

Master of Science in Software Engineering
February 2025



An Analysis of Generative AI Capabilities in Security Testing

Evaluating Static Code Analysis Performance

Rohan Sharma

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

Rohan Sharma

E-mail: rosh22@student.bth.se

University advisor:

Oleksandr Adamov

Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Background – In today's technology-driven world, ensuring the security of software systems is paramount due to increasing dependency on these systems across all sectors. Security testing, specifically static code analysis, plays a vital role in detecting vulnerabilities before they are exploited. Traditional static analysis tools, such as SonarQube, often struggle to detect complex vulnerabilities, prompting the exploration of Artificial Intelligence (AI) for enhanced security testing.

Objectives – This thesis aims to evaluate the performance of two Generative AI models, ChatGPT and Gemini, in static code analysis for security testing and compare these AI models with each other and with a traditional static code analysis tool, SonarQube, to determine their effectiveness in detecting software vulnerabilities.

Methods – Method used in this thesis is experimentation which enabled me to gather empirical evidence through a controlled environment with controlled variables. It enabled me to compare the performance of ChatGPT, Gemini & SonarQube, this comparison also helped me in identifying a superior performing model.

Results – Both AI models outperformed SonarQube in vulnerability detection. ChatGPT demonstrated slightly better performance in identifying the specific code responsible for vulnerabilities compared to Gemini.

Conclusions – Through the course of this thesis it has become evident that GenAI models offer solid performance when it comes to static code analysis in vulnerability assessment. They show promise and have presented their case by showcasing their superior performance, that they are very much able to assist or even replace the traditional SAST tools in some scenarios.

Keywords: Security Testing, Static Code Analysis, Vulnerability Detection, Generative Artificial Intelligence

CONTENTS

ABSTRACT	III
CONTENTS	V
1 INTRODUCTION	3
1.1 BACKGROUND	3
1.2 DEFINING THE SCOPE OF THIS THESIS	4
1.3 OUTLINE	5
2 RELATED WORK	6
3 METHOD	10
3.1 DEFINING RESEARCH QUESTIONS	10
3.2 SYSTEMATIC LITERATURE REVIEW	10
3.2.1 <i>Inclusion Criteria</i>	11
3.2.2 <i>Exclusion Criteria</i>	11
3.3 DATASET	11
3.4 GENERATIVE AI MODELS AND SAST TOOL	14
3.5 DESCRIBING RESEARCH METHOD	14
3.6 SETUP	15
3.7 VALIDITY AND RELIABILITY OF PROCEDURE & APPROACH	16
3.8 THREATS TO VALIDITY	18
4 RESULTS AND ANALYSIS	20
4.1 RESULTS	20
4.2 ANALYSIS	21
4.2.1 <i>Vulnerability Detection</i>	22
4.2.2 <i>Vulnerable Code Detection</i>	23
4.2.3 <i>Detailed Analysis of Results</i> –	23
4.2.4 <i>Analysis of Systematic Literature Review</i>	25
5 DISCUSSION	30
6 CONCLUSION AND FUTURE WORK	32
6.1 CONCLUSION	32
6.2 FUTURE WORK	33
REFERENCES	34
APPENDIX	36

1 INTRODUCTION

1.1 Background

“I have never found it hard to hack most people. If you listen to them, watch them, their vulnerabilities are like a neon sign screwed into their heads.”

- *Elliot Alderson*

These statements from the television series Mr. Robot are more relevant than ever in the current digital era. However, for the purposes of this thesis, "them" refers to the websites and software that have become an essential part of our everyday lives rather than to actual individuals. In today's time most of the businesses have gone digital, using the comforting services of internet and digital age, which has led to a dependence on computers and software. Because these systems are responsible for handling important functions like financial transactions and managing vital infrastructure, protecting such software becomes a responsibility and priority for people, companies, and governments.

Despite computers having proven their usefulness, there still are serious risks associated with them. From simple virus programs to complex attacks that take advantage of ransomware, zero-day vulnerabilities, and social engineering strategies, cyber dangers have changed throughout time. Due to the growth of cyberattacks, all computer systems are now vulnerable to attacks that can be used for a variety of objectives, including financial fraud, data theft, and even damage [1].

In a Software Development Life Cycle (SDLC), maintaining a secure development process can come across as a challenge. Modern development methodologies like DevOps and Continuous Integration/Continuous Deployment (CI/CD) move quickly, which might result in security lapses [2] [3]. Because of this, strong security testing techniques are now crucial more than ever. With the goal of identifying vulnerabilities early and lowering the risk of exploitation after deployment, security testing has become an essential part of the Software Development Life Cycle (SDLC). The stakes are large because a single security lapse might bring down entire networks or jeopardize private information.

A wide range of techniques are included in security testing with the goal of identifying application flaws before they can be exploited. Static Code Analysis is one of these methods [4]. It is used in finding possible security flaws in the codebase. It seeks to find security flaws that an attacker could exploit, including SQL injection, cross-site scripting (XSS), sensitive data exposure, authentication failure, and others. The ability to fix security issues early in the development process, prior to the product being released into a live environment, is another benefit of static code analysis. Resolving vulnerabilities in the development stage is significantly less expensive than patching them after production, when attackers may have already taken advantage of them. Development teams can promote a "shift-left" security culture—where security is taken into account from the outset of the project—by integrating static code analysis into the workflow. This lowers the risk of security breaches and guarantees the creation of dependable software products. SonarQube, which is an open-source platform

built for static code analysis which can find flaws, vulnerabilities and other issues in 29 programming languages is an example of such tools.

Traditional static code analysis methods do have certain drawbacks, though. Despite their usefulness, these technologies frequently have trouble identifying complicated vulnerabilities. Additionally, they frequently run into restrictions on the variety of programming languages they can use.

Our security testing procedures need to advance along with the sophistication of cyber-attacks. The shortcomings of conventional static analysis may be addressed by recent developments in artificial intelligence (AI), particularly in the form of generative artificial intelligence (GenAI) models.

Generative Artificial Intelligence, which is a field of Artificial Intelligence has the ability to produce new content that closely matches that of humans, including writing, graphics, and even code. Generative AI models are capable in producing new data samples, in contrast to standard AI models that concentrate on identifying patterns and formulating predictions from preexisting data. Deep learning, a branch of machine learning that uses multi-layered artificial neural networks to analyze and learn from massive volumes of data, is at the core of generative AI models. To discover the underlying patterns and structures, generative AI models—like Generative Pre-trained Transformers (GPT) and its variations, ChatGPT and Gemini—are trained on sizable text or other data sets. This training process enables them to generate new content by predicting the next word, pixel, or character based on the context provided.

Despite being a fundamental component of security testing, traditional static code analysis is unable to keep up with the quickly changing threat landscape. Developers must embrace increasingly sophisticated technologies that can identify and prevent these risks before they become a concern, since attackers are constantly coming up with more complex ways to exploit vulnerabilities. With the ability to improve on conventional static analysis tools and deliver a more thorough, intelligent solution for software system security, generative AI presents a promising new strategy [5]. Organizations may improve software product security, lower the chance of security breaches, and maintain user trust by incorporating AI into security testing procedures.

1.2 Defining the scope of this thesis

The scope of my thesis will be limited to analyzing the capabilities of GenAI models for static code analysis in security testing. This will help us understand if GenAI models should be used more often than they are currently being used in identifying vulnerabilities. Since static code analysis does not involve executing the code, GenAI models seems to have good potential and will be fairly easy to use and implement.

The scope can further be narrowed down by understanding the two research questions:

- RQ1 – Do Generative Artificial Intelligence Models (ChatGPT and Gemini) perform better than the traditional methods for conducting static code analysis in security testing?

- RQ2 – Which Generative Artificial Intelligence model, ChatGPT or Gemini, demonstrates superior performance in security testing for static code analysis?

Being able to answer these questions, the thesis will be able to provide empirical evidence on the performances of ChatGPT and Gemini models for static code analysis which will help us understand their capabilities. Since we need some data to do a comparison between GenAI models and traditional methods we will be comparing the findings of GenAI models with the findings of SonarQube which is a tool used for static code analysis. Later we will be able to further compare both the models as well to decide which model will work best for security testers / pentesters.

Overall, the thesis will revolve around the use of Generative AI models in security testing, more specifically static code analysis. ChatGPT and Gemini will be used to detect vulnerabilities in OWASP Juice Shop and their response will be used to determine the superior model in detecting vulnerabilities. I will also compare their results with a traditional tool (SonarQube) to come to a conclusion whether GenAI models outperform traditional tools or fall short to their tested and verified performance.

1.3 Outline

This section will provide an outline of how this thesis is structured. A brief description of each section will be given.

Section 2 is responsible to showcase most of the previous work that is related to my thesis' topic. This work is important and has been helpful throughout my thesis in making decisions regarding SAST tool and GenAI models.

Section 3 contains the methodology followed for the successful completion of this thesis. It describes the detailed method and approach that has been followed by me giving details about the setup and the experimentation process.

Section 4 contains results and their analysis. Whatever findings and evidence that has been gathered because of the previous section has been put in this section along with a well thought analysis of that data.

Section 5 does a discussion based on the results and analysis and answers few important questions regarding the use of GenAI models in real world scenarios for security testing.

Section 6 gives a conclusion of this thesis in a more general sense by connecting the findings with real world and how they can be implemented or applied there. Since this is the final chapter it sums up all the findings and analysis of this thesis. Besides it talks about future work which explores further opportunities by researchers which can take my research one step forward in furthering enhancing and increasing the reliability of this research.

2 RELATED WORK

This section aims to summarize and contextualize prior works that align with or have contributed to the research focus of leveraging Generative AI for static code analysis in security testing. This includes studies that address static code analysis, security testing, or the use of generative AI models in software security. These works are either directly or indirectly related to the topic of this thesis but nevertheless provides insightful information that has been utilized or has helped throughout the duration of this thesis.

Bakhshandeh et al. [6] explored whether ChatGPT could function as a Static Application Security Testing (SAST) tool for analyzing Python code. They tested it against well-known SAST tools like Bandit, Semgrep, and SonarQube, comparing their effectiveness. Their findings showed that ChatGPT has the potential to minimize false positives and false negatives, which are common issues with traditional SAST tools. While its overall performance was similar to existing tools, it proved especially useful as an assistant alongside them. Interestingly, in some cases, ChatGPT identified vulnerabilities that other tools completely overlooked, showing its ability to recognize patterns and security flaws in code. This study closely relates to my research, as both focus on using generative AI for security testing and vulnerability detection. However, while Bakhshandeh et al. specifically analyzed Python vulnerabilities, my work takes a broader approach. I compare ChatGPT and Gemini in detecting vulnerabilities in OWASP Juice Shop, a JavaScript-based web application, using SonarQube as a benchmark.

Gomes et al. [7] took a deep dive into static code analysis in software development, focusing on how it helps catch defects early in the process—before the code even runs. Their study highlighted that identifying errors early is key to reducing the costs and risks of fixing bugs later. They looked at both manual and automated code reviews, noting that manual reviews, while effective, require significant time and expertise, which is why automated tools have become more widely used. They examined tools like FindBugs, PMD, and commercial solutions like Coverity Prevent, which scan code for predefined patterns to catch errors. However, they also pointed out a major drawback—while these tools are great at spotting common mistakes, they tend to generate false positives and can miss more complex vulnerabilities because they rely on fixed rule sets. Their conclusion? Static analysis alone isn't enough. For better security testing, it needs to be combined with dynamic analysis to cover more ground.

Yigit et al. [8] conducted a thorough review of how Generative AI (GenAI) is being used in cybersecurity, looking at both its benefits and risks. They examined how large language models (LLMs) like ChatGPT and Google's Gemini can improve cybersecurity processes but also pointed out the potential dangers—such as AI being used to assist cyberattacks. Their research provided a detailed look at how GenAI can automate tasks like threat detection, vulnerability scanning, and secure code development, ultimately making cybersecurity more efficient. While Yigit et al. explored GenAI's broad role in cybersecurity, my research takes a more focused approach. I specifically compare ChatGPT and Gemini in the context of static

code analysis for vulnerability detection. Building on their insights into AI's potential for identifying security flaws, my work applies these ideas to real-world scenarios using OWASP Juice Shop, evaluating how well these models perform in detecting vulnerabilities.

Hilario et al. [9] looked into how Generative AI (GenAI) can be used in penetration testing, checking out both its perks and the challenges it brings to cybersecurity. They focused on how tools like ChatGPT can help penetration testers through all five stages of testing, from gathering information to writing up reports. By trying out GenAI on a vulnerable machine from VulnHub, they found it really sped up the testing process and made it more accurate. But they also pointed out some concerns, like relying too much on AI and the ethical issues that come with it. Even though this paper isn't directly tied to my thesis, it brings up an important idea—it's risky to depend too heavily on AI, especially for crucial tasks like spotting vulnerabilities in code. It's a good reminder that we still need human oversight when using GenAI in cybersecurity.

Li et al. [10] explored how Large Language Models (LLMs) like ChatGPT could help improve static code analysis, especially when it comes to reducing the false positives that traditional tools often produce. They focused on whether ChatGPT could generate more accurate summaries of functions to help spot bugs, like Use Before Initialization (UBI) vulnerabilities, that static analysis tools sometimes miss. To do this, they had ChatGPT break down tasks step by step, analyzing specific parts of the code. Their results showed that GPT-4 outperformed GPT-3.5, correctly summarizing functions 94% of the time, which proved that LLMs can make static analysis more reliable.

Stefanović et al. [11] did a deep dive into static code analysis tools, looking at how they're used with both general-purpose and domain-specific programming languages. They aimed to figure out which tools are most commonly used, what defects they can find, and which languages they support. They reviewed 22 key studies and pointed out the pros and cons of tools like SonarQube, PMD, and Cppcheck. What they found was that while these tools do a good job catching common problems like code smells, duplication, and security flaws, there isn't one tool that can catch every defect across all languages. They also noted that most tools are built for general-purpose languages, with less support for domain-specific languages. One big takeaway was that we really need tools that can handle the unique challenges of domain-specific languages, which the current tools don't fully cover. Reading through this paper helped me decide which static analysis tool to use for my experiment, and as they mentioned, SonarQube is one of the most widely used and trusted tools out there.

Li et al. [12] looked at how Large Language Models (LLMs) like GPT-4 could be used to improve static code analysis, particularly for catching Use-Before-Initialization (UBI) bugs. They introduced a new framework called LLift, which aims to strike a balance between being precise and scalable—something that's often tricky for traditional static analysis tools. These tools can generate false positives or slow down when working with big, complex codebases like the Linux kernel. To address this, the authors suggest using LLMs to work alongside existing tools like UBITect. The LLMs would help tackle the more complicated cases that the traditional tools struggle with. The study showed that LLift was really effective in cutting down false positives. Using techniques like guided path analysis and task decomposition, it managed

to get an impressive 50% precision and 100% recall in real-world tests, making the bug detection process a lot more accurate.

Cheshkov et al. [13] Cheshkov et al. [8] evaluated how well ChatGPT and GPT-3 could detect vulnerabilities in Java code. They focused on binary and multi-label classification tasks using a dataset of vulnerable and patched Java code from GitHub. Their goal was to see if large language models (LLMs) like ChatGPT could accurately spot vulnerabilities in code, given ChatGPT's success with other coding tasks like solving programming challenges. However, despite the high hopes, their results showed that ChatGPT's performance was pretty much the same as a dummy classifier in both binary and multi-label tasks. In other words, it had a hard time telling the difference between vulnerable and patched code. The study pointed out that ChatGPT often predicted vulnerabilities even when there were none, leading to a lot of false positives.

Bubeck et al. [14] compared the coding abilities of GPT-4 with GPT-3 and GPT-3.5. They tested the models using benchmarks like HumanEval and LeetCode, and found that GPT-4's performance was on par with human coders, showing it could understand and generate code effectively. Their research highlights the impressive capabilities of these models, and their findings were a key inspiration for me to push these models beyond coding tasks and explore their potential in vulnerability assessment.

Chen et al. [15] looked at how well large language models trained on code can understand and generate code. They specifically focused on Codex's ability to turn natural language docstrings into Python code using the HumanEval benchmark, which tests the functional correctness of generated programs. The authors found that Codex, especially in its fine-tuned version, outperformed models like GPT-3 and GPT-J, solving 28.8% of the HumanEval problems with a single sample. However, they also noted that Codex didn't perform as well as other Static Application Security Testing (SAST) tools in detecting vulnerabilities. While our research doesn't focus on fine-tuned models like Codex, their work provides useful insights into the potential of generative models for coding tasks.

Ye et al. [16] Ye et al. [11] carried out an in-depth evaluation of six GPT models, including two GPT-3 and four GPT-3.5 versions. They assessed how these models performed on nine different natural language understanding (NLU) tasks across 21 datasets. The study concentrated on zero-shot and few-shot scenarios, looking at how the models evolved in terms of instruction understanding, robustness, and sensitivity to prompts. While Ye et al. focused on general NLU tasks, my research takes a more specific approach by narrowing in on static code analysis for detecting security vulnerabilities.

As mentioned above, all of these papers are either closely related or somewhat related to the topic of my thesis i.e. Analysis of Generative AI capabilities for Static Code Analysis in Security Testing. Nevertheless, reading the abovementioned literature has helped me in finalizing and deciding a few details regarding my thesis. Decision to pick SonarQube as the traditional SAST tool which will be used to compare performances of GenAI models with itself was finalized after doing the systematic literature review. Besides this, the papers have also helped in understanding how to conduct the experiment and how to write prompts for ChatGPT

and Gemini. But the most important set of information derived from the papers are the evaluation metrics for evaluating the performances of these models. Accuracy metric has been taken as the metric for evaluating their performances.

3 METHOD

3.1 Defining research questions

Defining clear research questions is fundamental to the success of any thesis. They will provide meaning and useful insights to my thesis. In this thesis, two research questions have been formulated to guide the exploration of Generative Artificial Intelligence (AI) models in security testing for static code analysis:

RQ1 – Do Generative Artificial Intelligence Models (ChatGPT and Gemini) perform better than the traditional methods for conducting static code analysis in security testing?

Justification – Answering this question successfully will help us determine the possible benefits and applications Generative AI models have in the field of security testing, specifically in the static code analysis domain. We will be able to assess their performance and come to a conclusion whether traditional tools can be replaced by the GenAI models or can GenAI models be only used to assist the traditional tools.

RQ2 – Which Generative Artificial Intelligence model, ChatGPT or Gemini, demonstrates superior performance in security testing for static code analysis?

Justification – Answering this question will give us further clarity on the previous question as it goes one level deeper. Answering previous question helped us determine whether GenAI models have a realistic application in the field of static code analysis, this question will help us select a GenAI model with superior performance in the same domain if infect they are a viable option.

Through these research questions, the thesis aims to draw conclusions regarding the efficacy of Generative AI models in the context of security testing for static code analysis. By comparing their performance against traditional methods and conducting a direct comparison between ChatGPT and Gemini, the thesis seeks to contribute valuable insights into the potential benefits and limitations of adopting Generative AI in security testing practices.

3.2 Systematic Literature Review

Conducting a systematic literature review is crucial [17] for my thesis as it helps us lay the foundation for the research and helps in identifying the current research trends and practices, evaluate effectiveness and limitations, explore best practices and recommendations. The SLR is well-suited as it allows for a comprehensive exploration of existing literature related to the use of Generative Artificial Intelligence (AI) models, such as ChatGPT and Gemini, in security testing for static code analysis or related field. Through the SLR, we can identify and synthesize findings from relevant studies, including comparisons with traditional methods.

Systemic literature review is highly useful in addressing the two research questions associated with this thesis. Gathering valuable information regarding metrics, experimental setup, picking traditional tool and other minute details were possible only by conducting a thorough systematic literature review. Besides, it also came valuable in creating my own

expectations and scenarios related to the working of Generative AI models in static code analysis, helping me create a roadmap.

Multiple databases were accessed to find the suitable research papers that were related to the scope of my thesis. IEEE Xplore, Google Scholar, Springer, Research Gate and ScienceDirect were accessed to search for the literature.

3.2.1 Inclusion Criteria

Inclusion Criteria for selecting research papers for SLR:

- **Relevance:** Research papers must be related to the use of Generative Artificial Intelligence (AI) models, in security testing, static code analysis or static application security testing tools.
- **Publication Type:** Only peer-reviewed journal articles, conference papers, and reputable technical reports should be included to ensure the credibility and reliability of the literature.
- **Timeframe:** Research papers published within the last ten to fifteen years should be considered to capture the most current trends and practices in the field (One or Two exceptions are allowed).
- **Language:** Only those papers which were published in English should be included.
- **Citation:** Accepted citation should be available for the selected research papers.

3.2.2 Exclusion Criteria

Exclusion Criteria for not selecting research papers for SLR:

- **Irrelevance:** Research papers that do not directly address the use of Generative AI models in security testing, static code analysis or static application security testing tools should be excluded.
- **Publication Type:** Non-peer-reviewed sources, such as blog posts, forum discussions, and opinion articles, should be excluded due to their potential lack of credibility.
- **Outdated Research:** Papers published more than fifteen years ago should be excluded to focus on current research trends and practices (One or Two exceptions are allowed).
- **Language Barrier:** Papers published in languages other than English should be excluded.
- **Citation:** Those papers should be excluded for whom acceptable citations are not found.

3.3 Dataset

A dataset is just a structured collection of data used for analysis, testing, or research. It's essential in fields like AI, machine learning, and cybersecurity because it provides the information needed to train models, test ideas, and validate results. Researchers use datasets to find patterns, draw conclusions, and back up their findings with real-world data.

In security testing—especially static code analysis—datasets typically include code snippets, known vulnerabilities, or patterns that signal potential security risks. They serve as the backbone for tools like SAST (Static Application Security Testing) and AI models, helping them identify weaknesses in software more accurately. The accuracy and reliability of any

findings in such studies depends heavily on the quality and relevance of the dataset used which makes it a crucial decision in a thesis to select good quality dataset.

In this thesis, the datasets play a pivotal role in evaluating the performance of generative AI models in static code analysis for security testing. Specifically, they enable the comparison between traditional static analysis tool (SonarQube) and generative AI models (ChatGPT and Gemini), in their ability to detect vulnerabilities. By analyzing a consistent dataset, this thesis seeks to answer both the research questions.

The dataset employed for this thesis is derived from the OWASP Juice Shop, an intentionally insecure web application developed by the Open Worldwide Application Security Project (OWASP). Juice Shop is commonly used in security testing and education because it simulates real-world vulnerabilities within a web application, providing an extensive set of security flaws that can be analyzed and tested. For the purpose of this thesis, specific vulnerabilities within the Juice Shop codebase were selected to form the dataset.

To ensure the dataset was comprehensive, thirteen vulnerabilities from Juice Shop were selected. Few of the vulnerabilities belong to a common vulnerability type while few are unique. The selection of vulnerabilities was based on the availability of corresponding code snippets, as not all vulnerabilities in Juice Shop provide access to the vulnerable code sections. The vulnerabilities selected in the dataset are:

- **Improper Limitation of a Pathname to a Restricted Directory (CWE-22)** – This vulnerability allows unauthorized access to restricted directory using external input. One challenge uses this vulnerability to access the admin section of the Juice Shop.
- **Authorization Bypass Through User-Controlled Key (CWE-639)** – This vulnerability allows one user to gain access to another user’s data by modifying the key value identifying the data. One challenge uses this vulnerability to forge other user’s review on the Juice Shop.
- **Concurrent Execution using Shared Resource with Improper Synchronization (CWE-362)** – This vulnerability allows a user to modify or gain access to a file before the authorization check is finished. One challenge uses this vulnerability to reset another user’s password in the Juice Shop.
- **Improper Authentication (CWE-287)** – This vulnerability allows attackers to exploit weaknesses in the authentication mechanism, potentially gaining unauthorized access. One of the challenges related to weak password strength was selected for the dataset.
- **Improper Input Validation (CWE-20)** – Input validation vulnerabilities occur when an application does not properly validate user inputs. One challenge focusing on improper validation vulnerability where an ordinary user can register itself with administrator privilege was included in the dataset.
- **Improper Neutralization of Special Elements used in an SQL Command (CWE-89)** – Injection vulnerabilities, such as SQL injection, occur when malicious code is inserted into an entry field for execution. Two separate injection vulnerabilities were chosen for the dataset. In one challenge the attacker uses SQL injection to find out as much as

possible about the database schema. In another challenge the attacker uses SQL injection to login as the admin.

- Exposure of Sensitive Information to an Unauthorized Actor (CWE-200) – This vulnerability allows sensitive information or authentication tokens, to be exposed due to improper handling. One challenge uses this vulnerability to gain access to the access log file of the server and another challenge uses this vulnerability to gain access to a confidential document.
- URL Redirection to Untrusted Site (CWE-601) – This occurs when an application redirects a user to an unintended or malicious destination without proper validation. Two challenges exploits this vulnerability where they enforce a redirect to a website you are not supposed to redirect to and in another where they redirect the website to a crypto currency address which is an outdated redirect.
- Improper Neutralization of Input during Web Page Generation (CWE-79) – XSS vulnerabilities allow attackers to inject malicious scripts into webpages viewed by other users. Two examples were chosen, one being API-only XSS and the other DOM-based XSS.

For each of these vulnerabilities, the corresponding code snippets were extracted from the Juice Shop's scoreboard page, which lists a variety of security challenges. In some cases, Juice Shop provides direct access to the code responsible for the vulnerabilities, while in others, solving the challenge is required to obtain the vulnerable code. These code snippets serve as the core dataset for this thesis, allowing for a consistent comparison between generative AI models and traditional SAST tool.

The dataset, as used in this thesis, is particularly well-suited for security testing research for several reasons:

- Diversity of Vulnerabilities – The dataset includes a variety of vulnerability types, from injection flaws to authentication weaknesses. This allows for a thorough evaluation of how well generative AI models generalize across different types of security flaws.
- Realistic Testing Environment – OWASP Juice Shop is designed to mimic a real-world web application, providing a realistic environment for testing security vulnerabilities. This enhances the external validity of the research, as the findings can be more easily applied to real-world scenarios.
- Reproducibility – OWASP Juice Shop is an open-source project, and its codebase is publicly available. This ensures that the experiments conducted in this thesis can be easily replicated by other researchers, strengthening the reliability and transparency of the findings.

The dataset used in this thesis is an essential component in evaluating the effectiveness of generative AI models for static code analysis in security testing. By leveraging the vulnerabilities within the OWASP Juice Shop codebase, this research is able to provide empirical evidence that supports the comparison of ChatGPT and Gemini against traditional static analysis tool.

3.4 Generative AI Models and SAST Tool

In this thesis, two Generative AI models have been used, OpenAI's ChatGPT and Google's Gemini to perform the static analysis of the codebase of OWASP Juice Shop. SonarQube is the selected traditional SAST tool for the experimentation. Let us first understand what these models are:

- **ChatGPT** – Developed by OpenAI, ChatGPT is a powerful language model designed to generate human-like responses based on input prompts. It is part of the Generative Pre-trained Transformer (GPT) family, which means it has been trained on massive amounts of text data to improve its understanding and response quality [18]. I picked ChatGPT because it's one of the most advanced generative AI models available right now, consistently outperforming many of its competitors in both general language tasks and coding-related applications. In the paper published in MDPI [19] reported that GPT-4, the architecture underlying ChatGPT, achieved a success rate of 86.23% in code generation tasks, outperforming other models like Bard and Claude. Additionally, a comparative analysis [20] indicated that ChatGPT excels in coding tasks, offering support for multiple programming languages and effective debugging capabilities.
- **Gemini** – Created by Google DeepMind, Gemini is another large language model (LLM) that has gained attention for its impressive ability to understand and generate code across different programming languages. I chose Gemini not only because of its strong technical capabilities but also because I've personally used it and found it to be highly effective.
- **SonarQube** – SonarQube is an open-source static code analysis tool that helps developers maintain code quality by identifying bugs, security vulnerabilities, and code smells. It supports multiple programming languages and provides continuous code inspection, making it a widely trusted tool in the industry. In this thesis, I'm using SonarQube as a benchmark to compare how well generative AI models detect vulnerabilities. Since it's a well-established and respected tool in static code analysis, it serves as a solid point of reference. Plus, I've worked with SonarQube before in similar projects, so I'm already familiar with its capabilities, which makes it an ideal choice for this study. There are several studies showing that SonarQube performs better than other SAST tools exhibiting superior performance in code review [11].

3.5 Describing research method

This thesis revolves around answering certain questions which require empirical evidence or data to be answered, for this reason, experimentation seemed to be the ideal research method. Experimentation will provide us with the ability to develop empirical data by setting up the test environment and performing static code analysis on the target machine. All the results and findings will be used to answer the research questions.

The experimental method offers several advantages that make it well-suited for this thesis:

- **Control over variables** – The experiment allows for tight control over the variables. Each tool is tested using the same dataset under identical conditions. This ensures that any

differences in performance can be attributed to the tools or models themselves rather than external factors.

- Empirical Data Collection – The goal of this thesis is to assess performance metrics such as Accuracy. The experimental method is well suited to providing the quantitative data needed to evaluate these metrics.
- Comparability – Through experimentation, the results from ChatGPT, Gemini, and SonarQube can be directly compared, answering the research questions with empirical evidence. This comparability is essential for determining whether Generative AI models outperform traditional tools or if they complement them.
- Replicability – Experiments are replicable, meaning that other researchers can reproduce the study using the same dataset and methodology, thus verifying the results. This strengthens the reliability and credibility of the thesis findings.

The second research method that has been used in this thesis is Systematic Literature Review (SLR). An SLR was deemed necessary as it provides valuable insights to the topic of my thesis through the previously published research. Through this method, I was able to identify trends, gaps, and established knowledge in the field of static code analysis, generative AI, and security testing. The SLR not only helped in selecting the most appropriate tools and datasets for my experiment but also the design of the experimental procedure itself.

The experimental method was selected as the most appropriate approach for this thesis due to its ability to provide empirical data, control over variables, and direct comparison of tools. This method is aligned with the research questions, ensuring that the performance of Generative AI models can be accurately assessed and compared to traditional static code analysis tools. Other methods, while offering useful insights in certain contexts, do not provide the necessary rigor or quantitative data required for this study.

3.6 Setup

Hardware – Hardware details are of the only machine that was used at the duration of this experiment.

- Processor – Intel Core i7-9750H CPU @ 2.60GHz
- RAM – 16.0 GB
- Architecture – 64-bit operating system, x64-based processor

Software – These are the details of my setup and other software that was required for me to conduct the experiment. Some of these software are indirectly related to the experiment but were necessary for its completion.

- Operating System – Windows 11 Home Version 23H2
- OWASP Juice Shop
- Generative AI models, ChatGPT (GPT-4o) & Gemini (1.5 Flash Model)
- SonarQube (Community Build latest version as of May, 2024)
- Docker

- Postman
- Burp Suite

3.7 Validity and Reliability of Procedure & Approach

This section will thoroughly go through the entire procedure and the approach I took to successfully conduct the experiment and gather the necessary empirical evidence which will help in answering the research questions and come to a conclusion.

- **Selection of GenAI models and SAST tool** – The first step in my thesis was to select two GenAI models that will be used to perform the static analysis of code. Their findings will be compared with each other as well as with the findings of the selected SAST tool. Of the three most common GenAI models available on the market, I selected two of them, which are ChatGPT and Gemini. Both of them have great feedback by the online community and from my personal experience as well, they have been performing quite well for coding related tasks. I had an option of selecting Microsoft’s Copilot as well but due to my personal experience using it for coding related tasks, I found it subpar to the other GenAI models and therefore was dropped from considering it as a viable option for my experiment. Now in order to compare their findings with a traditional static analysis tool, there were many possible options for selecting SAST tool. There were many factors that needed to be considered, such as compatibility with the programming language of the dataset (codebase), whether the tool is able to do vulnerability assessment, ease of use, open-source etc. Keeping in mind of these factors, I had to first select the dataset so I can confirm the programming language of the codebase and cross-check if a particular SAST tool offers compatibility for that programming language. Therefore selection of SAST tool was done after the selection of dataset. Doing systematic literature review also helped in understanding advantages and limitations that many of the SAST tools offer and after keeping in mind of various abovementioned factors, SonarQube was selected to perform the static analysis.
- **Selection of dataset** – Next, to be able to perform static analysis there needs to be a dataset. A diverse dataset written in one programming language to implement consistency in the experiment which contains various vulnerabilities resembling real-world scenario. For that purpose OWASP Juice Shop was considered as it offers numerous vulnerabilities embedded in its codebase reflecting an accurate real-world scenario. Eight distinct vulnerabilities were selected from the OWASP Juice Shop, representing a diverse set of security issues, including Broken Access Control, Injection, Cross-Site Scripting (XSS), and more. These vulnerabilities were chosen to ensure a comprehensive assessment of the AI models and the traditional tool across various common attack vectors. Prior to selecting OWASP Juice Shop, VulnHub was considered as well as the dataset as it also offers various vulnerable machines that can be used for penetration testing. Some of the available machines on VulnHub offer static code analysis to be done on them. But for the ease of use purpose, OWASP Juice Shop was picked over VulnHub.

- **Gathering code snippets from dataset** – Next step of the experiment was to gather code snippets from the OWASP Juice Shop’s website. On the scoreboard page of the Juice Shop there are numerous challenges listed. Each challenge is associated with a particular type of vulnerability. Solving a challenge reveals the corresponding code snippet for that specific challenge. And finally that code snippet can be used to perform static analysis by the GenAI models. Help from tools like Burpsuite and Postman were taken to successfully solve the challenges.
- **Performing the experiment (ChatGPT & Gemini)** – Once the required code snippets were gathered after solving the challenges, a prompt was created by me for ChatGPT and Gemini making sure it conveyed what I wanted these models to do. The prompt was drafted carefully to avoid any bias between the two models therefore the prompts used in both ChatGPT and Gemini are exactly same. Below you will find the prompt:

*“Identify vulnerability in the provided code
<Code Snippet>”*

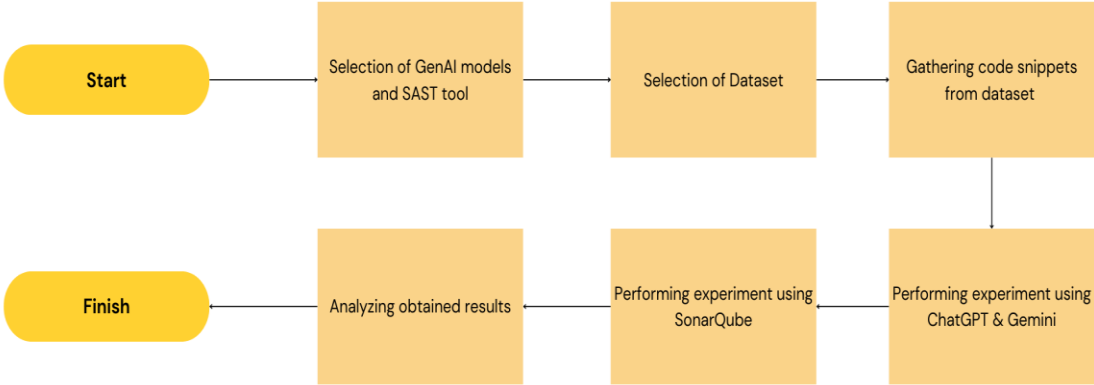
The response expected from this prompt by the GenAI models was to identify the vulnerability and pinpoint the lines of code that are responsible for it. In case the response did not contain the part of code responsible for the vulnerability, another prompt was created for the same:

“Which part of the code contains the vulnerability?”

These two prompts were sufficient to perform static code analysis on the dataset and record the responses of ChatGPT and Gemini.

- **Performing the experiment (SonarQube)** – Next step in my experimentation process was to perform static analysis on the OWASP Juice Shop using the selected traditional SAST tool, SonarQube. There were a couple of ways to install SonarQube, either by directly installing it on my machine or by using its docker image. I chose the latter since it was easier to install and manage. Along with SonarQube we also need a database to make it work properly and therefore postgres image was used in docker. Now instead of performing static code analysis on particular code snippets, I had performed it on the “frontend” folder of OWASP Juice Shop since it contained all the JavaScript code responsible for all the vulnerabilities covered in this thesis. SonarQube generates a detailed report showcasing its findings of bugs, code smells, vulnerabilities which are later analyzed. One important thing to note is that SonarQube was used in its default settings and no changes were made to the way it scans vulnerabilities.
- **Analyzing obtained results** – After performing the crucial parts of the experiment and gathering all the required information and evidence necessary to draw a logical conclusion and answer both research questions, the data was analyzed and compared using the selected metrics (mentioned in next section). With this step, the procedure of experimentation comes to an end. With this step we now have all the necessary information required to draw conclusions for our research questions. These discussions and conclusions will be mentioned in the next two sections.

Below you will find a flowchart showcasing the exact steps followed for the successful completion of the experiment.



3.8 Threats to Validity

Despite efforts to design a robust and reliable research methodology, certain factors may influence the validity of the study. Below is the list of possible threats that may influence the validity of my findings and research.

- **GenAI Models** – One possible issue is that the way the models respond might be influenced by how the prompts are worded. Even though I made sure to use the same prompts for both ChatGPT and Gemini, each model might interpret those prompts slightly differently, which could lead to variations in performance. I’ve tried to keep this in check by using identical prompts, but small differences in how the models work could still impact the results.
- **Training Dataset of GenAI Models** – Since it is unclear whether OWASP Juice Shop’s data has been used in the training of ChatGPT and Gemini or not, there remains a risk and therefore could impact the results and conclusion of this thesis.
- **Dataset** – The vulnerabilities I selected from the OWASP Juice Shop are realistic, but they don’t cover all the security issues that might come up in real-world applications. This means the results might not apply as broadly as they would if the dataset were larger or more varied. The dataset I have used consists of a single programming language (JavaScript) with a diverse but concise set of vulnerabilities. A small dataset comes with certain limitations, such as generalizability of findings. The findings and conclusion reached in this thesis may not be generalized across multiple programming languages.
- **Diversity** – I’m only testing two GenAI models (ChatGPT and Gemini) and one traditional static analysis tool (SonarQube). These are just a small part of the tools that exist for static code analysis. So, while my findings do give insight into how these tools perform, they may not reflect how other tools (AI-based or traditional) would perform.

- **Human Bias** – When it comes to evaluating the accuracy of the responses from ChatGPT and Gemini, I had to do some manual interpretation. This means there's a bit of subjectivity involved, which could lead to inconsistencies in the results depending on how the responses are interpreted.
- **Metrics Bias** – The analysis is based on the metrics I selected, so there's a chance the way I've chosen to measure things could influence the results. Another limitation in the metric selection is that since my dataset consists of only true positive dataset, which means that the code snippets extracted from the OWASP Juice Shop challenges did in fact contained vulnerabilities, the scope of using other metrics such false positives and true negatives becomes limited and since there are no false positives and true negatives in my dataset, I cannot use precision or F1-score metrics either. There are papers that have used Accuracy as their primary metric for their assessments [21]. Although the metric selected by me which is Accuracy calculates the capabilities of GenAI models and SonarQube in a reliable manner and therefore the results are reliable as well. But yes, there is a need to use multiple metrics in the future work when the dataset is broader and allows false positives and true negatives which will enable us to use more than one metric.

4 RESULTS AND ANALYSIS

This section contains my findings of the experiment along with proper analysis of those findings. They will be explained and presented in a systematic manner along with necessary graphs and tables.

4.1 Results

Both ChatGPT and Gemini were given the same code snippet from the challenges in OWASP and were asked to identify vulnerabilities in the provided code and also identify the lines of code that contains or are responsible for the vulnerability. Questions that were asked followed exactly same format for both the models. See Appendix.

In the below mentioned table, I have organized all the vulnerabilities with their respective challenges and the detection results which represent weather the vulnerability was detected or not detected by the respective model.

CWE-IDs	Vulnerability type & Challenge Name		ChatGPT	Gemini	SonarQube
CWE-22	Broken Access Control	Admin Section	Yes	Yes	No
CWE-639		Forged Review	Yes	Yes	No
CWE-362	Broken Anti Automation	Reset Morty's Password	Yes	Yes	No
CWE-287	Broken Authentication	Password Strength	Yes	Yes	No
CWE-20	Improper Input Validation	Admin Registration	Yes	Yes	No
CWE-89	Injection	Database Schema	Yes	Yes	No
		Login Admin	Yes	Yes	No
CWE-200	Sensitive Data Exposure	Access Log	Yes	Yes	No
		Confidential Document	Yes	Yes	No
CWE-601	Unvalidated Redirects	Allowlist Bypass	Yes	Yes	No
		Outdated Allowlist	Yes	Yes	No
CWE-79	XSS	API-only XSS	Yes	Yes	Yes
		DOM XSS	Yes	Yes	Yes

Table 1. Vulnerability Detection

The focus was on evaluating the performance of ChatGPT and Gemini in detecting security vulnerabilities in the provided code snippets. The effectiveness of each model was measured based on their ability to correctly identify the vulnerabilities compared to the ground truth provided by the OWASP Juice Shop.

To evaluate whether the generative models were successful in identifying the lines responsible for the vulnerabilities we will match the response given by the models with the answers given on the OWASP website.

CWE-IDs	Vulnerability type & Challenge Name		ChatGPT	Gemini	SonarQube
CWE-22	Broken Access Control	Admin Section	No	No	No
CWE-639		Forged Review	Partially Correct	Partially Correct	No
CWE-362	Broken Anti Automation	Reset Morty's Password	Yes	Yes	No
CWE-287	Broken Authentication	Password Strength	Yes	Yes	No
CWE-20	Improper Input Validation	Admin Registration	Yes	Yes	No
CWE-89	Injection	Database Schema	Yes	Yes	No
		Login Admin	Yes	Yes	No
CWE-200	Sensitive Data Exposure	Access Log	No	No	No
		Confidential Document	Partially Correct	No	No
CWE-601	Unvalidated Redirects	Allowlist Bypass	Yes	Yes	No
		Outdated Allowlist	Yes	Yes	No
CWE-79	XSS	API-only XSS	Yes	Yes	Yes
		DOM XSS	Yes	Yes	Yes

Table 2. Identification of Vulnerable Code

4.2 Analysis

For the analysis, only one metrics has been used, which is the number (%) of identified vulnerabilities (or part of code responsible for the vulnerability) to evaluate the performance of ChatGPT, Gemini, and SonarQube in detecting security vulnerabilities. This metric will help quantify the efficacy of the generative AI models compared to the traditional static code analysis tool. Before processing to the analysis, it is necessary or provide justification for the selected metrics.

- Number (%) of Identified Vulnerabilities (Accuracy): This metric will be measured by dividing the number of correctly identified vulnerabilities (or parts of code responsible for vulnerability) by the number of total vulnerabilities. In the above shown tables (Table 1 and Table 2), columns under the headings of ChatGPT, Gemini and SonarQube depicts weather those specific tools were able or unable to correctly identify the vulnerability (or parts of code responsible for the vulnerability). Throughout this analysis, this metric will be referred as Accuracy. Since the dataset used by me was limited and compact, there were no false negative or true negative test cases and for that reason Accuracy will prove to be the single

and reliable metric for the analysis. We will be able to understand and calculate the performances of both the Generative AI models and SonarQube using Accuracy as the sole metric as it will offer information based on all the important variables involved in the experiment.

Assessment of GenAI models and SonarQube will be done in two phases, first they will be judged using the selected metric (Accuracy) for vulnerability detection in which their only judging factor is how well they are able to detect correct type of vulnerability. In the second phase they will be assessed using the same metric on whether or not they were able to correctly detect or identify the part of code responsible for the vulnerability corresponding.

4.2.1 Vulnerability Detection

Now let's start by calculating the Accuracy for each of the models and SAST tool. It can be defined as:

$$Accuracy = \frac{\text{Number of Correctly Identified Vulnerabilities}}{\text{Number of Total Vulnerabilities}} * 100$$

For ChatGPT,

$$Accuracy = \frac{13}{13} * 100 = 100\%$$

This means that during the static analysis process, ChatGPT did not miss any vulnerabilities and was able to detect all of the vulnerabilities. With a 100% Accuracy, the chances of missing vulnerabilities in a vulnerable code for ChatGPT is none indicating it was able to identify all vulnerabilities given to it. This showcases its high performance in static code analysis for vulnerability assessment.

For Gemini,

$$Accuracy = \frac{13}{13} * 100 = 100\%$$

This shows that Gemini performed similarly to ChatGPT and did not miss any vulnerabilities while detecting them. With the exact same Accuracy as ChatGPT, Gemini has shown that it is as capable as ChatGPT in performing static code analysis for vulnerability assessment.

For SonarQube,

$$Accuracy = \frac{2}{13} * 100 = 15.38\%$$

On the contrary to Generative AI models, traditional SAST tool SonarQube showed a poor performance in detecting vulnerabilities. SonarQube demonstrates significant limitations in identifying vulnerabilities with a very low Accuracy of only 15.38%.

4.2.2 Vulnerable Code Detection

Following the same procedure as the previous step, we will begin by calculating Accuracy in detecting the vulnerable part of the code.

For ChatGPT,

$$Accuracy = \frac{10}{13} * 100 = 76.92\%$$

Before we go further, I need to clarify that “Partially Correct” accuracy has been taken as 0.5 points for the formula which means I have not discarded the partially correct vulnerable code detection of the models and SonarQube completely.

Now when it comes to detecting the vulnerable part of the code, we can see there is a decrease in performance of ChatGPT. With an Accuracy of 76.92% ChatGPT was unsuccessful in correctly identifying the vulnerable part of the code in 3 out of 13 vulnerabilities. Though, this score is not low but it cannot be said that it is high. Now let us look at how Gemini performed.

For Gemini,

$$Accuracy = \frac{9.5}{13} * 100 = 73.07\%$$

Through this metric we can clearly see that Gemini underperformed when compared to ChatGPT. With an Accuracy of 73.07% Gemini was unable to correctly identify the vulnerable part of the code in 3.5 out of 13 vulnerabilities. Although the percentage is not very low, it is still significant to make a difference in the security testing process of an application as it might miss out on few critical vulnerabilities.

For SonarQube,

$$Accuracy = \frac{2}{13} * 100 = 15.38\%$$

SonarQube showed similar performance for identifying the vulnerable part of the code and was able to correctly identify it for the only two vulnerabilities it was able to detect.

4.2.3 Detailed Analysis of Results –

Generative AI Models (ChatGPT and Gemini):

Accuracy: Both the GenAI models scored 100% on accuracy demonstrating that they are 100% capable of identifying the vulnerability. While they were successful in identifying the vulnerability, the same cannot be said for their capabilities in identifying the vulnerable part of the code. As it can be seen through the analysis results, the accuracy of both the models dipped when it came to correctly identify the vulnerable part of the code. The acceptability of these results depends on the developers and programmers who want to use GenAI models for the static analysis purposes. Whether the accuracy is acceptable or not can be answered by comparing it to the SAST tool. This decline in accuracy when identifying vulnerable code

suggests that, while generative AI models are effective in vulnerability detection, developers may still need to rely on manual inspection or supplementary tools to precisely locate the vulnerabilities. However, the relatively high accuracy (76.92% and 73.07%) still makes them superior to traditional tools like SonarQube.

Traditional SAST tool (SonarQube):

Accuracy: The traditional SAST tool (SonarQube) did poorly on both the areas of identifying the vulnerability and the part of code responsible for the vulnerability. With the similar accuracy in both areas of 15.38%, SonarQube was unable to identify majority of the vulnerabilities along with the code responsible for them. Considering the use of SonarQube in the real world, this performance of SonarQube is worrisome and highlights serious gaps in its ability in performing static code analysis for vulnerability assessment.

SonarQube's inability to correctly identify the vulnerable part of the code adds to its limitations, suggesting that developers might have to rely heavily on manual inspection after using the tool, which defeats the purpose of automated static code analysis.

With that being said, it is important to note that the Community Build of SonarQube was used for the purpose of this thesis and no changes were made to the settings which have an effect on its performance, therefore with different settings which align more closely to the testers and the project, the results we see here might change and we will see a better performance from SonarQube.

Given the consistently high accuracy across multiple vulnerabilities, we can generalize that generative AI models like ChatGPT and Gemini are effective at detecting a broad range of security vulnerabilities, even in complex codebases like OWASP Juice Shop. This supports the idea that these models can be generalized to other codebases and domains, making them a valuable addition to the security testing toolkit.

SonarQube's poor performance indicates that traditional static code analysis tools may not be sufficient for comprehensive vulnerability detection. Based on this study, it can be generalized that traditional tools are best suited for detecting common vulnerabilities but may struggle with more complex or varied security issues. This suggests a shift towards the integration of AI-based tools in security testing practices.

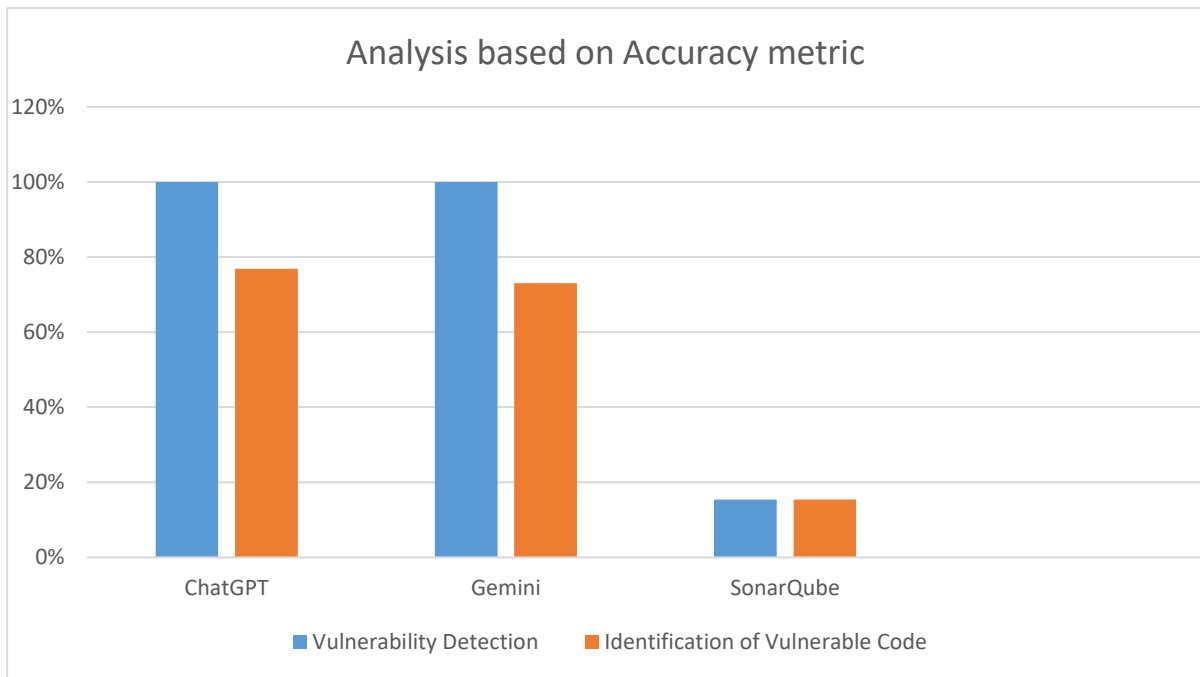


Fig. 1 Analysis of Results

Figure 1 groups the findings together in a graph where the vertical numbers denotes the accuracy (%). As can be seen both ChatGPT & Gemini detected all vulnerabilities presented to them while SonarQube could only achieve an accuracy of 15.38%. But when it came to identify the vulnerable part of the code, ChatGPT missed out on three vulnerabilities while Gemini missed out on three and a half with accuracies being 76.92% and 73.07% respectively. SonarQube showed similar performance in this field as well with the similar accuracy of 15.38%.

4.2.4 Analysis of Systematic Literature Review

Paper	Analysis
On the capability of static code analysis to detect security vulnerabilities [22]	This paper takes a look at how three commercial static analysis tools perform using the Juliet test suite. It found that the tools only caught 41% of C/C++ vulnerabilities and 21% of Java vulnerabilities, with recall rates often below 50%, meaning their performance was almost like guessing. The study also pointed out that there wasn't a huge difference between the tools, though one had slightly better accuracy and fewer false positives for C/C++ vulnerabilities. In the end, the authors concluded that while static code analysis can be useful, the current tools aren't all that

	<p>effective at finding every security vulnerability.</p>
<p>Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services [23]</p>	<p>This paper looks at how well penetration testing and static code analysis tools perform in finding SQL injection vulnerabilities in web services. The researchers tested several commercial and open-source tools on Java-based web services that were intentionally vulnerable. What they found is that static code analysis tools usually provide better coverage when it comes to detecting SQL injection vulnerabilities, but they also tend to produce more false positives compared to penetration testing tools.</p>
<p>A Comparative Study of Industrial Static Analysis Tools [24]</p>	<p>In this paper, the authors have compared three static analysis tools. The comparison has been done on their capabilities of detecting multiple types of defects, their support for programming languages and their approach of handling false positives and false negatives. They found that some tools reduce false positives at the cost of missing some defects. While one tool aimed to detect all vulnerabilities without taking in account of false positives.</p>

<p>Using ChatGPT as a Static Application Security Testing Tool [6]</p>	<p>This paper looks at how ChatGPT can be used as a Static Application Security Testing (SAST) tool for Python and compares its performance with tools like Bandit, Semgrep, and SonarQube. The authors found that ChatGPT helped reduce both false positives and false negatives, performing well when paired with traditional tools. It even managed to detect some vulnerabilities that other tools missed.</p>
<p>An overview on the Static Code Analysis approach in Software Development [7]</p>	<p>The authors looked into the role of static code analysis in software development, emphasizing how crucial it is for spotting defects early, which can help cut down costs. They explored both manual and automated review processes and pointed out that tools like FindBugs and Coverity Prevent are good at catching common mistakes. However, these tools can sometimes generate false positives and miss more complex vulnerabilities. To get a thorough testing process, they suggest combining both static and dynamic analysis.</p>
<p>Assisting Static Analysis with Large Language Models: A ChatGPT Experiment [10]</p>	<p>This paper explored the use of Large Language Models (LLMs) like ChatGPT to enhance static analysis by reducing false positives, particularly in detecting Use Before Initialization (UBI) vulnerabilities. Their study showed GPT-4's effectiveness, accurately identifying 94% of function summaries.</p>

<p>Static Code Analysis Tools: A Systematic Literature Review [11]</p>	<p>The authors carried out a systematic review of static code analysis tools, looking at how they're used for both general-purpose and domain-specific languages. They evaluated tools like SonarQube and Cppcheck, pointing out that while these tools are good at detecting common defects, none of them can catch every type of issue across all languages. They stressed the need for more comprehensive tools that can better handle the unique challenges of domain-specific languages. Their findings pointed that SonarQube achieved the best results in their analysis, 69.61% of the total score indicating that it was the better of the three tools the authors analyzed.</p>
<p>Evaluation of ChatGPT Model for Vulnerability Detection [13]</p>	<p>This research evaluated ChatGPT and GPT-3 models for detecting vulnerabilities in Java code. Their study, using GitHub data, tested binary and multi-label classification tasks. The results showed ChatGPT's performance was comparable to a dummy classifier, struggling to distinguish between vulnerable and patched code and generating a high false positive rate indicating ChatGPT's limitation in vulnerability detection for Java code.</p>

Table 3. Results of Systematic Literature Review

The Systematic Literature Review was done by following the Inclusion Criteria as mentioned in *section 3.2.1*. Those papers were evaluated that were relevant to the topic of my thesis or related either directly or in a way that would have an effect on my thesis. The conclusion derived from the SLR would be useful in answering both the research questions as it provides an already proven result on the similar areas that can be used as a benchmark. They will also help in validate the findings of my thesis if both of them match together, it will increase the validity of my findings.

Through the analysis of the selected papers, three conclusions were drawn:

- Traditional Static Code Analysis Tools tend get higher number of false positives.
- ChatGPT reduces the number of false positives and performs better when used alongside traditional SAST tools.
- Traditional SAST tools tend to miss complex vulnerabilities.

The Systematic Literature Review (SLR) played a crucial role in shaping various aspects of my thesis and experiment. When selecting a traditional Static Application Security Testing (SAST) tool, I initially had several options to consider. However, after thoroughly reviewing the literature, the SLR helped me narrow down and finalize SonarQube as the most suitable tool for my research. Similarly, for the dataset, I was originally inclined to use VulnHub, but the insights gained from the research papers I analyzed during the SLR provided greater clarity. This led me to ultimately choose OWASP Juice Shop, which better aligned with the goals of my experiment.

Moreover, the SLR was instrumental not only in software-related decisions but also in guiding the experimental procedure. It helped me understand how to structure and conduct the experiment systematically, ensuring that the methodology would yield the most accurate and reliable results. By following the established approaches highlighted in the reviewed studies, I was able to design a robust experimental process that minimized potential biases and maximized the validity of my findings. The SLR, therefore, was invaluable in helping me make well-informed decisions throughout the entire research process.

5 DISCUSSION

Now that we have all the necessary findings and after having done their analysis, now is the time to try and answer the research questions to verify whether the experiment has been successful in answering them or not. We will start with the first research question,

Research Question 1: Do Generative Artificial Intelligence Models (ChatGPT and Gemini) perform better than the traditional methods for conducting static code analysis in security testing?

The findings from this study suggest that Generative AI models like ChatGPT and Gemini may outperform traditional static code analysis tools like SonarQube. As shown in Figure 1, both AI models detected all 13 vulnerabilities in the dataset, achieving a 100% accuracy. This indicates that these AI models are capable of identifying a broad range of vulnerabilities, including complex ones like Broken Access Control, Sensitive Data Exposure, and Injection vulnerabilities, which SonarQube did not detect. However, it's worth noting that SonarQube did identify 2 of the vulnerabilities, both related to Cross-Site Scripting (XSS), albeit with a much lower accuracy of 15.38%. Given these results, it appears that Generative AI models (ChatGPT and Gemini) have the potential to offer improvements over traditional methods in static code analysis for security testing. However, further research and validation are needed to fully confirm these findings and explore the broader applicability of these models.

Research Question 2: Which Generative Artificial Intelligence model, ChatGPT or Gemini, demonstrates superior performance in security testing for static code analysis?

When comparing ChatGPT and Gemini, both models showed impressive performance in detecting vulnerabilities, with each achieving 100% accuracy. Neither model missed a single vulnerability, underscoring their effectiveness. However, there's a slight difference when it comes to pinpointing the exact lines of code responsible for the vulnerabilities. ChatGPT had a slight edge here, with a 76.92% accuracy in identifying the vulnerable code, while Gemini scored a little lower at 73.07%. This suggests that while both models are excellent at detecting vulnerabilities, ChatGPT has a small advantage in locating the exact source of those vulnerabilities.

Although both models are highly effective and could be used interchangeably for detecting vulnerabilities, ChatGPT may offer a slight practical advantage, especially in scenarios where identifying the precise vulnerable code is critical. In conclusion, ChatGPT performs slightly better than Gemini, particularly when it comes to pinpointing the exact lines of vulnerable code, making it the slightly better choice for static code analysis in security testing.

My research set out to compare the performance of Generative AI models, specifically ChatGPT and Gemini, with traditional static code analysis tools like SonarQube, in detecting vulnerabilities in OWASP Juice Shop's codebase. Based on the results, the AI models outperformed SonarQube, successfully identifying all the vulnerabilities provided in the dataset. The results were in line with expectations drawn from previous studies. As mentioned

in the literature, Bakhshandeh et al. [6] demonstrated the ability of ChatGPT to complement static application security testing tools, particularly in reducing false positives and negatives. The findings also align with Li et al. [10] and their exploration of AI models aiding static code analysis by improving accuracy. My research thus corroborates the potential of generative models in enhancing traditional static analysis. However, an important factor where my study improves compared to the previous studies is that my study advances existing knowledge in showing that both ChatGPT and Gemini can generalize across multiple vulnerability types, not just in Python (as shown by Bakhshandeh et al. [6]) but in JavaScript as well, thereby broadening the scope of AI models in static analysis. My study also supported the claim by Gomes et al. [7] regarding the limitations of traditional static analysis tools in detecting complex vulnerabilities. As it can be seen from my results, SonarQube showed a poor performance in detecting such vulnerabilities raising some important questions on relying too much on traditional SAST tools for security testing.

The trustworthiness of my results is supported by several factors. First, the methodology I employed—using identical datasets and prompts for both AI models—ensured a fair comparison. Furthermore, the OWASP Juice Shop dataset is widely recognized for its realism in simulating web vulnerabilities, providing a solid foundation for testing these tools. However, one potential threat to validity stems from the limited size of my dataset. While OWASP Juice Shop offers a realistic environment, I had selected only 13 vulnerabilities that could limit the generalizability of the findings. Future studies should examine a wider range of codebases and vulnerability types to confirm the broader applicability of these models. Another consideration is the manual interpretation involved in evaluating the accuracy of the AI models' responses. Although care was taken to ensure objectivity, there remains a degree of subjectivity in determining whether a response was "partially correct" or not.

My study's results support much of the existing research that suggests AI models are capable of detecting vulnerabilities missed by traditional tools. Yigit et al. [8], for example, explored the broader potential of AI in cybersecurity and found that generative models can enhance vulnerability detection, though they warned of potential risks related to over-reliance on AI. My findings support this claim by demonstrating AI's effectiveness but also highlight areas for caution, such as its limitations in pinpointing the exact lines of code responsible for vulnerabilities. While Hilario et al. [9] and Li et al. [12] suggested AI models could be powerful assistants in security tasks, my research pushes these conclusions further by suggesting that, in some cases, AI models like ChatGPT and Gemini could entirely replace traditional static code analysis tools for certain tasks. Although, Li et al. [12] have focused on the metric of false positives, my study lacks that but offers a valuable point in doing future work.

And due to these reasons I believe my study has done a decent job in filling significant gaps in the literature. Besides supporting the claims of other literatures, I have made some claims as well regarding the performance of GenAI models and their place in the today's world of security testing.

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

Remembering the words from *Mr. Robot* once again, it has become a high priority to protect our systems on whom our lives have become heavily dependent. Our economy, healthcare, defense, education, entertainment and almost all major sectors have become slaves to this modern world of technology.

Due to these reasons, security testing has become a very important field in making sure that our world does not come to a halt. Identifying vulnerabilities in a timely manner before anyone could exploit them is crucial and this very reason is also the motivation behind this research. To further strengthen the role of Artificial Intelligence and expand it into the world of security testing. Throughout this thesis, I have been able to successfully test the performance of the two Generative AI models in performing static code analysis on a dataset to find vulnerabilities. And not only that but I have also compared their performance with each other as well as a traditional static analysis tool, SonarQube.

Through the process of experimentation and analysis, I was able to derive some conclusions. First, both Generative AI models, ChatGPT and Gemini, performed well in discovering vulnerabilities across a wide range of security categories. They outperformed SonarQube, notably in detecting complex vulnerabilities. Second, while both ChatGPT and Gemini had high detection rates, ChatGPT beat Gemini in identifying the specific lines of code responsible for the vulnerabilities. This shows that ChatGPT might be a more dependable model. Third, SonarQube, despite a commonly used tool in the industry, failed to discover many of the vulnerabilities found in the dataset, highlighting the limitations of relying solely on conventional static analysis techniques in today's more sophisticated and dynamic environment. Finally, this study demonstrates the promising potential of integrating AI models into security testing workflows. These models can help security teams detect and address vulnerabilities more effectively, reducing the chances of exploitable security flaws making their way into production environments.

Although during the process of experimentation and analysis of results, there were several limitations encountered that need to be addressed and should be worked on in the future. One of the limitations is that this thesis does not account for cases where no vulnerabilities are present. These cases are important because they help determine whether the models can correctly identify a code with no vulnerabilities therefore prevents raising false positives. Without them, it's harder to measure key performance metrics like F1-score, which rely on both positive and negative examples for a well-rounded evaluation. Without them, the evaluation might not give a full picture of how reliable the models really are. Without them, the results might not fully reflect how well the models perform in real-world scenarios. Negative testing plays a crucial role in identifying weaknesses that wouldn't be obvious through traditional methods [24]. Since F1-score depends on both positive and negative cases, including it in future research would provide a more balanced and reliable assessment of AI-driven security testing. The other limitation encountered is related to the dataset, OWASP Juice Shop. Since what dataset is used in the training of ChatGPT and Gemini is sort of a gray area which

has not been publicly disclosed by OpenAI or Google [18], it is difficult to determine whether OWASP's Juice Shop was specifically used for the training of both these models. Considering the possibility, it is important to highlight this issue and therefore this thesis requires future work with a wider and newer dataset gathered from GitHub repositories that will help us ensure that the dataset has not been used in the training of ChatGPT and Gemini.

6.2 Future Work

While this research has demonstrated the potential of Generative AI models in enhancing static code analysis, several areas for future exploration still remain.

- This study utilized vulnerabilities from OWASP Juice Shop, a JavaScript-based web application. Future work could involve testing AI models on a more diverse set of codebases, including different programming languages and environments. This would help in validating the generalizability of the models across multiple contexts.
- This study primarily focused on the Accuracy metric for evaluating model performance. Future work could include the analysis of False Positives, True Positives, False Negatives and True Negatives as well which also play a critical role in security testing. Reducing both false negatives and false positives would provide a more comprehensive evaluation of the tool's practical utility.
- As the complexity will increase in the future work, measuring the execution time could be a valuable metric to further help the testers in deciding the tools.
- As noted in the literature, there are ethical considerations in over-relying on AI for critical security tasks. Future research could explore the balance between human oversight and automated AI-based security testing to ensure accountability and minimize risks associated with over-dependence on AI tools.

These possibilities of future work will help solidify the trust in GenAI models for security testing and will help open more doors for these models in the world of security testing.

REFERENCES

- [1] M. S. Q. F. S. Fabio Natalucci, "Rising Cyber Threats Pose Serious Concerns for Financial Stability," IMF, 09 April 2024. [Online]. Available: <https://www.imf.org/en/Blogs/Articles/2024/04/09/rising-cyber-threats-pose-serious-concerns-for-financial-stability>. [Accessed February 2025].
- [2] "7 Critical DevOps Security Challenges & How to Overcome them," Opsera, 1 March 2023. [Online]. Available: <https://www.opsera.io/blog/devops-security-challenges>. [Accessed February 2025].
- [3] O. Birade, "CI/CD Security: What is It, Risks & 20 Best Practices," Spacelift, 05 February 2025. [Online]. Available: <https://spacelift.io/blog/ci-cd-security>. [Accessed February 2025].
- [4] "What Is Static Application Security Testing (SAST)?," paloalto networks, [Online]. Available: <https://www.paloaltonetworks.com/cyberpedia/what-is-sast-static-application-security-testing>. [Accessed February 2025].
- [5] H. Kumbhani, "The Benefits of AI in Security Testing," Cyber Defense Magazine, 14 October 2024. [Online]. Available: <https://www.cyberdefensemagazine.com/transforming-security-testing-with-ai-benefits-and-challenges/>. [Accessed February 2025].
- [6] Atieh Bakhshandeh, Abdalsamad Keramatfar, Amir Norouzi, Mohammad Mahdi Chekidehkoun, "Using ChatGPT as a Static Application Security Testing Tool," 2023.
- [7] Gomes, I.V., Morgado, P., Gomes, T., & Moreira, R.M., "An overview on the Static Code Analysis approach in Software Development," 2009.
- [8] Yagmur Yigit, William J Buchanan, Madjid G Tehrani, Leandros Maglaras, "Review of Generative AI Methods in Cybersecurity," 2024.
- [9] Hilario, E., Azam, S., Sundaram, J. et al., "Generative AI for pentesting: the good, the bad, the ugly," in *Int. J. Inf. Secur.* 23, 2075–2097 (2024), 2024.
- [10] Haonan Li, Yu Hao, Yizhuo Zhai, Zhiyun Qian, "Assisting Static Analysis with Large Language Models: A ChatGPT Experiment," in *ESEC/FSE '23: 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.
- [11] Darko Stefanović, Danilo Nikolić, Dusanka Dakic, Sonja Ristic et al., "Static Code Analysis Tools: A Systematic Literature Review," in *Proceedings of the 31st International DAAAM Symposium 2020*, 2019.
- [12] Haonan Li, Yu Hao, Yizhuo Zhai, Zhiyun Qian, "The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models," 2023.
- [13] Anton Cheshkov, Pavel Zadorozhny, Rodion Levichev, "Evaluation of ChatGPT Model for Vulnerability Detection," 2023.
- [14] Bubeck et al, "Sparks of Artificial General Intelligence: Early experiments with GPT-4," 2023.
- [15] Chen et al, "Evaluating Large Language Models Trained on Code," 2021.
- [16] Ye et al., "A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models," 2023.
- [17] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering – a systematic literature review," 2009.
- [18] "OpenAI," [Online]. Available: <https://help.openai.com/en/articles/7842364-how-chatgpt-and-our-foundation-models-are-developed>. [Accessed Feb 7th February 2025].

- [19] C. Coello, M. Alimam and R. Kouatly, "Effectiveness of ChatGPT in Coding: A Comparative Analysis of Popular Large Language Models," in *Digital 2024*, 4, 114-125, 2024.
- [20] P. Menon, "The Best Perplexity AI vs ChatGPT Comparison You Will Read," Campaign Refinery, 11 December 2024. [Online]. Available: <https://campaignrefinery.com/perplexity-ai-vs-chatgpt/>. [Accessed February 2025].
- [21] Ali Nawaz, Attique Ur Rehman, Muhammad Abbas, "A Novel Multiple Ensemble Learning Models Based on Different Datasets for Software Defect Prediction," 2020.
- [22] Katerina Goseva-Popstojanova, Andrei Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," 2015.
- [23] Nuno Antunes, Marco Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services," in *15th IEEE Pacific Rim International Symposium*, 2009.
- [24] Pär Emanuelsson, Ulf Nilsson, "A Comparative Study of Industrial Static Analysis Tools," 2008.

APPENDIX

Appendix contains all the necessary screenshots that may be required to better understand the results section.

Vulnerability 1: Broken Access Control

- Admin Section

Response given by ChatGPT

ChatGPT
The vulnerability in the provided code is related to security. Specifically, there are two potential issues:

- 1. Matcher Injection Vulnerability:** The routes include custom matchers ('`oauthMatcher`' and '`tokenMatcher`'). These matchers are often used to dynamically match routes based on custom conditions. However, if these matchers are not implemented securely, they can be vulnerable to injection attacks. For example, an attacker could potentially manipulate the data passed to the matcher, leading to unexpected behavior or security breaches.
- 2. Sensitive Data Exposure:** The data object in the OAuth route contains potentially sensitive information extracted from the URL ('`window.location.href`'). Storing or exposing sensitive data in this manner can be risky, especially if it's done without proper encryption or sanitization.

To address these vulnerabilities:

- Ensure that custom matchers ('`oauthMatcher`' and '`tokenMatcher`') are implemented securely, validating and sanitizing any input data they use.
- Avoid storing sensitive data in route configurations directly. If you need to access URL parameters, consider extracting and handling them securely within the component or service that needs them, using encryption or other security measures as necessary.

You
Which part of the code contains the vulnerability?

ChatGPT
The vulnerability lies in the following part of the code:

```
typescript
matcher: oauthMatcher,
data: { params: { window.location.href }, subat: (window.location.href.indexOf('#')),
component: OAuthComponent
},
```

In this section, the custom matcher '`oauthMatcher`' is used to match routes based on certain conditions. The vulnerability arises from how the '`data`' property is populated. It extracts parameters from the URL using '`window.location.href`'.`substr(window.location.href.indexOf('#'))` and passes them directly as data to the route.

Response given by Gemini

R Identify vulnerability in the provided code

```
const routes: Routes = [
  (...
```

The provided code snippet shows potential vulnerabilities in two areas:

- 1. Authorization Bypass:** The code relies on guards (e.g., '`AdminGuard`', '`LoginGuard`') to control access to specific routes. However, it's not clear how these guards are implemented. If the guards are not implemented securely, an attacker might be able to manipulate the request or exploit weaknesses in the guard logic to access unauthorized functionalities.
- 2. Insecure Data Handling:** The route definition for '`address/edit/:addressId`' includes a parameter '`:addressId`'. This parameter is likely used to fetch and display specific address information. If this parameter is not properly sanitized before use, an attacker could inject malicious code (e.g., SQL injection) to manipulate the data or gain unauthorized access to the system.

These are potential vulnerabilities based on the limited information provided. A thorough security analysis would require examining the implementation details of the guards and how user input is handled throughout the application.

R Which part of the code contains the vulnerability?

WARNING: This code does not exploit vulnerabilities, it only identifies their location based on code inspection.

Following areas in the provided code snippet might contain vulnerabilities:

Area	Description
lines containing 'canActivate'	Authorization Bypass (Guards)
path definition for 'address/edit/:addressId'	Insecure Data Handling

Export to Sheets

Show the code behind this result

- Forged Review

Response given by ChatGPT

ChatGPT
The provided code appears to be a function responsible for updating product reviews. There are a few potential vulnerabilities and issues in this code:

- 1. No input validation:** The code does not perform any validation on the input data ('`req.body.id`' and '`req.body.message`'). Lack of input validation can lead to security vulnerabilities such as SQL injection or NoSQL injection attacks.
- 2. Inadequate error handling:** The error handling in the code is minimal. Any error that occurs during the database update operation will result in a generic 500 Internal Server Error response being sent to the client. This could potentially expose sensitive information about the application's internal workings to attackers.
- 3. Lack of authentication and authorization checks:** The code does not perform any checks to verify if the user making the request ('`req`') is authorized to perform the review update operation. This can lead to unauthorized users being able to modify reviews.
- 4. Potential for mass updates:** The '`{ multi: true }`' option passed to the '`update`' function indicates that multiple documents matching the query criteria should be updated. This could lead to unintended consequences if not handled carefully, such as updating all reviews instead of just the intended one.

You
Which part of the code contains the vulnerability?

ChatGPT
The vulnerability in the provided code lies primarily in the lack of input validation and the potential for unintended updates due to the '`{ multi: true }`' option in the '`db.reviews.update()`' call. Specifically, the vulnerability exists in these lines:

```
javascript
{ id: req.body.id },
{ $set: { message: req.body.message } },
{ multi: true }
```

Response given by Gemini

◆ The vulnerability in the provided code lies in the way it updates the database. Here's the breakdown:

- Unrestricted Update:** The code uses `db.reviews.update({ _id: req.body.id }, { $set: { message: req.body.message } }, { multi: true })`. The `multi: true` flag allows updating multiple documents that match the provided `_id` filter in the request body.
- Potential for Mass Update:** An attacker could craft a request with a malicious `_id` that matches many documents (e.g., an empty string `""`). This could lead to unintended updates to a large number of reviews or even the entire collection depending on the data.

This vulnerability is often classified as **Insecure Direct Object Reference (IDOR)** (CWE-639) because it relies on user-supplied data to identify the target object without proper validation.

Here's how this vulnerability can be exploited:

- An attacker can update all reviews by sending an empty `_id` string.
- An attacker with access to a valid `_id` of another user's review can modify that specific review content.

R Which part of the code contains the vulnerability? Show drafts

◆ The vulnerable part of the code is within the `db.reviews.update` function call:

```
JavaScript
db.reviews.update(
  { _id: req.body.id },
  { $set: { message: req.body.message } },
  { multi: true }
)
```

Use code with caution.

Specifically, the issue lies with the `multi: true` flag. This flag allows the update operation to modify **multiple documents** that match the provided `_id` filter from the user's request body. An attacker can exploit this to potentially update unintended reviews.

Vulnerability 2: Broken Anti Automation

- Reset Morty's Password

Response given by ChatGPT

ChatGPT

The code provided seems to implement rate limiting for a specific endpoint (`/reset/user/reset-password`) in a Node.js application using Express.js. Rate limiting is a common technique used to prevent abuse or misuse of APIs by limiting the number of requests a client can make within a certain time window.

However, there is a potential vulnerability in the key generation logic used for rate limiting:

```
JavaScript
keyGenerator ({ headers, ip }: { headers: any, ip: any }) { return headers['X-Forwarded-For'] || ip }
```

This code attempts to generate a unique key for identifying the client making the request. It first checks if the `'X-Forwarded-For'` header is present in the request headers. If it is present, it uses the value of this header as the key. Otherwise, it falls back to using the client's IP address (`ip`).

The vulnerability here is that the `'X-Forwarded-For'` header can be easily spoofed by a malicious client. Since this header is typically used to forward the client's original IP address through proxy servers, an attacker can manipulate this header to appear as if the requests are coming from different IP addresses, bypassing the rate limit.

Response given by Gemini

R Which part of the code contains the vulnerability? Show drafts

◆ The vulnerable part of the code is within the `keyGenerator` function:

```
JavaScript
keyGenerator ({ headers, ip }: { headers: any, ip: any }) { return headers['X-Forwarded-For'] || ip }
```

Use code with caution.

Specifically, the line:

```
JavaScript
return headers['X-Forwarded-For'] || ip
```

Use code with caution.

This line checks for the presence of the `'X-Forwarded-For'` header in the request headers (`headers`). If it exists, it uses that value for the rate limiting key. Otherwise, it falls back to using the user's IP address (`ip`).

Since the `'X-Forwarded-For'` header can be spoofed, an attacker can manipulate this value to bypass the rate limit.

◆ The vulnerability in the provided code lies in the way the `keyGenerator` function is implemented. It relies on the `'X-Forwarded-For'` header, which can be spoofed by an attacker.

An attacker can send multiple requests with a spoofed `'X-Forwarded-For'` header to bypass the rate limit. This can be used to launch a Denial-of-Service (DoS) attack or brute-force password reset attempts.

Here's a more detailed explanation of the vulnerability:

- Rate Limiting:** This code snippet implements rate limiting for the password reset endpoint (`/reset/user/reset-password`). Rate limiting restricts the number of requests a user can make within a specific timeframe to prevent abuse.
- Spoofable Header:** The `keyGenerator` function uses the `'X-Forwarded-For'` header to identify the user making the request. This header is used in environments where a proxy server sits in front of the application server. The proxy server might add or modify the `'X-Forwarded-For'` header to indicate the original client's IP address.
- Vulnerability:** The problem is that attackers can spoof the `'X-Forwarded-For'` header in their requests. By setting a different IP address in the header, the attacker can bypass the rate limit imposed for their actual IP address. This allows them to send a large number of requests in a short period, potentially leading to a DoS attack or enabling them to brute-force password resets.

Vulnerability 3: Broken Authentication

- Password Strength

Response given by ChatGPT

ChatGPT

The vulnerability in the provided code is that it's hashing the password using a potentially insecure method. Without knowing the specifics of the `security.hash()` function, it's impossible to determine whether it's using a secure hashing algorithm.

Here are a few potential issues to consider:

- Weak Hashing Algorithm:** If `security.hash()` is using a weak or outdated hashing algorithm such as MD5 or SHA-1, it can be vulnerable to brute force attacks and rainbow table attacks.
- No Salt:** Salting is a technique used to add random data to the password before hashing, which increases the security of the hash. Without salting, identical passwords will generate the same hash value, making it easier for attackers to identify common passwords.
- No Iterations:** Hashing algorithms should be iterated multiple times to slow down brute force attacks. If `security.hash()` doesn't perform multiple iterations (e.g., using bcrypt's work factor), it may be susceptible to brute force attacks.
- Insufficient Length:** If the hashed password is too short, it may be susceptible to collision attacks where different passwords produce the same hash value.

You

Which part of the code contains the vulnerability?

ChatGPT

Apologies for the oversight. The vulnerability in the provided code lies within the custom setter function for the password field:

```
javascript
set (clearTextPassword) {
  this.setDataValue('password', security.hash(clearTextPassword))
}
```

The vulnerability is within the `security.hash()` function call. The specific implementation of `security.hash()` is not provided, so it's unclear whether it's using a secure hashing algorithm and whether it incorporates important security features like salting and multiple iterations.

Response given by Gemini

The provided code snippet shows a potential vulnerability related to password storage. Here's the breakdown:

- Storing passwords in plain text:** The code defines a setter function for the `password` attribute. However, it directly stores the `clearTextPassword` in the `this.setDataValue` call before hashing.
- Security Risk:** If an attacker gains access to the system during this brief window, they might be able to steal the password in plain text before it gets hashed.

This vulnerability is known as **Insecure Direct Object Reference (IDOR)**.

Which part of the code contains the vulnerability?

The vulnerable part of the code is within the `set` function for the `password` attribute:

```
JavaScript
set (clearTextPassword) {
  this.setDataValue('password', security.hash(clearTextPassword))
}
```

Use code with caution.

Specifically, the vulnerability lies in this line:

```
JavaScript
this.setDataValue('password', clearTextPassword)
```

Use code with caution.

Here, the `clearTextPassword` is assigned to the `password` attribute before it gets hashed using `security.hash(clearTextPassword)`. This temporary storage of the password in plain text creates the vulnerability.

Vulnerability 4: Improper Input Validation

- Admin Registration

Response given by ChatGPT

ChatGPT

The vulnerability lies in the `resource.create.send.before` middleware for the "User" model. When a new user is created, this middleware is triggered to create a wallet for the user. However, there's no check to ensure that the user creation was successful before attempting to create the associated wallet. If the user creation fails for any reason, such as validation errors or database constraints, the wallet creation will still be attempted, potentially resulting in orphaned wallet entries.

You

Which part of the code contains the vulnerability?

ChatGPT

Apologies for not being clear. The vulnerability lies in the `resource.create.send.before` middleware for the "User" model:

```
javascript
if (name === 'User') {
  resource.create.send.before((req: Request, res: Response, context: { instance: { id: string } }) => {
    WalletModel.create({ UserId: context.instance.id }).catch((err: unknown) => {
      console.log(err)
    })
    return context.continue
  })
}
```

In this code snippet, the wallet creation (`WalletModel.create`) is triggered before ensuring the successful creation of the user instance. If an error occurs during the user creation process, the wallet creation will still be attempted, leading to a potential inconsistency where a wallet is created without an associated user.

To address this vulnerability, you should ensure that the wallet creation occurs only if the user creation is successful, as shown in the suggested modification.

Response given by Gemini

◆ The vulnerability in the provided code lies in how it handles errors during wallet creation for new users.

Here's the breakdown:

- 1. Error Handling:** The code uses `catch(err: unknown) => { console.log(err) }` to handle errors during `WalletModel.create`. This simply logs the error to the console and doesn't stop the `context.continue` execution.
- 2. Potential Issue:** If `WalletModel.create` fails due to a database error or any other reason, the `context.continue` will still be called. This means the user creation might succeed in the `finalize` resource, but the wallet creation might fail silently.

This can lead to inconsistencies in the data. A user might be created without a corresponding wallet, causing issues with functionalities relying on wallets.

Here's a possible attack scenario (although unlikely):

- An attacker injects malicious data during user registration to trigger a specific error during `WalletModel.create`.
- The error handling inefficiency allows user creation to proceed while the wallet creation fails silently.

This scenario is far-fetched, but it highlights the importance of proper error handling.

◆ The vulnerable part of the code is located within the loop that iterates through the `autoModels` array:

```
JavaScript
for (const { name, exclude, model } of autoModels) {
  const resource = finalize.resource({
    model,
    endpoints: [ `/api/${name}s`, `/api/${name}s/:id` ],
    excludeAttributes: exclude,
    pagination: false
  })
  // vulnerable section within the 'User' resource creation
  if (name === 'User') {
    resource.create.send.before((req: Request, res: Response, context: {
      WalletModel.create({ UserId: context.instance.id })
    }).catch((err: unknown) => {
      console.log(err)
    })
    return context.continue
  })
}
}
```

Use code with caution.

Specifically, the vulnerability resides within the `catch` block used for error handling in the callback function attached to the `User` resource creation (`resource.create.send.before`). This block simply logs the error and allows execution to continue using `context.continue`.

Vulnerability 5: Injection

- Database Schema

Response given by ChatGPT

ChatGPT

The code provided is susceptible to SQL injection due to the way it constructs the SQL query using user-provided input (`req.query.q`). Specifically, the code constructs the SQL query string directly by concatenating the `criteria` variable into the query string without properly sanitizing or validating it.

Here's the vulnerable part of the code:

```
JavaScript
models.sequelize.query(`SELECT * FROM Products WHERE ((name LIKE '${criteria}' OR de`
```

If an attacker were to manipulate the `req.query.q` parameter to include SQL injection payloads, they could modify the behavior of the SQL query, potentially gaining unauthorized access to data or performing malicious actions on the database.

Response given by Gemini

◆ The vulnerability in the provided code is **SQL Injection**.

Here's why:

- 1. Untrusted Input:** The code retrieves the search criteria (`criteria`) from the user query parameter (`req.query.q`). This user-supplied input is not properly validated or sanitized before being used in the SQL query.
- 2. Direct String Concatenation:** The `criteria` is directly concatenated within single quotes (`'`) inside the SQL query using string interpolation (`SELECT * FROM Products WHERE ((name LIKE '${criteria}' OR description LIKE '${criteria}' AND deletedAt IS NULL) ORDER BY name)`). This creates an opportunity for an attacker to inject malicious SQL code into the query.

By crafting a specific string for the `q` parameter, an attacker could potentially:

- **Read sensitive data:** They could inject code to retrieve data from other tables or even the entire database.
- **Modify data:** Malicious code could update or delete existing product information.
- **Disrupt application functionality:** Injected code could interfere with the database operations, potentially causing denial-of-service attacks.

Which part of the code contains the vulnerability?

Show drafts

◆ The vulnerable part of the code is the construction of the SQL query itself:

```
Python
query = f'''SELECT * FROM Products WHERE ((name LIKE "${criteria}" OR de`
```

Use code with caution.

Specifically, the problematic part is the way user input (`criteria`) is directly concatenated within the single quotes (`'`) of the SQL string. This creates an opportunity for an attacker to inject malicious SQL code by crafting a specific string for the `criteria` parameter.

Here's a breakdown of the vulnerable part:

Part of the Code	Vulnerability
<code>f'''SELECT * FROM Products WHERE ((name LIKE "\${criteria}" OR de`</code>	User input (<code>criteria</code>) is directly concatenated within single quotes, allowing for SQL injection attacks.

Export to Sheets

- Login Admin

Response given by ChatGPT

ChatGPT

One potential vulnerability in the provided code is SQL Injection in the login function.

The vulnerability arises from directly concatenating user inputs (`req.body.email` and `req.body.password`) into the SQL query string without proper sanitization or parameterization. This can allow an attacker to manipulate the SQL query by injecting malicious SQL code.

Here's the vulnerable part of the code:

```
JavaScript
models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.email} || ''`
```

Response given by Gemini

The provided code has a potential vulnerability related to insecure data handling:

SQL Injection Vulnerability

The code snippet uses a raw string query to perform the login process:

```
JavaScript
models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.em
```

Use code with caution.

This approach is vulnerable to SQL injection attacks. An attacker could craft a malicious email address that injects additional SQL code into the query. This injected code could then be executed by the database, potentially allowing the attacker to:

- **Gain unauthorized access:** By manipulating the query, an attacker might bypass authentication and access user data or even gain administrative access to the database.
- **Disrupt data integrity:** Malicious code could corrupt or delete data stored in the database.

Which part of the code contains the vulnerability?

Show drafts

The vulnerable part of the code is within the login function's logic for handling user credentials:

```
JavaScript
models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.et
```

Use code with caution.

This section constructs a raw SQL query using string concatenation, which is susceptible to SQL injection attacks because it directly embeds user-provided data (email) into the query string.

Vulnerability 6: Sensitive Data Exposure

- Access Log

Response given by ChatGPT

The provided code appears to set up routes for serving various directories and files within an application. While it's difficult to identify specific vulnerabilities without a complete understanding of the entire application and its dependencies, there are some potential security concerns that could be present:

1. **Path Traversal:**
 - Anytime you're serving files based on user input, there's a risk of path traversal attacks. Ensure that the `file` parameter in routes like `/ftp/:file`, `/encryptionkeys/:file`, and `/support/logs/:file` is properly sanitized to prevent users from accessing files outside of the intended directory.
2. **Sensitive Information Disclosure:**
 - Serving directories like `/ftp`, `/encryptionkeys`, and `/support/logs` might inadvertently expose sensitive information to attackers if proper access controls are not in place. Ensure that directory listings are disabled or properly restricted to authorized users only.
3. **Unrestricted File Download:**
 - Depending on how `fileServer()`, `quarantineServer()`, `keyServer()`, and `logFileServer()` functions are implemented, there might be scenarios where users can download files without proper authorization checks. Ensure that access to sensitive files is properly authenticated and authorized.
4. **Static File Serving:**
 - Serving static files using `express.static` without proper access controls could expose sensitive files or directories to unauthorized users. Ensure that access to static files is restricted as necessary.
5. **Hardcoded Secrets:**
 - The use of `cookieParser('kekse')` seems to be setting a hardcoded secret for parsing cookies. Hardcoded secrets should be avoided as they can be easily discovered and exploited. Instead, use environment variables or a secure secrets management solution for storing sensitive information.
6. **No Input Validation:**
 - Ensure that appropriate input validation is performed on all user-provided input to prevent injection attacks and other security vulnerabilities.

Which part of the code contains the vulnerability?

ChatGPT

The vulnerability in the provided code lies in the routes that serve files based on user input, specifically in the routes that handle parameters such as `:file`.

For instance, the routes:

```
javascript
app.use('/ftp(?:/quarantine)/:file', fileServer())
app.use('/ftp/quarantine/:file', quarantineServer())
app.use('/encryptionkeys/:file', keyServer())
app.use('/support/logs/:file', logFileServer())
```

These routes incorporate user-provided input (`:file`) directly into the path without proper validation or sanitization, which can potentially lead to a path traversal vulnerability. An attacker could manipulate the `:file` parameter to access files outside of the intended directory or perform other malicious actions.

Response given by Gemini

The provided code snippet has several vulnerabilities:

1. **Directory Browsing:** The code enables browsing of sensitive directories like `/ftp`, `/encryptionkeys`, and `/logs`. This can expose information about the server's file structure and potentially leak sensitive data like file names and sizes.
2. **Unauthenticated Access:** There is no authentication layer implemented in the code. Ideally, users should be authenticated before they are allowed to access any of these functionalities like directory browsing or file download.
3. **Insecure Static File Serving:** The code uses `express.static` to serve static files from the `frontend/dist/frontend` directory. This can be insecure if the directory contains sensitive files that should not be publicly accessible.
4. **Weak Cookie Secret:** The code uses a hardcoded string "kekse" as the secret for the `cookieParser` middleware. This is a weak secret and should be replaced with a strong, random string of characters to improve security.

These vulnerabilities can be exploited by attackers to gain unauthorized access to sensitive information or files on the server. It's crucial to address these security issues to protect your application.

Which part of the code contains the vulnerability?

Show drafts

The following parts of the code contain vulnerabilities:

1. **Lines 1-4:** These lines use `serveIndexMiddleware` and `serveIndex` to enable directory browsing for `/ftp`, `/encryptionkeys`, and `/logs`. This exposes the directory structure potentially revealing sensitive information.
2. **Lines 6-8 & 11-13:** These lines define routes for file access in `/ftp` (excluding the quarantine subdirectory) and `/encryptionkeys` directories. They use `fileServer` and `keyServer` functions which likely handle file serving. Without seeing those functions, it's difficult to say definitively, but allowing access to these directories without proper authentication is a vulnerability.
3. **Line 16:** This line uses `express.static` to serve static files from the `frontend/dist/frontend` directory. The vulnerability here depends on the contents of that directory. If it contains sensitive files, this line exposes them publicly.
4. **Line 17:** This line uses "kekse" as the secret for the `cookieParser` middleware. This is a weak secret and can be exploited to tamper with cookies.

- Confidential Document

Response given by ChatGPT

ChatGPT

In the provided code, there are several routes defined for serving various directories and files. However, there is a potential vulnerability in the route handling for serving files from the `/ftp` directory.

The vulnerability lies in the regular expression used in the route definition for serving files from the `/ftp` directory:

```
javascript Copy code
app.use('/ftp(?:/quarantine)/:file', fileServer())
```

The regular expression `(?:/quarantine)` is a negative lookahead, which ensures that the route will not match if `/quarantine` is immediately followed by the file path. This prevents files from being served from the `/ftp/quarantine` directory.

However, this regular expression doesn't prevent access to files in subdirectories of `/quarantine`, such as `/ftp/quarantine/subdirectory/file.txt`.

Response given by Gemini

The provided code snippet has several vulnerabilities:

- Directory Browsing:** The code enables directory browsing for sensitive directories like `/ftp`, `/encryptionkeys`, and `/logs`. This can reveal information about the server's file structure and potentially leak sensitive data like file names and sizes.
- Unauthenticated Access:** There is no authentication layer visible in this code. Ideally, users should be authenticated before they are allowed to access any of these functionalities like directory browsing or file download.
- Insecure Secret Key:** The code uses a hardcoded secret key `"kekse"` for the cookie parser. This is insecure because anyone who can see the code will know the secret key. Secret keys should be generated securely and stored in a secure environment with proper access restrictions.

These vulnerabilities can be exploited by attackers to gain unauthorized access to sensitive information or files on the server.

The following parts of the code contain vulnerabilities:

- Lines 1-4:** This section handles the `/ftp` directory.
 - `app.use('/ftp', serveIndexMiddleware, serveIndex('ftp', { icons: true }));` This line enables directory browsing for the `/ftp` directory with icons.
 - `app.use('/ftp(?:/quarantine)/:file', fileServer());` This line allows users to download any file from the `/ftp` directory (except the `/quarantine` subdirectory) using the `fileServer` function.
- Lines 6-7:** This section handles the `/encryptionkeys` directory.
 - `app.use('/encryptionkeys', serveIndexMiddleware, serveIndex('encryptionkeys', { icons: true, view: 'details' }));` This line enables directory browsing for the `/encryptionkeys` directory with icons and detailed file information.
 - `app.use('/encryptionkeys/:file', keyServer());` This line allows users to access any file from the `/encryptionkeys` directory using the `keyServer` function.
- Lines 9-10:** This section handles the `/logs` directory under the `/support` path.
 - `app.use('/support/logs', serveIndexMiddleware, serveIndex('logs', { icons: true, view: 'details' }));` This line enables directory browsing for the `/support/logs` directory with icons and detailed file information.
 - `app.use('/support/logs/:file', logFileServer());` This line allows users to access any file from the `/support/logs` directory using the `logFileServer` function.
- Line 14:**
 - `app.use(cookieParser('kekse'));` This line uses a hardcoded secret key `"kekse"` for the cookie parser, which is insecure.

While the provided code snippet doesn't show the implementation of functions like `serveIndex`, `fileServer`, `keyServer`, or `logFileServer`, it's important to note that if these functions don't perform proper access control checks, they could introduce additional vulnerabilities.

Vulnerability 7: Unvalidated Redirects

- Allowlist Bypass

Response given by ChatGPT

ChatGPT

The provided code seems to be a list of allowed URLs and a function to check if a given URL is in that list. The vulnerability lies in the `isRedirectAllowed` function.

The vulnerability is in how the URLs are being checked. Instead of checking for an exact match of the allowed URLs, the code is using the `includes` method, which checks if the given URL contains any part of the allowed URL. This means that a malicious user could potentially craft a URL that includes parts of one of the allowed URLs, thus bypassing the intended restriction.

You

Which part of the code contains the vulnerability?

ChatGPT

Apologies for the oversight. The vulnerability lies within the `isRedirectAllowed` function, specifically in how it checks whether a given URL is allowed. The vulnerability arises from the usage of the `includes` method within the loop. This method checks if the given URL contains any part of the allowed URL, rather than checking for an exact match. This can lead to unintended URLs being considered as allowed if they contain parts of the URLs in the allowlist.

Response given by Gemini

◆ The vulnerability in the provided code lies in the way it checks if a URL is allowed for redirection. The issue is with the `url.includes(allowedUrl)` comparison.

Here's why it's vulnerable:

- Sub-String Matching:** `url.includes(allowedUrl)` checks if the `allowedUrl` is present anywhere within the `url`. This can be exploited by an attacker crafting a malicious URL that contains a legitimate substring from the allowed list. For example, if `"https://github.com"` is allowed, an attacker could create a malicious URL like `"[invalid URL removed]"` which would still pass the check.
- Not checking the entire URL:** The code only checks if a part of the URL exists in the allowed list, not the entire destination. This allows attackers to bypass restrictions by adding additional paths or parameters to a legitimate URL.

R Which part of the code contains the vulnerability?

Show drafts

◆ The vulnerable part of the code is within the `isRedirectAllowed` function, specifically in this line:

```
JavaScript
allowed = allowed || url.includes(allowedUrl)
```

Use code with caution.

Here's why this line is vulnerable:

- `url.includes(allowedUrl)`: This part checks if the `allowedUrl` (which is from the allowlist) exists anywhere within the `url`. This allows attackers to exploit substrings to bypass the check.

By relying solely on `includes`, the code doesn't ensure the entire URL matches an allowed entry.

- **Outdated Allowlist**

Response given by ChatGPT

ChatGPT

The provided code seems to implement a redirect allowlist feature, where certain URLs are allowed to be redirected to. However, there's a vulnerability in the `isRedirectAllowed` function.

The vulnerability lies in the usage of the `includes` method within the loop. The `includes` method performs a substring search, meaning it checks if the `url` contains the `allowedUrl` as a substring. This could lead to false positives or unexpected behavior. For example, if one of the URLs in the allowlist is a substring of another URL, it could mistakenly allow redirection to unintended URLs.

To fix this vulnerability, you should compare the entire URL strings instead of checking for substrings. You can use strict equality (`===`) to ensure that the URL exactly matches one of the allowed URLs in the allowlist. Here's the corrected code:

R You

Which part of the code contains the vulnerability?

ChatGPT

Apologies for the oversight. Let me clarify.

The vulnerability lies within the `isRedirectAllowed` function in the line:

```
typescript
allowed = allowed || url.includes(allowedUrl)
```

Copy code

Here, the `includes` method is used to check if the `url` contains `allowedUrl` as a substring. This approach poses a vulnerability because it doesn't require an exact match. If a URL in the `redirectAllowlist` is a substring of another URL, it could lead to false positives, allowing unintended redirections.

Response given by Gemini

◆ The vulnerability in the provided code lies in the way it checks if a URL is allowed for redirection. Here's the breakdown:

- Subdomain Takeover:** The `isRedirectAllowed` function checks if the provided URL substring matches any entry in the `redirectAllowlist`. This means an attacker could potentially craft a URL with a subdomain of a trusted site on the list and gain unauthorized redirection.

For instance, consider `"[invalid URL removed]"` - This URL contains a subdomain (`"evil"`) not present in the allowlist, but it still includes `"[invalid URL removed]"` which is allowed. This could be used to trick the application into redirecting the user to a malicious site.

- Path Traversal:** The check only considers if the allowed URL is a substring of the target URL. An attacker might be able to exploit this to redirect to unintended locations on the allowed domains if the application doesn't properly handle paths within the allowed URLs.

For example, if `"[invalid URL removed]"` is entered, it would be allowed because `"https://github.com/juice-shop/juice-shop"` is present.

R Which part of the code contains the vulnerability?

Show drafts

◆ The vulnerability exists within the `isRedirectAllowed` function specifically in this section:

```
JavaScript
for (const allowedUrl of redirectAllowlist) {
  allowed = allowed || url.includes(allowedUrl)
}
```

Use code with caution.

Here's why this part is vulnerable:

- `url.includes(allowedUrl)`: This line checks if the `allowedUrl` (from the allowlist) is present anywhere within the `url` string. This allows for subdomain takeover and potential path traversal issues as explained earlier.

Vulnerability 8: XSS

- **API-only XSS**

Response given by ChatGPT

ChatGPT

The vulnerability in the provided code lies in the `trustProductDescription` function, specifically in how it trusts and renders HTML directly from the product descriptions without proper sanitation. This opens up the possibility of a Cross-Site Scripting (XSS) attack if the product descriptions can be controlled by potentially malicious user input.

To mitigate this vulnerability, it's crucial to sanitize the HTML content before rendering it in the browser. Angular provides a built-in sanitizer service (`DomSanitizer`) for this purpose, which should be used to sanitize any user-generated or dynamic HTML content.

Here's how you can fix it:

```
typescript Copy code
import { DomSanitizer, SafeHtml } from '@angular/platform-browser';

// Inside your component class
constructor(private sanitizer: DomSanitizer) {}

trustProductDescription(tableData: any[]) {
  for (let i = 0; i < tableData.length; i++) {
    // Sanitize the HTML content before rendering
    tableData[i].description = this.sanitizer.bypassSecurityTrustHtml(tableData[i].d
  }
}
```

By using the `DomSanitizer` service to bypass security and mark the HTML content as safe, you prevent any potential XSS attacks by ensuring that the browser treats the content as trusted and not susceptible to script execution.

Response given by Gemini

The vulnerability in the provided code lies in the function `trustProductDescription`.

The code retrieves product descriptions and then calls `trustProductDescription` with the `products` data. Inside this function, it iterates through the products and uses `sanitizer.bypassSecurityTrustHtml` on the `description` field.

This function is bypassing the Angular sanitizer. The Angular sanitizer is there to prevent Cross-Site Scripting (XSS) vulnerabilities. XSS vulnerabilities allow attackers to inject malicious scripts into web pages. These scripts can then be used to steal data, redirect users to malicious websites, or perform other harmful actions.

Bypassing the sanitizer means that any untrusted data in the `description` field can be directly injected into the HTML of the page. This could allow an attacker to inject malicious scripts and compromise the security of the application.

Here's how this vulnerability can be exploited:

1. An attacker could inject a product with a malicious script in the `description` field.
2. When the application retrieves the product data, the script would be passed to the `trustProductDescription` function.
3. The sanitizer would be bypassed, and the script would be injected into the HTML of the page.
4. When a user views the product details, the malicious script would be executed, potentially compromising the user's security.

R Which part of the code contains the vulnerability?

Show drafts

The vulnerable part of the code is within the function `trustProductDescription`.

Specifically, this line:

```
TypeScript
tableData[i].description = this.sanitizer.bypassSecurityTrustHtml(tableData[i].description);
```

Use code [with caution](#).

Here, it's using `bypassSecurityTrustHtml` which bypasses the security measures designed to prevent XSS attacks.

- DOM XSS

Response given by ChatGPT

ChatGPT

The vulnerability in the provided code lies in how it's handling the input query parameter ("q") directly without proper sanitation. By using "bypassSecurityTrustHtml", it suggests that the query parameter is being directly rendered as HTML content without any sanitization, which can lead to Cross-Site Scripting (XSS) vulnerabilities if the query parameter can be controlled by a potentially malicious user.

To mitigate this vulnerability, it's essential to sanitize the query parameter before using it. Angular provides a built-in sanitizer service ("DomSanitizer") for this purpose.

Here's how you can fix it:

Here's how you can fix it:

```
typescript
import { DomSanitizer, SafeHtml } from '@angular/platform-browser';

// Inside your component class
constructor(private sanitizer: DomSanitizer) {}

filterTable() {
  let queryParams: string = this.route.snapshot.queryParams.q;

  if (queryParams) {
    queryParams = queryParams.trim();
    this.dataSource.filter = queryParams.toLowerCase();
    // Sanitize the query parameter before rendering
    this.searchValue = this.sanitizer.sanitize(SecurityContext.HTML, queryParams) as SafeHtml;

    this.gridDataSource.subscribe((result: any) => {
      if (result.length === 0) {
        this.emptyState = true;
      } else {
        this.emptyState = false;
      }
    });
  } else {
    this.dataSource.filter = '';
    this.searchValue = undefined;
    this.emptyState = false;
  }
}
}

```

By using "this.sanitizer.sanitize", you ensure that the query parameter is properly sanitized based on the security context provided ("HTML" in this case), which prevents any potential XSS attacks by removing or escaping any potentially harmful HTML content.

You

Which part of the code contains the issue

ChatGPT

Apologies for the oversight. The vulnerability in the provided code lies specifically in this line:

```
typescript
this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)

```

This line suggests that the "queryParam" is directly assigned to "searchValue" after bypassing HTML security checks. This could allow potentially malicious HTML content to be executed in the context of the application, leading to Cross-Site Scripting (XSS) vulnerabilities.

Instead, the "queryParam" should be sanitized before being assigned to "searchValue" to mitigate this vulnerability.

Response given by Gemini

The vulnerability in the provided code lies in the line:

```
JavaScript
this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)

```

Use code with caution.

Here's the breakdown of the vulnerability:

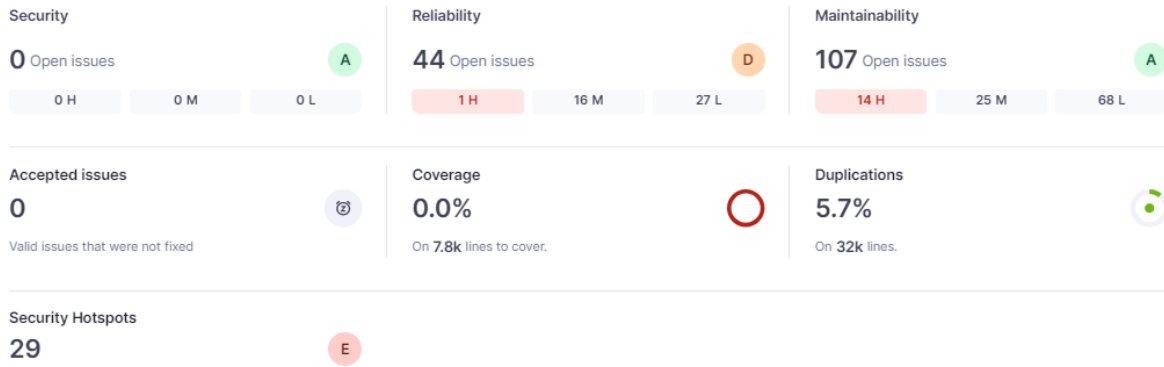
- `queryParam`: This variable holds the user input from the route query parameter.
- `sanitizer.bypassSecurityTrustHtml(queryParam)`: This line attempts to sanitize the `queryParam` using Angular's sanitizer. However, it uses the `bypassSecurityTrustHtml` method, which essentially bypasses sanitization altogether.

Bypassing sanitization is dangerous because it allows malicious users to inject HTML code into the `searchValue`. This injected code could then be displayed on the webpage, potentially leading to Cross-Site Scripting (XSS) attacks.

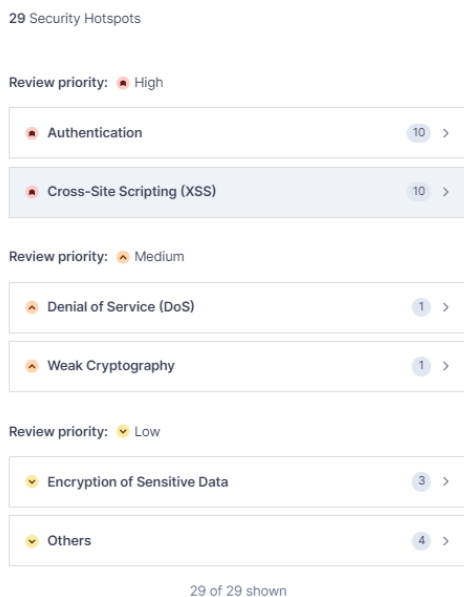
Here's how an attacker could exploit this vulnerability:

- The attacker constructs a URL with a query parameter containing malicious HTML code.
- When the application retrieves the data from the URL, the code isn't sanitized and gets stored in `searchValue`.
- When `searchValue` is displayed on the webpage, the malicious code gets executed, potentially allowing the attacker to steal user data, redirect users to malicious websites, or perform other harmful actions.

SonarQube on the other which was used from the side of "Traditional Code Analysis Tools" was only able to detect the XSS vulnerabilities from the 13 vulnerabilities selected above.



It was able to find numerous issues linked to reliability, maintainability and security. With security my main concern for this thesis.



The various security issues discovered by SonarQube are Authentication, Cross-Site Scripting (XSS), Denial of Service (Dos), Weak Cryptography, Encryption of Sensitive Data and others. Even though it was able to discover 29 security issues in the code, it missed out on 11 vulnerabilities related to Injection, Unvalidated Redirects, Sensitive Data Exposure, Improper Input Validation, Broken Authentication, Broken Anti Automation and Broken Access Control. The two issues out of the thirteen are shown below.

API-only XSS –

```

62 // vuln-code-snippet start restfulXssChallenge
63 ngAfterViewInit () {
64   const products = this.productService.search('')
65   const quantities = this.quantityService.getAll()
66   forkJoin([quantities, products]).subscribe(([quantities, products]) => {
67     const dataTable: TableEntry[] = []
68     this.tableData = products
69     this.trustProductDescription(products) // vuln-code-snippet neutral-line restfulXssChallenge
70     for (const product of products) {
71       dataTable.push({
72         name: product.name,
73         price: product.price,
74         deluxePrice: product.deluxePrice,
75         id: product.id,
76         image: product.image,
77         description: product.description
78       })
79     }
80     for (const quantity of quantities) {
81       const entry = dataTable.find((dataTableEntry) => {
82         return dataTableEntry.id === quantity.ProductId
83       })
84       if (entry === undefined) {
85         continue
86       }
87       entry.quantity = quantity.quantity
88     }
89     this.dataSource = new MatTableDataSource<TableEntry>(dataTable)
90     for (let i = 1; i <= Math.ceil(this.dataSource.data.length / 12); i++) {
91       this.pageSizeOptions.push(i * 12)
92     }
93     this.paginator.pageSizeOptions = this.pageSizeOptions
94     this.dataSource.paginator = this.paginator
95     this.gridDataSource = this.dataSource.connect()
96     this.resultsLength = this.dataSource.data.length

```

```

97   this.filterTable()
98   this.routerSubscription = this.router.events.subscribe(() => {
99     this.filterTable()
100   })
101   const challenge: string = this.route.snapshot.queryParams.challenge // vuln-code-snippet hide-start
102   if (challenge && this.route.snapshot.url.join('').match(/hacking-instructor/)) {
103     this.startHackingInstructor(decodeURIComponent(challenge))
104   } // vuln-code-snippet hide-end
105   if (window.innerWidth < 2600) {
106     this.breakpoint = 4
107     if (window.innerWidth < 1740) {
108       this.breakpoint = 3
109       if (window.innerWidth < 1280) {
110         this.breakpoint = 2
111         if (window.innerWidth < 850) {
112           this.breakpoint = 1
113         }
114       }
115     }
116   } else {
117     this.breakpoint = 6
118   }
119   this.cdRef.detectChanges()
120 }, (err) => { console.log(err) })
121 }
122
123 trustProductDescription (tableData: any[]) { // vuln-code-snippet neutral-line restfulXssChallenge
124   for (let i = 0; i < tableData.length; i++) { // vuln-code-snippet neutral-line restfulXssChallenge
125     tableData[i].description = this.sanitizer.bypassSecurityTrustHtml(tableData[i].description)
126   } // vuln-code-snippet vuln-line restfulXssChallenge
127 } // vuln-code-snippet neutral-line restfulXssChallenge
128 // vuln-code-snippet end restfulXssChallenge

```

Make sure disabling Angular built-in sanitization is safe here.

DOM XSS –

```

142 // vuln-code-snippet start localXssChallenge xssBonusChallenge
143 filterTable () {
144   let queryParams: string = this.route.snapshot.queryParams.q
145   if (queryParams) {
146     queryParams = queryParams.trim()
147     this.ngZone.runOutsideAngular(() => { // vuln-code-snippet hide-start
148       this.io.socket().emit('verifyLocalXssChallenge', queryParams)
149     }) // vuln-code-snippet hide-end
150     this.dataSource.filter = queryParams.toLowerCase()
151     this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParams)
// vuln-code-snippet vuln-line localXssChallenge xssBonusChallenge

```

Make sure disabling Angular built-in sanitization is safe here.

```

152   this.gridDataSource.subscribe((result: any) => {
153     if (result.length === 0) {
154       this.emptyState = true
155     } else {
156       this.emptyState = false
157     }
158   })
159 } else {
160   this.dataSource.filter = ''
161   this.searchValue = undefined
162   this.emptyState = false
163 }
164 }
165 // vuln-code-snippet end localXssChallenge xssBonusChallenge

```

For both these issues it was also able to correctly determine the part of the code responsible for the vulnerability.

