt);
$result = mysqli_stmt_get_result($stmt);

if (mysqli_num_rows($result) > 0) {
```

Figure 20: The updated code for display-reviews.php, with the SQL injection vulnerability eliminated

While it clearly took much more code to accomplish what originally took only one line of code, using prepared statements is worth it, and they are essential for ensuring that SQL database manipulations within code are performed without introducing any dangerous vulnerabilities. After this file was updated with the prepared statement, the SonarQube tester was run once again. As can be seen in **Figure 21**, there were only 2 SQL injection vulnerabilities remaining, and the display-reviews.php vulnerability was resolved successfully.


SQL Injection 2

Make sure that formatting this SQL query is safe here.

TO REVIEW

Make sure that formatting this SQL query is safe here.

TO REVIEW

Figure 21: The result of the second test of the program, with 1 of the 3 vulnerabilities resolved

With the first of the two files updated to include more secure coding practices, the get-ratings.php file was then opened and examined. The portions of code that were marked as vulnerabilities by SonarQube can be seen in **Figure 22** below.

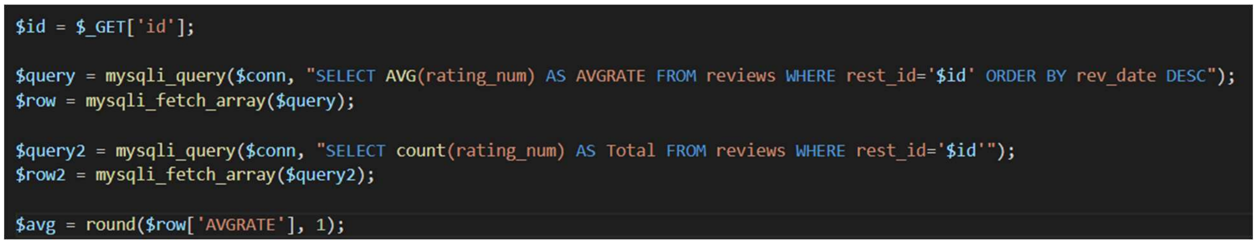
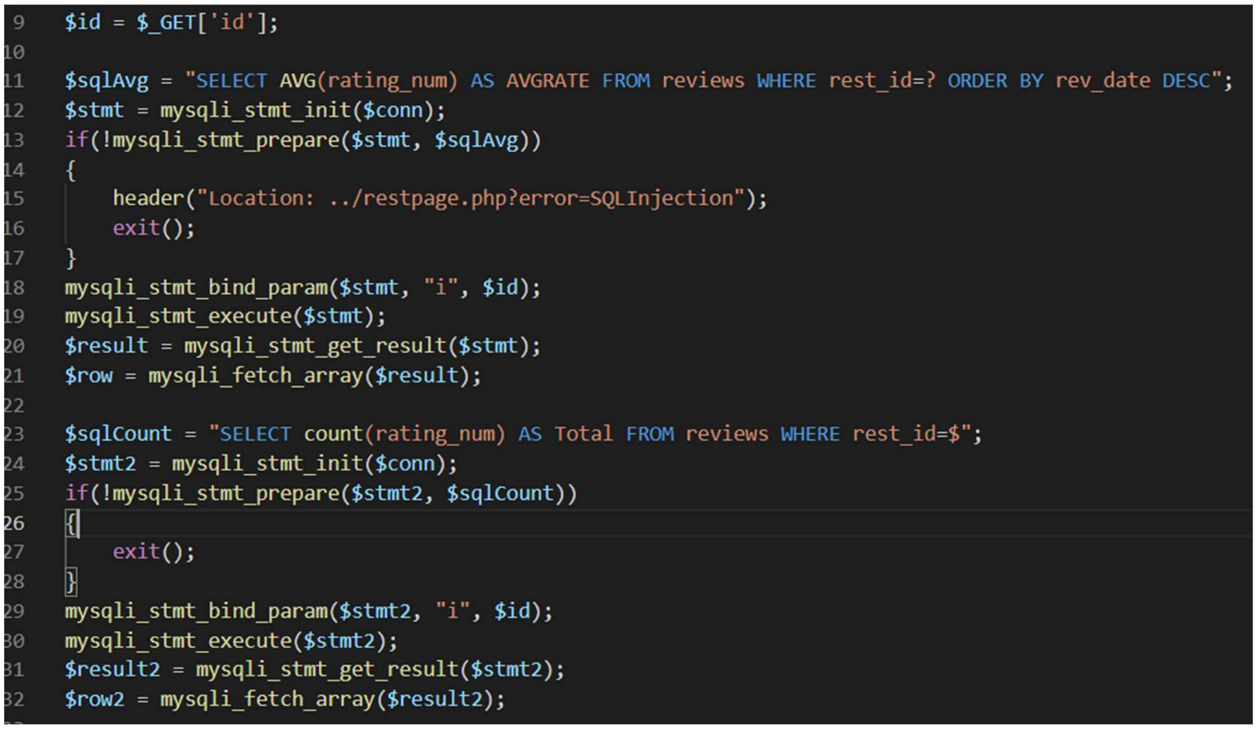

```
$id = $_GET['id'];

$query = mysqli_query($conn, "SELECT AVG(rating_num) AS AVGRATE FROM reviews WHERE rest_id='$id' ORDER BY rev_date DESC");
$row = mysqli_fetch_array($query);

$query2 = mysqli_query($conn, "SELECT count(rating_num) AS Total FROM reviews WHERE rest_id='$id'");
$row2 = mysqli_fetch_array($query2);

$avg = round($row['AVGRATE'], 1);
```

Figure 22: The vulnerable section of code within the get-ratings.php file

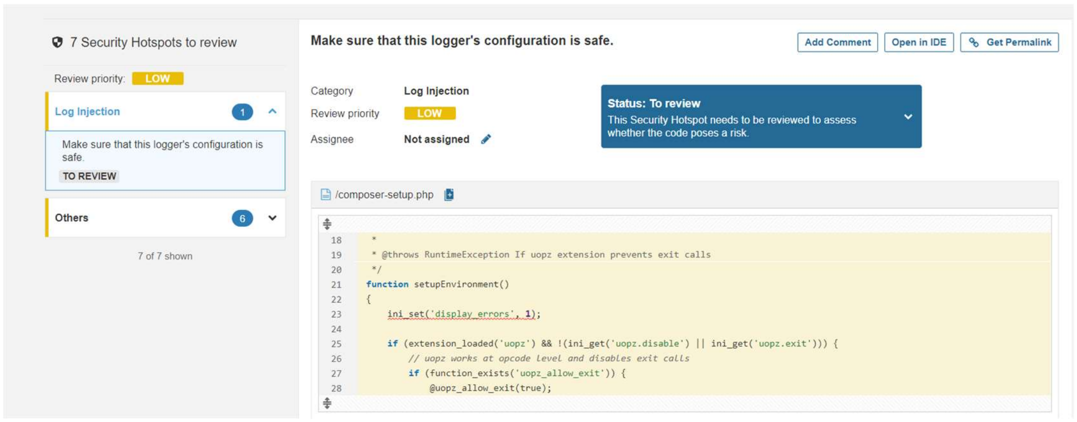
Just as with the previous section, the login-helper.php file and its prepared statements were used as a template to base these prepared statements on. The same functions that were used in the previous file and its vulnerability were used again in this file. The vulnerability does occur twice here, so the prepared statements are repeated once for each of the SQL statements that were called. The only other different between the prepared statement handling in this file and the previous file was the addition of the $row and $row2 variables, which are used later in the file for outputting the ratings to the review page. The variables are set to the data for each of the two respective statements using the mysqli_fetch_array function on the result of the SQL injection to get the data as an array. This is important because the variables are used as arrays later in the file, and they were this way in the original file before the prepared statements were added. The new get-ratings.php file with prepared statements added can be seen in **Figure 23**.



```
$id = $_GET['id'];

$sqlAvg = "SELECT AVG(rating_num) AS AVGRATE FROM reviews WHERE rest_id=? ORDER BY rev_date DESC";
$stmt = mysqli_stmt_init($conn);
if(!mysqli_stmt_prepare($stmt, $sqlAvg))
{
    header("Location: ../restpage.php?error=SQLInjection");
    exit();
}
mysqli_stmt_bind_param($stmt, "i", $id);
mysqli_stmt_execute($stmt);
$result = mysqli_stmt_get_result($stmt);
$row = mysqli_fetch_array($result);

$sqlCount = "SELECT count(rating_num) AS Total FROM reviews WHERE rest_id=$";
$stmt2 = mysqli_stmt_init($conn);
if(!mysqli_stmt_prepare($stmt2, $sqlCount))
{
    exit();
}
mysqli_stmt_bind_param($stmt2, "i", $id);
mysqli_stmt_execute($stmt2);
$result2 = mysqli_stmt_get_result($stmt2);
$row2 = mysqli_fetch_array($result2);
```

Figure 23: The updated get-ratings.php file with the new prepared statements added to resolve the vulnerabilities.

To ensure that the code segments above were successfully rid of the SQL injection vulnerabilities, SonarQube was run once more on the software. As can be seen in **Figure 24**, the SQL injection vulnerabilities no longer appear in the Security Hotspots section. Note that the other security hotspots and vulnerabilities found by SonarQube were either not covered in this class or were intentional features of the program (such as not having a password on the database when running the software locally).


7 Security Hotspots to review

Review priority: LOW

Log Injection 1

Make sure that this logger's configuration is safe.

TO REVIEW

Others 6

7 of 7 shown

Make sure that this logger's configuration is safe.

Category: Log Injection

Review priority: LOW

Assignee: Not assigned

Status: To review

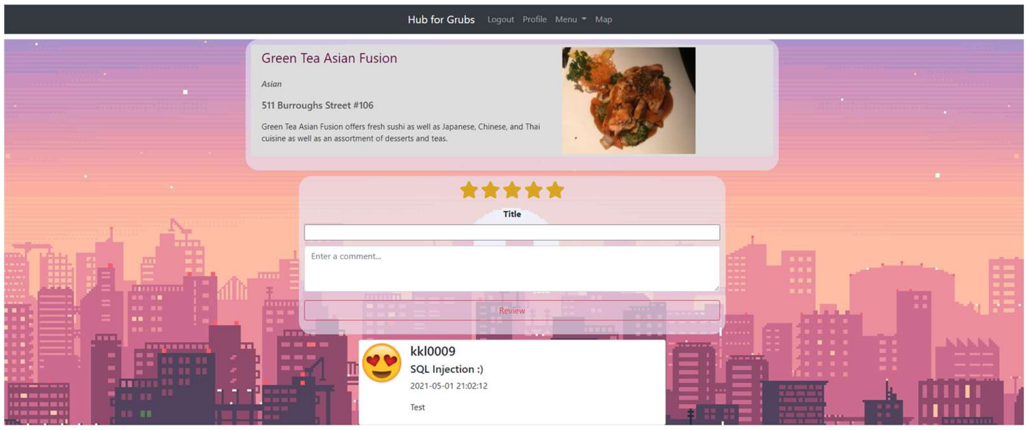
This Security Hotspot needs to be reviewed to assess whether the code poses a risk.

/composer-setup.php

```
     *
     * @throws RuntimeException If uopz extension prevents exit calls
     */
    function setupEnvironment()
    {
        ini_set('display_errors', 1);

        if (extension_loaded('uopz') && !(ini_get('uopz.disable') || ini_get('uopz.exit'))) {
            // uopz works at opcode level and disables exit calls
            if (function_exists('uopz_allow_exit')) {
                @uopz_allow_exit(true);
```

Figure 24: The Security Hotspots section on the final run of SonarQube, now free of SQL injection errors in the column on the left

Additionally, to ensure that the functionality of the software was not hindered in any way, a restaurant review page was navigated to from within the web application's local instance. The page appeared how it was supposed to, but to make sure it would still work as intended, a review was added to see if it broke anything. Luckily, the prepared statements preserved the functionality of the original program, as can be seen in **Figure 25**.


Green Tea Asian Fusion

Asian

511 Burroughs Street #106

Green Tea Asian Fusion offers fresh sushi as well as Japanese, Chinese, and Thai cuisine as well as an assortment of desserts and teas.

kkl0009

SQL Injection :)

2021-05-01 21:02:12

Test

Figure 25: A review page for a restaurant after the prepared statements were added

Conclusion

In conclusion, the SQL injection vulnerabilities found by the SonarQube tool within the software were successfully resolved. The prepared statements successfully prevented a malicious user of the software from using SQL injection attacks to access or manipulate information they should not have access to. Therefore, the software is now more secure than it was before, although that does not mean that it is free of vulnerabilities. It is possible that SonarQube was unable to find certain vulnerabilities and that there still exist some within the code. That being said, at least the SQL injection vulnerabilities were found and fixed, as if there was important data saved in the database for whatever reason, it could have been compromised.

SAST is a very important part of the software development process. Security is unfortunately too often overlooked by software developers, because programmers often have incentive to complete programs more quickly and adding security measures takes precious time. The side effects of these coding practices which overlook security are that software is often produced that is filled with vulnerabilities. When a software with overlooked vulnerabilities becomes widely used, the information of all users of that software could be in danger. Therefore, I feel a larger emphasis should be placed on security testing in general, which includes SAST. SAST doesn't usually take too long, as it often just requires the programmer to run an automated testing software, so there is not much reason not to test your code using SAST tools before publishing software. Luckily, these days many companies are placing a greater emphasis on security in the software development process, and hopefully more developers recognize the importance of SAST.

I thought this lab was unique, particularly given how open-ended it was. I liked how there were many different directions you could go with it, and it was really a different experience for everyone in the class. It did seem a bit too open-ended at times, particularly the part where we had to find a software with vulnerabilities. I do not blame you for that, I just did not expect it to be so difficult to find software with vulnerabilities we have discussed in class that have not already been patched. That is why I ended up performing tests on my own code from CS 230 last year. Performing the tests on code I already know well did make the lab a bit easier to understand, but it was also interesting to see how I was improving on code I made last year. I also noticed that my group used prepared statements for much of the code, but for whatever reason did not choose to use them in the files discussed in the lab. Overall, though, this lab was interesting, and I think it would be a good assignment to have next time you teach this class.