Loky : Static Code Analyser to Test Vulnerabilities

Nidhi Priya
nidhipriya@vt.edu
Department of Computer Science
Virginia Tech

Prachi Pundir
prachipundir@vt.edu
Department of Computer Science
Virginia Tech

***Abstract*—This document illustrates a developer centric model Loky for Static Code Analysis to improve Security of Web Applications empowered by early vulnerability detection and taint analysis. Its core is a configurable rule engine comprising of popular attacks - SQL injections, Cross Site Scripting, Server Side Request Forgery, Linting errors, Hard-coded Credentials**

***Index Terms*—Security, SAST, Vulnerabilities, Rule Engine, SQL Injection, XSS, SSRF, Linting**

I. Introduction

Academics in the software engineering field have spent decades inventing a variety of methods for finding defects in code, but the approaches mostly fall into one of two types of scanning: static application security security testing (SAST) and dynamic application security testing (DAST) [15]. When most developers think of scanning applications for weaknesses, they think of static analysis, which involves inspecting the coding for problems. A vulnerability is a flaw or weakness in a system design or implementation (the way the algorithms are coded in the chosen programming languages) that could be exploited to violate the system security policy [16]. Static code analysis is particularly good at detecting common problems like directory traversals, cross-site scripting, sql injection, and other vulnerabilities. Dynamic analysis examines how a program interacts with its surroundings while it is executing. Dynamic excels at troubleshooting server and deployment configuration errors.

A secure software demands effective techniques for vulnerability detection in the course of its development cycle. The exercise of detecting security flaws before the deployment phase eliminates the dangers that vulnerabilities may additionally impose on the system. Static evaluation and dynamic evaluation strategies offer two complimentary tactics for checking vulnerabilities [13]. Static evaluation entails the scanning of supply code or binary code , putting off the need of executing it [14]. This approach is fast and has no run time overhead. However, static evaluation are pretty imprecise and generate big false positives and fake negatives. The hassle of false positives and negatives is less in case of dynamic analysis because they examine with the aid of strolling the test cases, but this technique calls aims to ensure a positive confidence level in detecting vulnerabilities [9]. This paper aims at using static code analysis to find vulnerabilities in web applications, without executing the code. We introduce our Static Code Analyzer - Loky for identification of vulnerabilities during the early stages of development.

II. PROBLEM STATEMENT

A. Motivation

How can we tell whether software : Developed, Installed or Bought is safe to run ?
The earlier a bug is caught, the cheaper it is to fix.

Applications are evolving with every phase in the software development lifecycle. As we develop and test an application from requirement to post-release, security remains the non-negotiate attribute. We encounter several security vulnerabilities in both static and dynamic environments. Identifying the bugs early on is certainly a boon. Static code analysis showcases various predetermined vulnerabilities or potential errors within a static state by utilizing a number of testing mechanisms like Lexical analysis, Taint analysis, Data flow graphs, Control flow graphs, etc. [11]

B. Attacks

Since the target was to have a Static code analyzer that finds issues by reviewing the code on non - runtime environment, without executing the program, hence it is important for us to reproduce the exploitable vulnerabilities [4]. There could be numerous vulnerabilities in the application. For this project, we limit the scope to the following:

1) **SQL Injection** : It is a common vulnerability over backend/databases that consumes malicious SQL code for manipulation to access information that was not intended to be displayed.
2) **Cross-Site Scripting** : These attacks are a type of injection, in which malicious scripts are injected into benign and trusted websites.
3) **Server Side Request Forgery** : This exploitation type induces the server-side application to make HTTP requests to an arbitrary domain of the attacker's choice.
4) **Linting Errors** : The vulnerabilities induced by programming errors, bugs, stylistic errors, and suspicious constructs etc. where the quality of code and the product can be compromised.
5) **Hardcoded Credentials** : This attack enables the bypass of the authentication configured by the administrator by using hardcoded credentials such as a password or cryptographic key.

Through this study, we target to help express code standards, surface early bugs, attain greater coverage and ensure faster results for maximizing security protection via Static Code Analysis. [12]

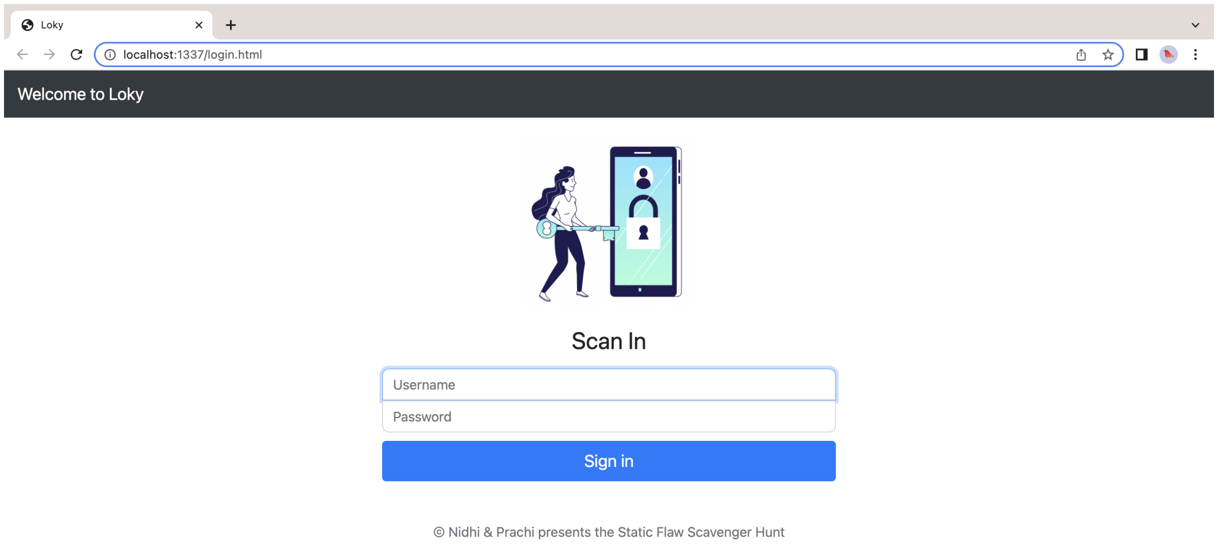
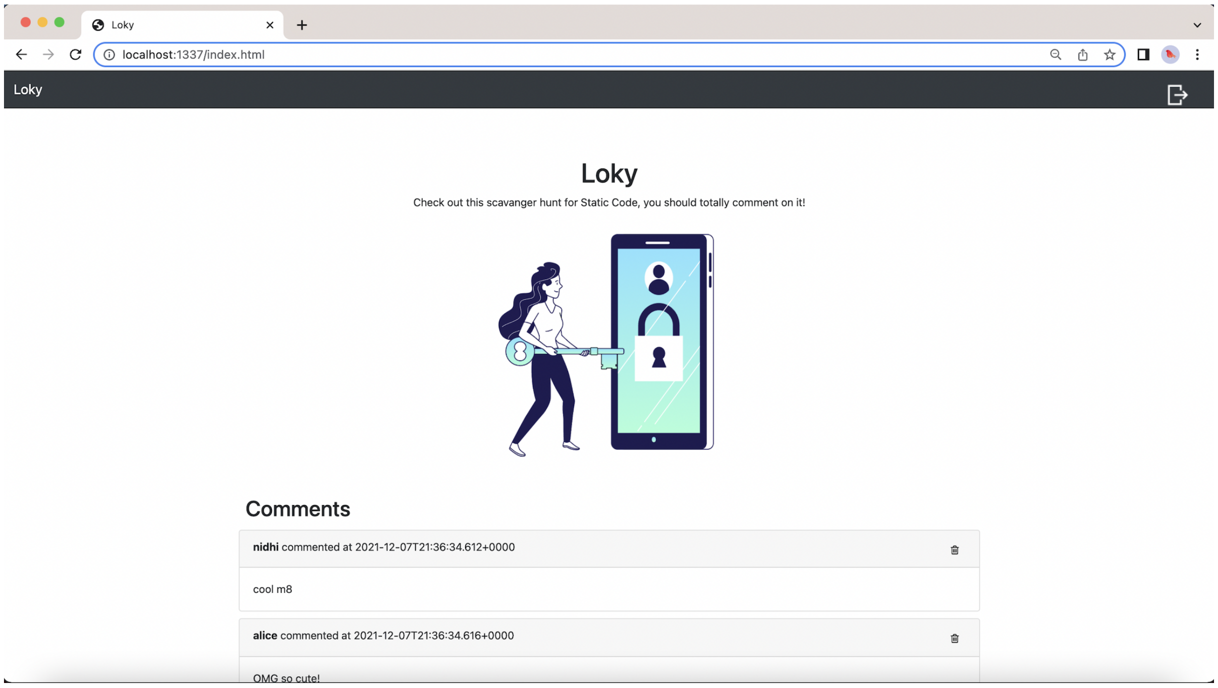

Loky
localhost:1337/login.html
Welcome to Loky
Scan In
Username
Password
Sign in
© Nidhi & Prachi presents the Static Flaw Scavenger Hunt

Fig. 1. User Interface of Loky

a) This is a platform developed to reproduce vulnerabilities. The rules from the rule engine are prescribed on the Web Application. Our intent is to exploit the system and ensure Loky finds all the relevant Vulnerabilities while we move from the Login screen through the authentication mechanism within the application.:



Check out this scavanger hunt for Static Code, you should totally comment on it!

nidhi commented at 2021-12-07T21:36:34.612+0000

cool m8

alice commented at 2021-12-07T21:36:34.616+0000

OMG so cute!

Fig. 2. User Interface of Loky

b) This figure illustrates unauthorized access to the system. We understand that Static code analysis tools are being increasingly used to improve code quality. This tool statically analyze the code to find bugs, security vulnerabilities, security spots, duplication, and code smell. Several use cases are identified, selected by conducting a systematic literature review, exploiting the system and detecting vulnerabilities in the field of static code analysis. [6] :

III. LIMITATIONS OF CURRENT WORK

Static code analysis being the initial testing technique on code before it is deployed on the server helps in reducing the cost overhead of fixing vulnerabilities while the system is up and running and save applications from attacks that cannot be resolved using manual [**?**]. There are many types of vulnerabilities and many approaches to detect such flaws and perform security tests. All the known approaches present pros and cons but none of them stands as a perfect solution. Static analysis is one of the approaches and it can be defined as the analysis of a software without its execution. Our research showed that 97 percent of typical Java applications are made up of open source libraries. Unfortunately, 79 percent of developer as never update third-party libraries after including them in a codebase resulting in unnecessary vulnerabilities. [6] Since security is an area that needs to be explored thoroughly, the motive of static analysis is to increase false negatives and decrease false positives, which is where the existing SAST tool's performance declines. Also the errors displayed are sometimes high level and needs a decent amount of expertise in testing or debugging in order to fix those bugs. With Loky we have created a robust systems that works on finding all the false negatives and as part of the extension of Loky a recommendation system that can help understand the developer about the type of vulnerability and how it can be removed.

IV. APPROACH

We have devised Loky – the Static code analyser. This tool is the baseline for vulnerability identification, mitigation, prevention and cure in the code. Loky scans the completed binary code of an application, accurately discovering, analyzing, and contextualizing security flaws.

A. System Design

Components	**Technical Stack**	**Host Location**
Frontend	ReactJs	User interface hosted on localhost:1337/login.html
Backend	SpringBoot	API hosted on tomcat webserver at localhost:8080
Databases	PostgreSQL	Relational Database management system hosted on Docker Containers
Deployment	Docker	Light-weight Docker Containers, Docker Compose, Docker Hub

B. Components of Loky

1) **Input Seed** : The input seed will be supplied to the rule engine for scans and by default, all rules will be running against this file. We can provide the root path of the files we are going to use as input seed within the Analyzer. In our evaluations, we considered java code [10].
2) **Rule Engine** : Loky is dictated by the Rule engine. The Rule Engine has a set of defined rules to capture vulnerabilities that can compromise the code quality. The engine does it using a pattern under each defined rule. These rules include the following [1]–

R1: SQL Injection
Steps to reproduce: The ability of an attacker to gain access to the system by running arbitrary SQL into an insecurely created query [12].
Methodology: The SQL injections are built by the escape characters such as **'** that permits the attacker to manipulate the input and fire another query directly into the database.
Use Case: In our application, we will be abusing the Authentication Mechanism in Loky via SQL Injection.
Input: SQL String that consumes the username and password. Using *'md5'* as an encryption mechanism to hash the password. Then compares the hashed password to the database.
Attack: The attacker executes an arbitrary SQL over our endpoint *'/login'*.The field that would be exploited would be '*username*'. This is so because our password is hashed before the attacker reaches the database. To gain access using *'username'* we try, *'admin'* or 1=1 limit 1 --
Output:

Scenarios	**Error Code**	**Status**
Login with right credentials	200	Success
Login with wrong password	401	Unauthorized
Expose SQL injection by updating password	500	Internal Server Error
Login with modified password	200	Success

R2: Cross Site Scripting
Steps to reproduce: The ability of the attacker to be able to inject arbitrary JavaScript or malicious code over a web application and have that JavaScript be rendered by another [**?**].
Methodology: The vulnerabilities enforcing the attack are widespread and can be found at any place within the web application that takes input from the user and generated output without validation or encoding.
Use Case: In our case, we will be abusing the Authentication Mechanism in Loky where authentication tokens can easily be sent to another site via JavaScript to an unsuspecting user. The client browser cant detect the malicious/ vulnerable script and will go ahead with execution of the script. Since the host assumes that the script came from a trusted source, the script can access

any cookies, session tokens, jwt tokens, authorization code, timestamps or other sensitive information retained by the browser used within the session.
Input: In the index.html we render a username value as a ‘data‘ attribute.
Attack: In case the user is able to amend the username value, stored on the server, by adjusting the HTML to run their own scripts on others’ computers, there will be a potential attack.
Output: We noticed not all values are Sanitized before rendering them. Iteration: Take the *’sessionToken’* from the local Storage. Have a *’netcat listener’* to trace the results.

R3: Server Request Forgery Attack
Steps to reproduce: The attacker can abuse functionality on the server to import data from URL, read or update and thus tamper internal resources by supplying or modifying a URL which the code is running on.
Methodology: The attacker will be able connect to internal services [8] and will be able to access server configurations, the payload, the metadata, the http enabled databases. It will also enable the post requests towards internal REST interfaces, Database HTTP interface, Files hosted on server, which should not be exposed. When the tampered request reaches the server, the server-side code picks up the altered URL and reads/publishes data to the altered URL. To recapitulate, an attacker can practically scan the entire network and retrieve sensitive information [3].
Use Case: In our application, an endpoint will reach out to a website and scrape the HTML for valid links.
Input: Hit the endpoint *‘/links.‘*
Attack: We abuse the web crawling by executing Server Request Forgery Attack by iterating over the internal IPs against the endpoint *‘/links.‘*.
Output: The docker container will give a list of IPs. The web server now can access the internal information, hence, acquiring access to sensitive information from this internal site. In our case, extracting the list of internal email addresses and names from the *’/links’* endpoint.

R4: Linting
Steps to reproduce: Finding defects which lead to bugs or inconsistencies with code health and thus the product health. Induce issues dealing with class design problems, code indentation, styling and formatting [2].
Methodology: Identifying Linting bugs via Loky, improves the quality, readability, re-usability of the code and may reduce the cost of development. Checkstyle is the imported modules used for identifying the following:

- Indexing beyond arrays
- De-referencing null pointers
- Unreachable code
- Non-portable constructs
- Block Checks
- Class Design
- Unused Headers and Imports
- Javadoc Comments
- Size Violations
- Whitespace

R5: Hardcoded credentials
Steps to reproduce: If a malicious user tries accessing the application by feeding in the default password and logging in with complete access.
Methodology: We can configure the rule engine with patterns - ”pattern”: (?i) [password, pwd, private key, pass, rsa token, digipass]

C. Rule Engine to CWE Mapping

Common Weakness Enumeration (CWE) Top 25 is a list of software weaknesses that serves as a yardstick for security tools. [5]
Every weakness/ vulnerability detected by Loky has a relationship matrix with a CWE for a given Rule Id [5]. This enables us to keep a uniform approach and describe the issues in terms that are more generic the vulnerability. The exact mapping can be found here.

D. Rule Selection

By default, Loky executes all rules over the supplied input seed. The analyser can be customized. It facilitates a convenient option of choosing a subset of rules that can be included or excluded using a flag against each rule of the engine.
The next step is to load the Analyzer configuration, for which the following methodology is used:

1. Checkout source code
2. Run Loky with arguments
3. Verification done on the basis of Designing, Formatting, detecting Complexities

E. Priority Matrix

Determine a score against each error with respect to – “Type”, “Severity”, “Confidence”, “Bug Patterns”.
Basis the above, a score of –
“High”, “Medium”, “Low” would be granted.

F. Output

The results are reported to the terminal but it can also be written in an output file, eg , txt, json, csv and yaml.
The Raw Results are in the following format:
xyz lines of code scanned
Audit Start
Vulnerability Type
Priority Score
Audit End
Number of warnings found
Number of errors found

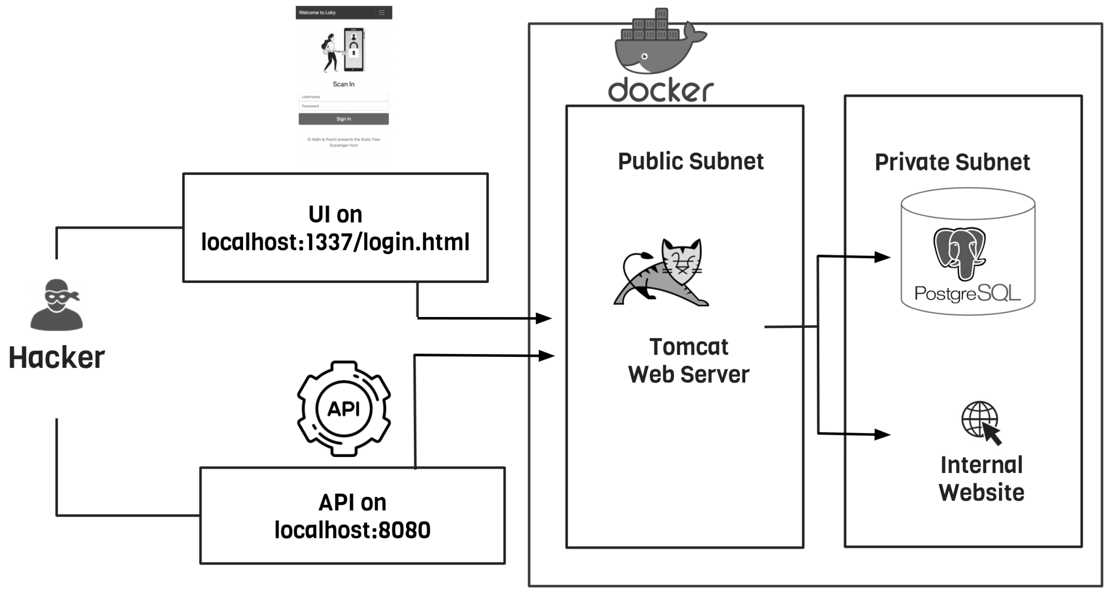


Fig. 3. Loky Design

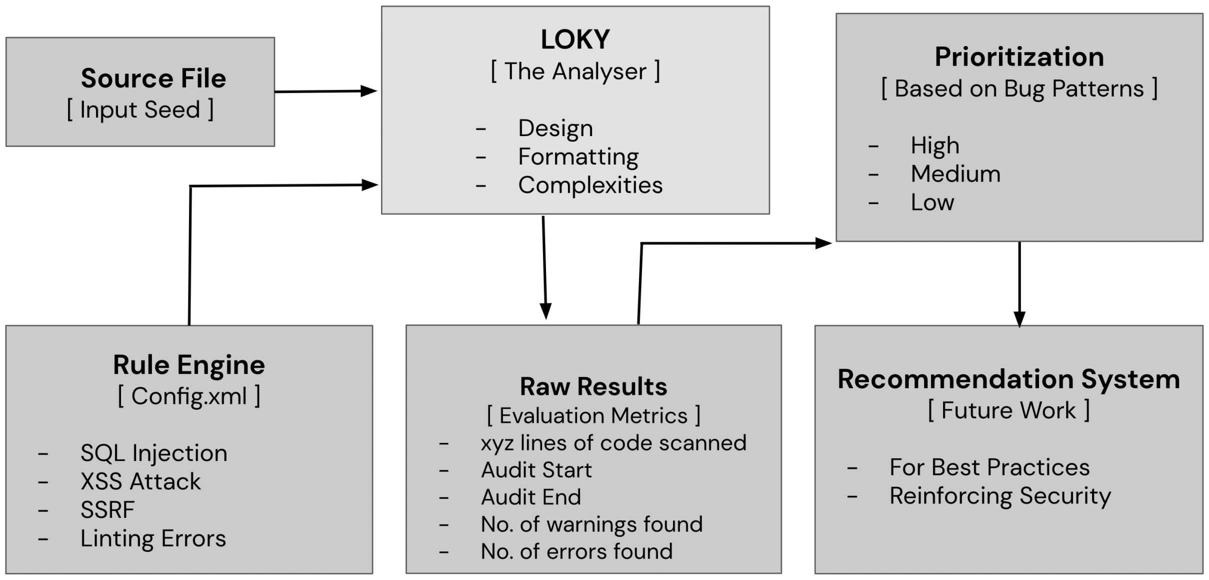


Fig. 4. Loky Architecture

V. EVALUATION

Vulnerabilities	**Actions**	**Analyzer Results**
SQL Injection	Exploit the endpoint *'/login'* using a SQL injection	Successfully identified
Cross-Site Scripting	Navigate to *'http://localhost:1337'* and use it to log into the site	Successfully identified
Server Side Request Forgery	Iterate over all the internal IPs. Exploit the endpoint *'/links'* using SSRF	Successfully identified
Linting errors	Indentation, FileTabCharacter, JavaDocPackets, Unused Imports	Successfully identified
Hard-coded Credentials	Exploit the application using Keyword detection for passwords	Successfully identified

VI. CONCLUSIONS & FUTURE WORK

In summary, Loky's efficiency in finding security vulnerabilities using rule engine further extends the idea of static code analysis in terms of identifying relatively large percentage of application security flaws and to great extent was able to reduce false positives and capture maximum false negatives.

From future prospective, we intend to primarily create a plugin for the tool for continuous integration and continuous development. Secondly, since our Rule engine is configurable, more areas for security vulnerable can be enlisted by adding their rule set in the rule engine of Loky. Third, in addition to the console representation of the Auditing tool, we aim to create a user interface that generates an automated feedback for security analysis over each file. Automation and Artificial Intelligence is the future. A significant amount of research has been done on Static Analysing using Machine learning Algorithms to find vulnerabilities is an area that can be explored. Since the motive is to reduce false negatives in order to have a secure code, multiclass classification using boosters would be the best way to go. Optimizing the hyperparameters of the model, varying the method of pre-processing the input data and developing an architecture that brings together multiple models to predict different specific vulnerabilities was the most interesting. [7] Loky is an evolving solution for a burning problem. More robust implementation of Loky can be achieved by tracking code segments that are prone to attack.

ACKNOWLEDGEMENT

We thank Dr. Muhammad Ali Gulzar for his discussions and suggestions on early stages of this work. His untiring efforts and sincere guidance is commendable. Loky is a prototype deliverable of Course CS6704 . From contributions prospective, since we wanted to come up with a unique idea, Nidhi's contribution towards recommending to create Loky - Static Code Analyser was something out of the box. Prachi worked in collaboration on the idea to ensure a working prototype was created. We implemented the following components: 1. Frontend - React.js based Website, 2. Backend - Running on Apache Tomcat server 3. Database - Postgres 4. Output from which was aggregated on docker containers. The idea of creating the website was to ensure that we could deduce the problem statement to reproduce vulnerabilities for static code analysis. Based on this we created an Analyzer-Loky which could detect these vulnerabilities over the Frontend, Backend and Database so created. In addition, we would also like to thank our class students for giving us motivation and a platform to deliver our presentation.

REFERENCES

[1] Agile security. https://www.veracode.com/security/agile-security.

[2] Check style. https://checkstyle.org/index.html.

[3] Info security magazine. http://www.infosecurity-magazine.com/view/36214/target-breach-affecting-40-million-was-likely-an-inside-job.

[4] Owasp foundation. https://owasp.org/www-community/vulnerabilities. Accessed: 2010-09-30.

[5] Use of hard-coded credentials. https://cwe.mitre.org/data/definitions/798.html.

[6] Qirat Ashfaq, Rimsha Khan, and Sehrish Farooq. A comparative analysis of static code analysis tools that check java code adherence to java coding standards. In *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*, pages 98–103. IEEE, 2019.

[7] Tiago Baptista, Nuno Oliveira, and Pedro Rangel Henriques. Using machine learning for vulnerability detection and classification. In *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[8] Debasish Chakroborti and Sristy Sumana Nath. Web service performance enhancement for portable devices modifying soap security principle. In *2017 20th International Conference of Computer and Information Technology (ICCIT)*, pages 1–7. IEEE, 2017.

[9] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[10] C Ebert. Junit: Unit testing and coding in tandem. *IEEE Software*, 22:12–15, 2005.

[11] Jai Narayan Goel and BM Mehtre. Vulnerability assessment & penetration testing as a cyber defence technology. *Procedia Computer Science*, 57:710–715, 2015.

[12] Melina Kulenovic and Dzenana Donko. A survey of static code analysis methods for security vulnerabilities detection. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1381–1386. IEEE, 2014.

[13] Francesc Mateo Tudela, Juan-Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan-Antonio Sicilia Montalvo, and Michael I Argyros. On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications. *Applied Sciences*, 10(24):9119, 2020.

[14] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. Explaining static analysis-a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32. IEEE, 2019.

[15] Nicholas Saccente, Josh Dehlinger, Lin Deng, Suranjan Chakraborty, and Yin Xiong. Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 114–121. IEEE, 2019.

[16] Robert W. Shirey. Internet security glossary, version 2. rfc, 4949:1–365. doi:10.17487/RFC4949.