ANALYSIS OF JAVA'S COMMON VULNERABILITIES AND EXPOSURES IN GITHUB'S OPEN-SOURCE PROJECTS

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Semiu Ayobami Akanmu

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering

June 2022

Fargo, North Dakota

North Dakota State University
Graduate School

Title

ANALYSIS OF JAVA'S COMMON VULNERABILITIES AND EXPOSURES IN GITHUB'S OPEN-SOURCE PROJECTS

By

Semiu Ayobami Akanmu

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Muhammad Zubair Malik
Chair

Dr. Pratap Kotala

Dr. Supavich (Fone) Pengnate

Approved:

July 25, 2022
Date

Dr. Simone Ludwig
Department Chair

ABSTRACT

Java developers rely on code reusability because of its time and effort reduction advantage. However, they are exposed to vulnerabilities in publicly available open-source software (OSS) projects. This study employed a multi-stage research approach to investigate the extent to which open-source Java projects are secured. The research process includes text analysis of Java's Common Vulnerabilities and Exposures (CVE) descriptions and static code analysis using GitHub's *CodeQL*. This study found (a) cross-site scripting, (b) buffer overflow (though analyzed as array index out of bounds), (c) data deserialization, (d) input non-validation for an untrusted object, and (e) validation method bypass as the prevalent Java's vulnerabilities from the MITRE CVEs. The static code analysis of the compatible seven (7) Java projects out of the 100 top projects cloned from GitHub revealed a 71.4% presence of the array index out-of-bounds vulnerability.

ACKNOWLEDGMENTS

I, first and foremost, acknowledge the mercy and blessing of Allah (SWT) on me, especially during this graduate school program. I appreciate the graduate assistantship support from the Department of Computer Science, North Dakota State University. Without the department's tuition waiver assistance and the monthly stipend, I would not have completed this program. I also appreciate my thesis advisor, Dr. Muhammad Zubair Malik, and my thesis committee members, Dr. Pratap Kotala and Dr. Supavich (Fone) Pengnate.

My particular regards to my parents, who planted the seed of formal education in me and exposed me to the values and benefits of education. Lastly, I appreciate the support from my wife, Folasade, and my kids, Ayomiposi and Ayomikun. Your sacrifice finally paid off. I love you!

DEDICATION

This thesis is dedicated to all humans of all races, whose dreams kept them awake and whose wins are testimonies that “impossibility” does not have the right to exist at all times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SDLC ……... Software development lifecycle

OSS ……. Open-source software

JRE ……. Java runtime engine

CVE ……. Common vulnerabilities and exposures

CWE …… Common weakness enumeration

OOP ……. Object-oriented programming

JSON …… JavaScript object notation

XML ……. Extensible markup language

SOA ……. Service oriented architecture

SQL ……. Structured query language

CSV ……. Comma-separated values

NVD …… National vulnerabilities database

DoE …… Denial of entry

DoS …… Denial of service

XSS ……..... Cross-site scripting

HTML …… Hypertext markup Language

OWASP ……. Open web application security project

ORM ……. Object relational modelling

NER ……... Name entity recognizer

1. INTRODUCTION

Software security approaches emphasize the extension of security design and integration to requirement engineering, software architecture, and coding beyond the prevalent security testing [1]. These approaches, which explain the need to integrate security best practices to every stage of the software development lifecycle (SDLC), include, but are not limited to, risk analysis, abuse case modeling, and static code analysis [2]–[4]. Implementing these approaches is essential for all software projects. Still, open-source software (OSS) projects are direr because many OSS libraries that speed up the development process have been sources of known vulnerabilities [5].

Cybercriminals often exploit vulnerabilities in the software to perpetrate fraud, data and identity theft, and denial-of-service attacks [6]. The OSS projects, like others, are always alerted of vulnerabilities, and the communities are urged to work on fixing them. The Java OSS projects have also experienced damaging and costly attacks due to vulnerabilities exploited through a Java development platform or Java Runtime Engine (JRE) [7], [8]. The Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) have documented these experiences and their associated impacts [9].

Java developers rely on code reusability because of its time and effort reduction advantage and thus are exposed to vulnerabilities in these publicly available OSS projects. The advocacy for secure coding practices, such as input validation, output encoding, and session management [10], has been further stressed and expected to be adopted by all, including OSS project contributors. However, the extent to which this has been done is unknown. Also, the anticipated impact of secure coding among the OSS projects and Java is rarely investigated. The Java language's approach to security, including encapsulation and access control mechanism, has not stopped it from failing security testing in interesting ways [10]. Based on understanding these situations, this study

answered a central question: How secured are the codes written in the top GitHub's open-source Java projects?

In answering the central question, the following are the research questions answered:

i. What are Java's vulnerabilities prevalent in its MITRE CVE's vulnerability descriptions?

ii. What are the Java's vulnerabilities suggested in the commit messages of GitHub's open-source projects?

iii. How secured are Java's open-source projects in GitHub from Java's MITRE CVE vulnerabilities?

In the light of these research questions, the objectives achieved by this study are:

i. Identification of Java's vulnerabilities from the MITRE CVE's vulnerability descriptions.

ii. Identification of Java's vulnerabilities suggested by the projects' commits logs.

iii. Security Assessment of GitHub's Java projects code for the identified Java vulnerabilities.

GitHub is chosen as the public repository for fetching Java's OSS projects because it supports all popular programming languages and provides numerous development supports [12]. A multi-stage research approach that includes text analysis and static code analysis using GitHub's *CodeQL* [11] is employed by this study to answer the research questions and achieve their respective objectives. The remaining parts of this work are organized as follows: Chapter 2 discusses the background and motivation of the study. Examples of Java's vulnerabilities are discussed, and past studies on building security into the software construction phase of the SDLC are presented. Chapter 3 presents the research approach, enumerating the techniques and procedures, employed by this study. Chapter 4 presents the findings, and Chapter 5, as the concluding section, discusses the implications and limitations of the study.

2. BACKGROUND AND MOTIVATION FOR STUDY

This study stems from the need for securing software by starting from writing secure code. It emphasizes integrating security into the software construction stage of the SDLC. Software security has been primarily a post-development activity, with security and penetration testing, among others. Understanding the characteristics of the Common Vulnerabilities and Exposures (CVEs) and how they can be exploited is as essential in defensive security as it is for offensive security. The analysis of Java's CVEs in GitHub's open-source projects is an essential preliminary study into understanding the security status of the public code repositories. The security of these public code repositories would further suggest a compliance rate to secure coding practices. In this section, prominent Java CVEs are discussed, with due attention to the coding characterizations of the weaknesses. Also, the concept of "Building Security In," popularized by McGraw [1] as it relates to software construction security best practices, was discussed.

2.1. Java's common vulnerabilities and exposures

Vulnerabilities are generally weaknesses in the system design or code implementation. They can be exploited for attack manifestation. Some vulnerabilities are general to many programming languages, and some are specific to Java. The CVE program [9] identifies, defines, and catalogs publicly announced cybersecurity vulnerabilities. The initiative helps cybersecurity professionals globally to coordinate efforts addressing vulnerabilities. This study adopts Holzinger et al.'s [7] discussion of the Java CVEs. [7] specified unauthorized use of restricted classes, loading arbitrary classes, the unauthorized definition of privilege classes, reflective access to methods and fields, confused deputies, caller sensitivity, method handles, serialization and type confusion, and privileged code execution. These weaknesses are discussed in the following sub-sections.

2.1.1. Unauthorized use of restricted classes

Java platform is highly affected by unauthorized use of restricted classes. Custom classes defined with such an exploit can run arbitrary codes and disable security managers without further security checks. The object-oriented programming (OOP) paradigm's information hiding is often violated, thus exposing sensitive functionality to untrusted code. Preventing this experience might finally be done by the Java Module System [11], but there is still a considerable knowledge gap in this area.

The misconception of inner class restriction is prevalent among Java programmers, and the belief that enclosing classes can only access the inner class fields is incorrect for many Java compilers. The compilation is often done into independent classes with scopes extended throughout the package. Therefore, when compiled, the private fields of the outer Class changed to package scope and gave package scope access. Code listing in Figure 2.1 is an example of an inner class with access to the private variable of its enclosing class. In the code listing example in Figure 2.1, the `outer_value` variable would become accessible to all other classes in the same package, thus violating the intended scope restriction.

```
package testresearch;
public class outerclass {
      private String outer_variable = "private outer variable";
      class innerclass {
            void printprivate() {
            System.out.println("private field"+outer_variable);

} }

public static void main(String a[]){...} }

}
```

Figure 2.1. Inner class restricted class vulnerability.

2.1.2. Loading arbitrary classes

Java platform has dynamic class loading, which is a central security feature. By design, the class loaders ensure that all code load only classes it can access. However, malicious code can abuse a system class to invoke a caller-sensitive method, such as `Class.forName(String)`, as a confused deputy. The caller-sensitive method will use the immediate defining class loader to load the requested class. For example, the immediate caller of `forName` is a trusted system class; but an untrusted code can request the loading of arbitrary restricted classes. Figure 2.2 lists the simplified code illustration for arbitrary class loading.

```
1 // Method loads arbitrary classes
2 private Class GetClass1 (String s) {
3     JmxMBeanServer server =(JmxMBeanServer)JmxMBeanServer;
4     newMBeanServer ("",null ,null , true );
5     MBeanInstantiator i= server.getMBeanInstantiator ();
6     return i.findClass (s ,(ClassLoader) null );
7 }
```

Figure 2.2. Simplified code illustration for arbitrary class loading [7].

The `MBeanInstantiator`, in the listing in Figure 2.2 (line 3), is the trusted Class with vulnerability because it provides an unrestricted, public interface to load arbitrary classes. A special class loader that will not define a privileged context for a custom class can allow an untrusted code in cases of complex call sequences.

2.1.3. Unauthorized definition of privilege classes

Arbitrary code execution can be caused by defining a class, with all permissions, in a protected domain. Exploits can use restricted classes to define a custom class and thus requires an attack vector that abuses the vulnerability to gain access. Unauthorized access to

MethodHandles is called internal methods of class loaders, bypassing security checks implemented in the publicly accessible methods.

2.1.4. Reflective access to methods and fields

Malicious codes use improper reflection in system classes and caller-sensitive methods to bypass information hiding. This weakness, which is found in the sun.awt.SunToolkit, has been used to access private members of the Java class. There are also experiences of using confused deputies to invoke caller-sensitive methods, such as getDeclaredFields and getDeclaredMethods in Java.lang.Class. For example, as shown in the code listing in Figure 2.3, a private variable, say private_variable, can be called by a public method, say public_method (). It would increase the private variable's scope; therefore, its content, which might be confidential information, can be revealed.

```
1 public String public_method ( ) {
2
3           return private_variable;
4 }
```

Figure 2.3. Access violation vulnerability.

2.1.5. Confused deputies

Confused deputies are privilege escalators, legitimately tricked by another program into misusing its authority. They can be used to invoke the caller-sensitive methods, though it will not allow bypassing permission checks since its privileges are limited. The MethodHandle.invokeWithArguments can be used by untrusted code as a wrapper to MethodHandle.invokeExact, which will call the target method. Figure 2.4 lists the simplified code illustration of the confused deputy.

```
1 Class A {
2     public Object invoke (Method m, Object [] args ) {
3           return m.invoke (this, args);
4 }
5 // ...
6 }
```

Figure 2.4. Simplified code illustration of confused deputy.

2.1.6. Caller sensitivity

Caller-sensitive methods behave according to the trust level of their callers. They can skip permission checks when the immediate caller is seen to be trusted. Therefore, they are not primarily vulnerabilities but can be abused if called through a confused deputy. Exploits that use caller-sensitive methods can use `Class.forName` to load arbitrary classes and get reflective access to class members (fields, methods, and constructors) that should not ordinarily be accessed. However, Holzinger et al. [7] noted that empirical evaluation of the security check issues with caller-sensitive methods is required because callers are not explicitly aware of their privileges.

2.1.7. Method handles

`MethodHandles`, just as the reflection API, bypass information hiding. The lookup objects called by `MethodHandles.lookup` are facilitated by a confused deputy and used by malicious code in accessing system class members. The lookup object retrieved from the confused deputy grants such undue access when `MethodHandles` is used because it is less strict for type checking or an alternative to reflection API. Figure 2.5 illustrates how `MethodHandles` is exploited to access Class's members.

```
1 // Method loads arbitrary classes
2 private Class getClass2 (String s) {
3     MethodType mt = MethodType.methodType(Class.class, String.class);
4     MethodHandles.Lookupl = MethodHandles.publicLookup();
5     MethodHandle mh = l.findStatic(Class.class," forName ",mt);
6     return (Class)mh.invokeWithArguments (new Object []{s});
7 }
```

Figure 2.5. Exploiting `MethodHandles` for unauthorized access [7].

2.1.8. Serialization issues and type confusion

Serialization is the process of turning data objects into formats that can be saved to storage, sent as communication parts, or restored later. Therefore, data deserialization is reversing the serialization process. It involves rebuilding data back to objects from some formats. JavaScript Object Notation (JSON) and extensible Markup Language (XML) are the most popular data formats for serializing data. Native deserialization mechanisms provided by many programming languages, Java inclusive, can be repurposed for attack when an untrusted data object is involved. The Java's `ObjectInputStream#resolveClass()` method can be exploited for arbitrary classes' deserialization. Figure 2.6 shows a Java deserialization routine.

```
public class ValueObject implements Serializable {
   private String value;
   private String sideEffect;
   public ValueObject() {
       this("empty");
   }
   public ValueObject(String value) {
       this.value = value;
       this.sideEffect = java.time.LocalTime.now().toString();
   }
}
ValueObject vo1 = new ValueObject("Hi");
FileOutputStream fileOut = new FileOutputStream("ValueObject.ser");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(vo1);
out.close();
fileOut.close();
```

Figure 2.6. Java deserialization routine [12].

The data deserialization-related vulnerabilities leverage class loading and type safety security features in Java for attacks [7]. Figure 2.6 depicts the serialization of an object from a serializable class (ValueObject). The object's value, `Hi`, can be changed during deserialization without calling the constructor. An invalid object can, therefore, be created. The attack can be perpetrated by manually creating a serialized object inserted into the `AtomicReferenceArray`.

2.1.9. Privileged code execution

Privilege code execution is a type of vulnerability that allows attackers to execute code in a way that successfully bypasses arbitrary permission checks. They are different from and more radically powerful than confused deputies because they do not rely on caller sensitivity. Exploits

achieve privileged code execution through abuse system classes and trusted method chaining. In abuse system classes, the privileges are elevated, and attack methods are called with arbitrary arguments. However, malicious code creates a thread that executes the attacker-provided method in trusted method chaining. The code listing in Figure 2.7 shows how a system property of the privilege `System` class can gain unintended information, precisely the name value.

```
public static String getProp(final String name)
{
      return (String) AccessController.doPrivileged(new PrivilegedAction()
            {
            public Object run()
            {
// privileged code goes here, for example:
            return System.getProperty(name);
            }
      });
}
```

Figure 2.7. Privilege access as-a-result of abuse of `System` classes.

2.2. Building security in software construction

The need for software security approaches that are system development focused and beyond the operational and application-level security is dire. Against this backdrop, general software security best practices have been suggested for every stage of the SDLC. These approaches are needed as extensions of security design and integration to requirement engineering, software architecture, coding, and testing. Specifically, for the software construction phase, static analysis tools, program analysis, obfuscation and masking, verification and model checking, knowledge graph modeling, and machine and deep learning techniques are recorded as adapted strategies for building security into coding.

Static code analysis tools – which are either proprietary [6], [13] or open-source [14]–[16] – search the code or analyze compiled versions to identify vulnerabilities, among others. The tools provide immediate feedback to developers, help in the speedy build process, and fasten the release

period. Unfortunately, they may give false positives and miss specific security issues, such as authentication. They are primarily unsuitable for codes that cannot be compiled and are generally language-specific [17]. In the same vein, Taint analysis, one of the evaluation analyses of Collective Program Analysis proposed by Upadhyaya and Rajan [18], detects and reports vulnerabilities in the program source code. The analysis checks if data from external inputs like consoles are read to the outputs and reports associated vulnerabilities. In a similar study, taint analysis is used in detecting malicious input in embedded systems [19]. It helps track tainted data from its source to its application point in the source code.

Formal software verifications and model checking methods have also been used to prove code correctness and assess compliance with specified security constraints, such as no memory, type safety violations, and logging sensitive information [3], [20]. However, its adoption and wide acceptability have been limited due to its inability to scale. Obfuscation and masking techniques were also invented to enhance software security [21], [22]. These include selecting a subset of code to obfuscate or transform to the desired level that resists reverse engineering and removes potential security vulnerabilities. The implementations involve a finite-state machine (FSM), which decomposes programs for simple predicate extraction [21], and code transformation, which uses opaque constructs from aliasing and concurrency [22].

With applicability in automated detection of vulnerabilities in source code, a knowledge graph of vulnerabilities data was constructed by Jia et al. [23]. The study used the Stanford Named Entity Recognizer (NER) as a machine learning training model for an extractor of cybersecurity-related entities. Their work, *useGazette* parameter, though focusing on broader cybersecurity entities, is helpful in training recognizers in automated detection of vulnerabilities. A similar anticipated study for detecting vulnerabilities is [24] on classifying service-oriented architecture

(SOA) vulnerabilities. The study proposed a comprehensive classification to identify the systems' vulnerabilities, including building vulnerability management tools for software code, due to existing additional SOA vulnerabilities.

Notably, there are significant new studies on detecting program source code vulnerabilities using machine and deep learning techniques. These range from vulnerabilities detection in open source dependencies [5], [25], to programming languages like C [26]–[29], C++[26], [27], and Java [30]. Building security in software is the major motivation of these studies, but there are other important specific problems. For instance, open-source software (OSS) libraries, widely used to speed up the development process, have been sources of publicly known vulnerabilities [5]. The static scanning tools have also been inadequate for vulnerability detection in complex but low-level languages like C [28]. There is also an invention for just-in-time vulnerability detection in source code [30]. Also, a proposed minimum intermediate representation learning technique reduces the false-positive rate [29].

GitHub's *CodeQL* [31] and Facebook's *PySa* [32] are currently open-source projects for vulnerability detection in source code. *CodeQL* is a semantic code analysis engine that allows querying code as data to detect variants of a vulnerability. Using taint analysis, *CodeQL* can be used on codebases to discover bad patterns [31]. A similar, but a specific tool for Python language, is *PySa* – an acronym for Python Static Analyzer. It detects and prevents security and privacy issues in Python code. It is built on a type checker to analyze data flows through the code to identify web application security issues, including cross-site scripting and SQL injection [32]. Table 2.1 summarizes the "build security in" techniques for the software construction phase.

Table 2.1. Build-Security-In techniques for the software construction phase.

Techniques	Strengths	Limitation
Static code analysis tool [13], [15], [32]	Suitable for flagging vulnerabilities in source code or after being compiled. It can be integrated into IDE to provide immediate feedback.	Give a high number of false positives. Miss specific security issues, such as authentication.
Program analysis [18], [19]	Optimal in detecting malicious input or flow within the source code.	Limited in scope, therefore, mostly need supporting techniques for optimal performance.
Obfuscation and masking [21], [22]	Suitable for concurrency security control. Applicable in state-dependent code.	It is unable to scale.
Verification and model checking [3], [20]	Suitable for actualizing complete and sound techniques.	It is unable to scale.
Knowledge graph (ontology) modeling [23], [24]	It is suitable for rule-based verification and formalization and can easily integrate into other technologies.	Requires supporting techniques like program analysis for abstract syntax tree construction for optimal performances.
Machine and Deep learnings [5], [26], [28]	It handles multivariate data optimally. Suitable for pattern recognition on all types of datasets.	It has a high rate of false positives. It requires intensive data for optimal performance.

3. RESEARCH APPROACH

A multi-step research approach is adopted in proffering answers to the research questions of this study. Text and static code analysis are the main components of this research process. Considering the leverage provided by computational methods and tools, text analysis helps extract information from documents. It is also used in identifying and exploring interesting patterns from unstructured textual data [30]. Text analysis's use cases include, but are not limited to, text categorization, text clustering, entity extraction, production of taxonomies, sentiment analysis, and entity relation modeling [31]. The text analysis techniques are used to identify the prevalent Java vulnerabilities published by the MITRE CVEs.

Static code analysis tools have been reported for their extensive use and merits in providing immediate feedback to developers, helping in the speedy build process, and fastening the release period [10]– [14]. They are used for finding bugs or security vulnerabilities in the code by scanning and not executing the code. Figure 3.1 presents the research design process. Sub-sections 3.1 and3.2 provide the details of the steps involved in the components. Also, the code implementation of these components is presented[1].

[1] https://github.com/Semiu/java-codesecurity/tree/main/java-cve-analysis

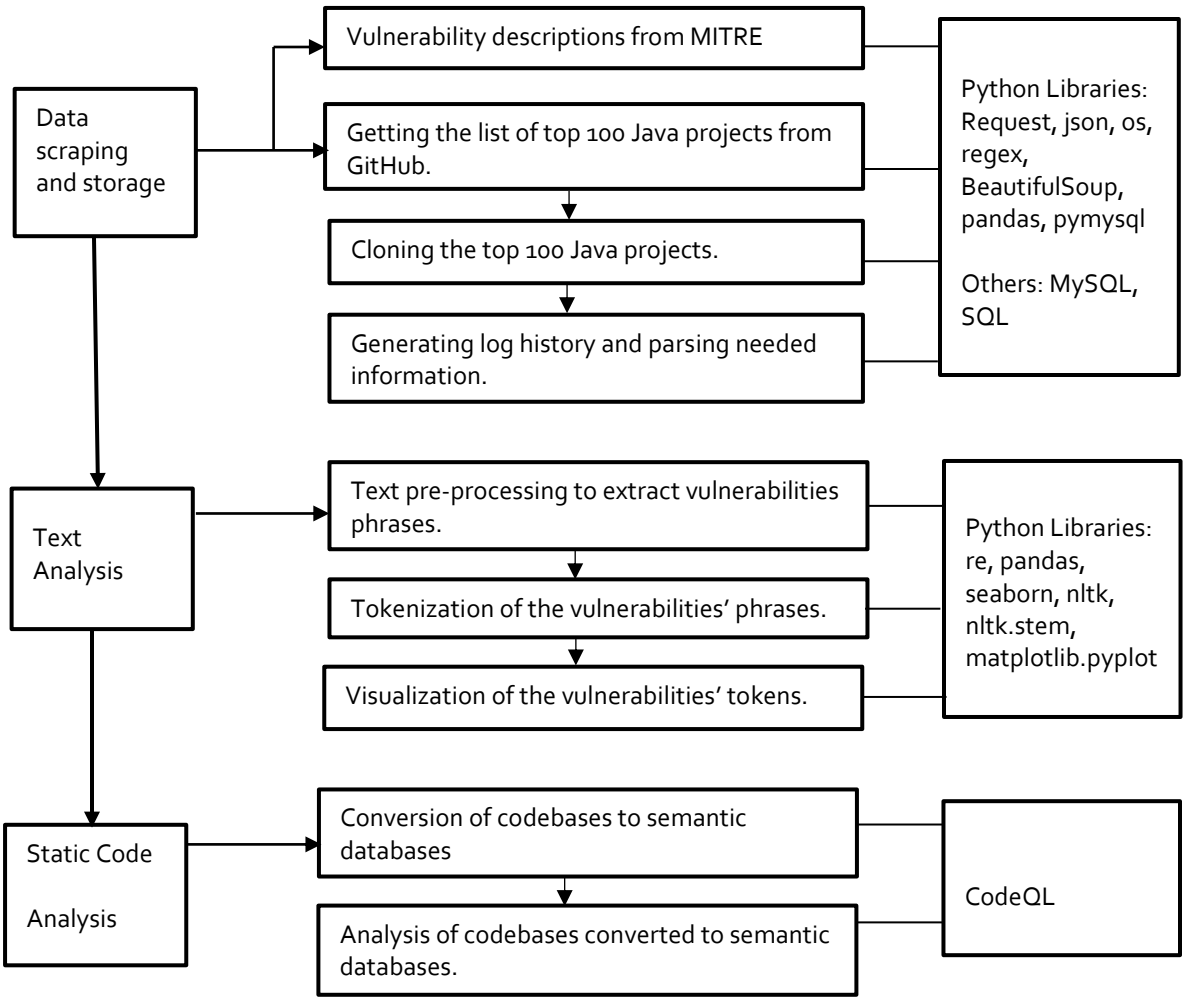

Figure 3.1. Research approach.

3.1. Text analysis to identify prominent Java CVE in MITRE and vulnerabilities-related themes in the commit's logs of the Java's GitHub open-source projects

Data curation and text analysis are the two steps in this phase of the research approach. The first and second research questions, answered by this phase, identified Java's CVEs in the MITRE's vulnerability descriptions and commit messages of GitHub's Java open-source projects. The Java's CVE descriptions are extracted from the MITRE website, cleaned, and saved in comma-separated values (CSV) files, using different web scraping techniques. Web scraping is

extracting text from web pages [33]. Python libraries, such as `BeautifulSoup` and `Request`, implement web scraping scripts. The list of the top 100 GitHub Java open-source project names is created using the number of stars, and each of these projects is cloned to a local machine. Also, commit logs are generated for each of the open-source projects, and they are parsed and saved in a CSV file, using their respective hash numbers (sha) as identifiers.

The extracted MITRE's vulnerability descriptions and Java's open-source projects git commit messages are pre-processed for vulnerability phrases. The pre-processing allows clarity and specifics of words that represent the important contents. These phrases are then tokenized using `NLTK` and `Wordnet Lemmatizer`: Python libraries for removing stop words, stemming, and Lemmatization. Stop words are littered words and mostly have no significance to the generality of the analyzed body of text. Examples are "the," "and," "at." Stemming is the process of reducing words to their base form, while Lemmatization groups different forms of words into single items for analysis [33]. The vulnerability-representative tokens derived are visualized to identify the prevalent ones. These tokens suggest the prevalent Java's CVE vulnerabilities published on MITRE's website.

3.2. Analysis of GitHub's Java projects using *CodeQL* to identify security vulnerabilities present

The third research question evaluates the security of Java open-source projects on GitHub using the static code analysis method. The static code analysis uses *CodeQL* [31] to identify the security vulnerabilities. *CodeQL* is a semantic code analysis engine that allows querying code as data to detect variants of a vulnerability. The extracted MITRE's vulnerability descriptions for Java suggest the vulnerabilities analyzed. The codebases of the Java open-source projects are firstly converted to semantic databases for working compatibility with *CodeQL*. Queries are then

written to detect the suggested vulnerabilities, and the analysis results are presented. This study adopts applicable code queries from the *CodeQL's* documentation[2].

[2] https://codeql.github.com/codeql-query-help/java/

4. FINDINGS

4.1. Identification of prominent Java CVEs in MITRE CVEs

Identifying the prominent Java CVEs from the NVD was based on identifying the prevalent tokens in the vulnerability's texts reported in MITRE. Figure 4.1 presents the top twenty (20) prominent tokens.

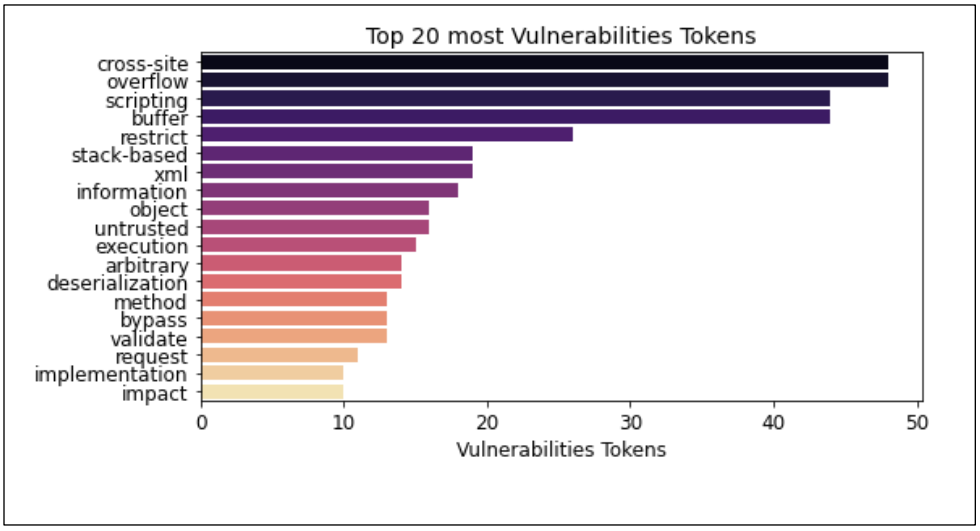

Figure 4.1. Top twenty (20) prominent tokens from the text analysis of Java's CVEs in NVD.

The top 20 prominent tokens from the text analysis of Java's CVEs in MITRE suggest the following web vulnerabilities, using domain knowledge as heuristics. These are (a) cross-site scripting, (b) buffer overflow, (c) data deserialization, (d) input non-validation for an untrusted object, and (e) validation method bypass. These vulnerabilities were investigated where appropriate, and the findings are reported in sub-section 4.3. The explanation for the presence of buffer overflow as a vulnerability reported by the MITRE, despite the Java's defence against it is explained in Chapter 5, section 5.1. Similarly, identifying the security vulnerabilities-related themes in the commit's logs of GitHub Java's open-source projects followed the same process.

The details of the text analysis are also presented in sub-section 3.2. Figure 4.1 presents the top twenty (20) prominent tokens from the text analysis of Java's open-source projects' git commits.

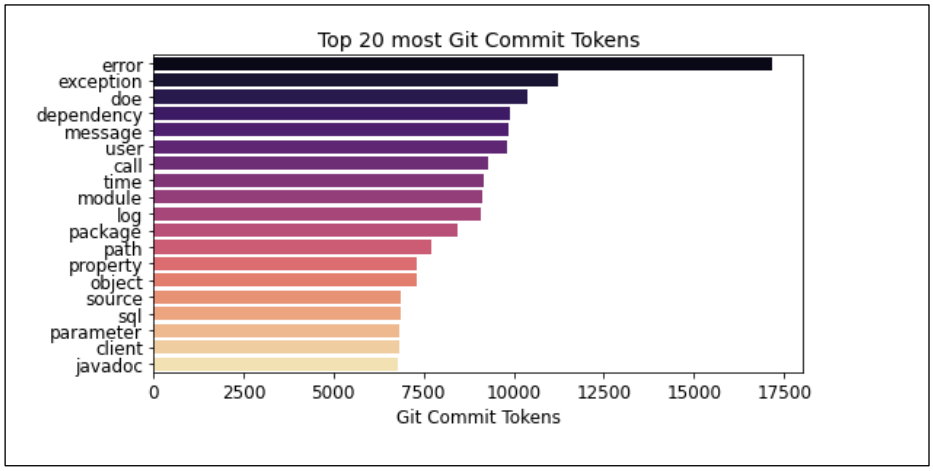

Figure 4.2. Top twenty (20) prominent tokens from the text analysis of Java's open-source projects' git commits.

The git commit messages are not correctly worded to ease text analysis in identifying the vulnerability themes and tokens. Nevertheless, SQL injection and Denial of entry (DoE), also understood as Denial of Service (DoS), are the vulnerabilities arguably suggested by the prominent tokens identified.

4.2. Identification of security vulnerabilities in GitHub Java's open-source projects

The process of identifying the security vulnerabilities in GitHub's Java open-source projects is multi-stage, as described in Chapter 3. Findings reported in this section include (a) the list of the 100 open-source projects cloned for analysis, indicating the build message from the codebase-database conversion process, (b) the summary of the successful and unsuccessful conversions, (c) the breakdown of the build messages of the unsuccessful conversions, (d) the descriptions of the successfully-converted codebases that were later analyzed for the presence of

the specified vulnerabilities, and (e) the results of the analysis of the specified vulnerabilities in the codebases. Table 4.1 lists the 100 Java open-source projects from GitHub.

Table 4.1. List of the first 100 (based on star counts) Java's open-source projects on GitHub.

	Project Name	Star count	Build message
1	CyC2018/CS-Notes	120756	No suitable build command
2	Snailclimb/JavaGuide	96870	No suitable build command
3	iluwatar/java-design-patterns	63975	mvn.cmd not recognized
4	MisterBooo/LeetCodeAnimation	62528	No suitable build command
5	elastic/elasticsearch	53668	SocketException
6	spring-projects/spring-boot	53458	FileNotFound Exception
7	doocs/advanced-java	51923	No suitable build command
8	kdn251/interviews	50095	could not detect suitable build
9	macrozheng/mall	46312	mvn.cmd not recognized
10	ReactiveX/RxJava	44239	Successful
11	spring-projects/spring-framework	41432	Successful
12	google/guava	40302	mvn.cmd not recognized
13	square/okhttp	39372	Successful
14	square/retrofit	37517	SDK location not found
15	TheAlgorithms/Java	35025	Could not find the build command
16	apache/dubbo	34657	MAVEN/DependencyResolutionException
17	PhilJay/MPAndroidChart	32765	NoClassDefFoundError
18	bumptech/glide	30698	NoClassDefFoundError
19	airbnb/lottie-android	30633	Android SDK location not found; software internal component missing
20	kon9chunkit/GitHub-Chinese-Top-Charts	30122	No build detected
21	Blankj/AndroidUtilCode	28848	NoClassDefFoundError
22	zxing/zxing	27166	mvn.cmd not recognized
23	netty/netty	25976	CompilationError problem
24	crossoverJie/JCSprout	25779	mvn.cmd not recognized
25	JakeWharton/butterknife	25615	SDK location not found
26	proxyee-down-org/proxyee-down	25526	mvn.cmd not recognized
27	skylot/jadx	25108	Successful
28	ityouknow/spring-boot-examples	24798	mvn.cmd not recognized
29	eugenp/tutorials	24780	mvn.cmd not recognized
30	NationalSecurityAgency/ghidra	24728	Gradle not found
31	alibaba/arthas	24709	MojoFailureException in the build
32	geekxh/hello-algorithm	24079	No build detected

Table 4.1. List of the first 100 (based on star counts) Java's open-source projects on GitHub (continued).

	Project Name	Star count	Build message
33	ctripcorp/apollo	23805	mvn.cmd not recognized
34	alibaba/druid	23339	mvn.cmd not recognized
35	greenrobot/EventBus	23178	NoClassDefFound error
36	alibaba/fastjson	23035	mvn.cmd not recognized
37	scwang90/SmartRefreshLayout	22186	NoClassDef error
38	CymChad/BaseRecyclerViewAdapterHelper	21490	SDK location not found
39	Netflix/Hystrix	21075	IllegalArgument Exception
40	xkcoding/spring-boot-demo	20523	mvn.cmd not recognized
41	lenve/vhr	20236	mvn.cmd not recognized
42	SeleniumHQ/selenium	19804	No suitable build
43	signalapp/Signal-Android	19798	No SDK location
44	hollischuang/toBeTopJavaer	19604	No suitable build
45	ReactiveX/RxAndroid	19328	SDK location not found
46	google/gson	19178	IllegalArgument Exception
47	qiurunze123/miaosha	19073	XMLpullException
48	zhangdaiscott/jeecg-boot	18903	mvn.cmd not recognized
49	alibaba/easyexcel	18851	mvn.cmd not recognized
50	seata/seata	18679	SocketException
51	dbeaver/dbeaver	18594	mvn.cmd not recognized
52	wuyouzhuguli/SpringAll	18490	Cannot detect build command
53	libgdx/libgdx	18029	Taskexecution exception
54	apache/kafka	18017	Gradle not recognized
55	halo-dev/halo	18006	Successful
56	looly/hutool	17884	mvn.cmd not recognized
57	square/picasso	17854	NoClassDef error
58	alibaba/canal	17787	mvn.cmd not recognized
59	alibaba/spring-cloud-alibaba	17589	pom.xml file does not exist
60	Baseflow/PhotoView	17495	Missing SoftwareInternalComponent
61	xuxueli/xxl-job	17274	mvn.cmd not recognized
62	google/ExoPlayer	17255	NoClassDef error
63	jenkinsci/jenkins	16902	mvn.cmd not recognized
64	nostra13/Android-Universal-Image-Loader	16782	NoClassDef error
65	didi/DoraemonKit	16741	No suitable build command
66	facebook/fresco	16490	SDK location not found
67	alibaba/nacos	16336	mvn.cmd not recognized
68	bazelbuild/bazel	16212	Could not find a suitable build

Table 4.1. List of the first 100 (based on star counts) Java's open-source projects on GitHub (continued).

	Project Name	Star count	Build message
69	apache/skywalking	16077	Could not find a suitable build
70	shuzheng/zheng	15847	mvn.cmd not recognized
71	CarGuo/GSYVideoPlayer	15720	SDK location not found
72	redisson/redisson	15695	mvn.cmd not recognized
73	Tencent/tinker	15634	Could not determine Java version
74	apache/flink	15503	mvn.cmd not recognized
75	alibaba/Sentinel	15325	mvn.cmd not recognized
76	linlinjava/litemall	15318	mvn.cmd not recognized
77	mybatis/mybatis-3	15048	mvn.cmd not recognized
78	dianping/cat	14994	mvn.cmd not recognized
79	forezp/SpringCloudLearning	14956	Cannot detect build directory
80	android10/Android-CleanArchitecture	14708	could determine Java version from 15
81	brettwooldridge/HikariCP	14568	mvn.cmd not recognized
82	oracle/graal	14537	No build command
83	winterbe/java8-tutorial	14376	No build command
84	elunez/eladmin	14241	mvn.cmd not recognized
85	EnterpriseQualityCoding/FizzBuzzEnterpriseEdition	14080	IllegalArgument Exception
86	openzipkin/zipkin	14006	Successful
87	JeffLi1993/springboot-learning-example	13971	mvn.cmd not recognized
88	lottie-react-native/lottie-react-native	13847	NoClassDefFound error
89	hdodenhof/CircleImageView	13746	NoClassDefFound error
90	apache/rocketmq	13516	mvn.cmd not recognized
91	lgvalle/Material-Animations	13510	Gradle not recognized
92	LMAX-Exchange/disruptor	13436	Successful
93	apache/shardingsphere	13349	mvn.cmd not recognized
94	alibaba/ARouter	12858	local.properties file is missing
95	dyc87112/SpringBoot-Learning	12762	mvn.cmd not recognized
96	orhanobut/logger	12676	could determine Java version from 15
97	Tencent/QMUI_Android	12652	SDK location not found
98	TeamNewPipe/NewPipe	12538	SDK location not found
99	Bigkoo/Android-PickerView	12531	NoClassDefFound error
100	Curzibn/Luban	12326	IllegalArgumentException

For various reasons, only seven (7) codebases were successfully converted to databases that could be analyzed for vulnerabilities using *CodeQL*. The reasons for the unsuccessful conversion, as shown in the build messages, are presented in Table 4.3. Table 4.2 presents the summary of the codebase-to-database conversion success rate. The conversion success rate is also graphically represented in Figure 4.3.

Table 4.2. Summary of the codebase-to-database conversion for the vulnerability analysis.

Codebase-to-Database state	Quantity
Successful	7
Unsuccessful	93
Total	100

Figure 4.3 depicts a graphical representation of the summary of the codebase-to-database conversion of the 100 Java open-source projects from GitHub.

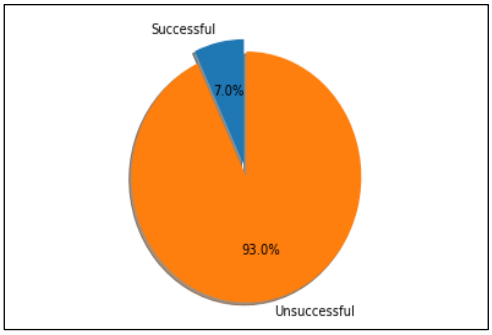

Figure 4.3. Pie-chart – graphical representation of codebase-to-database conversion summary.

Table 4.3. Breakdown of the causes of the unsuccessful codebase-to-database conversions.

	Causes	Frequency
1	Illegal Argument Exception	4
2	No Class Definition Exception	11
3	Android SDK location not found	10
4	mvn.cmd not recognized	35
5	Java version could not be determined	3
6	Gradle not recognized	4
7	local.properties file missing	1
8	No build command in the source	4
9	Missing software internal component	1
10	pom.xml does not exist	1
11	Task Execution Exception	1
12	Socket Exception Error	2
13	XML pull Exception	1
14	Mojo Failure Execution in the build	1
15	File not found exception	1
16	MAVEN Dependency resolution Exception	1
17	Compilation Error	1

Figure 4.4 depicts a graphical representation of the breakdown of the causes of the unsuccessful codebase-to-database conversions.

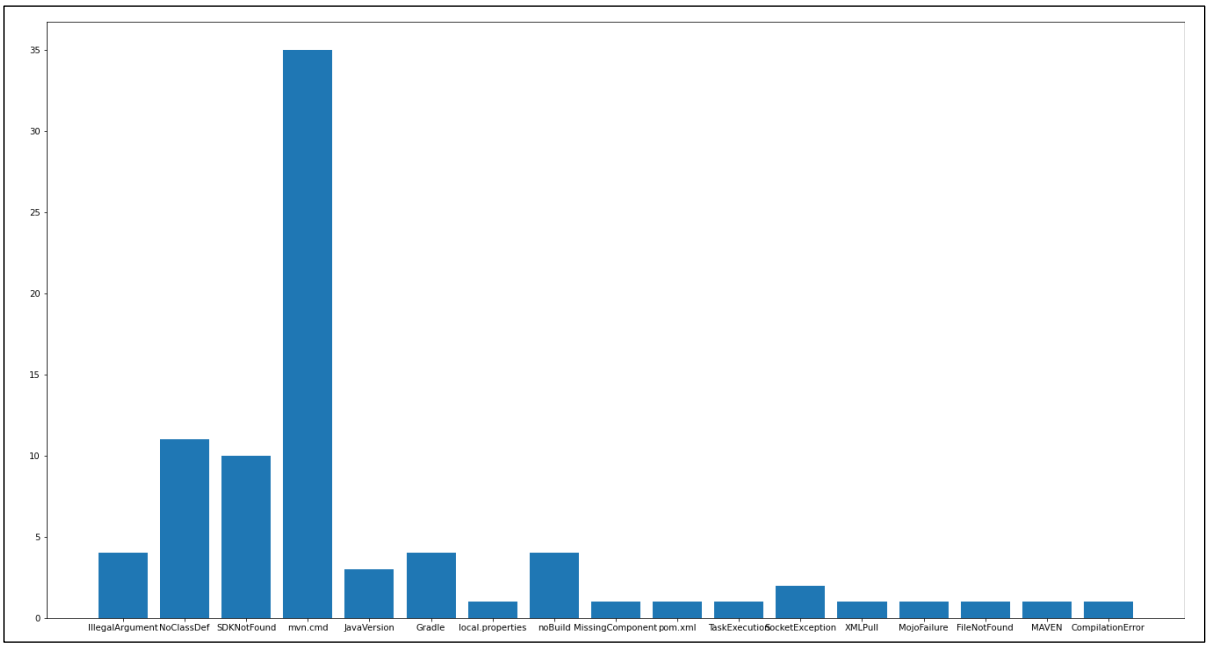

Figure 4.4. Bar chart – graphical representation of codebase-to-database conversion summary.

The successfully converted codebases are seven (7): RxJava, Spring framework, OkHTTP, Jadx, Halo, Zipkin, and Exchange Disruptor. Table 4.4 presents the detailed information of the codebases, including their respective descriptions and uniform resource locators (URLs).

Table 4.4. List of the successful codebases from the codebase-to-database conversion process.

	Name of project	GitHub Star count	Description	Uniform Resource Locator	Lines of Code (LOC)
1	ReactiveX/RxJava	44239	It is a library for composing asynchronous and event-based programs.	https://github.com/ReactiveX/RxJava	323206
2	spring-projects/spring-framework	41432	The home of the Spring framework. Spring provides everything required for creating enterprise applications.	https://github.com/spring-projects/spring-framework	805165
3	square/okhttp	39372	This HTTP client supports all requests to the same host when sharing a socket, reducing the request latency, among others.	https://github.com/square/okhttp	92178
4	skylot/jadx	25108	This command line and Graphical User Interface (GUI) tools produce Java source code from Android Dex and Apk files.	https://github.com/skylot/jadx	125739
5	halo-dev/halo	18006	A modern personal and independent blogging system.	https://github.com/halo-dev/halo	40967
6	openzipkin/zipkin	14006	A distributed tracing system used for gathering timing data which are needed for solving latency problems in service architecture.	https://github.com/openzipkin/zipkin	113750
7	LMAX-Exchange/disruptor	13436	A high-performance inter-thread messaging library.	https://github.com/LMAX-Exchange/disruptor	19925

The successfully converted codebases are analyzed with *CodeQL* and the provided code snippets to query the vulnerabilities identified in sub-section 4.1. Table 4.5 presents the findings of the analysis. Notably, array index out of bounds, as a type of buffer overflow, was analyzed because of the presumption that Java's technology has been built to avoid the occurrence of buffer overflow. Further discussions are provided in section 5.

Table 4.5. Security evaluation of GitHub open-source projects.

Vulnerability	Database	Result	Details (where necessary)

Array index out of bound	ReactiveX/RxJava	2	at TestHelper.java file, line 2454, 72
			code below shows lines 2453 to 2455
			`for (int i = 0; i < classes.length; i += 2) {`
			`  assertError(list, i, (Class<Throwable>)classes[i], (String)classes[i + 1]);`
			`  }`
			at TestHelper.java file, line 2499, 72
			code below shows lines 2498 to 2500
			`for (int i = 0; i < classes.length; i += 2) {`
			`  assertError(list, i, (Class<Throwable>)classes[i], (String)classes[i + 1]);`
			`  }`

	spring-projects/spring-framework	3	at Frame.java file, line 653, 35
			code below shows 652 to 654
			`if (kind == STACK_KIND) {`
			`  initializedType = dim + inputStack[inputStack.length - value];`
			`  }`
			at PathPatternTests.java file, line 1182, 40
			code below shows line 1181 to 1183
			`for (int i = 0; i < keyValues.length; i += 2) {`
			`  expectedKeyValues.put(keyValues[i], keyValues[i + 1]);`
			`  }`
			at ViewResolverRegistryTests.java file, line 219, 22
			code below shows lines 218 to 220
			`for (int i = 0; i < nameValuePairs.length ; i++, i++) {`
			`  Object expected = nameValuePairs[i + 1];`
			`{`

	square/okhttp	1	at CallTest.java file, line 3955, 37
			code below shows lines 3954 to 3956
			`for (int i = 0, size = headers.length; i < size; i += 2) {`
			`  builder.addHeader(headers[i], headers[i + 1]);`
			`  }`

	skylot/jadx	2	at SignatureParserTest.java file, line 111, 41 `List<ArgType> list = (List<ArgType>) objs[i + 1];`

Table 4.5. Security evaluation of GitHub open-source projects (continued).

Vulnerability	Database	Result	Details (where necessary)

Array index out of bound			at TestArrayforEachNegative.java file, line 28, 12
			code below shows lines 27 to 29
			`for (int i = 0; i <= a.length; i++) {`
			`sum += a[i];}`

	halo-dev/halo	0	Not applicable

	openzipkin/zipkin	2	at Endpoint.java file, line 379,27
			code below shows lines 378 to 382
			`for (int i = 0; i < ipv6.length; i += 2) {`
			`if (ipv6[i] == 0 && ipv6[i + 1] == 0) {`
			`if (zeroIndex < 0) zeroIndex = i;`
			`continue;`
			`}`
			at Endpoint.java file, line 414,18
			`byte low = ipv6[i++];`

	LMAX-Exchange/disruptor	0	Not applicable
Method Bypass	ReactiveX/RxJava	0	Not applicable
	spring-projects/spring-framework	0	Not applicable
	square/okhttp	0	Not applicable
	skylot/jadx	0	Not applicable
	halo-dev/halo	0	Not applicable
	openzipkin/zipkin	0	Not applicable
	LMAX-Exchange/disruptor	0	Not applicable
Cross-site scripting (due to user-provided value)	ReactiveX/RxJava	0	Not applicable

	spring-projects/spring-framework	51	(Note: Few selected details are presented)
			The user-provided value
			`out.write(in);`
			at ServletWebRequest.java file, line 372, 28
			The code below shows from line 371 to 373
			`HttpServletRequest request = getRequest();`
			`StringBuilder sb = new StringBuilder();`
			`sb.append("uri=").append(request.getRequestURI());`
			at line 382, 18
			`String user = request.getRemoteUser();`

Table 4.5. Security evaluation of GitHub open-source projects (continued).

Vulnerability	Database	Result	Details (where necessary)

Cross-site scripting (due to user-provided value)			at ForwardedHeaderFilter.java, line 394, 12
			`if (this.requestUri == null) { return this.delegate.get().getRequestURI(); }`
			at MultipartFileResource.java, line 85, 39
			the code shows lines 84 to 86
			`public String getDescription() { return "MultipartFile resource [" + this.multipartFile.getName() + "]"; }`
			at UrlPathHelper.java file, line 435, 10
			code below shows lines 434 to 436
			`if (uri == null) { uri = request.getRequestURI(); }`
			at UrlResource.java file, line 186, 11
			code below shows line 185 to 187
			`try { return con.getInputStream(); }`
			at httpComponentAsynClientHttpResponse.java file, line 81, 28
			`HttpEntity entity = this.httpResponse.getEntity(); return (entity != null ? entity.getContent() : StreamUtils.emptyInput());`
			at SimpleServerhttpRequest.java, line 98, 62
			`this.responseStream = (errorStream != null ? errorStream : this.connection.getInputStream());`
			at MultipartFile.java, line 149, 22
			`default void transferTo(Path dest) throws IOException, IllegalStateException { FileCopyUtils.copy(getInputStream(), Files.newOutputStream(dest)); }`
			at MultipartFileResource.java file, line 77, 10
			`return this.multipartFile.getInputStream();`
			at DefaultMultipartHttpServletRequest.java, line 85, 21
			`String[] values = getMultipartParameters().get(name);`
			at RequestPartServletServerHttpRequest.java, line 100, 23
			`String paramValue = this.multipartRequest.getParameter(this.requestPartName);`

Table 4.5. Security evaluation of GitHub open-source projects (continued).

Vulnerability	Database	Result	Details (where necessary)
			at UrlResouce.java file, 186, 11 `try { return con.getInputStream(); }`
	square/okhttp	0	Not applicable
	skylot/jadx	0	Not applicable
	halo-dev/halo	0	Not applicable
	openzipkin/zipkin	0	Not applicable
	LMAX-Exchange/disruptor	0	Not applicable
Deserialization	ReactiveX/RxJava	0	Not applicable
	spring-projects/spring-framework	1	at HttpInvokerServiceExporter.java, line 146, 16 Code below shows lines 95 to 99 `protected RemoteInvocation readRemoteInvocation(HttpServletRequest request) throws IOException, ClassNotFoundException { return readRemoteInvocation(request, request.getInputStream()); }`
	square/okhttp	0	Not applicable
	skylot/jadx	0	Not applicable
	halo-dev/halo	0	Not applicable
	openzipkin/zipkin	0	Not applicable
	LMAX-Exchange/disruptor	0	Not applicable
Improper validation of user-provided array index	ReactiveX/RxJava	0	Not applicable
	spring-projects/spring-framework	81	(Note: Few selected details are presented) at Frame.java, line 486, `@Override public InputStream getInputStream() throws IOException, IllegalStateException { return this.multipartFile.getInputStream(); }`

Table 4.5. Security evaluation of GitHub open-source projects (continued).

Vulnerability	Database	Result	Details (where necessary)

			at MultipartFileResource.java, line 77
			`if (multipartRequest != null) {`
			`List<MultipartFile> files = multipartRequest.getFiles(name);`
			`…`
			`{`
			at UrlResource.java, line 186
			Code below shows line 185 to 187
			`try {`
			`return con.getInputStream();`
			`}`
			at ServletWebRequest.java, line 372
			Code below shows lines 371 to 372
			`StringBuilder sb = new StringBuilder();`
			`sb.append("uri=").append(request.getRequestURI());`
			at ForwardedHeaderFilter.java, line 185
			Code below shows line 184 to 186
			`if (!FORWARDED_HEADER_NAMES.contains(name)) {`
			`headers.put(name, Collections.list(request.getHeaders(name)));}`

	square/okhttp	0	Not applicable
	skylot/jadx	0	Not applicable
	halo-dev/halo	0	Not applicable
	openzipkin/zipkin	0	Not applicable
	LMAX-Exchange/disruptor	0	Not applicable
Improper validation of user-provided size used for array construction	ReactiveX/RxJava	0	Not applicable
	spring-projects/spring-framework	0	Not applicable
	square/okhttp	0	Not applicable
	skylot/jadx	0	Not applicable
	halo-dev/halo		
	openzipkin/zipkin	0	Not applicable
	LMAX-Exchange/disruptor	0	Not applicable

5. DISCUSSION AND CONCLUSION

5.1. Discussion

5.1.1. Java's vulnerabilities identified from the MITRE CVE's vulnerability descriptions

Cross-site scripting (XSS), buffer overflow, data deserialization, input non-validation for an untrusted object, and validation method bypass are the prevalent Java vulnerabilities suggested by the text analysis of MITRE CVEs. XSS allows attackers to inject malicious code into the web browser, such as JavaScript programs. Buffer overflow attacks are specified by overwriting process memory segments [34]. Data deserialization is rebuilding data back to objects from formats like JSON and XML. It can, therefore, be used for attack when the data object is untrusted [7]. The input non-validation for untrusted data can happen in different instances, including deserialization. All data from untrusted sources, including user-facing sites and backend feeds, should be subject to input validation [35]. The validation method bypass, in most cases, happens when a malicious code poses as a trusted object and therefore enjoys the privilege and unmerited execution [36].

XSS is also caused by user input from the HTML output that escapes validation. It resembles the validation method bypass principle. But in specifics, XSS injects scripts for damage, whether persistent or reflected. XSS can be exploited for attack because every user-facing application requires input. With the malicious code, attackers can access the victim's credentials, such as cookies and passwords [34]. The log injection attack is also carried out through exploitation [37]. Therefore, securing Java applications from XSS and log injection requires server-side input validation because the client-side can be easily bypassed [36], [38].

Buffer overflow can be prevented by modifying the stack-allocated data and bound checking. The modification of the stack-allocated data presents canary values in programs that

help intercept the buffer overflow attack. Languages like Java natively employ bound checking, which checks permission to each allocated memory block. It prevents data into unallocated space because they do not have direct memory access [34]. Therefore, the identification of buffer overflow as a prevalent Java vulnerability from the text analysis of the MITRE's CVEs is best understood by the possibility of exploitation through the Java Virtual Machine, which is developed in C++ language, or Java Runtime Engine (JRE) [7], [8]. Also, the JVM is highly affected by unauthorized use of restricted classes, making custom classes defined with an exploit to run arbitrary codes and disable security managers without further security checks [7]. Array index out of bounds, as a type of buffer overflow that throws `ArrayIndexOutofBoundsException` is analyzed in the codebases.

Data deserialization in Java allows exploitation of the arbitrary Class and the type-safety features using type confusion, which means the object type passed to the code is not verified [7]. An attacker would insert a modified serialized object that can trigger a malicious code when deserialized [12]. On the other hand, improper input validation is when software does not validate input properly. It allows input in an unexpected and unsanitized form, leading to altered control or arbitrary code execution. Lastly, validation method bypass is manifested through confused deputies. These are programs that trick programs into misusing authority or bypassing validation. The experience would subsequently privilege code execution.

5.1.2. Java's vulnerabilities identified from the open-source projects' commits logs

SQL injection and Denial of entry (DoE), also known as Denial of Service (DoS), are the vulnerabilities suggested by the git commits of the top 100 Java projects on GitHub. Though the commit messages' wordings inhibit valuable insights, the tokens derived from the text analysis still helped. SQL injection manipulates applications by passing input containing SQL commands

to the database for execution. It can add, modify and delete records in a database [39], [40]. On the other hand, DoS is an explicit attempt to prevent legitimate users from using a service [41]. Though DoS is only 5% of the 2016 OWASP survey of web application attacks [42], it is, nevertheless, essential to prevent it [41].

Static analysis tool, input validation [10], [39], [43]–[45], injection detection tool [40], machine and deep learning models [40], [46], [47] are some of the techniques that can prevent SQL injection. Java development frameworks and libraries [38], especially for the Model-View-Controller development, are now developed with Object Relational Modelling (ORM) technology for data query and plain SQL statement execution alternatives. Considering its numerous attack approaches, the best way to prevent DoS is a hybrid of attacker and victim side defenses through overlay networks [41].

5.1.3. Security assessment of GitHub's Java projects source code for the identified Java vulnerabilities

Cross-site scripting, array index out of bound, data deserialization, input non-validation (or improper validation) for an untrusted object, and improper validation of user-provided array construction (as a validation method bypass) are the vulnerabilities investigated in GitHub's Java projects. As a type of buffer overflow, array index out of bounds was analyzed for two reasons. First, since the analysis is done on program source code, without the involvement of a Java platform or JRE, the presumption that Java technology has been built to avoid buffer overflow is held. Second, GitHub's *CodeQL*, the semantic code analyzer, most likely because of the first reason, provides documentation only for the array index out of bounds vulnerability.

The security assessment identified two (2) instances of array index out of bounds in each ReactiveX/RxJava, skylot/jadx, and openzipkin/zipkin. It found three (3) instances in spring-

projects/spring-framework, and one (1) in square/okhttp. The total number of cases of array index out of bound is ten (10) across four (4) out of the seven (7) codebases analyzed, implying 71.4% presence. Method bypass and improper validation of user-provided size used for array construction were not found in any codebases analyzed. Due to user-provided value, fifty-one (51) instances of XSS were found in only spring-projects/spring-framework codebase. A similar result is found for the improper validation of the user-provided array index, where eighty-one (81) instances were found only in spring-projects/spring-framework codebase. A single instance of deserialization vulnerability was found in spring-projects/spring-framework codebase.

5.2. Limitations of the findings

The main limitation of the findings of this study, which would understandably affect its generalizability, is the few numbers of successfully converted codebases that were ultimately used for the code analysis. Out of the one hundred (100) top open-source Java projects cloned from GitHub, only seven (7) were successfully converted. These are, therefore, the codebases compatible for code analysis using *CodeQL*. Though the reasons for the unsuccessful conversion are beyond the researcher's fix, future research should extend the pool of the top open-source projects enough to achieve at least thirty (30) compatible codebases.

The identified vulnerabilities reported in this study were based on heuristics, considering the prevalent tokens from the texts analyzed. Future research could employ n-gram analysis which provides more insights than tokens. Name Entity Recognizer (NER) for cybersecurity texts, where a processed text can be fed and recognized by the present name entity, such as vulnerability, vector attack, and agent, should also be developed. The non-existence of name entities for cybersecurity texts and themes affects the ability to gain a deeper understanding and make definitive conclusions from the text analysis of the vulnerabilities' descriptions extracted from the MITRE's NVD.

5.3. Conclusion of the study

The need to extend security design and integration to requirement engineering, software architecture, and coding beyond the prevalent security testing is justified. It is an essential approach toward promoting secure coding and ultimately reducing the experiences of vulnerable software, attacks, and the associated cost. This study's response to minimizing the occurrence of damaging attacks due to possible vulnerabilities in Java's OSS is the analysis of Java's CVEs in GitHub's Open-Source Projects using text and code analyses. It identified the prevalent vulnerabilities and evaluated the security state of the open-source projects. The text analysis of the Java's CVEs extracted from the MITRE's NVD identified cross-site scripting, buffer overflow, data deserialization, improper validation, and validation method bypass. SQL injection and Denial of Service vulnerabilities are identified from the git's commit.

The code analysis of the compatible codebases showed that array index out of bounds is a common vulnerability in Java's GitHub's open-source projects. Notably, the code analysis using *CodeQL* to identify the prevalent vulnerabilities in the Java open-source projects in GitHub investigated array index out of bounds instead of buffer overflow and the other identified vulnerabilities. In conclusion, despite its limitations, this study answered the central question of how secure the codes written in top GitHub's open-source projects are. It showed that the top projects on GitHub are not secured. These findings further emphasize the need for the adoption of secure coding practice.

REFERENCES

[1] G. McGraw, “Software Security: Building Security In,” in *2006 17th International Symposium on Software Reliability Engineering*, Raleigh, NC, Nov. 2006, pp. 6–6. doi: 10.1109/ISSRE.2006.43.

[2] J. McDermott and C. Fox, “Using Abuse Case Models for Security Requirements Analysis,” in *Proceedings 15th Annual Computer Security Applications Conference (ACSAC’99)*, Phoenix, AZ, USA, Dec. 1999, p. 11. doi: 10.1109/CSAC.1999.816013.

[3] K. Li, “Towards Security Vulnerability Detection by Source Code Model Checking,” in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, Paris, France, Apr. 2010, pp. 381–387. doi: 10.1109/ICSTW.2010.23.

[4] A. Ekelhart, S. Fenz, M. Klemen, and E. Weippl, “Security Ontologies: Improving Quantitative Risk Analysis,” in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS’07)*, Jan. 2007, pp. 156a–156a. doi: 10.1109/HICSS.2007.478.

[5] S. E. Ponta, H. Plate, and A. Sabetta, “Detection, assessment and mitigation of vulnerabilities in open source dependencies,” *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 3175–3215, Sep. 2020, doi: 10.1007/s10664-020-09830-x.

[6] Edgescan, “2016 Vulnerability Statistics Report.” EdgeScan Continous Vulnerability Management, 2016. [Online]. Available: www.edgescan.com

[7] P. Holzinger, S. Triller, A. Bartel, and E. Bodden, “An In-Depth Study of More Than Ten Years of Java Exploitation,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna Austria, Oct. 2016, pp. 779–790. doi: 10.1145/2976749.2978361.

[8] V. Jain, J. Gomex, and A. Singh, “A DAILY GRIND: Filtering Java Vulnerabilities,” *Security Reimagined*, p. 33.

[9] “CVE - Home.” http://cve.mitre.org/about/index.html (accessed Aug. 15, 2021).

[10] J. Viega, T. Mutdosch, G. McGraw, and E. W. Felten, “Statically Scanning Java Code for Security Vulnerabilities,” *IEEE Softw.*, vol. 17, no. 5, pp. 68–74, 2000.

[11] “The State of the Module System.” http://openjdk.java.net/projects/jigsaw/spec/sotms/ (accessed Aug. 19, 2021).

[12] “Serialization and deserialization in Java | Snyk Blog.” https://snyk.io/blog/serialization-and-deserialization-in-java/ (accessed Aug. 15, 2021).

[13] TechTarget, “How to Deliver DevSecOpsVeracode.” VeraCode.

[14] “PyCQA/bandit.” Python Code Quality Authority, Sep. 15, 2020. Accessed: Sep. 14, 2020. [Online]. Available: https://github.com/PyCQA/bandit

[15] “Checkmarx – Application Security, Made Easy,” *Checkmarx*. https://www.checkmarx.com/ (accessed Sep. 14, 2020).

[16] “Security Code Scan.” https://security-code-scan.github.io/ (accessed Sep. 14, 2020).

[17] “Source Code Analysis Tools | OWASP.” https://owasp.org/www-community/Source_Code_Analysis_Tools (accessed Sep. 03, 2020).

[18] G. Upadhyaya and H. Rajan, “Collective program analysis,” in *Proceedings of the 40th International Conference on Software Engineering - ICSE ’18*, Gothenburg, Sweden, 2018, pp. 620–631. doi: 10.1145/3180155.3180252.

[19] A. Fehnker, R. Huuck, and W. Rödiger, “Model checking dataflow for malicious input,” in *Proceedings of the Workshop on Embedded Systems Security - WESS ’11*, Taipei, Taiwan, 2011, pp. 1–10. doi: 10.1145/2072274.2072278.

[20] M. Payer, *Software Security: Principles, Policies and Protection*. 2019.
[21] S. Chen, J. Xu, Z. Kalbarczyk, and K. Iyer, "Security Vulnerabilities: From Analysis to Detection and Masking Techniques," *Proc. IEEE*, vol. 94, no. 2, pp. 407–418, Feb. 2006, doi: 10.1109/JPROC.2005.862473.
[22] C. S. Collberg, C. D. Thomborson, and D. W. K. Low, "Obfuscation techniques for enhancing software security," US6668325B1, Dec. 23, 2003 Accessed: Jul. 10, 2019. [Online]. Available: https://patents.google.com/patent/US6668325B1/en
[23] Y. Jia, Y. Qi, H. Shang, R. Jiang, and A. Li, "A Practical Approach to Constructing a Knowledge Graph for Cybersecurity," *Engineering*, vol. 4, no. 1, pp. 53–60, Feb. 2018, doi: 10.1016/j.eng.2018.01.004.
[24] L. Lowis and R. Accorsi, "On a Classification Approach for SOA Vulnerabilities," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Seattle, Washington, USA, 2009, pp. 439–444. doi: 10.1109/COMPSAC.2009.173.
[25] Y. Li, L. Ma, L. Shen, J. Lv, and P. Zhang, "Open source software security vulnerability detection based on dynamic behavior features," *PLOS ONE*, vol. 14, no. 8, p. e0221530, Aug. 2019, doi: 10.1371/journal.pone.0221530.
[26] R. L. Russell *et al.*, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," *ArXiv180704320 Cs Stat*, Nov. 2018, Accessed: Sep. 09, 2020. [Online]. Available: http://arxiv.org/abs/1807.04320
[27] J. A. Harer *et al.*, "Automated software vulnerability detection with machine learning," *ArXiv180304497 Cs Stat*, Aug. 2018, Accessed: Sep. 09, 2020. [Online]. Available: http://arxiv.org/abs/1803.04497
[28] B. Chernis and R. Verma, "Machine Learning Methods for Software Vulnerability Detection," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics - IWSPA '18*, Tempe, AZ, USA, 2018, pp. 31–39. doi: 10.1145/3180445.3180453.
[29] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning," *Appl. Sci.*, vol. 10, no. 5, p. 1692, Mar. 2020, doi: 10.3390/app10051692.
[30] University of Waterloo, "Deep Defect and Vulnerability Prediction"
[31] "CodeQL - GitHub Security Lab." https://securitylab.github.com/tools/codeql/ (accessed Sep. 09, 2020).
[32] "Pysa: Open Source static analysis for Python code," *Facebook Engineering*, Aug. 07, 2020. https://engineering.fb.com/security/pysa/ (accessed Sep. 09, 2020).
[33] F. Berends, "Library Guides: Text mining & text analysis: Introduction." https://guides.library.uq.edu.au/research-techniques/text-mining-analysis/introduction (accessed Aug. 26, 2021).
[34] P. Shital and C. R., "Web Browser Security: Different Attacks Detection and Prevention Techniques," *Int. J. Comput. Appl.*, vol. 170, no. 9, pp. 35–41, Jul. 2017, doi: 10.5120/ijca2017914938.
[35] "Deserialization - OWASP Cheat Sheet Series." https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html (accessed Aug. 15, 2021).
[36] J. Offutt, Ye Wu, Xiaochen Du, and Hong Huang, "Bypass Testing of Web Applications," in *15th International Symposium on Software Reliability Engineering*, Saint-Malo, Bretagne, France, 2004, pp. 187–197. doi: 10.1109/ISSRE.2004.13.

[37] S. Turner, “Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby,” p. 16.
[38] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, “Secure Coding Practices in Java: Challenges and Vulnerabilities,” *ArXiv170909970 Cs*, Sep. 2017, Accessed: Aug. 11, 2021. [Online]. Available: http://arxiv.org/abs/1709.09970
[39] V. B. Livshits and M. S. Lam, “Finding Security Vulnerabilities in Java Applications with Static Analysis,” p. 16.
[40] T. Pattewar, H. Patil, H. Patil, N. Patil, M. Taneja, and T. Wadile, “Detection of SQL Injection using Machine Learning: A Survey,” *Int. Res. J. Eng. Technol.*, vol. 06, no. 11, p. 8, 2019.
[41] Q. Gu and P. Liu, “Denial of Service Attacks,” p. 28.
[42] N. K. Sangani and H. Zarger, “Machine Learning in Application Security,” in *Advances in Security in Computing and Communications*, J. Sen, Ed. InTech, 2017. doi: 10.5772/intechopen.68796.
[43] S. Thakare and D. B. B. Meshram, “Java Program Vulnerabilities,” vol. 2, no. 3, p. 8, 2013.
[44] L. V. Satyanarayana, “STATIC ANALYSIS TOOL FOR DETECTING WEB APPLICATIONLVENUKLANERABILITIES,” *Int. J. Mod. Eng. Res. IJMER*, vol. 1, no. 1, pp. 127–133, 2009.
[45] A. Kaur and R. Nayyar, “A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code,” *Procedia Comput. Sci.*, vol. 171, pp. 2023–2029, 2020, doi: 10.1016/j.procs.2020.04.217.
[46] M. Vignesh and K. Kumar, “WEB APPLICATION VULNERABILITY PREDICTION USING MACHINE LEARNING,” *Int. J. Sci. Eng. Res.*, vol. 8, no. 5, pp. 80–90, 2017.
[47] L. K. Shar, L. C. Briand, and H. B. K. Tan, “Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning,” *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 6, pp. 688–707, Nov. 2015, doi: 10.1109/TDSC.2014.2373377.