COMSE6156 Project Report: projScan-I

Tianzhi Huang

May 2022

1 Synopsis

This project, called projScan-I (Project Scan Integrated), aims to construct a multidimensional code based tool in order to help programmers and software developers assess their project's level of security-related threats and vulnerabilities. Since there are various places where the projects may contain vulnerabilities, projScan-I is implemented to focus on three aspects of them and striving for revealing meaningful information about any security threats and vulnerabilities. These three detecting and scanning aspects are: 1) vulnerabilities caused by code error, misuse of functions, modules, and APIs; 2) vulnerabilities caused by importing and using vulnerable dependencies including those provided Third-Party Libraries; 3) vulnerabilities caused by having duplicate (or high similarity) code snippets (functions, classes, modules) within the same project. projScan-I not only scans the projects, searching for vulnerabilities in these three aspects, but also saves the raw scanning result data, parsing the data of different types of file, and eventually generates a concrete, summarized, and detailed textual report for each of the three detecting aspects. projScan-I is easily deployed and could be run on Linux environment with most of the virtual machine supported, and it is currently supporting scanning projects written in Java and Python.

The novelty of this project could be illustrated in three perspectives. Firstly, unlike other scanning systems which focus mainly on a single aspect of detecting software vulnerabilities, projScan-I tends to cover all those three aspects within a single running time period. Moreover, with more computing abilities and CPUs provided, projScan-I has the capability of doing those three aspects of scanning concurrently, which significantly enhance its efficiency. Secondly, projScan-I generates textual file reports with fixed information hierarchy and saves them in the machine. Unlike some detection tools that does not organize their scanning results and only display the result in the console, reports generated by projScan-I are well-organized and categorized, which is more readable to the programmers and could be easily parsed and re-used by other program as their inputs. Lastly, projScan-I is compact and does not require too much configuration or set-up procedures to run on Linux environments. Users could set up the project and run it successfully by following the guide within several minutes with ease.

The essential values that projScan-I brings up to the community are 1) the capability of doing multiple aspects of scanning and detection within a single run, and the efficiency improvement it carries out to the users; and 2) the capability of generating highly readable, organized, categorized, and detailed scanning reports, which could be used as the inputs for other programs. In real world, completing more tasks within shorter period of times means higher efficiency and more profits. Instead of probably running several scanning tools, which

may require completely different set-ups and environment configurations, and possibly having multiple machines in order to run them, projScan-I presents the capability of running on a single machine, and with enough computing abilities provided, running those three underline detection processes concurrently. This way of time and resource saving has the potential of dramatically speeding up the vulnerabilities detecting process. Moreover, projScan-I provides high extensibility by having the ability to be not only the ending procedure of a series of programs running sequence. Since its reports are text-based and have highly summarized and categorized information, any project could easily read or parse those reports after running projScan-I and proceed to any further necessary steps.

In terms of the availability to the user community, I will create another Github repo containing only the projScan-I project (by removing all course related files like assignments, proposals and so on). I will make the repo public to all the Github users so that they could access the projScan-I project and try it out if they want to. All the necessary configuration files, source code files, and running scripts will be put in the repo, and a detailed README file containing the guideline of setting up, configuration, deploying, and running the project will also be provided to the community.

2 projScan-I Methods & Techniques

projScan-I is implemented and configured to run on Linux platform with additional software tools and packages needed. The main program of projScan-I is written in Java so that the most recent version of JDK (Java Development Kit) and JRE (Java Runtime Environment) is required to be installed on the environment. Python 3.6 or higher is required for the detection of code duplication and similarity and Docker is required for the static code analysis and dependencies detection. Two more dependencies, provided as jar files, Json-simple and Jsoup are used when parsing the raw scanning data results. Figure 2.1 below illustrates the overall project structure and workflow of projScan-I.

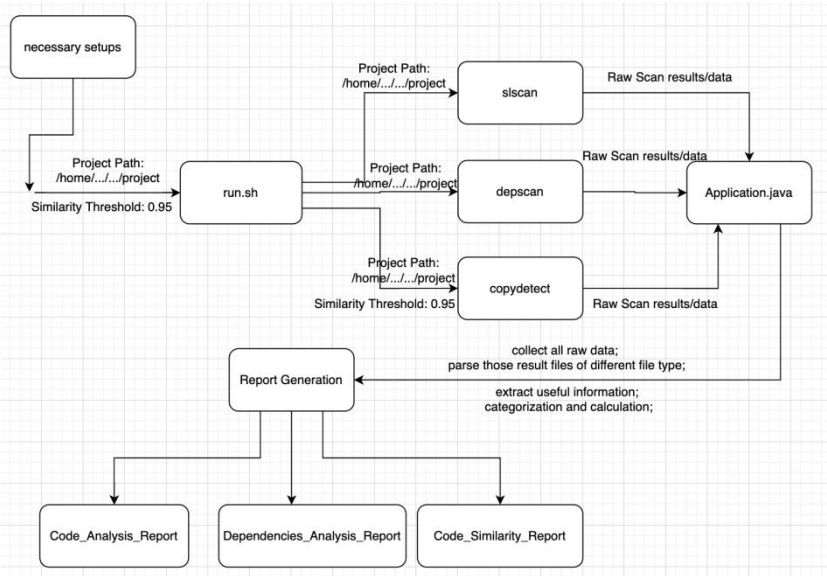

Figure 2.1 projScan-I Structure and Workflow

projScan-I is implemented and extended based on three sub-systems, as shown in the figure 2.1. The slscan is responsible for static code analysis, looking for code errors and any type of misuse. The depscan scans for vulnerabilities in imported and used dependencies. Lastly copydetect is used for determine code duplication by evaluating code similarities between code snippets in every source files. There is a provided bash script called run.sh, which serves as the starting point of the projScan-I. The users run the script along with two parameters: the path of the project being scanned, and the code similarity threshold. The script passes those parameters to the three sub-systems and triggers their execution of scanning. The main program of projScan-I, named Application.java, waits and searches for the raw scan result data, collects those data from different source directories, parses those raw results, analyzes and categorizes those data information, extracts meaningful information, and eventually, writes the generated results to three separate files: Code_Analysis_Report, Dependencies_Analysis_Report, and Code_Similarity_Report.

In the main program of projScan-I, Application.java, two additional tools/packages are being used during the process of data parsing and categorizing. The Json-simple package is used to better extracting, categorizing, and representing JSON data while still maintaining the data hierarchy, and the Jsoup package processes the similar functionalities against the HTML source files and data, the example of using those tools are shown in the figure 2.2 and figure 2.3 below.

```
private static void parseDepAnalysis(String filePath, String depAnaFilePath) {
    Map<String, Integer> countMap = new HashMap<>();
    List<DependenceResult> resultMap = new ArrayList<>();
    try{
        List<String> lines = Files.readAllLines(Paths.get(depAnaFilePath));
        for (String line: lines) {
            JSONParser parser = new JSONParser();
            JSONObject jsonObject = (JSONObject) parser.parse(line);
            DependenceResult resultObject = new DependenceResult();
```

Figure 2.2 Example Usage of Json-simple

```
private static void parseCodeSimiAnalysis(String filePath, String codeSimiFilePath) {
    File htmlFile = new File(codeSimiFilePath);
    Document document;
    try {
        document = Jsoup.parse(htmlFile, "UTF-8");
        Elements p = document.getElementsByTag("p");
        Elements s = document.getElementsByClass("highlight-red");
        Elements g = document.getElementsByClass("highlight-green");
```

Figure 2.3 Example Usage of Jsoup

Here in figure 2.2, after reading the lines from the raw scan result file, the JSONParser in Json-simple will parse the each line of the file as a JSONObject, which will be available to be accessed by certain field to extract specific information. Meanwhile, in figure 2.3, the parse

API in the Jsoup package will parse the whole raw data file (which is in HTML format) with the encoding method specified as “UTF-8”. After parsing the file to a Document object in Jsoup, it could be accessed by attributes in HTML file like tag, class, name, id, and so on, in which retrieving information within specific tags or attributes is fairly easy and effortless.

3 Research Questions

As proposed in the project progress report, three research questions are studied and answered. One side note is that, the fourth research question, after consideration, is eliminated from the list and in the Section 6 Problem Encountered I explain the reason why I eventually choose to get rid of that question and why it is reasonable to make this decision.

There are total of four test projects being used here to study those research questions. Service-operation and Service-client are two Java SpringBoot projects that I implemented with my teammate from COMS4156 Fall 2021. They are built by maven and, for the research purpose, are of different scale. Service-operation is developed for handling various transaction-related situation as well as nearly all the business logic for the application, thus it contains fairly larger amount of Controller, Service, DTO, DAO, Utilities, and so on. On the other hand, Service-client is designed as a middle layer between the actual clients and back end services. Its responsibilities is to redirect HTTP requests to the corresponding service application to process, and its scale is significantly smaller than Service-operation. Connect4 and CULifting are two Python-based projects. Connect4 is borrowed from Assignment 1 of the COMS4156 Fall 2021. CULifting is a flask based web application that involves more complex data processing and transferring, and its scale is fairly larger than Connect4. To summarize, Service-operation (Java, Large Scale), CULifting (Python, Large Scale), Service-client (Java, Small Scale), Connect4 (Python, Small Scale) are being used as the test projects set.

Besides, in order to compare projScan-I’s performance against other tools, I introduce three tools, slscan-basic, slscan-dep, and plaggie to serve as the comparison tools for the corresponding three subsystems. slscan-basic is the basic version of slscan and it skip some advanced scanning options and only perform necessary static code analysis. slscan-dep is the dependency detection version of slscan which will perform similar dependencies vulnerabilities detection. plaggie is another tool for measuring code similarity and detecting code snippets with the number of percentage they overlap with others.

One thing to mention here is that, because the nature of the projScan-I characteristics, which combines three subsystems with different functionality together, it is super hard to find any open source project/tool which processes similar characteristics. To compromised, I choose those three comparing tools and by combing the research data of those three together, they could be view as a hypothetical “project” that has the similar characteristics and functionalities as projScan-I.

RQ 1: How long does projScan-I take to generate results and what level of efficiency it has?

In order to study the time projScan-I takes to generate results, I run the projScan-I on those four test projects and record the total time used for a single run. Because the design benefit, which the executable of the projScan-I is the run.sh bash script, I use Linux time command to measure the time elapsed for projScan-I running on those test projects, as well as running those three comparing tools, the result is shown in the table 3.1 below.

tools	time measure	Service-operation	Service-client	Connect4	CULift
projScan-I	Total time elapsed	237.2590s	166.2700s	7.8890s	25.2970s
	CPU time spent in user mode	4.1256s	3.5213s	0.4012s	0.3918s
	CPU time spent in kernel mode	0.2600s	0.2510s	0.2200s	0.2370s
slscan-basic	Total time elapsed	33.1900s	25.0223s	15.3510s	23.3170s
	CPU time spent in user mode	2.3988s	1.8200s	1.1030s	1.1787s
	CPU time spent in kernel mode	0.2500s	0.2300s	0.3712s	0.2600s
slscan-dep	Total time elapsed	202.2590s	188.6980s	0.0090s	0.0090s
	CPU time spent in user mode	3.1122s	3.5720s	0.0010s	0.0000s *
	CPU time spent in kernel mode	0.2503s	0.2300s	0.0010s	0.0010s
plaggie	Total time elapsed	40.0137s	38.2618s	0.5741s	3.2877s
	CPU time spent in user mode	1.2245s	1.2399s	0.3718s	0.2213s
	CPU time spent in kernel mode	0.2510s	0.2368s	0.1550s	0.2188s

Table 3.1 Experiment Result of RQ1

Based on the result shown in the table 3.1, some observations could be made. First of all, the scale difference between Service-operation and Service-client, Connect4 and CULift is obvious enough to influence the time needed for completing the execution, which shows that project scale is one of the factors that could have impacts on projScan-I running time. Secondly, since we consider projScan-I being compared with a hypothetical “project” consists of slscan-basic, slscan-dep, and plaggie each doing a corresponding task as slscan, depscan, and copydetect in projScan-I, according to the total time elapsed, projScan-I has efficient advantages against this set of three projects. Especially when running on Service-operation, projScan-I completes the task even faster than slscan-dep, which is only one step of the whole process. This kind of situation also happens on Connect4 when slscan-basic takes even longer than the whole running process of projScan-I. Thirdly, notice that in the table 3.1, slscan-dep runs very fast on Connect4 and CULift, with elapsed time only 0.009s. Moreover, the CPU time spent in user mode for CULift is even 0.0000s. The reason behind is probably that there is no dependency detected in both Connect4 and CULift, the time elapsed will be relatively small, some may not be able to be reflected by the Linux time command. Overall, it appears that projScan-I has slight efficiency advantages against the comparing tools, and this kind of advantages are reflected by smaller time elapsed of completing a single detection run.

Limitations

There are several limitations of the above experiment for RQ1. Firstly, using Linux time command to measure time elapsed may causes some issues. The time it measures may vary on different virtual machines with different kind of configuration and computing capability. Besides, the work load caused by running other program on the back end may affect the speed of running both projScan-I and other three comparing tools. That why, in my observation above, I did not conclude anything based on the absolute value of the time elapsed. All the

observations and conclusions are drown based on the comparison between those time measurements. Secondly, the choice of those three projects that combining to form a hypothetical "project" may heavily influence the result of RQ1. As I mentioned at the beginning of this section, one of the main issue of conducting experiments and comparing projScan-I's performance against other projects, is the fact that I can not find proper comparison for projScan-I. projScan-I is developed by extending on three subsystems that perform features that including those provided by each single system, and this is part of why the "I" stands for "Integrated". However, I seemed to have trouble finding another project which processes the similar functionalities as projScan-I. To compromise, I introduce three tools representing three subsystems comparing to projScan-I's and assess their performance individually. This brings up the issue of 1) whether the overall performance is simply equals to the combination of three subsystems sequentially (maybe there would be some sort of concurrency going on if the comparing tool is a single integrated project), and 2) having a different choice of those three comparing tools would possibly have very different result, and that why I only concluded that projScan-I has efficiency advantages against those three-tool combination. In order to address those issues, more work needs to be done in the future.

RQ2 & RQ3 How effective is projScan-I in terms of the number of vulnerabilities it detects; And how correct they are?

Here I combine RQ2 and RQ3 in the project progress report together since both of them require analysis of the same experimental results. Because, according to the projScan-I's design the detection results will be included in the generated reports, I run the projScan-I several times on each of the test projects until I get the consistent results. After running projScan-I, I retrieve those generated reports and record the information of number of vulnerabilities detected as well as their level of severity (if applicable). Then I run slscan-basic on those test projects to get the code analysis result, slscan-dep to get dependency analysis result, and plaggie to get code similarity result. I record those results and categorize them according to the type of detection, and try to draw observation and conclusion from it. The results are shown below in table 3.2, 3.3, and 3.4.

tools	severity	Service-operation	Service-client	Connect4	CULift
projScan-I	Total	4	3	6	5
	High	2	1	3	0
	Critical	2	2	0	0
	Low	0	0	1	1
	Medium	0	0	2	4
slscan-basic	Total	3	3	6	5
	High	1	1	3	0
	Critical	2	2	0	0
	Low	0	0	1	1
	Medium	0	0	2	4

Table 3.2 Experimental Result of Code Vulnerabilities

tools	severity	Service-operation	Service-client	Connect4	CULift
projScan-I	Total	16	17	0 *	0
	Medium	4	4	0	0
	Low	9	10	0	0
	Critical	3	3	0	0
slscan-dep	Total	16	16	0	0
	Medium	4	4	0	0
	Low	9	9	0	0
	Critical	3	3	0	0

Table 3.3 Experimental Result of Dependency Vulnerabilities

tools	Service-operation	Service-client	Connect4	CULift
projScan-I	17 (11.04%)	17 (10.97%)	0 (0%)	4 (6.56%)
plaggie	20 (12.90%)	21 (13.55%)	0 (0%)	4 (6.56%)

Table 3.4 Experimental Result of Code Similarity (threshold = 0.95)

Based on the results shown in table 3.3 to 3.5, several observation and conclusion could be made. First of all, in terms of code static analysis, projScan-I achieves nearly the same results as slscan-basic excepts it detects and categorizes one more High severity vulnerability in Service-operation. One possible reason is that because slscan, which projScan-I extends on, is the advanced version of slscan-basic. Taking this factor into consideration, it is reasonable that projScan-I could achieve more precise results and why their detecting result are quite similar. Secondly, in term of dependency analysis, because of no dependency detected, the results for Connect4 and CULift are all 0. projScan-I detects one more Low severity dependency vulnerability in Service-operation than slscan-dep, which suggests that the underline dep-scan possibly achieves higher precision when detecting dependency vulnerabilities comparing to slscan-dep. Thirdly, the code similarity result was running with threshold setting to 0.95, which means the report will only include the files that overlap over 95% of the content with another. I think 0.95 is a reasonable setting because within the scope of a project, there is possible that some portion of the code will be similar to others for some reason. It is not meaningful to catch all those duplication as potential threats. However, anything over 95% means there may be exactly several copies of the same code snippets, which brings up the possibility of inconsistent code version or misuse of any old/deprecated version of code. The result shows that plaggie detects more files with over 95% similarity than projScan-I's underline copydetect tool. Although the difference is not dramatic, a further human inspection is needed to determine whether there is truly a code duplicates issue here in those files.

Limitations

There are two major limitations within the experimental process of RQ2 and RQ3. The most important one is that, it is hard for me to study test metrics and benchmarks like false positive cases or false negative cases which are essential to those kind of research, because the lack of trustworthy truth ground. And this is the main reason why in the above analysis of the results, I only draw conclusions from comparison of another tool, not the absolute results of

individual cases. Because those projects are borrowed from previous assignments, and there are not used or analyzed by others, it is difficult for me to acquire the truth ground of how many code errors, vulnerable dependencies in each of the project. Although I could maybe try several tools on them and view the combination of those results as the truth ground, this kind of experimental truth ground has no credential, and it is not verified by trustworthy programs or origination. Thus, in this case of research, I am not able to acquire the trustworthy truth ground for the number of code errors and threats, and vulnerable dependencies in those four projects, so results of false positive or false negative is not applicable here, which definitely limits the conclusions that I could draw from the data. In order to address this issue, future work needs to find test projects with established data of truth ground in those two aspects, and repeats the experiment to product the results including false positive or false negative measurements

The second limitation is that the result of code similarity is somehow subjective. A larger number of detected files does not directly imply the better performance of detecting code duplicates. The result of code similarity scan need to be viewed by the software developers in order to determine which of them are really the vulnerability, and this type of judgment will also vary in different context and application. Besides, different techniques including tokenizing, categorizing will have impacts on the results in terms of similarity determination and calculation. Thus, more files detected with similarity above threshold may only imply the possible better capability of finding those files.

4 Deliverables

Link to the Github Repository: https://github.com/tianzhh9665/projScan-I

All the required deliverables will be placed in the Github repository with the link specified above. And the following part shows the file hierarchy inside the repository:

---- README
- ---- Brief introduction of the projScan-I project, its capability, novelty, and values
- ---- Set-up Instructions
  - ---- Step by step instructions of how to set up the running environment for projScan-I to run on the Linux platform
  - ---- Instructions of installing required software
  - ---- Instructions of checking all environment requirements met
  - ---- Instructions of where to put certain project file (Application.java) in
- ---- Running Instructions
  - ---- Instructions and commands to run the projScan-I
  - ---- Example running commands and running results

---- projScan-I
- ---- This folder contains all the code, scripts, source files, and included jar packages of projScan-I including: Application.java, run.sh, Jsos-simple-x.jar, Jsoup-x.jar.

This folder could be viewed as the deliver package of the projScan-I project.

---- Test Project

---- This folder contains the test projects that I used to evaluate the projScan-I project's performance as well as studying the research questions

---- Assignment Related Documentation

---- This folder contains all textual assignments files of this final project including: project proposal, revised project proposal, project progress report, project final report, and in-class project demo slides.

---- Generated Reports And Evaluation Results

---- This folder contains the reports generated by projScan-I when running against different test projects, three per project: Code_Analysis_Report_<project_name>, Dependencies_Analysis_Report_<project_name>, and Code_Similarity_Report_<project_name>.

---- Also it will contains the result generated by the evaluation of research questions.

---- Raw Scan Result Data

---- This folder will contains the raw data generated by the three sub-systems, which projScan-I reads and parses from.

5 Self-Evaluation

This is the project assignment which I have the opportunity to choose a topic that I am interested in and willing to spend time working on it. By walking through several iterations of assignments, from composing the project proposal, implementing the project, all the way to writing the final report, I have learned and accomplished several things which are meaningful for me throughout the journey of being a Software Engineer.

I was inspired by the researches and resources in my midterm paper while deciding the topic of the project. Having a relatively broad idea in mind, I was able to narrow down the scope of this project to detect software vulnerabilities in the three aspects I mentioned above. During this process, I learned that choosing a proper, workable goal of the project is critical before actually kicking things off. I need to make sure that 1) I am able to find enough workable and extendable software to build on, 2) those software have reasonable scale and can be installed, deployed, and run without requiring too much efforts on environment setting and configuration. While implementing the project, there are just so many things that could possibly go wrong. Thanks to my software developing experiences, I came up with a clear plan listing each parts' relation with another, figuring out the best start point to implement and doing this incrementally. A well-planned developing schedule let me to implement the project in a efficient and organized way, which helped me to finish the all the coding part of the project before the in-class demo. Moreover, I learned that never get stuck in on place for too long. There are so many online resources and solutions available all the time, and sometimes I found myself being in a dead-end road and did not realize to turn around and go back. In those situation when I was not making progress for a while, I should look for some hints or solutions on the Internet, or ask some one for help.

Lastly, while I was writing this final report, I realized that having or proposing an primitive idea to develop a project is fairly easy. When asked about those ideas, I may be able to give many of them with different approaching angles of a specific problem. However, putting those kind of idea into reality, real project is hard, and it requires people to have organized and workable plan, stick to the development schedule, solve problems one after another, cooperate with teammates, and eventually perform necessary tests to assess reliability and performance.

6 Problem Encountered

During the process of developing projScan-I, I have encountered several problems and they are summarized below:

- During the process of environment setting up, I initially set up all the environment requirements and installed necessary software in the Virtual Machine running on the VMare of my local machine. Then I discovered that the kernel setting and the version of the Virtual Machine is not compiled with some of the software. I then solve this problem by reusing a Virtual Machine created for COMSE6111 class which has the relatively new and compilable version of computing engine.
- Because of the nature of the Java project which includes lots of dependencies, it needs to be built by either Maven (like SpringBoot, SpringCloud project) or other tool to produce a X.jar or X.build file in order to be scanned by deepscan. I think this is a reasonable prerequisite for the users to provide those build file for scanning since for those projects, it somehow makes little sense to be scanned when they are un-built, especially for dependency scanning.
- One of the research questions is related to evaluate the false positive, false negative number of the results produced by projScan-I. After diving into this research question, I found out that it is quite difficult to establish the truth ground for those projects, especially those I wrote from the COMS4156 class. Without having a trustworthy truth ground, it is not possible to output those false positive or false negative cases, as well as evaluating its performance based on this criteria.

PLANNED BUT DIDN'T DO

During the in-class demo, when talking able future work to do, I was saying that I planned to combine those three generated reports into one single one for better organization. After considering its effectiveness, I feel like having three separate reports corresponding to three aspects of detection is more clear and organized. Besides, considering one of the projScan-I value is having the capability for other program to use the reports it generates as the inputs, having them separated makes more sense and provides more flexibility for other programs to reuse the generated results.

Another thing is that I eliminated the research question 4 in the project progress reports when

studying those research questions. This research question focus on “Is the final report readable and meaningful to the project developers (does it contain meaningful and useful information for the project developers to locate the problems)”. After consideration, I think it is too subjective to evaluate this question as “readable” or “meaningful”. Those kind of judgments and evaluations are vary from people with different background, working experience, and their definition of what is “readable” and “meaningful”. Assessing this aspect would require more time and efforts to maybe conduct several user studies or questionnaires, which I do not think I am able to wrap those up before the submission.

7 Reuse of Software

There is list of software and projects that I refer to and reuse in implementing projScan-I. As described on the Section 2 “projScan-I Methods & Techniques”, first three of them are parts of the sub-systems, the rest of them are test projects which are used during the process of evaluating projScan-I’s performance as well as those three research questions, and tools that are used in the implementation of the project.

- depscan: https://github.com/AppThreat/dep-scan
- slscan: https://slscan.io/en/latest
- copydetect: https://github.com/blingenf/copydetect
- Sangria (from COMS4156): https://github.com/tianzhh9665/SANGRIA
- Connect4 (from COMS4156): https://github.com/tianzhh9665/COMSW4156-ASSIGNMENT-I1
- CULifting: https://github.com/tianzhh9665/COMS4170_Project
- Json-simple: https://code.google.com/archive/p/json-simple
- Jsoup: https://jsoup.org
- slscan-dep: https://slscan.io/en/latest/getting-started/#scanning-java-projects
- slscan-basic: https://slscan.io/en/latest/#sample-invocation

8 Appendices

```
th2888@cs6111vm:~/cs6156$ ./run.sh /home/th2888/cs6156/SANGRIA/service-operation/ 0.95
Project Path Being Scanned: /home/th2888/cs6156/SANGRIA/service-operation/
Code Similarity threshold: 0.95
Start Scanning Processes. This may take some time to finish...
```

Figure 8.1 Example of Running projScan-I on Linux

```
Generating Analysis Reports...
Reports have been generated in /home/th2888/cs6156/SANGRIA/service-operation/ as Code_Analysis_Report_*, Dependencies_Analysis_Report_*, and Code_Similarity_Analysis_Report_*
th2888@cs6111vm:~/cs6156$
```

Figure 8.2 Example of projScan-I's Prompt When Completed

```
Java Project Code Static Analysis Result

Scanned Project Path: /home/th2888/cs6156/SANGRIA/service-operation/

Code Threat and Vulnerabilities Found Summary:

1).Source File Result:
total    0
high    0
critical    0
low    0
medium    0

2).Infrastructure Security:
total    0
high    0
critical    0
low    0
medium    0

3).Class File Result:
total    4
high    2
critical    2
low    0
medium    0

4).Secrets Audit:
total    0
high    0
critical    0
low    0
medium    0
```

Figure 8.3 Example of Code Static Analysis Report 1

```
Found Vulnerabilities Detail Information:

1).Source File Result:

None

2).Infrastructure Security:

None

3).Class File Result:

1.Issue Details:

This random generator (java.util.Random) is predictable
At CommonUtils.java:[lines 12-52]
In class com.sangria.operation.utils.CommonUtils
In method com.sangria.operation.utils.CommonUtils.generateUniqueId(String, int)
At CommonUtils.java:[line 40]
Value java.util.Random.

File Location: file:///home/th2888/cs6156/SANGRIA/service-operation/src/main/java/com/sangria/
operation/utils/CommonUtils.java
At Line 40, code content:               int randomInt = new Random().nextInt(20);

Issue Level: error
Issue Severity: HIGH
```

Figure 8.4 Example of Code Static Analysis Report 2

```
Project Dependencies Analysis Report

Scanned Project Path: /home/th2888/cs6156/SANGRIA/service-operation/

Dependency Analysis Found Vulnerabilities Summary:

MEDIUM    4
LOW    9
CRITICAL    3

Found Vulnerabilities Detail Information:

1.
ID: CVE-2022-22965
Package Name: org.springframework:spring-beans
CVSS Score (A higher score indicates higher severity): 2.0
Severity: LOW
Related Urls:
1.https://nvd.nist.gov/vuln/detail/CVE-2022-22965
2.https://github.com/spring-projects/spring-framework/commit/002546b3e4b8d791ea6acccb81eb3168f51abb15
3.https://github.com/spring-projects/spring-boot/releases/tag/v2.5.12
4.https://github.com/spring-projects/spring-boot/releases/tag/v2.6.6
5.https://github.com/spring-projects/spring-framework/releases/tag/v5.2.20.RELEASE
6.https://github.com/spring-projects/spring-framework/releases/tag/v5.3.18
7.https://spring.io/blog/2022/03/31/spring-framework-rce-early-announcement
8.https://tanzu.vmware.com/security/cve-2022-22965
9.https://github.com/spring-projects/spring-framework
Threat Description:
# Remote Code Execution in Spring Framework
Spring Framework prior to versions 5.2.20 and 5.3.18 contains a remote code execution vulnerability
known as `Spring4Shell`.

## Impact

A Spring MVC or Spring WebFlux application running on JDK 9+ may be vulnerable to remote code
execution (RCE) via data binding. The specific exploit requires the application to run on Tomcat as a
WAR deployment. If the application is deployed as a Spring Boot executable jar, i.e. the default, it
is not vulnerable to the exploit. However, the nature of the vulnerability is more general, and there
may be other ways to exploit it.

These are the prerequisites for the exploit:
- JDK 9 or higher
- Apache Tomcat as the Servlet container
- Packaged as WAR
- `spring-webmvc` or `spring-webflux` dependency

## Patches

- Spring Framework [5.3.18](https://github.com/spring-projects/spring-framework/releases/tag/v5.3.18)
and [5.2.20](https://github.com/spring-projects/spring-framework/releases/tag/v5.2.20.RELEASE)
- Spring Boot [2.6.6](https://github.com/spring-projects/spring-boot/releases/tag/v2.6.6) and
[2.5.12](https://github.com/spring-projects/spring-boot/releases/tag/v2.5.12)
```

Figure 8.5 Example of Dependency Analysis Report

```
Code Similarity Analysis Report

Scanned Project Path: /home/th2888/cs6156/SANGRIA/service-operation/

Code Similarity Found Summary:

Number of files tested: 154
Number of reference files: 154
Test files above display threshold: 17 (11.04%)

1).
Test file: /home/th2888/cs6156/SANGRIA/service-operation/src/main/resources/application.yml (99.61%)
Reference file: /home/th2888/cs6156/SANGRIA/service-operation/target/classes/application.yml (99.61%)
Token overlap: 254

Similar Code Snippet:

spring:
  profiles:
    active: dev
  application:
    name: service-operation


mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  mapper-locations: classpath:com/sangria/operation/mapper/*Mapper.xml
  type-aliases-package: com.sangria.operation.entit


spring:
  profiles:
    active: dev
  application:
    name: service-operation


mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  mapper-locations: classpath:com/sangria/operation/mapper/*Mapper.xml
  type-aliases-package: com.sangria.operation.entit
```

Figure 8.6 Example of Code Similarity Analysis Report

```
drwxrwxr-x 2 th2888 th2888   4096 Apr 24 09:04 logs
-rw-rw-r-- 1 th2888 th2888   6790 Dec 16 23:38 mvnw.cmd
-rw-rw-r-- 1 th2888 th2888   3095 Dec 16 23:38 pom.xml
drwxrwxr-x 4 th2888 th2888   4096 Apr 24 08:39 src
drwxrwxr-x 8 th2888 th2888   4096 Apr 24 09:11 target
th2888@cs6111vm:~/cs6156/SANGRIA/service-operation$
```

Figure 8.7 Example of Test Project Service-operation (Java Project) Structure

```
-rw-rw-r-- 1 th2888 th2888  10068 May  7 17:19 server.py
drwxrwxr-x 2 th2888 th2888   4096 May  7 17:20 static
drwxrwxr-x 2 th2888 th2888   4096 May  7 17:20 templates
th2888@cs6111vm:~/cs6156/SANGRIA/CULift$
```

Figure 8.8 Example of Test Project CULift (Python Project) Structure

Notes: Raw experiment result data could be found in the Github repository under **Generated Reports And Evaluation Results** folder.