The Construction of a Static Source Code Scanner Focused on
SQL Injection Vulnerabilities in Java

Carla Zurita Rubin de Celis

Follow this and additional works at: https://digitalcommons.montclair.edu/etd
Part of the Computer Sciences Commons
ABSTRACT

SQL injection attacks are a significant threat to web application security, allowing attackers to execute arbitrary SQL commands and gain unauthorized access to sensitive data. Static source code analysis is a widely used technique to identify security vulnerabilities in software, including SQL injection attacks. However, existing static source code scanners often produce false positives and require a high level of expertise to use effectively.

This thesis presents the design and implementation of a static source code scanner for SQL injection vulnerabilities in Java queries. The scanner uses a combination of pattern matching and data flow analysis to detect SQL injection vulnerabilities in code. The scanner identifies vulnerable code by analyzing method calls, expressions, and variable declarations to detect potential vulnerabilities.

To evaluate the scanner, malicious SQL code is manually injected in queries to test the scanner's ability to detect vulnerabilities. The results showed that the scanner could identify a high percentage of SQL injection vulnerabilities.

The limitations of the scanner include the inability to detect runtime user input validation and the reliance on predefined patterns and heuristics to identify vulnerabilities. Despite these limitations, the scanner provides a useful tool for junior developers to identify and address SQL injection vulnerabilities in their code.

This thesis presents a static source code scanner that can effectively detect SQL injection vulnerabilities in Java web applications. The scanner's design and implementation provide a useful contribution to the field of software security, and future work could focus on improving the scanner's precision and addressing its limitations.

Keywords: Static analysis, Source Code Scanner, Security Testing, SQL Injection
MONTCLAIR STATE UNIVERSITY

THE CONSTRUCTION OF A STATIC SOURCE CODE SCANNER FOCUSED ON
SQL INJECTION VULNERABILITIES IN JAVA

By

Carla Zurita Rubin de Celis

A Master’s Thesis Submitted to the Faculty of
Montclair State University

In Partial Fulfillment of the Requirements

For the Degree of

Master of Science

May, 2023

College of Science & Mathematics

Department of Computer Science

Thesis Committee:

Dr. Kazi Zakia Sultana

Dr. Jiayin Wang

Dr. John Jenq
THE CONSTRUCTION OF A STATIC SOURCE CODE SCANNER FOCUSED ON SQL INJECTION VULNERABILITIES IN JAVA

A THESIS

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

by

Carla Zurita Rubin de Celis

Montclair State University

Montclair, NJ

May, 2023
ACKNOWLEDGEMENTS

I would like to thank:

- Dr. Kazi Zakia Sultana, my thesis advisor, for her unwavering support and guidance throughout the research process. Her expertise and insight have been invaluable in shaping this thesis and improving my analytical and technical skills.

- Dr. Jiayin Wang, for her time and valuable feedback during the review process. Her insight and expertise have helped refine this research's scope and direction.

- Dr. John Jenq, for his time and valuable support during this process. His perspicuity and expertise in this field were valuable assets in improving and finalizing this research.

- My loving husband Warren, none of this would have been possible without him. He is always my biggest believer and provider of love and encouragement. He is my light in the darkest tunnels.

- My Mom and Dad, who always give me unconditional love and support. Foment me to better myself for a brighter tomorrow and made me who I am today!

- My siblings, Daniel, Gabriel, and Ariana, for always rooting for me and letting me know I can count on them no matter the circumstances.

- My mother-in-law and father-in-law, Evelyn and Phil, for giving me the support, love, and care to build and restart my life in this country.
TABLE OF CONTENTS

ABSTRACT .......................................................................................................................... 1

CHAPTER 1: INTRODUCTION .......................................................................................... 11

PROBLEM STATEMENT ............................................................................................... 16

OBJECTIVES AND SCOPE OF THE STUDY ................................................................. 16

CHAPTER 2: RELATED WORK ....................................................................................... 18

DISCUSSION OF THE MOST COMMONLY USED TOOLS AND APPROACHES IN THE FIELD .............................................................................................................................. 24

CHAPTER 3: THEORETICAL FRAMEWORK AND CONCEPTUAL MODEL .......... 28

DESCRIPTION OF THE KEY CONCEPTS AND COMPONENTS OF THE SCANNER ................................................................................................................................. 37

DIAGRAMS AND OTHER VISUAL AIDS TO HELP ILLUSTRATE THE CONCEPTUAL MODEL ......................................................................................................................... 42

CHAPTER 4: METHODOLOGY ....................................................................................... 42

DESCRIPTION OF THE RESEARCH METHODOLOGY AND DESIGN ............... 42

EXPLANATION OF THE TOOLS AND TECHNOLOGIES USED IN THE IMPLEMENTATION OF THE STATIC SOURCE CODE SCANNER ......................................................... 44

DISCUSSION OF THE TESTING AND EVALUATION STRATEGY ..................... 47

CHAPTER 5: IMPLEMENTATION, PRODUCTION AND RESULTS ...................... 48

DESCRIPTION OF THE IMPLEMENTATION AND PRODUCTION OF THE STATIC SOURCE CODE SCANNER ............................................................................................................. 48
LIST OF FIGURES

Figure 1 Patterns over time of security breaches in Verizon report ........................................... 12
Figure 2 OWASP Top Ten Vulnerabilities ..................................................................................... 13
Figure 3 Software Security practices applied in every stage of SDLC ........................................... 14
Figure 4 Example of a Control Flow Graph. Node 1 represents the entry block and Node 6
represents the exit block .................................................................................................................. 30
Figure 5 Description of SQL injection attack ............................................................................... 32
Figure 6 Unsafe SQL query ............................................................................................................. 33
Figure 7 Example of Abstract Syntax Tree Data Structure ............................................................ 39
Figure 8 Process of the Static Source Code Scanner .................................................................... 42
Figure 9 Maven path file in system settings .................................................................................. 49
Figure 10 System variables windows to paste file path ................................................................. 49
Figure 11 Maven project Group ID and Artifact ID ...................................................................... 50
Figure 12 Maven dependencies added to the Scanner Project ...................................................... 50
Figure 13 Scanner class that implements the GUI elements for Scanner ...................................... 51
Figure 14 Applications of the dropping and button features to scan the file from the user ........ 52
Figure 15 Method calls to parse and compile the source code file ............................................... 53
Figure 16 Display of parsing results and list of vulnerabilities found .......................................... 54
Figure 17 End of the Scanner class syntax .................................................................................... 55
Figure 18 First lines of VisitorScannerFinal class ....................................................................... 56
Figure 19 Class OuterVisitor and method calls to check for vulnerabilities ............................... 57
Figure 20 Patterns and Set of rules for the scanner ...................................................................... 58
Figure 21 Following part of the VisitoScannerFinal class. Application of rules and patterns .... 59
Figure 22 Code that searches for PreparedStatement or Statement declarations .......................... 60
Figure 23 Final lines of the VisitorScannerFinal class for Try and Catch Statements.................. 61
Figure 24 GUI of Scanner with source code dropped in. ......................................................... 62
Figure 25 Final Result in GUI layout ..................................................................................... 65
LIST OF ILLUSTRATIONS

Table 1 Scanning Results ............................................................................................................. 63
Table 2 False Negatives chart ..................................................................................................... 63
Table 3 Parsing Errors Chart ....................................................................................................... 64
CHAPTER 1: INTRODUCTION

Web and big software applications are the main sustainers of today's society. Given the seriousness of their function, it is imperative to secure and prevent attacks that may cause harm to their execution or cause any leakage of pivotal data that belongs to businesses, government agencies, and individuals. Software development is a complex process that involves many stages, from designing and coding to testing and maintenance.

The increasing data volume and the variety of data formats of modern data-intensive systems unveiled the boundaries of traditional relational database management systems. (Cherry, et al., 2020), (Phillips, 2023)

Software security is the protection of programs, applications, and digital interactions that contain essential data from unauthorized access, data breaches, and data destruction that must be considered throughout the software development process. (Yang, 2020)

According to the Open Web Application Security Project (OWASP), code scanning is a crucial part of the Security Development Cycle (SDC), considering that it finds security flaws in the logic and syntax of applications at a high confidence level. Applications with secure coding practices can lead to safe, secure, and reliable software systems. (OWASP, OWASP Top 10:2021, 2021)

Since the Covid-19 pandemic in 2020, cybercrime increased by 300% because of the change in payment methods from cash to online and the shift of the work structure from working at the office to remote work. (Vulpis, 2021)

A study by the University of Maryland found that hackers (an individual who uses computers to gain unauthorized access to data) attack data systems through the internet every 39 seconds. (Cukier, 2007), (Li, 2021), (Abner Mendoza, 2018)
Furthermore, despite the best efforts of software developers, software security is still a new and recent field, making the most secure and best software practices difficult to adopt. (McGraw, Software Security, 2016)

Moreover, bugs and vulnerabilities often creep into software, causing security breaches, crashes, and other problems, making software security an imperative field to be applied in every Software Development Life Cycle (SDLC). (McGraw, Software Penetration Testing, 2005)

In the past year of 2022, Verizon released its data breach investigation report, which showed that the second highest pattern over time in web application attacks was executed through online credentials. These attacks are launched against basic web applications, commonly developed by junior software developers. After the initial compromise, they have a small number of additional Actions. It is the "get in, get the data, and get out" pattern. (Verizon, 2022)

![Figure 1 Patterns over time in incidents](Verizon, 2022)

According to the Software Security Report of Veracode, a cybersecurity organization, 44.2% of the source code scanned by their tools are made in Java, meaning that security flaws and fixture of vulnerabilities are increasing. (Veracode, 2023)
According to the OWASP top 10 vulnerability report, released in 2021, a standard awareness document for developers and web application security, the most critical security risks for web applications are broken access control and injection. The report shows injection attacks as the third riskiest vulnerability attack. (OWASP, OWASP Top 10:2021, 2021)

This attack occurs when the input data is not sanitized, filtered, or validated; also, when the web application has dynamic queries that are non-parameterized, and malicious data is used to search for parameter and extract additional data. The most common injections are SQL and NoSQL. SQL injection happens when a hacker inserts his destructive and malicious SQL code into an external database, and then, runs those queries with the goal of extracting private and valuable information or may destroy the database. (Divya Jain, 2015), (Anley, (more) Advanced SQL Injectio, 2002)

One of the solutions to this attack, according to Gary McGraw, Ph.D., in his released paper "Software Security" from Cigital, is the correct application of secured code practices, like considering security in the early stages of SDLC, can solve software security problems. (McGraw, Software Security, 2016)
These software security practices can be fitted into every stage of SDLC and different software artifacts, as shown in the image below.

Figure 3: Software Security practices applied in every stage of SDLC (McGraw, Software Security, 2016)

As seen, one of the biggest defenses against these attacks and practices in secure software is Static Analysis tools, preferably applied in the coding stage. Organizations can include static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools in the Continuous Integration pipeline to identify flaws and vulnerabilities before production deployment. Static code analysis (SAST) is code analysis that can assist in the identification of security vulnerabilities, which developers can use in examining and analyzing their source code to identify potential problems without executing the program. (Yang, 2020), (Ivanov, 2005)

One type of static analysis technique is code scanning, which involves analyzing the source code for common programming errors and vulnerabilities. Code scanning can be used to identify issues such as buffer overflows, SQL injections, null pointer exceptions, and cross-site scripting (XSS) vulnerabilities. (Yang, 2020), (Bardas), (Junjie Wang, 2022)

Other additional benefits of performing this type of scanning are finding syntax, data flow, or bugs in earlier stages, which prevent future errors and attacks, saving the organization or company money and time. They can also detect overcomplexity in code by refactoring, simplifying, and
making complicated or large coding blocks smaller and more manageable, apart from creating efficient coding practices for developers.

Previous studies have shown that SAST scanners can help detect software defects faster and cheaper than human inspection or software testing. Because of this, they have been widely adopted by professional software developers and regularly integrated with current open-source projects and commercial software organizations. SAST tools are key weapons for to protect vulnerabilities that induce the previously described attacks. According to a study produced by computer scientists in Brazil and Germany, more than 80% of software developers agree or strongly agree that the issues reported by SAST tools are relevant for improving the design and implementation of software. (Marcilio, 2020)

Although there are two types of SAST options available; one is open source, and the other one is commercial; both can automatically scan source code and generate reports on potential problems. However, many of these tools are expensive or difficult to use, and they may not be suitable for all types of software. (Junjie Wang, 2022), (Zurita, 2022)

According to a study by Google and North Carolina State University, software developers do not use static source code scanner tools because they are challenging to configure and do not accommodate the customizations developers want. The presentation of the result could be higher, and there needs to be more information to help developers discover the problem and why it is a problem. (Johnson, 2019)

Good static analysis tools should be easy to use, meaning their results must be understandable to regular developers so that they educate their users about good programming practices. They should provide an automated and educative way to ensure that source code remains to predefined design and style guidelines that the software developer needs to produce a more uniform code while
minimizing the shortcomings without the need to spend more time understanding the output of the tool. (Gomes, 2009)

PROBLEM STATEMENT

This thesis proposes the development of a de novo static source code scanner using Java programming language to find SQL injection vulnerabilities in queries using the Java Data Base Connectivity API for the use of junior developers.

The scanner will be designed to analyze Java code files for potential SQL defects and vulnerabilities following specific security prevention rules and patterns. Along with using and applying the Java Parser tool, the scanner will extract information from the source code, identify security issues and warn the software developer about the vulnerabilities, stating the warnings of the error and recommendations to fix it.

This projects aims to develop a tool that is

- Fast and efficient, that can scan long vulnerable code in seconds.
- User-friendly, making it accessible for all amateur software developers, no matter their expertise level in secure programming.
- Simple, by adding a “Drop and Scan” feature to the software. The developers won’t need further installation or documentation.
- Educational, applying educational features along with the results, that show the why it happened and how it happened. With the main goal of encouragement in the practice of security in their daily coding skills without additional struggle.

OBJECTIVES AND SCOPE OF THE STUDY

The primary research question addressed by this thesis will be:
• "Can a static source code scanner be developed in Java using a parser that effectively identifies common SQL programming errors and vulnerabilities?"

The thesis will explore the following sub-questions:

• What are the requirements and steps for a static source code scanner from scratch?
• What are the most common SQL Injection vulnerabilities that a static source code scanner can identify?
• What are the advantages and disadvantages of using a parser-based approach to static analysis?
• What are the design principles and technical requirements for developing an effective static source code scanner in Java for junior developers?
• How can the scanner be evaluated and validated to ensure its effectiveness and accuracy in detecting SQL vulnerabilities?
• What is the prevention procedure to defend queries from SQL injections?
• How can this static source code scanner be more user-friendly than the current source code scanners?
• How can this static source code scanner help minimize the technical debt of developers?

The thesis will be structured as follows. Chapter 2 will provide an overview of the static analysis and code scanning literature, focusing on previous research, methodologies and effectiveness of these type of tool in Java. Chapter 3 will describe the theoretical framework and conceptual model underpinning the scanner's design, including the key concepts and components. Chapter 4 will describe the methodology used to develop and test the scanner, including the research design, data collection and analysis methods, and testing evaluation techniques. Chapter 5 will present the
implementation of the scanner in Java, including the design and development of the parser and other vital components. Chapter 6 will present the results of the testing and evaluation of the scanner, including an analysis of its performance and effectiveness. Chapter 7 will discuss the study's implications for static analysis and code scanning, as well as the limitations and future directions of the research. Finally, chapter 8 will provide a conclusion and summary of the essential findings and contributions of the study.

In summary, this thesis proposes the development of a static source code scanner in Java using a parser to improve software quality and security in Java queries from SQL injection attacks for junior developers. The study will explore the design and implementation of the scanner, as well as its effectiveness and limitations, by testing various vulnerable queries. It will contribute to the broader literature on static analysis and code scanning, and it has the potential to inform the development of future tools and techniques for software development and cybersecurity.

CHAPTER 2: RELATED WORK

Up to the current time, there are research developments in static code analysis that led to the creation of many tools and techniques for identifying software defects and vulnerabilities before its launch. However, most of the research papers are about using existing scanners to identify and classify security vulnerabilities. Some papers and books describe the procedure and experience to develop static source code scanner from the ground; however, most have a more general view and try to correct as many vulnerabilities as possible in various attacks but not in a profound manner. One of the most related research papers is from Stanford University called "A System and Language for Building System-Specific, Static Analyses," written by Xie, Engler, and Hallem, which describes an algorithm and process to build a source code scanner using the language metal. (Seth Hallem, 2002)
The oldest research in building a static source code analyzer dates back to the early 1970s. One of the earliest works in this field, it was the publication of the paper "Program Design by Informal English Description" by Harlan Mills in 1972. Mills proposed an approach for designing software based on natural language descriptions of program functionality, which could be used to generate code automatically. The approach relied on static analysis techniques to verify that the generated code was correct and met the design requirements. (Abbot, 1983)

Another initial research in this field was the publication of the paper "A Survey of Program Design Techniques" by B. Nejmeh in 1985. The paper reviewed various program design techniques, including formal methods and static analysis techniques. (Brian Nejmeh, 1985)

Some of the earliest approaches to static analysis were using formal methods involving mathematical models and logic to reason about program behavior. These can be useful for proving software properties, such as safety or correctness. However, they can be time-consuming and challenging to apply to large software systems.

In the following decades, research in building static source code analyzers continued to evolve with the development of more sophisticated techniques and tools. In the 1980s and 1990s, researchers focused on using formal software verification and validation methods and developing automated tools for checking program correctness. (Mike Hinchey, 2008)

In the early 2000s, there was a shift towards using code scanning tools and other techniques for identifying software defects and vulnerabilities. This was partly driven by the increasing complexity of software systems and the need to identify potential issues before software is released. Lately, machine learning techniques have also been applied to static analysis and code scanning, moving the tendency of research papers towards calculating and observing the effects of the artificial intelligence application in code scanning and its efficiency. One example is source code
summarization, a methodology for understanding the code's functionality and automatically generating its descriptions. Machine learning is used to identify software code patterns that indicate potential security issues. Nonetheless, this approach also requires large amounts of high-quality data to train the models, and the human models may need help to interpret. (Shraddha Biraria, 2021), (Hongliang Liang, 2019)

Most scanning tools projects use pattern equivalents and other heuristics to identify potential defects or vulnerabilities. Through the analysis of the user input side, code scanners can effectively identify archetypes that make SQL injection or Cross-site scripting vulnerabilities a risk externally. Previous research has also focused on evaluating and validating static analysis approaches. For example, researchers have developed benchmark suites for evaluating the effectiveness of code scanning tools, such as the Juliet Test Suite. These benchmark suites provide a standardized set of test cases that can be used to evaluate the performance and effectiveness of different static analysis tools, just like the OWASP Benchmark Project. (OWASP, OWASP Benchmark Project, 2023)

Additional studies have targeted the improvement of the accuracy and efficiency of static analysis techniques. For example, researchers have proposed using context-sensitive analysis to boost the precision of dataflow analysis. The context-sensitive analysis involves analyzing the program in its execution context, granting a more precise flow of data through the program. One of the examples is SonarQube, which offers a measure of technical debt. (Baldasarre, 2020)

We can find documents integrating static analysis techniques into the software development process. For example, researchers have proposed using continuous integration and deployment (CI/CD) pipelines that incorporate static and dynamic analysis tools into the development process. This approach allows developers to identify potential issues early in the development cycle,
reducing the likelihood of defects entering the production environment. One example is FindBugs, a plug-in software for Java language. (Rangnau, Buijtenen, Fransen, & Turkmen, 2020)

Previous explorations have also focused on the limitations and challenges of static analysis, such as producing false positives, which can be time-consuming to resolve. The difficulty is identifying all potential issues in software code, particularly in cases where the syntax is complex or uses advanced programming techniques. According to Mushtaq, Rasool, and Shehzad, it is estimated that the size of software in the year 2025 will be more than 1 trillion lines of syntax. This reflects the importance of source code analysis in the future. (Mushtaq, 2017)

The previously mentioned tools and techniques have applied formal methods like dataflow analysis, scanning, machine learning, and other approaches, evaluating its accuracy, efficiency improvement, software development process integration, and addressing limitations and challenges of static scanning.

Nevertheless, this field still represents an important area of research in software engineering and cybersecurity. It enables developers, no matter their expertise level, to identify potential issues in programs before it is released, improving the security and reliability of software systems.

About research focused on SQL injection vulnerabilities, has focused on understanding how these vulnerabilities arise, identifying vulnerable systems and applications, and developing effective mitigation strategies. One of the early studies in this area was conducted by Halfond and Orso (2009), who proposed a technique for detecting SQL injection vulnerabilities in web applications. Their approach involved analyzing the application's input validation and sanitization logic to identify potential vulnerabilities through techniques like call monitoring and hotspot technique that was successful but did not apply static analysis techniques. (Orso, 2005)
Other studies have explored the prevalence of SQL injection vulnerabilities in real-world systems. For example, research a large-scale analysis of SQL injection vulnerabilities in web applications from Stamford University, discussed how a general class of security errors in Java applications like SQL injections can be represented as sink objects which are derivable from sources via a combination of rules. This research applied the technique of taint analysis and developed a precise and scalable project with more precision so their scanner can find as many vulnerabilities matches in the source code. (Lam, 2005)

There have also been studies made to understand the nature of SQL injections, exploring what they are, how they happen and what are the causes of it (Boneh, 2009), (Zainab S. Alwan, 2017), (Gregory T. Buehrer, 2005).

According to Anley, in Advanced SQL Injection in SQL Server, a typical SQL statement that looks like this:

```
SELECT id, forename, AND surname FROM authors.
```

This statement will retrieve the 'id,' 'forename', and 'surname' columns from the 'authors' table, returning all rows. The 'result set' could be restricted to a specific 'author' like this:

```
SELECT id, forename, AND surname FROM authors WHERE forename = 'john' AND surname = 'smith'
```

The critical point is that the string literals 'john' and 'smith' are delimited with single quotes. Presuming that the 'forename' and 'surname' fields are being gathered from user-supplied input, an attacker might be able to 'inject' some SQL into this query by inputting values into the application like this:

```
Forename: jo'hn Surname: smith.
```
The 'query string' becomes this:

```sql
SELECT id, forename, surname FROM authors WHERE forename = 'joh'n' and surname = 'smith'.
```

When the database processes this query, it will likely return an error and leak information or damage the database. (Anley, Advanced SQL Injection In SQL Server, 2002), (Anley, (more) Advanced SQL Injec, 2002) (Boneh, 2009)

Jain and Choudhary in 2015, found patterns and attacks that can generate SQL Injection like the Union Query, which is done by introducing a UNION keyword or PLUS operator into a vulnerable parameter that will return the union of the original and injected query. The SQL UNION operator fetched the results (rows) from participating queries. Furthermore, the stored procedure, where this centralized logic is built to access resources and complex queries, is moved into a stored procedure. (Divya Jain, 2015), (Anley, Advanced SQL Injection In SQL Server, 2002)

In addition to the detection and analysis of SQL injection vulnerabilities, researchers have proposed various mitigation strategies. One approach is to use parameterized queries, which separate user input from the SQL code and prevent injection attacks. Another approach is implementing input validation and sanitization logic that carefully checks user input for potential SQL injection attacks. These strategies effectively reduce the risk of SQL injection attacks in many cases. (Prato, 2008)

However, despite the efforts of researchers and developers, SQL injection vulnerabilities still need to be addressed. Recent studies have highlighted new techniques and attack vectors that can exploit these vulnerabilities—for example, Dong and Change (2017) proposed an adaptive detection system for malicious queries, called AMODS, which takes daily web traffic as input, and models queries in web requests. (Dong, 2017)
Overall, research into SQL injection vulnerabilities has played an essential role in improving our understanding of this problem and developing effective mitigation strategies. However, the persistence of these vulnerabilities suggests that more work is needed to address this issue thoroughly. Moreover, previous work and research in static code analysis have led to the developing of many tools and techniques for identifying software defects and vulnerabilities before the software is released. However, it still needs more improvement and more focus on the user side and their level of expertise in secured programming. Ongoing research will likely focus on further improvement of accuracy and efficiency and developing new approaches for addressing the limitations and challenges of syntax control and data flow. Along with the increment of software systems complexity and the growing importance of software security, this area will continue to play a critical role in software development and maintenance.

**DISCUSSION OF THE MOST COMMONLY USED TOOLS AND APPROACHES IN THE FIELD**

Several commercial and open-source code scanning tools, including SonarQube, CodeQL, and Bearer, are available. These tools typically support multiple programming languages and can provide detailed reports on potential issues. However, most of them have a complex installation procedure, a high learning curve, and diverse features that may be considered time-consuming and challenging for junior developers. Bearer, SonarQube, and Appsonar are the most user-friendly code-scanning tools because they can be called through APIs. AppSonar has the easiest scanning method, the "drop file and click" scan, but it is only commercial, not open source. (Zurita, 2022) CodeQL is a static analysis implemented in Query Language and is simply a query run on a particular database: the database contains a representation of the program to analyze (encoding, say, its abstract syntax tree or control flow graph), from which the query computes a set of result
tuples. A bug-finding analysis, for instance, could return pairs of source locations and error messages. Since the database describes the program as it was at one particular point in time. This is also the latest source code scanner released in recent years and has the highest learning curve. The language needs complex supporting tools from the installation and setting up stages. It cannot be called through an API and requires coding expertise to write and customize queries. Furthermore, when it comes to security confidence level, customizable scanners are a better fit. If the programmer knows the rules or code block that may lead to attacks or leakages, adapting the rules and queries will give the most efficient result. (Avgustinov, 1998), (Santos, 2021)

Most popular source code scanners have advanced installation features that can take longer, they also have a more complex integration environment, and require more memory or extend hardware/software components. However, after installation, most scanners work fast and efficiently. CodeQL is the only static security tool that is a query programming language that needs its files and extensions. It cannot work correctly without the source code in a database format. The other tools also need their environments and database connection to function correctly. All tools use the techniques of data flow and control flow graph analysis as scanning automation techniques, especially Appsonar, Bearer, CodeQL, and SEMGREP. (Zurita, 2022), (APPSONAR, 2023), (SEMGREP, 2023), (CodeQL, 2023), (Bearer, 2023)

All tools claim to protect against the 10 OWASP vulnerabilities, especially against the injection of malicious code, cross-site scripting, and insecure deserialization. SonarQube uses static code analysis to find issues in code, but some issues can only be detected at runtime, such as issues related to the configuration of third-party components. (OWASP, OWASP Top 10:2021, 2021)

Most tools rely on a static rule set of predetermined code patterns that should detect and report issues. Experienced vulnerability researchers develop these rules and allow users with less security
expertise to find bugs they would not otherwise be able to recognize. The drawback is that a tool only reports vulnerabilities in its rule set. If a tool finds no bugs in a program, this is not a guarantee that no bugs exist. (Zurita, 2022)

SonarQube provides a feature to mark an issue as a false positive, but this requires manual intervention. Something to look for is the production of false positives in tools, that is, issues that are not actual problems but are flagged as such. Integrating SonarQube with other tools in a software development pipeline can be challenging, especially for organizations that use multiple programming languages or build systems. While SonarQube is a powerful tool for finding issues in code, it has its limitations and should be used with other software development tools and techniques. A study by Brasilia and Paderborn universities researchers found that SonarQube is used by more than 85,000 organizations and encompasses rules from other SATs, such as FindBugs, and PMD. However, only 68% of major issues are fixed by the software developers, and one of the causes is the preliminary determination of security rules, which means "What are we looking for?". Also, they found that security rules are dynamic and can be deleted or inserted based on the software version of the SonarQube installation, and just a subset of these rules lead to issues reports. (Marcilio, 2020), (Justin Smith, 2015)

Tools like Bearer have higher complexity levels because of the platform they run in and the high learning curve to use them effectively. Bearer also depends on third-party APIs, which can be unreliable or down, leading to service interruptions. Additionally, it can become expensive if we need to use many integrations or require more advanced features. Another feature to remember is the sensitive data being handled and kept by the scanners, which can be problematic if any breach or leak happens. Bearer does point out the issues found, but there are no details in the issues, just
a reference word with a number of occurrences. There is no code or any suggestions to fix the issues. (Bearer, 2023)

Static analysis tools are scalable, making them flexible and adaptable for different projects and programming languages. Also, the tools are efficient with automation and iteration options because they can detect and cover all vulnerabilities in future projects. CodeQL is efficient in scanning injection and deserialization problems in the source code. As long as the tester knows the malicious code or the flaw's syntax, it can help us find those patterns in the code. The drawback of the scanner is the familiarity and expertise that the user must have with the language to call operators and functions. The syntax and logic are similar to the SQL language but not the same. As mentioned, CodeQL has an extensive learning curve because of the query syntax, which is peculiar and different from the other scanning tools. There are no predefined or guidance queries for specific vulnerabilities or data flows. The documentation is theoretical and needs more coverage of the basic steps like installation and creation of the ql files. The user must create everything from scratch. A few reviews, studies, or hands-on experiences are documented on Github repositories; therefore, knowledge of how to manage the GitHub platform is a must. (CodeQL, 2023), (Zurita, 2022)

Although Appsonar is the most user-friendly tool for software developers, it is expensive. Many security vulnerabilities are challenging to find automatically and manually. Creating rules and queries from scratch can be challenging, especially if we need the security scanner to act fast before the software launching stage. (APPSONAR, 2023)

Most tools rely on a static rule set of predetermined code patterns that they should detect and report. Experienced vulnerability researchers develop these rules and allow users with less security expertise to find bugs they would not otherwise be able to recognize. The drawback is that a tool
will be able to report only vulnerabilities that exist in its rule set. If a tool finds no bugs in a program, this is not a guarantee that no bugs exist. (Gomes, 2009)

CHAPTER 3: THEORETICAL FRAMEWORK AND CONCEPTUAL MODEL

Building a source code scanner in Java requires a well-defined theoretical framework and conceptual model. This episode will explore the key concepts and theoretical frameworks for building an effective source code scanner. According to Ganesh Samarthyam in his article "Joy of Programming: The Technology Behind Static Analysis Tools," the theoretical foundation of a static source code scanner is based on two main applications of program analysis: to optimize code and to find vulnerabilities also called “Bugs”. (Ganesh, 2011)

A compiler optimizer analyses programs to understand how they can generate more efficient code. Bug-detection tools analyze programs to see if there are any mistakes in the program, such as buffer overflows, that can lead to runtime errors. Static analyzers primary goal is finding bugs and errors in applications before they are deployed to production. This programming procedure relies on the deep and thorough analysis of the syntax, programming logic, and it can be performed at any stage of the software development life cycle. It is a technique used to automatically examine the source code, generate insights into its behavior, flow, and warn software developers of vulnerabilities and risks. (Ganesh, 2011), (Gregory T. Buehrer, 2005), (Hongliang Liang, 2019)

Alexandru G. Bardas, Ph.D., classifies analyzers into two categories. The ones that work directly on the program source code and analyzers that work on the compiled byte code. Each type has its advantages. When analyzing the program code directly, the source code static analyzer checks directly the source program code written by the programmer. On the other hand, working on byte
code is much faster. It can find all potential security violations without executing the application, given that it obviates the need for the source code to be accessible. (Bardas)

The direct static source code scanner is designed to perform automated analysis of the source code to detect security vulnerabilities, bugs, and other issues. The scanner of this research, will analyze the code at a high level of abstraction, looking for patterns and structures that may indicate a problem. Therefore, it is going to be classified as a source code-level analyzer.

Also, taking into account that the foundation of a source code level analyzer lies in software engineering, which aims to develop reliable, maintainable, and efficient software systems. This SQL injection scanner is designed to help achieve these goals by analyzing the source code of a program for potential security vulnerabilities and code quality issues.

Following OWASP, the techniques that most static scanners need to generate more exhaustive and logical results are based on:

Data Flow Analysis, which is a technique used in software engineering to analyze the behavior of a program by examining how data is transferred between different parts of the program. It is a static analysis technique that examines the paths of data through a program to identify variables, constants, and expressions that can influence the behavior of the program. Data flow will be applied to collect variables information while it is in a static state. However, it can also produce many false positives, which can be time-consuming to resolve. (OWASP, OWASP Top 10:2021, 2021)

Control Flow Graph (CFG), is a graph representation of software using nodes representing basic blocks. A node in a graph represents a block; directed edges represent jumps (paths) from one block to another. If a node only has an exit edge, this is known as an 'entry' block. If a node only has an entry edge, this is known as an 'exit' block.
Pattern matching, the most straightforward static analysis technique. This is a common source code auditing technique uses the grep tool to find all occurrences of a particular pattern or lack of specific practice in the source code. Most of these would be calls to the function in the standard Java libraries. These techniques can often be omitted and not used. (OWASP, OWASP Top 10:2021, 2021)

Taint Analysis, is a technique that attempts to identify variables that have been 'tainted' with user-controllable input and traces them to possible vulnerable functions, also known as a 'sink.' If the tainted variable gets passed to a sink without being sanitized, it is flagged as a vulnerability. (Bardas), (Obadoni, 2023), (OWASP, OWASP Top 10:2021, 2021)

This taint technique analysis, is used to identify how user-controlled data flows (known as "tainted" data) through the program and may be used in security attacks. This analysis tracks how this data flows through different parts of the program, including inside function calls and data structures, identifying possible security vulnerabilities where this user input may be misused by tagging data inputs from untrusted sources as "tainted," such as user input from a web form or a file on disk.
The scanner executes, and the analysis tracks how this tainted data flows through the program and how it is used in different functions and operations.

The analysis will flag this as a potential security vulnerability if the tainted data is used in a potentially dangerous operation, such as a database query or system command. This should be the primary technique applied in web application security because the user's input from web forms or other sources can be vulnerable to injection attacks, such as SQL injection or cross-site scripting (XSS) attacks. (OWASP, OWASP Top 10:2021, 2021)

Lexical Analysis, is a scanning tactic that converts source code syntax into 'tokens' of information to abstract the source code, making it easier to manipulate. This analysis offers a slight improvement over simple pattern matching. A lexer breaks down the source code and turns it into a stream of tokens, fundamentals components of the code, discarding whitespace. The tokens can represent keywords, operators, identifiers, literals, and other programming language elements. For example, it may recognize keywords such as "if" or "while," operators such as "+" or "-," or identifiers such as variable names. Afterward, these tokens are matched against a database of known vulnerability patterns and get an attribute assigned to them, such as its type and value.

For example, a token representing the integer literal "123" may be assigned the type "integer" and the value "123". This technique improves the accuracy of pattern matching because a lexer can handle irregular whitespace and code formatting. It also detects and reports syntax or lexical errors, like when an input character is not recognized as part of any token. Unfortunately, the benefits of lexical analysis are small, and the number of false positives reported by these tools still needs to be higher. (Hippisley, 2010), (OWASP, OWASP Top 10:2021, 2021)

The static scanning of this project will be based on the application of combination techniques like Data Flow Analysis, Control Flow, Lexical Analysis, and Pattern matching because the algorithm
will break down the code into tokens with a parser automatically. Then, it will scan the code token by token while creating an abstract syntax tree to find potential vulnerabilities or reassure that certain defensive variables were applied. At the end of the scanning process, this will generate a report or message alerting the software developer to the potential vulnerabilities and code quality issues, along with recommendations for addressing them.

Given that the goal of this static analyzer is the detection of SQL injection vulnerabilities in queries. Theoretical foundation in the concept of SQL injection is also required.

According to multiple studies, SQL injection happens when an attacker inserts malicious SQL code into an SQL query by manipulating data input into a website or application. Becoming a serious threat to any program that reads input from users and uses it to build and execute SQL queries to an underlying database.

<table>
<thead>
<tr>
<th>Username:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password:</td>
</tr>
</tbody>
</table>

**Normal Query**
SELECT * FROM USERS WHERE Username = 'Bob' AND Password = 'Password'

**Malicious Query**
SELECT * FROM USERS WHERE Username = 'OR '1'='1'; /* AND Password = '*/--'

*Figure 5 Description of SQL injection attack*

With the malicious code injected in the query, the attacker can run random SQL queries to bypass security mechanisms, generate an error or generate a data leak in the database. (Anley, Advanced SQL Injection In SQL Server, 2002), (Boneh, 2009), (Divya Jain, 2015), (SQL Injection Cheat Sheet, n.d.)
Because SQL queries are already previously built to communicate with the database. There are certain variable declarations and different syntax pattern commonly use by software engineers that make standard queries vulnerable. An unsafe SQL queries typically look like this:

```java
String query = "SELECT account_balance FROM user_data WHERE user_name = " + request.getParameter("customerName");

try {
    Statement statement = connection.createStatement( ... );
    ResultSet results = statement.executeQuery( query );
}
```

*Figure 6 Unsafe SQL query*

The unvalidated "customerName" parameter concatenated to the query allows attackers to inject any SQL code they want. This query shows a string concatenation of user input with the SQL query string along with the getParameter() method directly. Additionally, the query also lacks used of prepared statements and uses Statements instead, which allows the user input to be difficult to distinguish from SQL code.

Also, as shown on the example, the string concatenation is being used with an execution method, executeQuery(). This is making the database extremely vulnerable, given that is taking the user input directly into execution.

Unfortunately, this method for accessing databases is all too common. (OWASP, SQL Injection Prevention, n.d.)
Three defensive patterns are needed to prevent SQL injection in queries. These are:

1. **Application of Prepared Statements (Parameterized the Queries):**

   The use of prepared statements is simple to write and easier to understand. They should be used in replacement of Statement declarations because they parameterized queries. This means that PreparedStatements force the pre-definition of all the SQL code and then passes it parameter by parameter to the query execution methods later. This coding practice allows the database to distinguish between code and data, regardless of the provided user input. PreparedStatements ensure that an attacker won’t be able change the intent of a query, even if SQL malicious commands are inserted as user input.

   This static scanner will be able to search for the use and declaration of Prepared statements. If no Prepared Statements are declared, then the scanner will search for Statement declarations used for executing SQL queries, given that these are used often in replacement of Prepared Statements. (Pour, 2013), (OWASP, SQL Injection Prevention, n.d.)

2. **Avoidance of string concatenation:**

   These vulnerable practices can be a serious security risk in software applications that use SQL databases. When user input is concatenated directly into SQL queries without proper sanitization, attackers can inject their own SQL commands into the query, leading to unauthorized data access, modification, or deletion. Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures. (OWASP, OWASP Top ten 2017, 2017), (Chapela, 2005)
String concatenations are very difficult to detect because of their dynamic nature. That alone does not represent a big threat, but when combined with other harmful methods or variables through data flow, it can threat all the queries.

a. One situation with this risk, is when the concatenation is being called as arguments for vulnerable methods like "executeQuery," "executeUpdate," "execute," "executeBatch," "executeLargeBatch," and "executeLargeUpdate.". Because hostile data is directly used or concatenated when using these methods, the SQL command contains and combines the structure parameters and malicious data. The scanner will include h theory to identify methods that could potentially be vulnerable to SQL injection attacks, such as "executeQuery," "executeUpdate," "execute," "executeBatch," "executeLargeBatch," and "executeLargeUpdate." Then it will filter this methods by checking whether they are being called with string concatenations that could contain user input. If so, the scanner will consider them them potential vulnerabilities. (OWASP, OWASP Top ten 2017, 2017), (Lam, 2005), (Lwin Khin SHAR, 2012)

b. Another situation with string concatenations vulnerabilities, happens when the concatenations involve string literals like UNION keywords, or variables containing user input with special characters like' or --and as vulnerable parameters. (Divya Jain, 2015), (Lwin Khin SHAR, 2012)

c. Another technique to check for string concatenation parameters being passed is an expression involving binary operators like "+" declared next to string literals to be passed as parameters in conjunction. into a vulnerable parameter which will return the union of the original and injected query. (Anley, (more) Advanced SQL Injectio, 2002)
3. **Checking user input as a parameter**

This occurs when user input is used as a parameter in combination with methods like getParameter or getQueryString, which are functions applied to retrieve query parameters from a URL. It parses the query string portion of a URL and returns an object containing key-value pairs of the parameters. If malicious code is injected into the query string, it will cause data leakage. Therefore, they are considered a vulnerability if the user input is not sanitized correctly or validated before it is used in the method call. (OWASP, Query Parameterization Cheat Sheet, n.d.), (OWASP, SQL Injection Prevention, n.d.), (Lwin Khin SHAR, 2012), (Owasp, n.d.)

The conceptual model underpinning this static source code scanner's design is based on the program analysis concept and includes the following concepts:

1. **Code Scanner**: This is the central component of the source code scanner. It will analyze the source code and identifies potential errors and vulnerabilities. The code scanner will use algorithms based on the programming language semantics to identify code constructs that violate the language rules and constraints.

2. **Rule Base**: The rule base is a set of rules used by the code scanner to identify potential errors and vulnerabilities by applying the previous static scanning tactics above. The rule base is based on the programming language semantics and includes rules for identifying common programming errors and security patterns that cause vulnerabilities. The security rules that will be applied to this project will be based on user input dataflow, control flow and taint analysis aiming vulnerabilities and syntax that cause SQL injection. These rules will be produce based on the patterns
of string concatenation, Prepared Statements and User input in parameters that we previously explained.

3. **Reporting System:** The reporting system will be responsible for generating reports that summarize the identified issues. The reporting system includes tools for filtering and prioritizing the identified issues based on their severity and impact.

4. **GUI:** The source code scanner can be integrated with programming files to ensure that developers can quickly and efficiently address the identified issues through a user-friendly graphic user interface to obtain easy-to-read and observe feedback and the ability to generate fixes for the identified issues automatically. The scanner will be divided into two panels; one to display the source code to scan, another to display the errors scanned along with the recommended fixtures.

5. **Continuous updates and patches:** Continuous updates will involve actualizing the source code scanner with newer vulnerability rules so the software is up to date and also improving certain features, like friendliness, and others.

Building a source code scanner in Java requires a deep understanding of the programming language semantics, formal verification techniques, and code analysis algorithms. The conceptual model for building a source code scanner includes the code scanner, rule base, reporting system, integration with development tools, and continuous integration. By following the theoretical framework and conceptual model, developers can build an effective source code scanner that identifies potential errors and security vulnerabilities and helps ensure the integrity and security of the software system.

**DESCRIPTION OF THE KEY CONCEPTS AND COMPONENTS OF THE SCANNER**

The process to build this research contains eight parts:
1. **Determine the programming language:** The first component to build a static source code scanner is the definition of the programming language that this one has to use to build the scanner. In this situation, the static source code scanner to be built will use and read Java file extensions. Therefore, the syntax, semantics, tools, and libraries must be evaluated and understood to apply them and correctly create the software.

2. **Outline the scope of the scanner:** involves analyzing the codebase of a Java application for potential vulnerabilities that could be exploited by attackers using SQL injection attacks. This type of scanner would use a set of pre-defined rules and algorithms to search for patterns in the source code that could be indicative of SQL injection vulnerabilities.

3. **Select the parsing library:** Select the parsing library: Parsing analyzes a text or sentence in a particular language to determine its grammatical structure and identify its constituent parts, such as nouns, verbs, adjectives, and phrases. It involves breaking down a sentence into its parts and determining how they relate. Parsing is an essential component of natural language processing (NLP), a field in computer science and artificial intelligence that studies the interaction between humans and computers. (Gregory T. Buehrer, 2005), (Michal Duracik, 2020)

   The parser in the static source code is responsible for scanning the source code file and converting the syntax into an abstract syntax tree (AST). The AST structure represents the code structure more candidly to analyze the syntax components. Every component is a node; after the AST is built, a node visitor class reads the data node by node.

   In Java, the classifications in an abstract syntax tree (AST) generally correspond to the language's syntax and grammar rules. (Hongliang Liang, 2019)
In Java, the classifications in an abstract syntax tree (AST) generally correspond to the language's syntax and grammar rules. (Hongliang Liang, 2019)

Figure 7 Example of Abstract Syntax Tree Data Structure

The most common classifications in a Java AST parser, according to the book JavaParser: Visitor are:

- **CompilationUnit**: The top-level node in the AST, representing the entire Java program.
- **PackageDeclaration**: A node representing a package declaration, which specifies the package that the current source file belongs to.
- **ImportDeclaration**: A node representing an import declaration, which allows classes and other elements to be referenced by their simple names rather than their fully qualified names.
- **TypeDeclaration**: A node representing a class, interface, or enum declaration, which includes the name of the type, its modifiers (such as public or private), and its members.
• MethodDeclaration: A node representing a method or constructor declaration, which includes the name of the method, its return type, its parameters, and its body.

• FieldDeclaration: A node representing a field declaration, which includes the name of the field, its type, and its modifiers.

• VariableDeclarationExpression: A node representing a declaration of one or more variables in an expression context.

• Expression: A node representing an expression, which can be a literal, a variable, a method call, or an operation involving other expressions.

• Statement: A node representing a statement, which is an executable unit of code. Examples of statements include if statements, for loops, and assignment statements.

• Annotation: A node representing an annotation, which is metadata added to code that can be used by the compiler or other tools.

• VoidVisitorAdapter: A class used to visit various nodes in the AST and extract information from them.

• BlockStmt: A node representing a block of statements.

• NodeList: A node representing a list of nodes.

• TryStmt: A node representing a try-catch block.

• CatchClause: A node representing a catch clause in a try-catch block.

• MethodCallExpr: A node representing a method call expression.

• ClassOrInterfaceType: A node representing a class or interface type.

• Type: A node representing a type.

• ReferenceType: A node representing a reference type.

• VariableDeclarator: A node representing the declaration of a variable.
• ConstructorDeclaration: A node representing the declaration of a constructor.
• InitializerDeclaration: A node representing the declaration of an initializer.
• LambdaExpr: A node representing a lambda expression.

These classifications are going to guide to implement the parser and rules of the scanner. It will help delimit the data flow and filter information of the source code to find the right syntax and parameter. (Tomassetti), (Nicholas Smith, 2021)

4. **Scanner:** The scanner is a crucial component of this project. It will be responsible for reading the query file in a Java extension, which the user will drop into the scanner, and passing it on to the compiler and parser for analysis. The scanner will apply a set of pre-defined rules and algorithms to the source code file to identify the potential SQL injection vulnerabilities. Once the scanner has completed its analysis, it will pass the results on to the analyzer.

5. **Analyzer:** The analyzer will be responsible for processing the results of the scanner and determining whether they represent actual issues in the code on which line and adding them into the data structure where all the errors will be displayed and stored and display the warnings.

6. **Warnings:** They will be reports that will warn the user the issues the scanner and analyzer found. They will be displayed in human-readable format with detailed information about each issue, such as its location in the code and why it is recommended to fix. They will be constructive in a convenient and educative manner so the software developer can understand and also learn about the issues in the code.

7. **Rules and algorithms:** The scanner will apply a set of pre-defined rules based on the reviewed theory previously to search for patterns in the source code that could be indicative
of SQL injection vulnerabilities. These rules will be based on best practices for preventing SQL injection attacks, such as parameterized queries and string concatenation.

8. **Output and visualization:** The final component of the scanner is the output and visualization. Once the scanner has completed its analysis, it will provide a report detailing any vulnerabilities found in the code. The report will include information on the location of the vulnerability, the code causing the vulnerability, and recommendations for remediation. The report will be presented in a human-readable format, such as an HTML file, to make it easy for developers to understand and address the issues identified by the scanner.

**DIAGRAMS AND OTHER VISUAL AIDS TO HELP ILLUSTRATE THE CONCEPTUAL MODEL**

![Figure 8 Process of the Static Source Code Scanner](image)

**CHAPTER 4: METHODOLOGY**

**DESCRIPTION OF THE RESEARCH METHODOLOGY AND DESIGN**

Altogether, the research procedure of building a static source code scanner in Java from zero involves a combination of literature review, design, implementation, evaluation, and dissemination of results, aiming to improve the quality and security of software systems.
This methodology starts with the identification the problem domain, which contains issues such as security vulnerabilities, coding errors, or the performance of existing scanners. Because of evidence in diverse researches of static source code scanners in the past, the results were that most of the tools were not friendly for junior developers; they can be discouraging and difficult to understand.

Once the problem domain is identified, the next step is to design the scanner; this will be held by recollecting data and researching the architecture of source code scanners and their functionalities. This involves deciding on the types of issues to be detected, which are SQL injection vulnerabilities in queries, the programming languages to be supported, in this case, Java. The analysis techniques to be used like exploring the nature of SQL queries, studying the vulnerability by itself and how it occurs, also determine the specific packages, IDES and environments to use like the connection of SQL in Java using the Java Database Connectivity.

Then we proceed with the implementation and coding of the scanner, which consists of the third step. This follows the import and use of the Java Parser into the Eclipse IDE, which is a popular library for parsing and analyzing Java source code. Download the tool and learn how to use it and implement it in Java projects. This stage is also about writing the code to traverse the abstract syntax tree (AST) generated by Java Parser, ensuring the compiler works appropriately, creating the visitor with the correct methods, calls, and functions, and studying the hierarchy of the nodes in Java grammar.

Then we evaluate the code and ensure it was built correctly and functions efficiently. For this step, we implement and build vulnerable SQL query files that would be scanned and observe whether the scanner found the patterns that we are targeting on the query. This includes implementing and researching the Java Database Connection library, importing SQL packages in Java, and
determining the correct method calls. Right after the scanner reads the source code query, the software developer must get the correct warning on their display if vulnerabilities exist.

If the scanner is not working correctly and generating the correct output, it will be fix based on the evaluation results. This can be done by tweaking its parameters, determining the correct variables, locating the right blocks, improving its analysis techniques, or addressing any issues identified during the evaluation.

Finally, we will proceed with the analysis and publication of the results of this paper and evaluate whether all of our research questions were answered. The research method we follow will involve the study's written analysis, conclusion, and predictions so that future cybersecurity academics can benefit from the final findings.

EXPLANATION OF THE TOOLS AND TECHNOLOGIES USED IN THE IMPLEMENTATION OF THE STATIC SOURCE CODE SCANNER

Several tools and technologies are available in Java to construct a static source code scanner. Here are the ones that will be used and implemented to construct this static analyzer, are:

- **Java Compiler**: A Java compiler will assemble the Java file into bytecode. The scanner will analyze this bytecode to detect potential security vulnerabilities, bugs, or performance issues. (How To Write Your Own Compiler, 2014)

- **Apache Maven**: This built-in automation tool helps manage and execute the process of coding formation and packaging projects, particularly those written in Java. It is designed to manage project dependencies, build, test, and uniformly deploy Java applications. This software uses project object model (POM) files to manage a project's configuration, dependencies, and build process. The POM file is an eXtensible Markup Language file (XML) used to store and transport data in a structured format. This file contains
information on the Maven project, such as the project's group ID, artifact ID, version, and dependencies. To use Maven and Java Parser in Eclipse, we must download the necessary dependencies for the parser library, compile the source code, run tests, and package the application into an executable file (such as a JAR or WAR file). This tool is widely used in the Java community and will be integrated into an IDE such as Eclipse, IntelliJ IDEA, or NetBeans. (Tomassetti)

- **Abstract Syntax Tree (AST) Parser:** An AST parser will be applied to parse the Java source code and generate an AST. This will represent the structure of the code and form it into a tree with parent and child nodes. It can be used to analyze the code for potential issues in nodes. The library we are going to implement is the Java Parser library. (Cherry, et al., 2020)

- **Design Patterns:** Two design patterns for three classes. The visitor class will call the methods to visit the specific node to check, analyze and filter the code. The scanner class will contain the GUI components, and the file scanner feature, start the parsing process, and return the results displayed on the user interface.

- **Algorithms and Data Structures:** Two data structures will be used to analyze the file and detect potential issues in the code. One will be the AST, and the other will be string List arrays that will store the vulnerabilities found based on categories. The algorithms to apply the security rules were also made specifically for this research.

1. **String Literal Expressions with Special Characters**

   This rule identifies if any string literal expressions contain special characters such as "-" or "" that can be used in SQL injection attacks. The rule generates a warning message
and recommends removing these characters if found. (Maha Alghawazi, 2022), (Anley, (more) Advanced SQL Injectio, 2002), (OWASP, SQL Injection Prevention, n.d.)

2. **String Concatenation with Vulnerable Methods** This rule checks for string concatenation with vulnerable methods such as executeQuery, executeUpdate, and executeBatch, among others. If found, the rule generates a warning message and recommends reviewing the code to ensure that any user-supplied data is properly validated and sanitized before being passed to these vulnerable methods. (Boneh, 2009), (Maha Alghawazi, 2022)

3. **UNION-Based SQL Injection Attack**

This rule identifies if the code is vulnerable to a UNION-based SQL injection attack. If found, the rule generates a warning message and recommends using parameterized queries or prepared statements instead of constructing queries by concatenating user input. (Yunus, 2018), (Devi, 2016), (Yunus, 2018)

4. **Operator Plus Used in Concatenation**

This rule identifies if the operator plus is used in concatenation, which can potentially leak sensitive information. If found, the rule generates a warning message and recommends replacing the operator and reducing sensitive information used in the concatenation. (Singh), (SQL Injection Cheat Sheet, n.d.)

5. **Using Statements Directly in SQL Queries**

This rule identifies if there are any statements passed directly into SQL queries without proper validation or sanitization, which can introduce vulnerabilities into the program that could compromise its security. If found, the rule generates a warning message and recommends using prepared statements or parameterized queries, which separate the
query logic from the user input and ensure that the data is adequately sanitized before execution. (SQL Injection Cheat Sheet, n.d.), (OWASP, Injection Prevention Cheat Sheet in Java, n.d.), (OWASP, Query Parameterization Cheat Sheet, n.d.) (Owasp, n.d.)

6. **Variables Declarations** - This rule supports other set of rules by checking for variable declarations related to statements and prepared statements. If found, the rule generates a warning message and recommends using prepared statements or parameterized queries, which separate the query logic from the user input and ensure that the data is correctly sanitized before execution. (SQL Injection Cheat Sheet, n.d.), (OWASP, Injection Prevention Cheat Sheet in Java, n.d.), (OWASP, Query Parameterization Cheat Sheet, n.d.)

- **Regular Expressions**: Regular expressions will detect malicious patterns in the code that may indicate potential query issues. One example is misusing the prepared statement declaration, a primary signal of insecure code practice. We will also search for string concatenations and user input that is not parametrized.

- **IDEs**: The Integrated Development Environment (IDEs) will be Eclipse. This will provide built-in tools for analyzing Java source code. It is user-friendly and can be used with the required libraries like Java Parser and SQL.

The abovementioned tools will be the technologies required to build the static source code scanner.

**DISCUSSION OF THE TESTING AND EVALUATION STRATEGY**

This stage includes several phases to ensure the scanner will effectively detects potential issues. These are:
• **Testing Environment:** The testing environment will be ten weak SQL queries that simulate real-world conditions and include or exclude specific methods or variable calls that cause SQL injection attacks. Each query will be isolated, secure.

• **False Positive Rate:** This metric will measure the percentage of non-vulnerable code the scanner will flag as vulnerable in all the test cases.

• **False Negative Rate:** This metric will measure the percentage of vulnerable code the scanner fails to detect. A low false negative rate indicates that the scanner effectively detects vulnerabilities and does not miss important issues.

• **Precision:** This metric measures the proportion of true positives (vulnerabilities detected by the scanner) among all the alerts generated by the scanner.

• **Human Review:** Security experts will review the scanner's results to validate the findings and assess the severity of any vulnerabilities detected.

• **Continuous Improvement:** The scanner will be continuously updated and improved to keep up with evolving attack techniques and new SQL injection vulnerabilities.

All these testing and evaluating techniques will be applied during the evaluation stage and while the scanner is being developed to ensure that the data flow and logic are correct and works effectively.

**CHAPTER 5: IMPLEMENTATION, PRODUCTION AND RESULTS**

**DESCRIPTION OF THE IMPLEMENTATION AND PRODUCTION OF THE STATIC SOURCE CODE SCANNER**

The development process in this project will be divided into three parts: This first part will describe the step where the software tools like Maven and the Java parser library are first downloaded.
The initial stage was downloading the Maven software tool as a zip file into our system. Extract the Maven code file and add the file path to our system environment variables. Since the Microsoft operating system is being used, the file path must be added to the system settings, then click on advanced system settings. A new window will be shown on the screen, and finally, click the environment variables button.

![Figure 9 Maven path file in system settings](image)

Right after the window with all the environment variables is shown, we click the new button in the System Variables field. Then we set a name for the path and paste the Maven path file in the fields just like they show in the window.

![Figure 10 System variables windows to paste file path](image)
Following this, we open the Eclipse environment, create the project and the classes and select File > New > Other > Maven > Maven Project. We click the "Create a simple project" option and click next. We enter the group ID, artifact ID, and version. The project is called CarlaParser, the package is JParser, and we click Finish.

![Maven project Group ID and Artifact ID](image1)

Figure 11 Maven project Group ID and Artifact ID

Then the parser library must be added as a dependency in the Maven pom.xml file for the project, and add the Maven coordinates for the parser library as a dependency. Next, save the pom.xml file, and Maven will automatically download the library and its dependencies from the central Maven repository and add them to the project's class path.

![Maven dependencies added to the Scanner Project](image2)

Figure 12 Maven dependencies added to the Scanner Project
After saving the Java Parser library in the project, click right on the project icon in the Project Explorer view and select Maven > Update Project. This action will ensure that Eclipse reads the new dependency.

The second part of this research is creating the classes required for the project. The whole source code scanner will be divided into two classes. The main class will be called Scanner and VisitorScannerFinal.

The class Scanner, implements the SQL injection scanner which is the VisitorScannerFinal class. It includes methods for parsing Java code, identifying potential SQL injection vulnerabilities, and providing a graphical user interface (GUI) for the user to interact with the scanner.

The class imports several packages including java.io, java.awt, javax.swing, and com.github.javaparser. The java.io package provides classes for reading files and input streams, while java.awt and javax.swing packages provide classes for creating graphical user interfaces (GUIs). The com.github.javaparser package provides classes for parsing Java source code.

![Figure 13 Scanner class that implements the GUI elements for Scanner](image)

The main() method of the Scanner class initializes the GUI for the scanner by creating a JFrame object and setting its properties such as size, background color, and icon image. It also creates a
JButton object for initiating the scan and a JPanel object for the drop file feature, which allows the user to drag and drop a Java file for scanning. The method also creates a JTextArea object for displaying the source code of the scanned file and a JTextArea object for displaying the result of the scan.

![Figure 14 Applications of the dropping and button features to scan the file from the user](image)

Continuously, a DropTargetListener is created to listens to drag-and-drop events on the scanner. When a user’s file is dropped on the panel, the drop() method of the listener is called. This method accepts the dropped file and saves it to the program's File variable 'droppedFile'. It then calls the SourceCode() method which reads the contents of the dropped file and displays it on the application's JTextArea 'textSourcecode' within a JScrollPane. The displayed code will be highlighted and formatted to improve readability using various font and color settings.
Figure 15 Method calls to parse and compile the source code file

This part of the program defines an action listener for the "Scan" button, which triggers the scanner and parser when clicked. It first checks if a file has been dropped onto the drop panel. If no file is found, an error message is displayed using the JOptionPane class. If a file is found, the JavaParser library is used to parse the file and the parse result is stored in the CompilationUnit object. The parse result is checked for success and an error message is displayed if parsing was unsuccessful. If successful, the CompilationUnit object is used to begin the scanning process.
This part of the program is responsible for scanning and parsing the file dropped by the user, using the VisitorScannerFinal class to extract statements, concatenations, and prepared statements from the parsed file.

The extracted statements and concatenations are checked for potential errors, and the appropriate warning messages are displayed in the text area. Specifically, if any statements or concatenations can potentially lead to SQL injection attacks, the corresponding error messages are displayed.

This part of the program also checks if any prepared statements are present. If there are no prepared statements, a warning message is displayed, and the recommended fix is to add prepared statements to the query.

Finally, the program adds the drop panel and button panel to the frame, making it visible for the user.
The second class of the program, the VisitorScannerFinal class implements extends the VoidVisitorAdapter class, which is a visitor for AST nodes. The purpose of this class is to visit Java code and identify potential SQL injection vulnerabilities.

The class implements three private lists named statements, concatenation, and prepared statements. These lists are used to store the vulnerable code found during the visit. The class also defines two patterns, StatementClass and PreparedStatement, which are used to identify vulnerable code.

The class contains three methods: getStatements(), getConcatenation(), and getPreparedStatements(), which return the respective vulnerable code lists.
Then, the class defines a subclass of VoidVisitorAdapter called OuterVisitor, which overrides the visit method for AnnotationDeclaration and BlockStmt nodes. The visit method of AnnotationDeclaration nodes calls the visit method of InnerVisitor on the node, which is a nested class defined within OuterVisitor. The visit method of BlockStmt nodes also calls the visit method of InnerVisitor.

The VisitorScannerFinal class also defines several methods that are used to check for vulnerable code patterns. isStringConcatenation checks if an expression is a string concatenation using the BinaryExpr class and the plus operator. isStringLiteral checks if an expression is a string literal using the StringLiteralExpr class.

The class also defines a Set called processedExpressions, which is used to keep track of which string literals have already been processed to avoid duplicates. The visit method for StringLiteralExpr nodes checks if the string literal contains special characters such as "--" or "'", and adds a warning message to the concatenation list if it does. The warning message includes the expression, line number, and recommended fix.
Figure 19 Class OuterVisitor and method calls to check for vulnerabilities

The visitor defines patterns, based on the rules that were previously mentioned, of code structures that indicate vulnerable SQL queries, such as the usage of certain JDBC methods (executeQuery, executeUpdate, etc.) and the presence of string concatenation. It checks for the concatenation of string values that have special characters or that are being executed with vulnerable methods, and adds any vulnerable code found to specific lists (statements, concatenation, preparedStatements) that will later be used to display warnings and recommendations to the user.

The visitor also defines a method to check for vulnerable createStatement methods that are vulnerable to UNION-based SQL injection attacks. If such a vulnerability is found, the visitor adds the vulnerable code to the concatenation list.
The subsequent part of the code, checks if the current method call is the executeQuery method. If so, it checks if the first argument of the method call is a string literal expression that contains a single quote ('). If it does, it adds a warning message to the concatenation list, indicating that the code is vulnerable to SQL injection attacks and recommending that all single quotes in the user input string be deleted.

The next if statement checks if the current method call is either getParameter or getQueryString, both of which are commonly used to extract parameters from HTTP requests. If so, it adds a warning message to the concatenation list, indicating that the code is vulnerable to SQL injection attacks and recommending that all single quotes in the user input string be deleted.

The visit(BinaryExpr n, Void arg) method checks for the usage of the + operator in string concatenation expressions. If the operator is used, it adds a warning message to the concatenation list, indicating that the code is potentially exposing sensitive information and recommending that the operator be replaced and sensitive information used in the concatenation to be reduced.
The following code block overrides the visit method for VariableDeclarator nodes in the abstract syntax tree. It processes variable declarations outside of a try block and checks if they are of type `Statement` or `PreparedStatement`.

If a variable declaration is of type `Statement`, it adds a warning message to the statements list to indicate that using statements directly in SQL queries without proper validation or sanitization can introduce vulnerabilities that could compromise the program's security. It also provides a recommended fix of using prepared statements or parameterized queries, which separate the query logic from the user input and ensure that the data is properly sanitized before execution.

If a variable declaration is of type `PreparedStatement`, it adds the declaration to the `preparedStatements` list. The code also checks if a variable declaration is equal to a `VariableDeclarationExpr` instance of `StatementClass`. If so, it adds a warning message to the statements list with the same information as described above.
Finally, the code calls the visit method of the superclass to continue visiting the rest of the nodes in the tree.

![Figure 22 Code that searches for PreparedStatements or Statement declarations](image)

The final part of the code defines the visitor that traverses the AST nodes inside the try block, catch clause, and finally block. It checks the nodes for potential SQL injection vulnerabilities and adds warnings and recommended fixes to the concatenation, statements, and preparedstatements lists. The visit(TryStmt n, Void arg) method visits the try statement and its associated catch and finally blocks. It gets the try block and its child nodes and passes them to the InnerVisitor to be visited. If it exits it also gets the finally block, and passes its statements to be visited as well.

The InnerVisitor class is an inner class that extends VoidVisitorAdapter<Void> and contains the logic for visiting each node type inside the try, catch, and finally blocks. It overrides the visit methods for MethodCallExpr, StringLiteralExpr, BinaryExpr, and VariableDeclarator, that are the same methods and logic applied in the outer visitor class, to check for SQL injection vulnerabilities and add warnings and recommended fixes to the appropriate lists.
The project took longer than expected because of unfamiliarity with the types of node on Abstract Syntax Trees.

Ten test cases that include different types of vulnerable code, such as SQL injection vulnerabilities or string concatenation vulnerabilities, will be scanned to test this scanner. This malicious code will be run with the scanner to verify that this works correctly, identifies the vulnerabilities, and provides appropriate recommendations for fixing them.

When testing this static source code scanner, there must verification of clarity and accuracy of error messages that identify the correct vulnerability and provide actionable recommendations to fix them.

The first object that will be tested is the GU, the drop feature and the scan me button. The file drop and scan me button work efficiently and the source code to be scan is shown on the screen.
The following table shows the results of the ten vulnerable queries tested twice, each with a different and random type of vulnerable code based on the security rules applied in the program that will trigger the error warning. Eight of the queries were customized, one was a malicious query found on Gihub projects, and the last was a secured query.

The total scannings of vulnerable queries were twenty. The total of vulnerability injections was sixty-two in total, however only forty-nine were scanned correctly and thirteen were false negatives. One optimistic result was that there were zero false positive results.
The first problem testing the program was the output not showing the Statement and PreparedStatement declarations. This was because the functions in the parser were not calling the correct nodes of the AST. The method of scanning these variables was as a node type, not a node value.

Table 1 Scanning Results

<table>
<thead>
<tr>
<th>Unit Test Number</th>
<th>Malicious code injected</th>
<th>Malicious code detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Query 2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Query 3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Query 4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Query 5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Query 6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Query 7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Query 8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Query 9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Query 10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2 False Negatives chart

![False Negatives Chart](image.png)
Another issue was with query number eight, which showed a parsing error. After specific analysis, the problem was the syntax error in the import of the package in the query, which created an error in the query's logic, confused the parser, and stopped the creation of the AST.

Table 3 Parsing Errors Chart

One error in the display of the results was occurrence of two vulnerabilities happening on the same line of syntax, and repeating itself inside the try and catch blocks and the outside blocks, triplicating the results and messages on the display. This had to fixed I the logic of the program. Another challenge of implementing the scanner was to create an interface that can be user-friendly for junior developers and discover a mode to educate about the errors and recommend the fixings. Because of this, the GUI is separated into two panels, one that displays the source code that the developer would like to scan and the other panel to display the scanning results.
As seen on the image above the errors are displayed correctly, the warnings are displayed in right location. There are no error repetitions and second round of the testing there were no false negatives produced.

ANALYSIS OF THE PERFORMANCE AND EFFECTIVENESS OF THE SCANNER

The analysis and performance of this static scanner will be analyzed under four parameters.

- **Accuracy:** High. The scanner detects vulnerable SQL statements and reports them to the user on the error panel. This research aimed to build a static source code scanner specialized in SQL injection vulnerabilities. The scanner checks for various types of SQL injection vulnerabilities pre-defined in the security rules, such as string concatenation, vulnerable methods, vulnerable statements, and missing prepared statements. It details the types of vulnerabilities detected and where they are in the code. The rate of false negatives was high in the first round of testing. Nevertheless, this rate decreased to zero in the second round because the errors and bugs in the scanner were fixed.
• **Efficiency**: High, given that the scanner can scan the code in a reasonable amount of time without consuming excessive resources and was tested with long malicious queries of more than 200 lines of syntax. The scanner uses the JavaParser library to parse the code and correctly identify potential vulnerabilities in the given code. However, it has a high learning curve, and it takes time to read and classify all the node types the scanner is looking for. One example was the Statement and PreparedStatement declarations.

• **Usability**: Medium-High, the scanner is user-friendly and easy to use. The graphical user interface provides drop listeners and displays in an educative manner the recommendable fixings of the errors found. The user interface was designed and produced to provide a more intuitive and user-friendly experience. Users can drag and drop a file onto the GUI and click the "scan file" button to initiate the scanning process. It does not need specific installation requirements, and it can help software developers that do not have expertise or knowledge in SQL Injection attacks. Nevertheless, the tool scopes for the moment is only SQL Injection vulnerabilities.

• **Extensibility**: Medium. The scanner can be extendable to cover additional types of vulnerabilities or to support other programming languages. Nevertheless, it would require more profound research and a literature review of the specific vulnerabilities it would have to cover.

This static source code scanner effectively detects SQL injection vulnerabilities in Java code and is user-friendly for software developers, regardless of their expertise level. However, like any tool, it may not be perfect and should be used with other security measures to ensure the safety of the code.
CHAPTER 6: DISCUSSION

INTERPRETATION OF THE RESULTS IN THE CONTEXT OF THE RESEARCH QUESTIONS AND OBJECTIVES

In this section, the achievement of the objectives will be analyzed by profoundly answering the initial research questions asked at the beginning of this study.

The main research question was

- "Can a static source code scanner be developed in Java using a parser that effectively identifies common SQL programming errors and vulnerabilities?"

Based on the results and analysis, there is certainty that a static source code scanner can be developed in Java using a parser to effectively identify common SQL programming errors and vulnerabilities like string concatenations and prepared statements.

About sub-questions of the study:

- What are the requirements and steps for a static source code scanner from scratch?

The requirements and steps to build a whole static source code scanner from zero are first to determine the types of vulnerabilities that the scanner aims to find. Knowing and learning the concepts of lexical grammar or abstract syntax trees would make the visiting method calls and the categories of nodes simpler to understand. Additionally, this knowledge provides a more in-depth scope of the parameters that need to flow in the visitor class. Furthermore, constructing and finding the test units to verify the scanner's accuracy is a main requirement, because it defines the application of security rules.

- What are the most common SQL Injection vulnerabilities that a static source code scanner can identify?
The most common types of attacks and vulnerabilities identified with source code scanners are the initialization of variables that may be vulnerable, like statements or PreparedStatements. Additionally, combining specific patterns like using vulnerable methods like executeQuery with a string expression that uses a binary symbol or string concatenations that may come from random user input.

• What are the advantages and disadvantages of using a parser-based approach to static analysis?

This research does prove that a parser library can improve the scanner's performance and accomplish more in-depth source code scanning in an efficient manner, that saves time for the software developer that is looking to build his/her own static source code scanner. The advantages of using a parser to build a static source code scanner are the improvement and efficiency of the code by examining data flow and dividing the source code into data structures to make it more accessible and organized. Additionally, building an abstract syntax tree provides a better classification of the variables, method calls, and combinations of the syntax the scanner aims to find, increasing the accuracy of the results and making the scanner more reliable in finding the patterns of the security rules. Finally, implementing a parser library will automate the scanning process, making the tool more friendly and saving time and resources for the user and developer.

Nonetheless, enforcing a parser has disadvantages, like the implementation's complexity. Using a parser library can be very difficult if there is no previous experience. Understanding all the classification nodes, implementing the super visit, and determining the exact pattern the scanner needs to find can have a high learning curve.

Additionally, the production of false negatives and positives can be high if there is no debugging stage in the software development cycle.
• What are the design principles and technical requirements for developing an effective static source code scanner in Java for junior developers?

The design principles and technical requirements for developing an effective source code scanner are the acquaintance of how programming vulnerabilities and attacks work to help determine the patterns and output of the scanner, also knowing basic knowledge of Database management and its management of programming languages like SQL to uncover better and more efficient security customs for the scanner. Further, the expertise in data structures and parser libraries to understand how data flow works after the parser compiles the document.

• How can the scanner be evaluated and validated to ensure its effectiveness and accuracy in detecting SQL vulnerabilities?

The best way to evaluate static source code scanners is to develop various unit tests with different vulnerabilities to test every security rule the analyzer applies. Specifically, the unit tests with SQL injection code with a combination of different statements or the absence of variables that may weaken the queries. Therefore, knowledge of pattern combinations is a convenient resource to assess and ensure the scanner’s accuracy. Consider the user input data flow and vulnerable code inside and outside try-and-catch coding blocks.

• What is the prevention procedure to defend queries from SQL injections?

Overall with all this research, it is concluded that the best static prevention of SQL injection in queries is the detection of patterns that create string concatenations with unique characters that can enter the database and make it vulnerable to leaking any additional data. Always remember to apply parameterized queries by customizing the use and declaration of PreparedStatements instead of statements so these variables accept only secured user input.
• How can this static source code scanner be more user-friendly than the current source code scanners?

Yes, it can. By simplifying the user’s experience of the scanner, always taking into account the time and expertise that the user has. Also, by making the results more detailed, while adding educational facts about the vulnerabilities detected that showed why they happen and how to fix them.

Focusing on the presentation of the Graphic User Interface, is also a good method. Given that, it is the main ground of interaction between the user and the scanner. The addition of the feature “Drop and Scan file” is helpful in making the user more comfortable to use the scanner, because it effectively reduces the time spent trying to learn the tool.

• How can this static source code scanner help minimize the technical debt of developers?

By making their experience simpler and less complicated. This scanner can minimize the learning curve of the tool. The “Drop and Scan file” feature makes it a friendlier application. The display of the results are educational, so the developer will keep in mind the vulnerabilities and preventions that he/she can apply while programming.

DISCUSSION OF THE STRENGTHS AND LIMITATIONS OF THE SCANNER

The strengths of this scanner are that:

The scanner can detect potential SQL injection vulnerabilities by checking for vulnerable methods such as executeQuery and executeUpdate, and by checking for the presence of the UNION keyword in SQL queries. It can search for string concatenations with vulnerable methods, which can be used in SQL injection attacks. Additionally find statements directly in SQL queries, which can lead to security vulnerabilities. And it can identify string literals containing special characters such as "--" or "'", which can also be used in SQL injection attacks. It can detect the use of the "+"
operator in concatenation, which can potentially leak sensitive information and finally it is a user-friendly tool, it doesn’t need any requirement or previous knowledge about secure programming to be used. It has a “Drop file and click scan” method, which is extremely simple to use. Additionally, it has educational feature in the warnings displayed.

However, the limitations of this static source code scanner is that it may not detect all instances of vulnerable code, as it relies on patterns and regular expressions to identify potential vulnerabilities. It could potentially generate false positives or false negatives. It may be unable to detect vulnerabilities in more complex code structures, such as nested loops or conditionals.

**SUGGESTIONS FOR FUTURE AND FURTHER RESEARCH IN THE FIELD OF STATIC SOURCE CODE SCANNER**

The suggestions for future research in static source code scanner it is important to acknowledge that not all software developers have equal levels of expertise or experience in security practices. Therefore, designing static source code scanners that cater to a wider range of users and provide guidance on how to fix identified vulnerabilities can be helpful. This could include providing detailed explanations of the detected vulnerabilities and suggested fixes, as well as offering automated solutions for fixing certain types of vulnerabilities.

In addition, integrating the educational aspect into the static source code scanning process can also benefit more experienced developers. Providing them with access to relevant resources, such as up-to-date documentation and best practices, can help them stay informed about emerging threats and improve their ability to write secure code.

Finally, another area for future development is to explore ways to incorporate machine learning and artificial intelligence techniques into static source code scanning. This could potentially improve the accuracy and effectiveness of identifying and mitigating vulnerabilities, while
reducing the risk of false positives. This approach could also enable the scanners to adapt and improve their accuracy over time as they learn from analyzing more code samples.

CHAPTER 7: CONCLUSION

DISCUSSION OF THE LIMITATIONS OF THE STUDY FOR THE FIELD OF STATIC ANALYSIS

The limitations of this research were the limited time. If the timeline of this project had been longer, there would be more accuracy in the results of the scanner, and it would have reduced the potential numbers of false positives or false negatives. Another implication of this research was the absence of previous knowledge about how to start the process of a static source scanner. Much of this research was dedicated to reviewing literature and previous investigations that may explain how to start, the basic requirements for building a static source code scanner, and how parsers and lexical analyses work.

The profound expertise required in programming rules for vulnerabilities was also a challenge, given that specific attacks on web applications are specialized and can include various parts and fields of programming like front-end along with back-end procedures that need diverse programming languages and have a dynamic nature. Finally, the implication of designing the security rules and patterns the scanner will apply to prevent specific attacks like SQL injections was a prominent assignment because of the active dataflow that the vulnerabilities, variables, and methods calls have in the source code, along with the random user input. Delimiting the security rules to apply in the analyzer to gain high accuracy is challenging. Certain risks appearing in arbitrary threatening code but not shown in others will likely be missed.
SUMMARY OF THE KEY FINDINGS AND CONTRIBUTIONS

The static source code scanner for SQL injection vulnerabilities in Java provides a comprehensive approach to identifying potential vulnerabilities in code. The scanner can detect vulnerable code by analyzing string concatenation, prepared statements, vulnerable methods, and user input. The scanner provides a list of found vulnerable codes and recommendations on how to fix them.

One of the key findings of this scanner is that using statements directly in SQL queries without proper validation or sanitization can introduce vulnerabilities into the program that could compromise its security. The scanner recommends using prepared statements or parameterized queries to separate the query logic from the user input and ensure the data is correctly sanitized before execution.

The contribution of this scanner is to provide a tool that junior developers can effortlessly use to identify and fix potential vulnerabilities in their code, no matter their expertise in secure programming. The scanner also provides educational recommendations for developers to understand better the causes of vulnerabilities and how to manage them. Overall, the scanner improves software security and promotes secure development practices.
REFERENCES


APPSONAR. (2023). *APPSONAR*. Retrieved from APPSONAR:

https://www.appsonar.com/documentation


Caitlin Sadowski, E. A.-C. (2018). *For a static analysis project to succeed, developers must feel they benefit from and enjoy using it*. Communications of the ACM.


https://owasp.org/Top10/A01_2021-Broken_Access_Control/


https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_in_Java_Cheat_Sheet.html

OWASP. (n.d.). *Query Parameterization Cheat Sheet*. Retrieved from OWASP:
https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html


OWASP. (n.d.). *SQL Injection Prevention*. Retrieved from OWASP:
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

Phillips, R. (2023, May 27). *The top 15 most used software packages used by developers*. Retrieved from Platformexecutive:


practices-for-taking-on-the-sql-server-dba-role-as-a-
developer/#:~:text=Best%20practices%20for%20taking%20on%20the%20SQL%20Server,PRIMARY%20KEYs%20in%20your%20database.%20...%20More%20items


*What is software security and why is it important(7,803),(992,995)?* (2021, May 19). Retrieved from Contentful:
https://www.contentful.com/blog/software-security-to-deliver-digital-experiences-fast/


