Laboratory Report

Course ID: CPS 5981

Lab 3: Static Code Analysis

Student: Alexander Fisher

Instructor: Dr. Jing-Chiou Liou

Date: 4/20/2023

Description

Static code analysis is one method used in software security testing to reveal any inherent issues of quality or security in written code that aren't necessarily noticeable at first glance by the human eye. Manual code review by a human can leverage the overall expectations of what the code should do while analyzing code, but automated static code analysis tools cannot do that. Automated tools primarily reveal defects at their face-value, and not with respect to the system as a whole. Automated code analysis tools such as Fortify and Snyk can test code for common vulnerabilities such as SQL injection, and will be used to demonstrate the process and results of static code analysis throughout this lab.

A Lenovo Yoga C930 laptop was used for this lab from my home in Manville, Somerset County, New Jersey. The laptop has the Windows 11 Home Version 22H2 (build 22621.1555) as the operating system and is connected to the internet using a 2.4 GHz band Wi-Fi 4 (802.11n) protocol connection through its Intel(R) Wireless-AC 9260 160MHz network adapter to a NetGear mesh router, which is connected to a modem that receives internet via DSL connection through Optimum Online (Cablevision Systems).

Procedure and Notes

1. To begin, the code and programs that were to be used for manual code review were prepared.
    a. The Java code files that were supplied to me by Professor Liou via Gmail, which were titled "SCAExample.java" and "CWE90_LDAP_Injection__database_54a.java" were downloaded. For this lab, I also chose a PHP file from my undergraduate capstone project, "createUser.php". I saved these code files to my local machine at the following path: "C:_Alex Files\Kean University\Spring2023\CPS 5981 - Software Assurance\Lab 3".
    b. I then launched Visual Studio Code (VSC).
        i. If VSC is not installed on the computer used for this lab, the installer for it can be downloaded from: https://code.visualstudio.com/. Download the installer using the "Download Stable Build" button on the linked website, and follow instructions in the installer to install it to the computer being used. It was already installed on my laptop, so I skipped this installation process.
    c. In VSC, I opened the folder where I saved the code files by navigating to the toolbar at the top of the window, then navigated to File > Open Folder, and then in

the file manager that popped up, I selected the path mentioned before which the code files were placed inside of. VSC should now show something like **Figure 1** below.

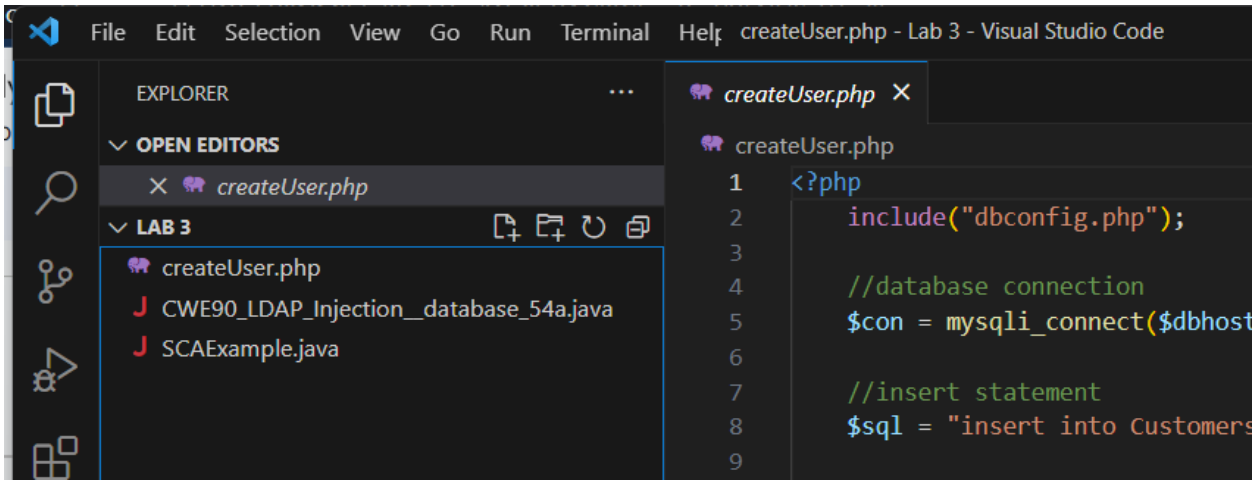

Figure 1: Folder with code files opened in VSC for manual review

2. Next, the **manual code review** of each file was done. This process was done by using VSC to detect any syntax errors, and by myself to detect any possible vulnerabilities caused by weak programming practices or code implementation flaws. It is important to note that manual code review often relies on the reviewer's knowledge and experience with secure coding practices and their fluency in the coding language of the file being reviewed, so different results found during this step are acceptable. With this in mind, I made sure to understand the file being reviewed to the fullest extent before hypothesizing any vulnerabilities.
    a. "SCAExample.java" manual review
        i. Brief explanation of my understanding:
            1. This file is a simple authentication program executed via command line interface (CLI) to open a given log file if the supplied password is correct.
            2. An example input could be: `login_history.txt test123`
                a. This example has argument 1 as the requested file to be opened, "login_history.txt", and argument 2 as the password for authentication, "test123".
        ii. Possible vulnerabilities
            1. One vulnerability that instantly caught my attention was that the password being compared is hardcoded into the java file. If anyone gets access to read this java file instead of executing it, they can see the correct password.

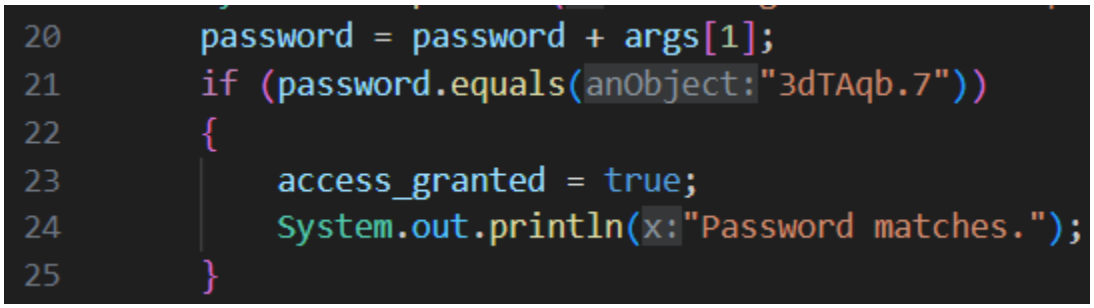

Figure 2: Hardcoded password

2. Depending on the privileges of the java file itself, files that would otherwise be off-limits to the user in terms of read privilege could be accessible to the user through the java file. If the user only has execution privileges for an example file like “login.php”, but the “SCAExample.java” file has read privileges on “login.php”, the user can essentially bypass their lack of read privileges by reading the “login.php” file through the java file’s output, so long as they have the correct password.

b. “CWE90_LDAP_Injection__database_54a.java” manual review
   i. Brief explanation of my understanding:
      1. This file shows two examples of retrieving data:
         a. The “bad()” function returns the name of a user whose id is equal to 1 from an SQL database after executing the SQL statement and throwing any errors if the attempt is unsuccessful.
         b. The “good()” function returns a hardcoded string “foo”, with no other additional operations.
   ii. Possible vulnerabilities:
      1. None found.
         a. The one vulnerability I did think about was SQL injection, since there was an SQL statement executed in the file, but it was a prepared statement that had no bound parameters (i.e., no user input, it would execute the same hardcoded query no matter what). I don’t think SQL injection is possible in this file.

c. “createUser.php” manual review
   i. Brief explanation of my understanding:
      1. This file takes an assortment of POST data sent from the client on a webpage to create a new user for the system.
         a. The inputs are: first name, last name, email, username, password, address, city, state, and zipcode.

b. Those input variables are then bound to an SQL query using a prepared statement that asserts that every input is of type “string”, which helps to resist SQL injection during execution.
c. The SQL query is a statement that, in brief, inserts a record into the Users table of the database, populating that record’s columns using the POST variables.

ii. Possible vulnerabilities:
1. None found.
a. Like the previous file manually reviewed, I also figured that the only possible vulnerability in this file could be SQL injection, but the proper use of prepared SQL statements ensures that inputs are always treated as strings and don’t change the behavior of the query or its results.

3. After manual code review was done, the **automated static analysis tools** were used. For this lab, I used **Fortify On Demand** and **Snyk**. Before usage, these tools must be set up and signed up for.
a. Fortify On Demand
i. Sign up for a free trial at: https://my.microfocus.com/myproducts after creating an account on the website.
ii. Once signed up and access is granted to Fortify On Demand, your My Products page should look like **Figure 4** below. Click on the white “Launch” button to use the application.

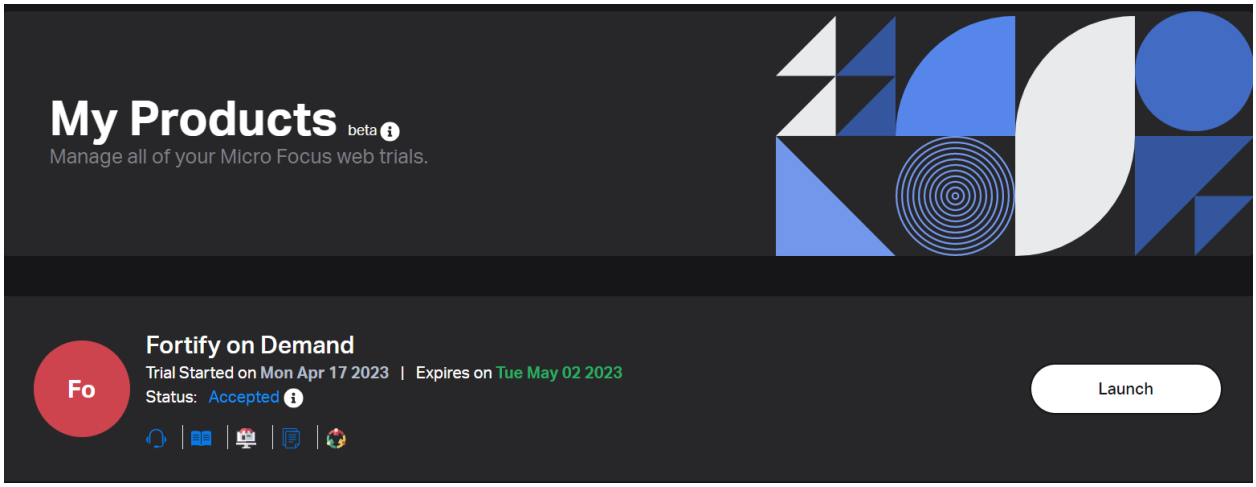

Figure 4: Fortify/MyProducts
(Click white “Launch” button to run Fortify on Demand)

iii. Once launched, the application will look like **Figure 5** below.

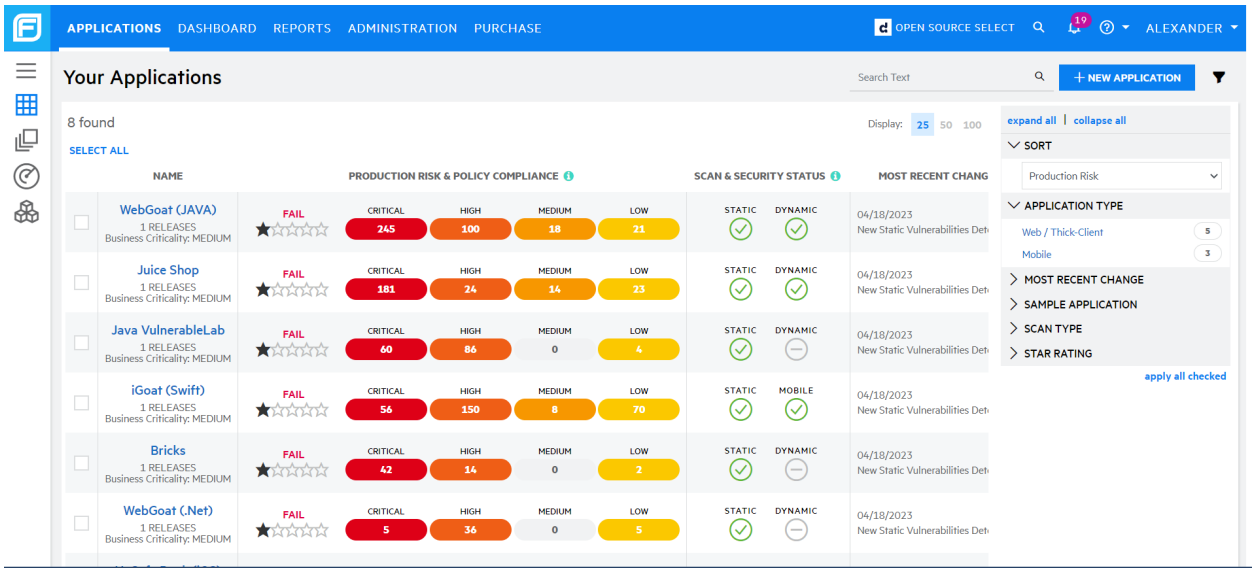

Figure 5: Fortify on Demand web application

iv. Once on the main page of Fortify on Demand, click the "New Application" button at the top right of the page and fill out the form that appears, as shown below in **Figure 6**. The application you create using the form is essentially a container for the code that needs to be scanned, and where all of the results and security ratings of the code will be stored.

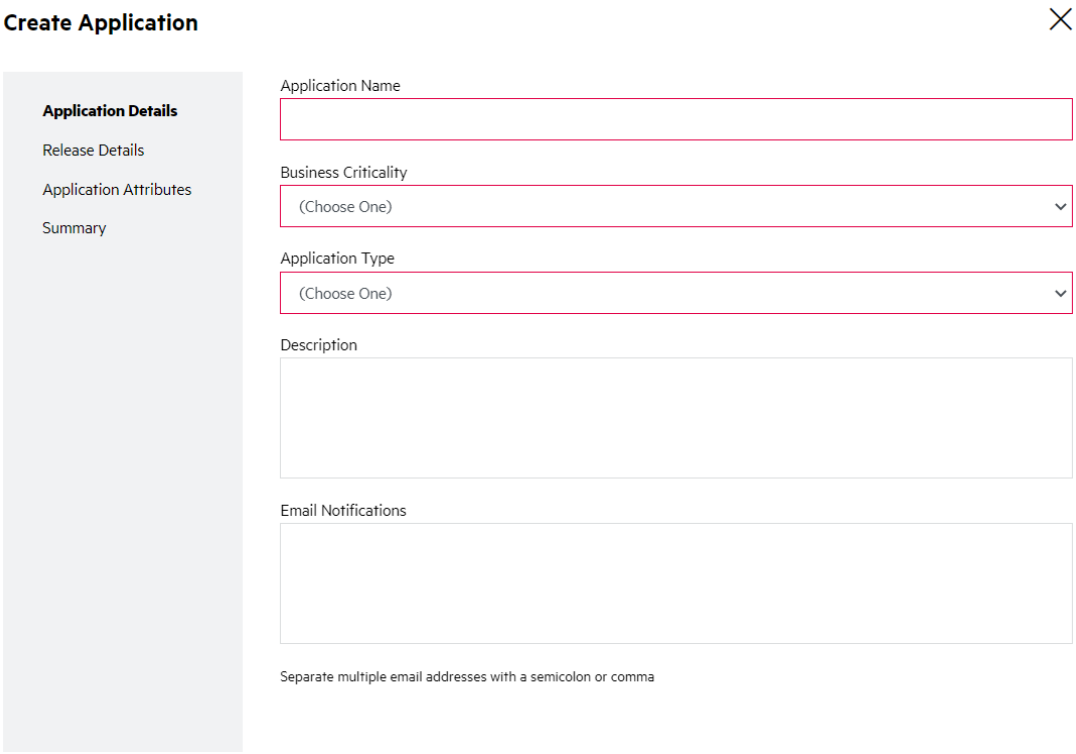

Figure 6: New Application form

v. Once the application is created, you must navigate to the newly created application's page, and set up a static scan. As shown in **Figure 7** below, click on the blue "Start Scan" button at the top right of the page to open a static scan setup form, which is then shown in **Figure 8**.

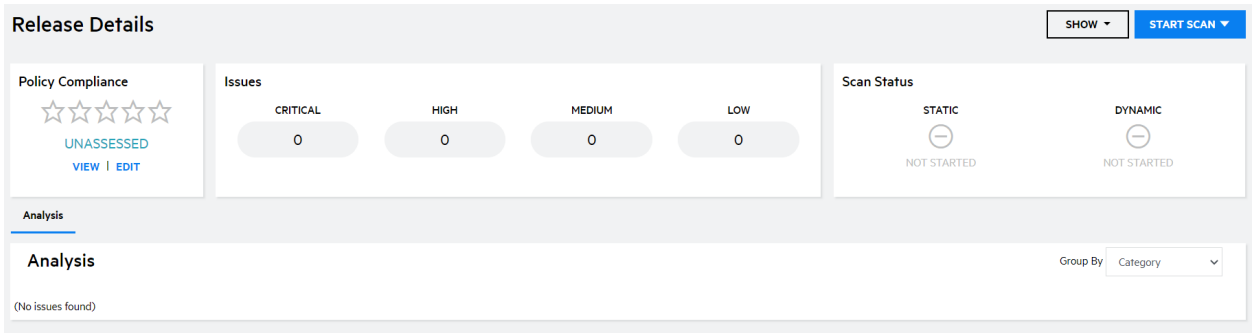

Figure 7: Application Overview page (Click blue "Start Scan" button)

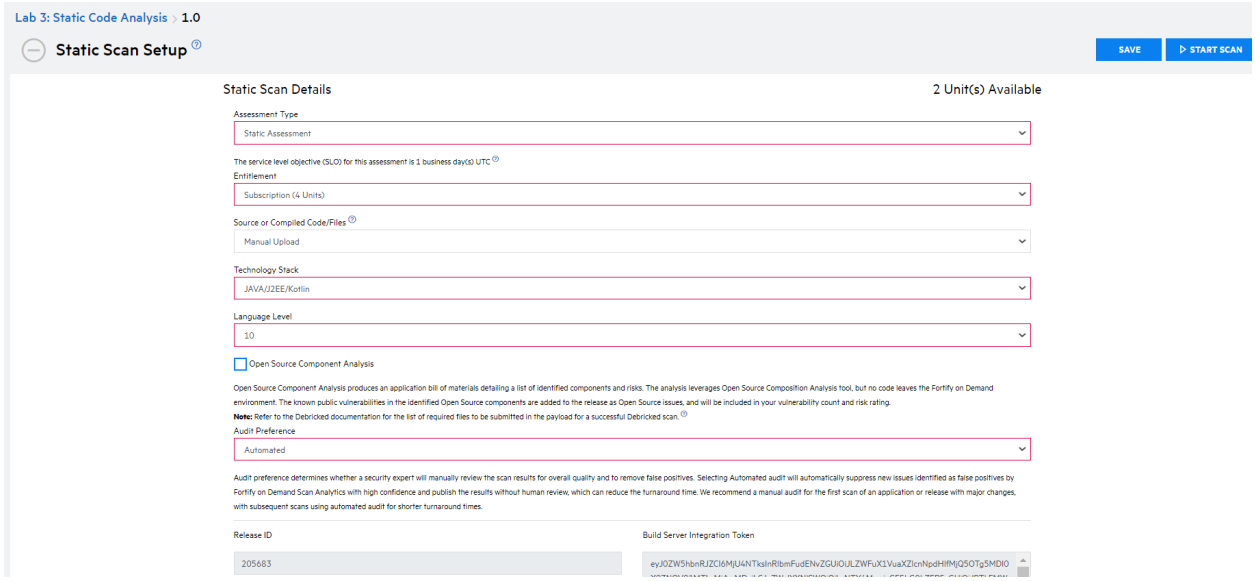

Figure 8: Static analysis scan setup for application

vi. Once the scan setup is completed, the scan can be then queued by uploading a zipped file containing the java code. **NOTE:** The Fortify on Demand automated analysis free trial does not support PHP code, so only the Java code was uploaded for scanning. The form to upload the zipped file is shown in **Figure 9** below.

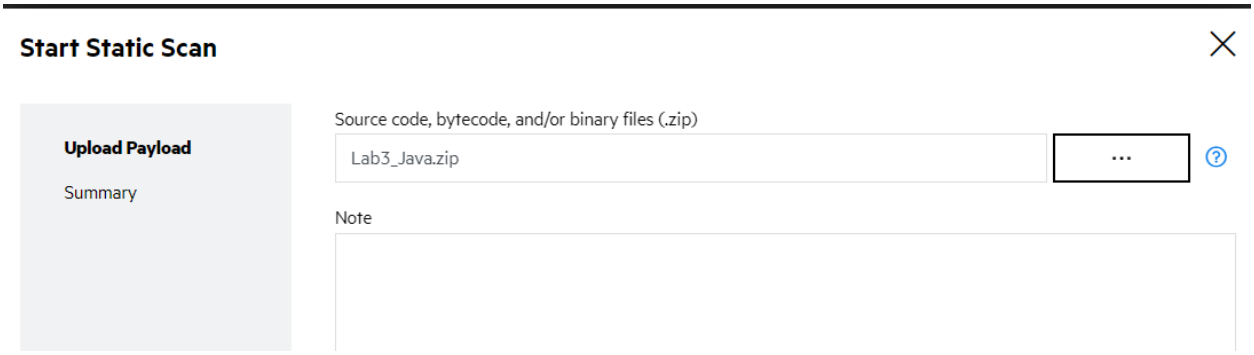

Figure 9: File upload for static analysis scan

vii. After uploading the files for scanning, the static analysis is queued for completion. Fortify mentioned that it usually would take 4 hours to two business days to finish. However, likely due to the small amount of code being analyzed, the static analysis results were returned within a few minutes after queuing the scan. The finished scan overview is shown in the application overview page, shown below in **Figure 10**.

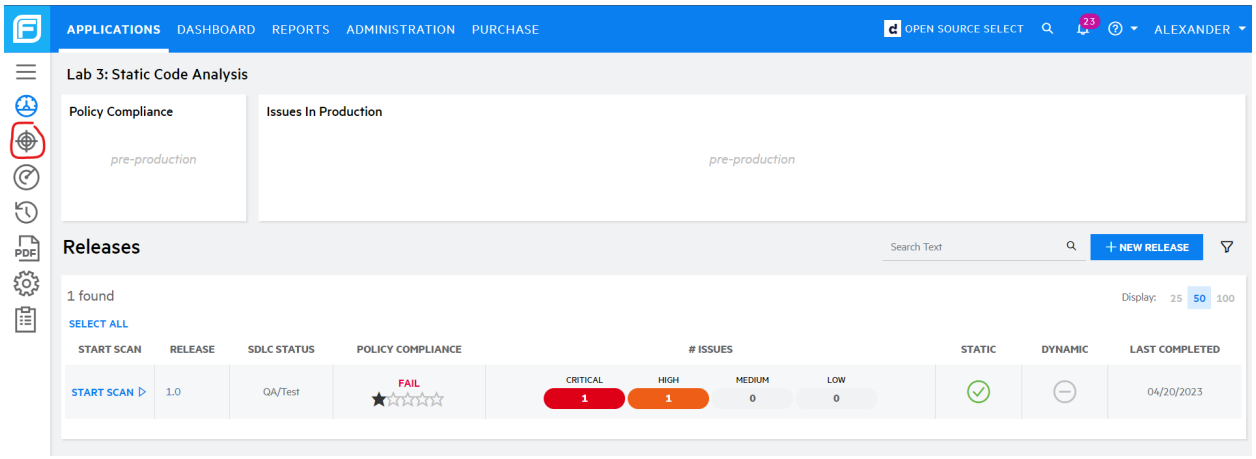

Figure 10: Static Scan results overview

viii. The issues tab was then navigated to by clicking on the third icon down on the left side pane of the page (circled in red in **Figure 10**). The issues page looks like what is shown in **Figure 11** below.

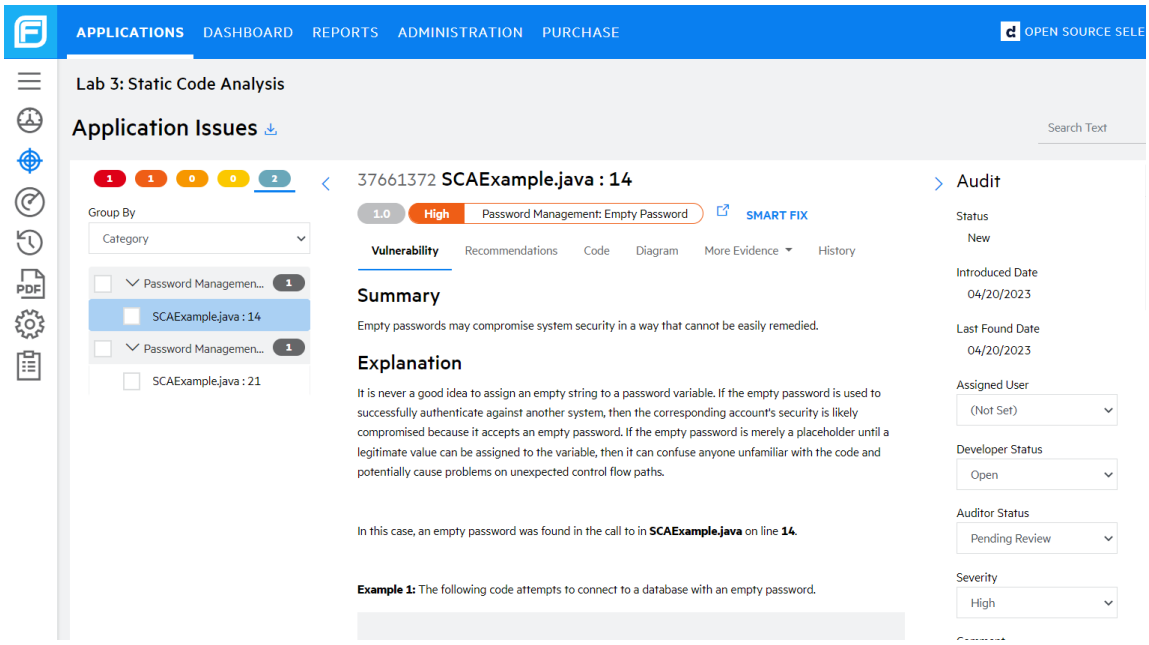

Figure 11: Issues tab of Application on Fortify

ix. Fortify found two issues:
    1. Password variable is initialized as empty string
        a. If an empty string works as the password, a user won't have to enter one to access the file, therefore it is a bad practice to initialize a password as an empty string.
    2. Password is compared with a hardcoded value
        a. Once the software is released, the password cannot be changed unless the software is patched.
        b. Also, if a user has access to read the file, they will see what the password is.

b. Snyk
    i. Sign up for free using an existing Github or Google account at: https://app.snyk.io/login/
    ii. Snyk connects straight to your Github account's repository to scan code for vulnerabilities, so the following Github setup must be done.
        1. If you don't already have a Github account, sign up at: https://github.com/
        2. Once logged into Github, create a new repository from the homepage by clicking the green "New" button, circled in red in **Figure 12** below. Fill out the form that appears. For reference, I named my repository "cps5981_lab3".

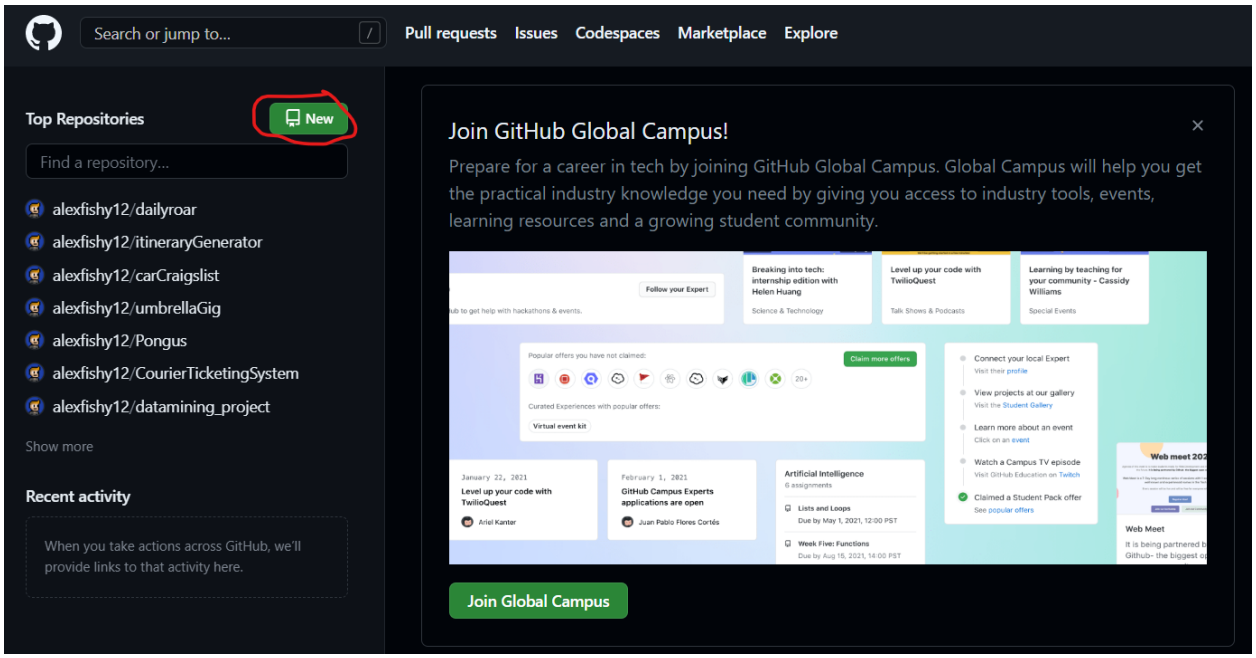

Figure 12: Github homepage (click green "New" button to create repository)

3. Once the repository has been created, navigate to its page by clicking on your profile icon on the top left of the page, then "Your Repositories". Then, click on the repository that you just created. Once on the repository page, click the "Add file" button to manually upload files to the repository, shown in **Figure 13** below.

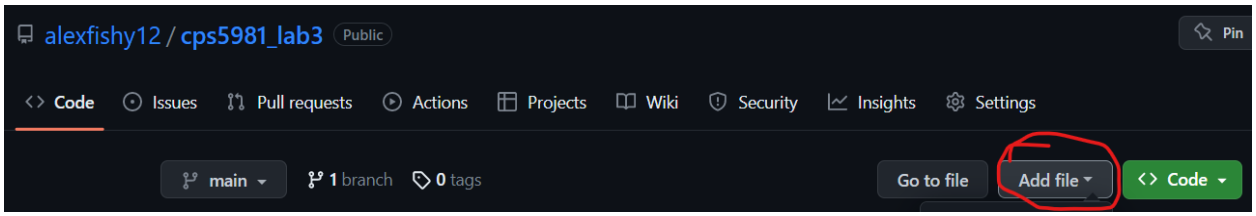

Figure 13: Repository page (Click "Add file" button)

4. Then, select the files in the file manager that Snyk will scan. I chose the three code files that I manually analyzed in the beginning of the lab.
5. Once the files to upload are chosen, select the option to commit directly to main, and then click "Commit changes". Now, your repository contents are ready to be scanned.

iii. Return to the Snyk website: (https://app.snyk.io/login/) and follow the sign-up wizard shown in **Figure 14** below.

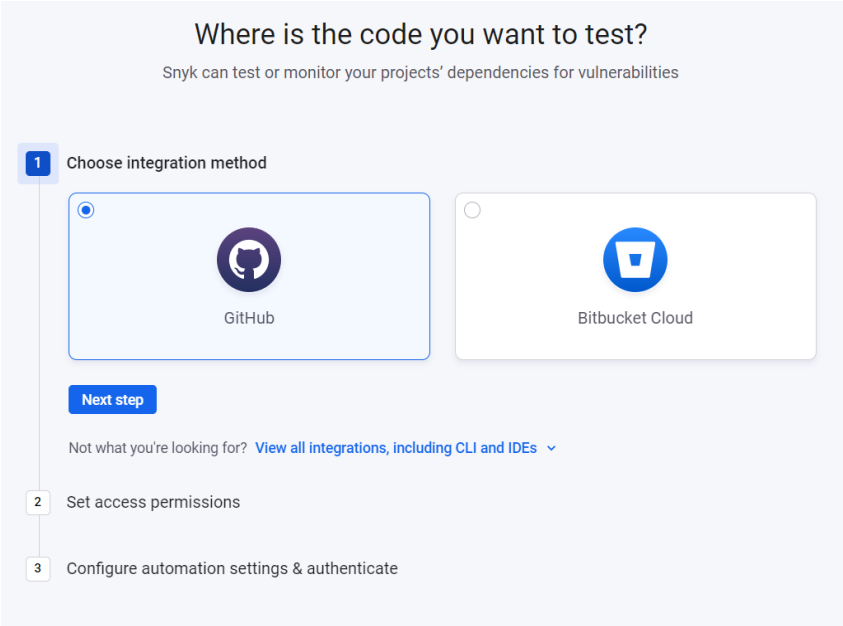

Figure 14: Attaching Github account to Snyk

iv. Once the Snyk account is set up and you are logged in, the dashboard page will look like what is shown below in **Figure 15**. Select the Github repository you uploaded files to on the list shown and then click the blue "Import and scan" button.

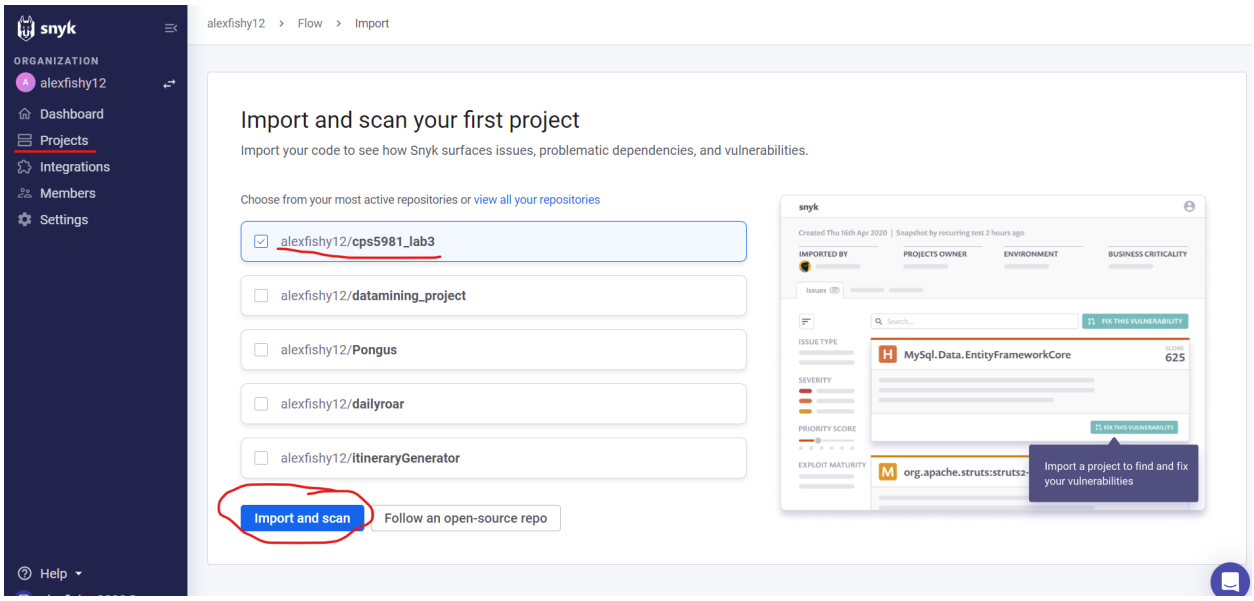

Figure 15: Snyk dashboard (choose repository and click "Import and scan")

v. After the repository has been imported and scanned, navigate to the projects tab of Snyk, the third option down on the left side pane of the dashboard (underlined in **Figure 15**). The projects page will display the

repository that has been imported with a quick overview, much like what Fortify had. This can be seen in **Figure 16** below. Click on the repository you imported to see the scan results.

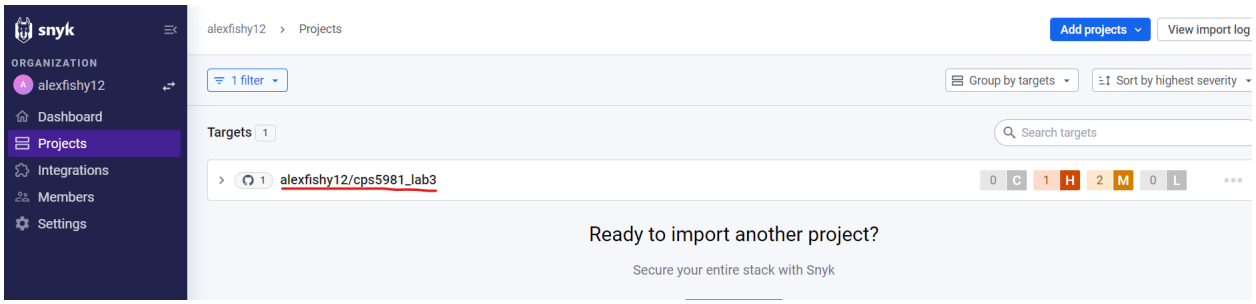

Figure 16: Projects page (Click on imported repository to see scan results)

vi. In **Figure 18** on the next page, an assortment of three screenshots are shown, displaying three separate issues that Snyk uncovered during the code scan. Unlike Fortify, Snyk was able to scan my PHP code, so all three files were scanned this time around. In total, Snyk found three issues (although two of them are duplicates):
    1. Path traversal vulnerability (this appeared twice for the same occurrence, and counted as two separate issues)
        a. In the file "SCAExample.java", the user input gets put directly into a FileOutputStream function, creating a vulnerability that potentially allows an attacker to write to arbitrary files
    2. SQL Injection
        a. In the file "createUser.php", SQL injection is possible where user inputted variable is directly added to SQL query string (shown in **Figure 17** below)

```
$result2 = mysqli_query($con, "select c_id from Customers where user_name='$username'");
if ($result2)
{
    while($row=mysqli_fetch_array($result2))
    {
        $userID = $row['c_id'];
        echo $userID;
```

Figure 17: SQL Injection vulnerability found in "createUser.php"

Figure 18: Resulting issues found during Snyk scan

Results and Reports

Issue Log

Filename	Issue description	Revealed by
SCAExample.java	Path traversal	Snyk
SCAExample.java	Hardcoded password	Manual review, Fortify
SCAExample.java	Password initialized to empty string	Fortify
createUser.php	SQL Injection	Snyk

SCAExample.java: 3 total issues detected
createUser.php: 1 total issue detected
CWE90_LDAP_Injection__database_54a.java: 0 issues detected

During the procedure of this lab, a couple key observations were realized:

- Tools can shed light to issues that are overlooked by manual code review
- It is useful to use more than one static code analyzer tool, since different tools can give different results

This lab enabled a clear comparison to be made between manual and automated static code analysis, and gave insight with regards to how automation tools can be of use. During manual code review, I could not find any issues with my own PHP code. However, Snyk found an SQL injection vulnerability. As it turns out, my manual code review process completely overlooked that issue which now seems obvious. Like I said during the manual review, that PHP code did use properly prepared SQL statements to avoid injection, but not for *every* SQL statement. I missed one, where I directly input a user inputted string into the query.

Fortify came up with two issues, one of which I was able to notice during my manual review, which was the hardcoded password. The other issue found, however, was of the password variable being initialized to an empty string, potentially enabling the user to bypass a password input by just leaving their password input blank and instead using the one that was initialized in the code.

I found it interesting that Snyk and Fortify found completely different issues with no overlap with each other's results. All in all, using multiple tools for static code analysis could actually be worth the time and money as the results of this lab have shown that doing so will cover more of the outstanding issues in analyzed code than if only one tool was used.

Reference and Acknowledgement

GitHub repository with all code used for this lab: https://github.com/alexfishy12/cps5981_lab3

Snyk website: https://app.snyk.io/

MicroFocus website (Fortify): https://www.microfocus.com/en-us/home