Daniel Evensen
Jostein Lind Aanby
Vebjørn Myklebust

Bridging the Gap Between Software Security and Development using CodeQL

Bachelor's thesis in Computer Science
Supervisor: Donn Morrison
May 2024

Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of Science and Technology

Daniel Evensen
Jostein Lind Aanby
Vebjørn Myklebust

Bridging the Gap Between Software Security and Development using CodeQL

Bachelor's thesis in Computer Science
Supervisor: Donn Morrison
May 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis evaluates the application of CodeQL as a static analysis tool to enhance cybersecurity practices, focusing on the knowledge gap identified by third-year computer science students at the Norwegian University of Science and Technology (NTNU). This gap highlights areas where the existing curriculum could be improved. The method involved using CodeQL on pre-existing applications, rather than implementing it in active development. This included phases of reviewing CodeQL documentation, environment setup, application on target repositories, and reflecting on the results.

Challenges in transitioning from theoretical learning to practical application revealed steep learning curves, and significant knowledge gaps in both vulnerability detection and CodeQL usage. Although no definitive vulnerabilities were identified, using CodeQL significantly enhanced the team's understanding of the tool and broader cybersecurity concepts. The study advocates for integrating tools like CodeQL into IT curricula to foster a security-first mindset among new developers. It emphasizes the importance of embedding security early in education to prepare for real-world challenges. This approach aims to reduce future vulnerabilities and enhance software security by integrating practical tools into software development education.

Sammendrag

Denne oppgaven evaluerer bruken av CodeQL som et statisk analyseverktøy for å styrke cybersikkerhetpraksiser, med fokus på kunnskapsmangelen observert av tredjeårs dataingeniørstudenter ved Norges Teknisk-Naturvitenskapelige Universitet (NTNU). Denne mangelen påpeker områder hvor den eksisterende læreplanen kan forbedres. Metoden inkluderte bruk av CodeQL på eksisterende applikasjoner, i stedet for å implementere det i en utviklingsprosess. Dette inkluderte faser for å gjennomgå CodeQL-dokumentasjon, oppsett av programmeringsmiljø, anvendelsen på kodebasen som ble undersøkt, og refleksjon av resultatene.

Utfordringene med overgangen fra teoretisk læring til praktisk anvendelse avdekket en bratt læringskurve og betydelige kunnskapsmangler i både sårbarhetdeteksjon og bruk av CodeQL. Selv om ingen definitive sårbarheter ble identifisert, forbedret bruken av CodeQL teamets forståelse av både verktøyet, og bredere cybersikkerhetskonsepter. Studien anbefaler integrering av verktøy som CodeQL i IT-læreplaner for å fremme en ”sikkerhet først” tankegang blant nye utviklere. Den understreker viktigheten av å integrere sikkerhet tidlig i utdannelsen for å forberede studenter på utfordringer i arbeidslivet. Denne tilnærmingen har som mål å redusere fremtidige sårbarheter og forbedre programvaresikkerheten.

Preface

This thesis marks the conclusion of a three-year Bachelor's program in Computer Science at the Norwegian University of Science and Technology (NTNU).

The original assignment to conduct a penetration test was requested by Donn Morrison, on behalf of the Department of Computer Science (IDI). During the course of the project the focus shifted towards the connection between software security and development, which aligned more with our goals and interests.

We would like to thank our supervisor, Donn Morrison, for his guidance and support throughout the project.

Trondheim, May 21th 2024

Daniel Evensen | Jostein Lind Aanby | Vebjørn Myklebust

Assignment Details

The description of the original thesis was ”Open-ended Pentest Project”. Through discussions with the supervisor the team decided to use CodeQL to conduct a penetration test. The initial thesis statement became ”Application of Static Analysis Tools on Open-Source Repositories”. During learning and applying CodeQL on the targets, the team realised a gap in their cybersecurity knowledge and changed the thesis statement to ”Bridging the Gap Between Software Security and Development using CodeQL”.

Contents

Figures

Acronyms

API	Application Programming Interface.
AST	Abstract Syntax Tree.
bqrs	binary query result set.
CI/CD	Continuous Integration and Continuous Delivery.
CLI	Command-Line Interface.
CR	carriage return.
CSRF	Cross-Site Request Forgery.
csv	comma-separated values.
CTF	Capture The Flag.
CVE	Common Vulnerabilities and Exposures.
CWE	Common Weakness Enumeration.
DAST	Dynamic Application Security Testing.
DHIS2	District Health Information System 2.
DOM	Document Object Model.
DoS	Denial of Service.
EMR	Electronic Medical System.
GDPR	General Data Protection Regulation.
GPT	Generative Pre-trained Transformer.
HIS	Health Information System.
HISP	Health Information Systems Programme.
HMIS	Health Management Information System.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
IoT	Internet of Things.
IT	Information Technology.
LF	line feed.
MITRE ATT&CK	MITRE Adversarial Tactics, Techniques, and Common Knowledge.
NTNU	Norwegian University of Science and Technology.
OOP	Object-oriented Programming.
OWASP	Open Web Application Security Project.
ReDoS	Regular expression Denial of Service.
regex	Regular expression.

SAST	Static Application Security Testing.
SDG	Sustainable Development Goals.
SQL	Structured Query Language.
SSRF	Server-Side Request Forgery.
UiO	University of Oslo.
URL	Uniform Resource Locator.
VS Code	Visual Studio Code.
XSS	Cross-Site Scripting.

1. Introduction

Threat actors and cybersecurity professionals are engaged in a constant tug-of-war, evolving and adapting in response to each other. While cybersecurity safeguards continuously improve, cyber threats are becoming increasingly sophisticated (Burt, 2020). Despite the critical threat cyber attacks pose to our increasingly digitized society, cybersecurity courses remain an optional discipline in many software engineering programs. This undermines the concept of ”Security by design”, which advocates for implementing security measures throughout the development process rather than merely an afterthought (Hansen and Kern, 2024).

This disconnect not only jeopardizes the security of digital solutions, but also creates a gap between software developers and dedicated cybersecurity teams. Bridging this gap is essential, not only to enhance the security of software but also to foster a more integrated approach where developers are proactive participants in cybersecurity efforts. For software developers with minimal or no cybersecurity knowledge, learning and effectively applying security tools and concepts might seem daunting.

This thesis posits that appropriate tools and educational approaches can significantly aid in this learning process. CodeQL, as one such tool, offers a promising pathway for new software engineers to bridge the gap between theoretical security knowledge and practical security application. This in turn underscores the vital role of integrated security education in software engineering curricula.

1.1 Relevance

The relevance of this thesis is supported by the dramatic increase in discovered vulnerabilities, which have risen by over 400 percent from 2016 to 2023 (‘Statistics Results’, 2024). This surge indicates an urgent need for software development practices to integrate robust security measures from the ground up. Cybersecurity is frequently absent from mandatory coursework in many computer science degree programs (‘NTNU Curriculum’, 2024). As a result, graduates often enter the workforce without a comprehensive understanding of secure coding practices. This lack of integration means that new developers are ill-equipped to address the complexities of modern cybersecurity threats, perpetuating a cycle of vulnerable software and increased risk.

The consequences of insecure code can be severe, with an estimated average cost of $4.45 million USD per data breach in 2023 ('Cost of a data breach report 2023', 2023). Vulnerabilities can also compromise personal privacy, potentially leading to identity theft and significant legal liabilities. Data breaches not only harm individuals but also erode public trust in digital services.

To effectively leverage security tools, comprehensive and accessible documentation is crucial. It facilitates easier adoption of the tool, helps users understand and utilize the tool's full capabilities, and reduces the learning curve associated with complex technologies.

While GitHub provides extensive documentation, Capture The Flags (CTFs), and a variety of blog posts designed to teach users how to use CodeQL for detecting vulnerabilities, these resources are often geared toward those who are already familiar with security practices.

This thesis positions CodeQL uniquely as both a static analysis tool, and an educational resource. Through hands-on experience, it offers a practical framework that enables developers to identify and mitigate vulnerabilities. Through the operational nature of how CodeQL works, it can also enhance their understanding of secure coding practices.

1.2 Purpose

The primary objective of this thesis is to evaluate the effectiveness of CodeQL as a means to enhance security knowledge among software developers, particularly those with limited exposure to cybersecurity. This study conducts an in-depth exploration of CodeQL, examining both its capabilities and the challenges it presents to novices in the field.

A significant aspect of this research is to identify gaps in cybersecurity knowledge that can be addressed through educational initiatives. By applying CodeQL, this thesis aims to pinpoint areas of confusion that new developers encounter, which can inform educational strategies to better integrate security into software engineering curricula.

Ultimately, this thesis seeks to demystify the process of adopting CodeQL as both a security tool and an educational resource. By clearly outlining how CodeQL can be used to both secure code and enhance understanding of software security, this work aims to fill educational gaps and raise security awareness among software developers, thereby making a significant contribution to the field of cybersecurity education.

1.3 Goals

1.3.1 Effect Goals

- Bridge knowledge gaps in security practices for software developers.
- Lower the bar for developers to use CodeQL actively in development.
- Make software developers more aware of possible vulnerabilities in their code during and after development.

1.3.2 Result Goals

- To find security vulnerabilities which a third-party could exploit with evil intent.
- Suggest improvement of education and documentation relating to secure software development.
- Identify the challenges for software developers to implement security tools effectively.

1.4 Scope

This thesis focuses on the utilization of CodeQL as a Static Application Security Testing (SAST) and learning tool for software developers with limited cybersecurity knowledge. The scope is confined to the cybersecurity and software development knowledge obtained through the computer science curriculum at NTNU, from the perspective of third-year students. CodeQL is applied not only to identify vulnerabilities but also as a continuous educational tool to enhance software security practices. In this study, CodeQL is primarily applied to Java REST applications. Although Java serves as the primary context, the insights and methodologies developed are intended to be adaptable across various programming environments.

Initially, the project aimed to employ CodeQL for penetration testing. However, upon identifying the knowledge gap in cybersecurity within the team, the focus was adjusted to use CodeQL as an educational tool as well as a traditional SAST tool. This shift expanded the scope beyond simple vulnerability detection to include an educational component aimed at understanding security issues.

This research did not integrate CodeQL into ongoing software development processes. Instead, it involved analyzing pre-existing, complete Java REST applications. Importantly, the insights gained from employing CodeQL on these applications are applicable to integrating the tool within development environments.

1.5 Structure

Chapter 2 presents an introduction to CodeQL and necessary theory. Chapters 3 and 4 detail the methodology and results of learning and applying CodeQL. Chapter 5 evaluates the effectiveness of using CodeQL as a learning tool to bridge the cybersecurity knowledge gap among developers. The findings and their implications for software development education are discussed in Chapter 6, which concludes the paper. The societal impact of this thesis is discussed in Chapter 7.

2. Theory

To understand this thesis, it is essential to grasp the underlying concepts and theories. This chapter provides an overview of software security practices and methodologies, including the foundational knowledge necessary to comprehend and apply these principles.

2.1 Static Code Analysis

Analysis is an integral part of the quality assessment for a system or application, especially considering security and potential vulnerabilities.

Static code analysis is defined as the analysis of source code without executing the application (Dewhurst, 2020). This is commonly done through SAST tools. These tools work by analyzing the codebase to identify potential vulnerabilities based on predefined rules and criteria. It is usually done during code review, or continuously during development by adding it to the Continuous Integration and Continuous Delivery (CI/CD) pipeline. This allows analysts and developers to address them before it is added to production, enabling correction before any faults can be exploited.

SAST tools can be applied manually and automatically. Some of the limitations of the automatic appliance, like accuracy and codebase-specific issues, can be handled by performing them manually.

Fixing vulnerabilities early in the development cycle can significantly reduce the cost compared to doing it in production (Hossain, 2018). Being able to incorporate it for automatic scanning in the CI/CD is also an upside, which will stop the integration process if any vulnerabilities are detected. Due to the analysis being static, it is able to cover 100% of the sourcecode, including the configuration files (Gillis, 2020). In comparison, Dynamic Application Security Testing (DAST) only covers the execution of a program, limiting its effectiveness.

SAST does come with downsides. The resulting output does not always provide context for the vulnerability, which can create a challenge in addressing the issue. It can also produce a large amount of false positives.

Taint Analysis

Taint analysis is a technique used to track the flow of sensitive data through a program where the variables are affected, or ”tainted”. This provides an overview of the different variables that are directly or indirectly affected by unsanitized user input. (Dewhurst, 2020)

Data Flow Analysis

Data flow analysis collects information on the dynamic flow of the data. This can provide insight into where data is improperly handled. The results can be represented in a data flow graph, where the path of the data is displayed and available for further analysis. (Dewhurst, 2020)

2.2 CodeQL

CodeQL is a semantic code analysis engine used for static code analysis. It lets a user query code as if it were data (‘CodeQL’, n.d.).

It is a declarative, object-oriented logic programming language designed to query complex, potentially recursive data structures within a relational data model (Avgustinov et al., 2016). It was initially developed by Semmle, which has been a part of GitHub since 2019 (Friedman, 2019).

CodeQL is tailored for static code analysis, allowing developers to identify vulnerabilities and bugs across various programming languages. It compiles into a variant of Datalog, which is a declarative logic programming language. In Datalog, you specify what you want to achieve rather than how to achieve it, meaning you declare the desired results, and the system determines how to compute them. This allows CodeQL to perform logical operations and handle recursive predicates efficiently, making it easier to write complex queries without procedural details (‘What is Datalog’, n.d.).

Query Structure

CodeQL has a similar syntax to Structured Query Language (SQL), with some differences.

```
/**
* Query metadata
*/

import /* ... CodeQL libraries or modules ... */

/* .. Optional, define CodeQL classes and predicates ... */

from /* ... variable declariations ... */
where /* ... logical formula ... */
select /* ... expressions ... */
```

Figure 2.1: Basic query structure in CodeQL (‘CodeQL Documentation’, 2024).

When familiar with the structure of SQL queries, one can instantly spot the differences in the structure used for CodeQL queries as seen in Figure 2.1. In CodeQL, the keywords are ordered differently: **from**, **where**, and then **select**. The **from** keyword specifies the data sources, **where** filters the data based on conditions, and **select** determines the data to be returned. Additionally, CodeQL includes metadata information in its queries, which helps distinguish between different queries and dictates how to display the results (‘Metadata for CodeQL queries’, n.d.). CodeQL also allows the definition of classes and predicates to be used within the query, providing a way to encapsulate logic and improve query organization.

Predicates are fundamental units in CodeQL that represent logical conditions or properties (Avgustinov et al., 2016). An example of this is the ”isSmall()” predicate in Figure 2.2 used to define an integer between 1 and 9.

```
predicate isSmall(int i) {
    i in [1 .. 9]
}
```

Figure 2.2: Predicate in CodeQL(‘CodeQL Documentation’, 2024).

CodeQL utilizes classes to define sets of values with similar properties. These classes can have methods and properties, similar to traditional Object-oriented Programming (OOP). Subclassing and inheritance are supported, where classes represent logical implications, and subclasses denotes a subset relationship (Avgustinov et al., 2016). Abstract classes are used to create templates that other classes can implement. These do not directly correspond to data but define a contract for subclasses to implement specific methods or predicates.

The class in Figure 2.3 represents integers between 1 and 10, and adds a method to square them. This is then used in the query.

```
class SmallInt extends int {
    SmallInt() { this in [1..10] }
    int square() { result = this*this }
}

from SmallInt x, SmallInt y, SmallInt z
where x.square() + y.square() = z.square()
select x, y, z
```

Figure 2.3: Class in CodeQL ('CodeQL Documentation', 2024).

CodeQL allows for multi-valued expressions, which means that predicates can return multiple values. This enables complex queries that mimic function calls in traditional programming. Similar to OOP, it supports method overriding and dynamic method dispatch, enabling polymorphic behavior by selecting methods at runtime according to the most specific type of the object involved. (Avgustinov et al., 2016)

Data Flow

Data flow analysis identifies the potential values a variable can take on at various points in a program and tracks how those values move through the code.

In CodeQL, data flow analysis is done by constructing a data flow graph. Unlike an Abstract Syntax Tree (AST), which represents the program's syntax, the data flow graph models the way data moves during runtime. AST nodes represent syntactic elements such as statements or expressions, showing how the code is structured. In contrast, nodes in the data flow graph represent semantic elements that carry values at runtime, reflecting how data flows through the program.

CodeQL supports both local and global data flow analysis. Local data flow considers only the data flow within a single function, making it efficient and precise. Global data flow, on the other hand, considers data flow between functions and through object properties, providing a comprehensive view of data movement across the entire program.

By constructing and analyzing the data flow graph, CodeQL can effectively track how data moves through a program, helping identify vulnerabilities and bugs that other static analysis tools might miss.
('About dataflow analysis', n.d.)

Practical Implementations of Queries

The practical implementation of queries starts by creating a CodeQL database from the target code. This database can be used to run queries against, essentially querying the actual target code itself. A typical query specifies a set of conditions to be validated against the created database, like identifying unused variables, detecting potential buffer overflows, or uncovering cross-site scripting vulnerabilities. The CodeQL engine optimizes the execution plan for these queries to efficiently process large datasets, which often encompass entire codebases, ensuring comprehensive analysis and accurate results. (Avgustinov et al., 2016)

Standard query suites

Included in the CodeQL language packages are a suite of community written queries hosted by GitHub. These queries are classified by the corresponding Common Weakness Enumerations (CWEs) items the query addresses. (‘CodeQL Documentation’, 2024)

2.3 MITRE

The MITRE Corporation is a not-for-profit organization which is well-known for its contribution to cybersecurity through the development of frameworks and systems (contributors, 2024).

MITRE ATT&CK

The MITRE Adversarial Tactics, Techniques, and Common Knowledge (MITRE ATT&CK) framework is a comprehensive knowledge base that provides detailed information about the tactics and techniques used by cyber adversaries during various stages of the attack lifecycle (Matke, 2023).

The MITRE ATT&CK framework is displayed in several matrices, each of which focuses on a specific platform or environment (‘ATT&CK Matrix for Enterprise’, 2024).

Each tactic and technique contain information on methods attackers use for accomplishing their goals, as well as suggestions for mitigations. It is a helpful tool for enterprises when it comes to making their products safe from attacks. It is particularly valuable for improving incident response and threat-hunting capabilities.

CWE

CWE is a community-developed list found on MITRE's websites, and contains common weaknesses in software and hardware. Having an updated list of common weaknesses helps system developers know which weaknesses might lead to vulnerabilities, allowing them to eliminate these possible vulnerabilities before deployment, making the development process easier and cheaper. ('About CWE', 2024)

CVE

Common Vulnerabilities and Exposures (CVE) is a standardized identifier for a known security vulnerability in software or hardware. Each CVE entry provides a reference number, a brief description of the vulnerability, and any relevant information to help users and organizations protect their systems from potential exploits ('CVE', n.d.).

2.4 OWASP Top Ten

Open Web Application Security Project (OWASP) Top Ten lists the most critical security vulnerabilities found in web applications. It is made using data from a number of contributors around the world. The data is then normalized and analyzed, creating an overview of the top security risks and vulnerabilities. Eight are taken from the data analysis, and two are taken from a survey given to contributors. The most updated list (OWASP Top Ten 2021) is:

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery (SSRF)

It is an essential part of the required knowledge when developing a secure system or application, as well as a component of testing systems and finding vulnerabilities.

OWASPs Top Ten is not a sufficient list or resource on its own. However, it is a good reference for coding, reviews, tool support and penetration testing.

It provides valuable insights and gives a starting point when approaching a penetration test or risk assessment.
('OWASP Top Ten', 2023)

2.5 Importance of Security Knowledge for Software Developers

Companies have a responsibility to protect the data collected from their users. In recent years, this responsibility has led lawmakers to formalize laws related to data protection, one of these being General Data Protection Regulation (GDPR). GDPR aims to enhance personal data security and standardize data protection laws across Europe, ensuring that individuals have greater control over their personal information (Wolford, 2024). For developers, being well-versed in the security and data protection requirements stipulated by GDPR is crucial for the responsible development of modern products and services.

Security by design

Security by design is an approach to software development aimed at creating systems that are as free of vulnerabilities as possible. This involves continuous testing, authentication safeguards, and adherence to best programming practices. Emphasizing security from the start counters the common tendency to treat security as an afterthought.

This approach is particularly crucial in the rapidly developing Internet of Things (IoT) environment, where almost any device can be networked and must be secure by default.

For developers, understanding and implementing security by design principles is essential. Integrating these principles ensures the development of secure, compliant, and trustworthy products.
(Hansen and Kern, 2024)

2.6 Vulnerabilities

This section discusses various types of software vulnerabilities, explaining their mechanisms, potential impacts and mitigations.

Log Injection

Log injection occurs when unvalidated user input is written to log files, allowing attackers to forge log entries or inject malicious content. This can lead to false log events, Cross-Site Scripting (XSS) attacks, and command execution. For example, an attacker can submit a specially crafted string to insert arbitrary log entries or even execute commands if the log file is accessible and processed by the application. To prevent this, user input should be validated and sanitized before logging. (‘Log Injection’, n.d.)

SQL Injection

SQL injection is an attack method where the attacker attempts to insert, or inject, harmful SQL statements as user input from the client to the application. These statements can allow the attacker to read sensitive data from the database, modify database data, and execute administrative operations on the database. Effective countermeasures include using prepared statements and parameterized queries, as well as sanitizing all user inputs. (‘SQL Injection’, n.d.)

HTTP Response Splitting

Hypertext Transfer Protocol (HTTP) response splitting occurs when untrusted data, often from an HTTP request, is included in an HTTP response header without proper validation. If the data contains carriage return (CR) and line feed (LF) characters, it can split the response, allowing attackers to control headers and create additional responses. This can lead to attacks such as cross-user defacement, cache poisoning, XSS, and page hijacking. To mitigate, input should be sanitized to prevent CR and LF characters from being included in response headers. (‘HTTP response splitting’, n.d.)

Unsafe Deserialization

Unsafe deserialization is an attack where untrusted data is deserialized, potentially leading to application abuse, Denial of Service (DoS), or arbitrary code execution. Attackers can exploit this by sending malicious serialized data to manipulate the logic of the application. To prevent this, validation and sanitization of all serialized data is required, as well as avoiding deserializing data from untrusted sources. (‘Unsafe Deserialization’, n.d.)

Path Injection

A path injection attack, also called path traversal, is a security vulnerability that occurs when an application takes user input to access file system directories and does not properly sanitize the input. This can allow attackers to access files and directories stored outside the intended directory root.

To do this the attacker utilizes path traversal, like dot-dot-slash (../) sequences, or absolute file paths in inputs to navigate the file system to restricted areas. This makes it possible to access arbitrary files and directories stored in file systems, including application source code or configuration and critical system files.

To protect against this, it is important to avoid passing user inputs directly to the file system's API where possible. If not possible, the recommended method is using whitelisted inputs. ('Path Traversal', n.d.)

ReDoS and Regex Injection

Regular expression Denial of Service (ReDoS) is a type of DoS attack where a Regular expression (regex) takes an excessively long time to compute, which can freeze the application. This can significantly disrupt system operations when exploited. Regex injection is the act of inserting a malicious regex pattern into an application, potentially causing a ReDoS. When it comes to handling regex related vulnerabilities, the safest approach is usually to adopt a whitelisting instead of blacklisting. ('ReDos', n.d.)

CSRF

Cross-Site Request Forgery (CSRF) is an attack where an authenticated user is manipulated into performing actions they did not intend on a web application. By leveraging social engineering techniques, such as sending a deceptive link via email or chat, an attacker can trick users into executing actions chosen by the attacker. For regular users, this can result in state-changing activities like transferring money or updating account details. If the victim is an administrator, a CSRF attack can potentially compromise the entire web application. ('CSRF', n.d.)

SSRF

SSRF is a vulnerability where an attacker exploits server functionality to access or modify internal resources. By supplying or modifying a Uniform Resource Locator (URL) that the server processes, the attacker can select URLs that allow them to read server configurations like relevant metadata, connect to internal services like HTTP-enabled databases, or send POST requests to internal services not meant to be publicly accessible. ('SSRF', n.d.)

XSS

XSS is a web security vulnerability that allows attackers to inject malicious scripts into web pages. XSS attacks exploit the way browsers process HyperText Markup Language (HTML) and JavaScript, enabling attackers to bypass access controls and perform unauthorized actions on behalf of the user. These attacks can lead to data theft, session hijacking, and other malicious activities. XSS vulnerabilities arise when a web application includes user input directly in web pages without proper validation or escaping. There are three main types of XSS attacks: Reflected, Stored, and Document Object Model (DOM)-Based. Effective defense against XSS is validating and sanitizing all user inputs. (‘Cross Site Scripting’, n.d.)

2.7 CTF

CTFs are computer security competitions where teams or individuals tackle various challenges. They are popular for starting cybersecurity careers due to their team-building and competitive nature. (‘Capture The Flag 101 Overview’, 2024).

2.8 GPT-4

Generative Pre-trained Transformer (GPT)-4 is a multimodal language model created by OpenAI, capable of generating text from both text and graphic inputs. Essentially, GPT-4 can understand and produce human-like text, making it useful for a variety of applications (Lutkevich, 2023).

3. Method

Research methods are systematic techniques used to conduct and evaluate studies, ensuring the reliability and validity of findings. The case study research method is a detailed investigation of a specific instance within a real-world context.

In this project, this involves the application of CodeQL as a SAST tool and educational resource. This approach allowed for an in-depth exploration of the practical challenges and learning outcomes associated with using security tools in software development.

3.1 Work and Role Distribution

The team, consisting of three third-year Computer Science students, established a collaboration agreement during the pre-project phase, which assigned specific roles including team-leader, document manager, and quality assurance manager. The distribution of the project workload was flexible. A Trello board was used to manage and track tasks, allowing for dynamic adjustments and balanced contributions from all team members.

3.2 Choice of Technology and Methodology

This section explains the selection of technologies and methodologies used in this study.

3.2.1 CodeQL

CodeQL was chosen due to its advanced capabilities in conducting static application security testing and its adaptability across various programming environments. Its ability to transform entire codebases into queryable databases allows for deep security analyses, making it particularly effective in identifying complex vulnerabilities and enhancing security awareness among developers.

3.2.2 GPT-4

GPT-4 was chosen as a supplementary tool due to its efficiency in processing extensive documentation and gathering relevant information. Additionally, it was selected for its capability to clarify the logic behind CodeQL queries and provide assistance in debugging commands.

3.2.3 Threat Modeling Frameworks

Threat modeling frameworks such as OWASP Top Ten, MITRE CWE and MITRE ATT&CK were chosen to guide and structure the analysis process. OWASP Top Ten provides a foundational overview of the most critical web application risks, directly informing the security checks implemented via CodeQL. Additionally, MITRE CWE and MITRE ATT&CK offers detailed insights into various technical components and attack techniques, enhancing the understanding of potential vulnerabilities and the methods by which they can be exploited.

3.2.4 Methodological Approach

The methodology of this project was intentionally designed to be open and exploratory to maximize learning opportunities and identify a broad range of knowledge gaps. This approach allowed the team to experiment with various aspects of CodeQL in real-world scenarios, thereby not only identifying but also attempting to understand possible vulnerabilities.

3.2.5 Administrative Strategy

To structure the project, it was split into three different phases, each representing a different type of activity.

Research

The first part of this project consisted of a research phase, where the team spent time researching the choice of technology, CodeQL, as well as learning how to use it.

Implementation

The second part of this project consisted of an implementation phase, where the team used the acquired knowledge and applied it in a real-world scenario, getting to know the target codebases. This phase ran in tandem with the documentation phase, and mostly consisted of writing CodeQL queries against targets while documenting the process and results.

Documentation

The final part consisted of the documentation phase, where results were documented as well as writing the final report.

Meetings

Meetings with supervisor were chosen to be scheduled on demand rather than with a set frequency to avoid unnecessary meetings.

Internal team meetings and discussions were scheduled on a much higher frequency. The team had daily stand-ups, presenting individual work and results, as well as the plan for the day. Every week started with the team discussing the plan for the coming week, and every week ended with a re-cap meeting. This was done to ensure a clear understanding of the results' implications.

3.3 Choosing Targets

The selection of targets for this study focused on open-source health applications, recognizing their critical importance in managing sensitive health information. The need for high security standards to protect patient privacy and data integrity made these applications suitable choices. Additionally, the applications being open-source made it easy to access for analysis.

After reviewing various open-source health applications, the team decided to conduct static analysis on two specific targets: Open Hospital and District Health Information System 2 (DHIS2).

3.3.1 Open Hospital

Open Hospital, developed by Informatici Senza Frontiere, aims to provide a free and open-source software solution for automating health centers, particularly in developing countries. The project's website describes its mission as offering ”a cost-effective and flexible software platform for Electronic Medical System (EMR)/Health Information System (HIS) adoption” ('About Open Hospital', 2024). The decision to choose Open Hospital as a target was influenced by its low incidence of CVE reports during preliminary research and an initial review of the source code.

3.3.2 DHIS2

DHIS2 is a web-based, open-source software managed by the Health Information Systems Programme (HISP) centre at the University of Oslo (UiO) and serves as a Health Management Information System (HMIS). It is employed in over 100 countries, with a particular emphasis on supporting low- and middle-income nations ('About DHIS2', n.d.). The decision to include DHIS2 as a secondary target was driven by a desire to broaden the scope of the study, allowing for a comparative analysis between different applications.

3.4 Research and Implementation of CodeQL

This section details the systematic approach to research and implement CodeQL in the case study. This involves a comprehensive review of the CodeQL documentation, practical experience through CodeQLs CTFs, and a survey to identify knowledge gaps.

3.4.1 Review of Documentation

The first phase involved a thorough review of the official CodeQL documentation; segmented into Background Information, Guides, Tools, and Reference Docs (‘CodeQL Documentation’, 2024).

The ’Background Information’ section provided a fundamental overview of CodeQL’s purpose and applications. It introduced core concepts essential for beginners, facilitating an initial understanding of the tool’s capabilities and its role in software security.

The documentation’s sections on ’CodeQL Tools’ and ’CodeQL Reference Docs’ provided comprehensive resources for advanced query management and syntax references. Utilized throughout the project, these resources were used for setting up the CodeQL environment and adjusting queries during the implementation phase. The ’CodeQL for Visual Studio’ section detailed the setup process, involving downloading the extension and accessing a CodeQL database to write and run queries.

The ’CodeQL Guides’ section delved into the technical specifics of writing and structuring queries, as well as some important features like data flow analysis and defining results of a query. Each guide featured practical exercises. This section also included a series of puzzles that used a training database called ’People’. Each puzzle came with instructions, constraints, and clues to help craft queries to identify the correct person.

3.4.2 Using CodeQL CTF

The CodeQL CTF challenges are based on real, discovered vulnerabilities in code. Each challenge includes a pre-configured CodeQL database reflecting the state of a codebase before a known vulnerability was addressed, along with detailed instructions on how to set up and begin the challenge. The challenge is then to discover the same vulnerability using CodeQL. The challenges are split into several steps, each having multiple questions.

The instructions guided participants through the process of discovering the vulnerabilities, starting with broad queries and incrementally refining them based on provided hints and guidance. The CTFs provided practical experience in applying CodeQL to real-world code scenarios.

3.4.3 Setup of CodeQL Environment

To execute queries, it is necessary to first create a CodeQL database. For this specific project this involved cloning the target repository locally and integrating the build command for the project within a single CodeQL Command-Line Interface (CLI) command to generate the database. The CLI was installed using a package manager. Both the Visual Studio Code (VS Code) CodeQL extension and the CodeQL CLI were utilized to write and execute queries against this database.

While using the CLI provides unformatted results, the VS Code extension includes a viewer that enables interaction with query results. To replicate the functionality of the VS Code extension, specifically the ability to run custom queries against a database and receive formatted results, simple shell scripts were developed.

3.4.4 Applying CodeQL on Target Repositories

After establishing the CodeQL databases and setting up the environment, the next phase involved running queries against the target repositories. The team employed OWASP to outline a framework for what vulnerabilities to look for. Initial efforts were concentrated on writing completely custom queries. After getting more familiar with CodeQL, the standard scan was used as a starting point instead of writing queries completely from scratch. The scan generated a categorized list of potential vulnerabilities, enabling further investigation at the source code locations of the results.

Two distinct methodologies were adopted to address the results based on the vulnerability category. For categories with few results, direct source code examination was performed to assess if the generated warnings were false positives. This involved reading the source code and documentation, alongside gaining an understanding of the specific vulnerability to distinguishing between safe and unsafe Java code implementations.

For categories yielding numerous results, a manual code audit of all results would be impractical; thus, the team opted to refine the original queries to reduce noise. The refinement process began by isolating the query from the standard GitHub CodeQL repository and running it separately within the CodeQL environment. The results were then grouped and analyzed focusing on either the sinks or the sources. If a result was identified as a false positive—often due to the standard query's inability to recognize custom security measures—a new segment of CodeQL code was crafted to exclude such secure implementations from subsequent results. This process of refining the query was done repeatedly, each cycle aiming to diminish the number of results until all potential issues within the category were examined.

3.4.5 Learning with CodeQL

To get a comprehensive understanding of the security vulnerabilities investigated, the OWASP and MITRE frameworks was used. The CWE numbers provided in the metadata of standard CodeQL queries and the naming of the vulnerability categories served as a springboard for further research through OWASP and MITRE.

Whenever a standard scan flagged a potential security issue, the team referred to the corresponding CWE entry within the MITRE database. This approach provided detailed descriptions of each vulnerability, including common examples of unsafe implementations and guidelines for secure coding practices. OWASP resources were similarly employed for practical and theoretical information on the respective vulnerability categories.

MITRE ATT&CK served as a learning platform, facilitating comprehensive exploration of potential attack vectors.

3.4.6 Identifying Knowledge Gaps

During the project, a systematic approach was implemented to identify knowledge gaps. The process to uncover these gaps involved a combination of discussion within the team and documentation of challenges encountered by team members while using the tool.

3.4.7 Surveying Cybersecurity Knowledge

To understand the prevalence of cybersecurity knowledge gaps among computer science students, a survey was conducted targeting this demographic. The survey was designed to gauge the respondents' self-assessed knowledge in several key areas: general cybersecurity, static code analysis, and the use of SAST tools. Additionally, participants were asked to evaluate how frequently they consider security practices during their software development processes, with responses ranging from one to ten.

This allowed for a quantification of knowledge gaps and identification of trends in security practices, providing a broader context to the findings from the hands-on work with CodeQL.

4. Results

This section presents the outcomes of the project, divided into scientific and administrative results. The scientific results focus on technical findings, while the administrative results cover managerial aspects.

4.1 Scientific Results

As detailed in Chapter 3.4, the approach was exploratory, aiming to comprehensively understand the application of CodeQL in identifying security vulnerabilities. The scientific results encompass the team's proficiency in CodeQL, the queries created during implementation, and the cybersecurity learning outcomes and challenges encountered.

CodeQL CLI

In the exploration of CodeQL's capabilities for integrating into software development, the CLI was integral. Its versatility allows integration across different environments, making it particularly useful for CI/CD pipelines. This adaptability ensures that CodeQL can be effectively employed by developers in various stages of the software development lifecycle.

Throughout the project, the CodeQL CLI was employed for several tasks. Specifically to create the CodeQL database, run suites of queries, run specific queries and decode the results from these queries:

codeql database create The command used to create a database from source code. The command takes arguments of output and build locations, language(s), build commands, and more.

codeql database analyze Command to run a suite of queries. If not specified it will run the standard pack of CodeQL queries provided by Github.

codeql query run This command runs one single query and provides the results in a binary query result set (bqrs) file. It takes the arguments; path to query, path to database and path to output bqrs file.

codeql bqrs decode The decode command was used to interpret the resulting bqrs file to a human readable format like comma-separated values (csv).

Figure 4.1 shows an example of using the command to create the database for Open Hospital's API repository:

```
codeql database create --language=java --source-root=../openhospital-api
--command="mvn clean install -DskipTests=true" ./dbs/api-db
```

Figure 4.1: Command for creating the CodeQL database from Openhospital's API repository.

The CodeQL CLI is absent of the results viewer that the VS Code extension has. This was mitigated by using third party CLI tools like "csvtotable" that could format the csv file to HTML, which provided a more functional result viewing experience.

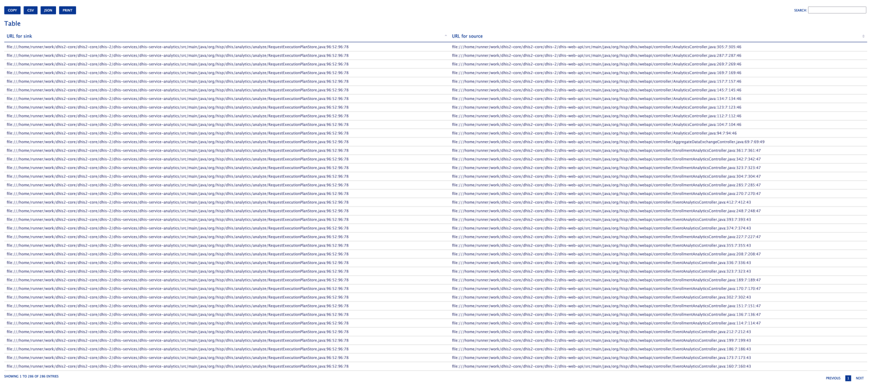


Figure 4.2: Results of a query formatted to HTML with "csvtotable".

Shell scripts were used to automate the process of running, interpreting and formatting the results of queries. Figure 4.3 is an example of a shell script taking an argument for the query file and running it against a pre-defined database. The results are decoded from binary and formatted into HTML so the results of a query can be viewed in a browser, very similar to the VS Code CodeQL extension.

```
#!/bin/sh

# The first argument is the filname of the query
QUERY_FILE="$1"
# The second argument is the database name
DATABASE_NAME="core-db"

# Construct the full paths
QUERY_FULL_PATH="queries/%QUERY_FILE"
DATABASE_FULL_PATH="dbs/$DATABASE_NAME"

codeql query run "$QUERY_FULL_PATH" --database="$DATABASE_FULL_PATH"
-- output="query_results.bqrs"
codeql bqrs decode --format=csv
--output=results/query_results.csv --entities=url query_results.bqrs
echo "Results:"
tail +2 ./results/query_results.csv | wc -l
csvtotable results/query_results.csv results/results.html -o
open results/results.html
```

Figure 4.3: Script used for running a query against a predefined database, with formatted results.

Standard Scan

Using the command to run the standard query suite provides a categorized list of possible vulnerabilities. It also shows the code locations where these warnings were generated. Figure 4.4 and Figure 4.5 shows the results of running the standard scan against the targets.

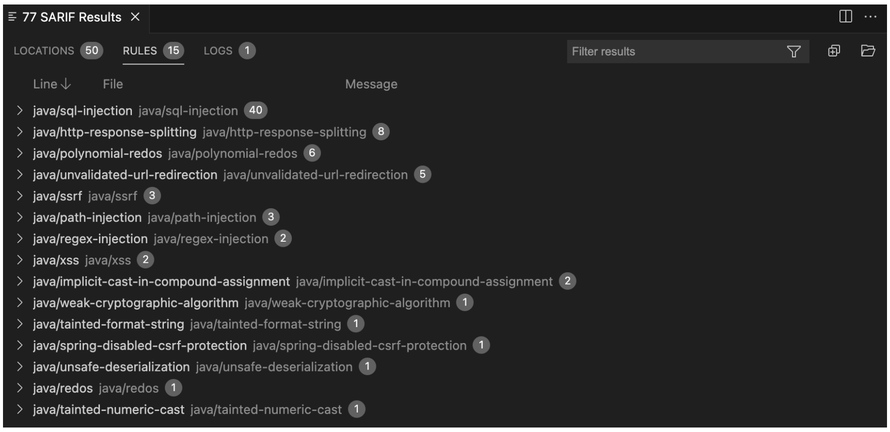


Figure 4.4: Standard scan results for DHIS2.

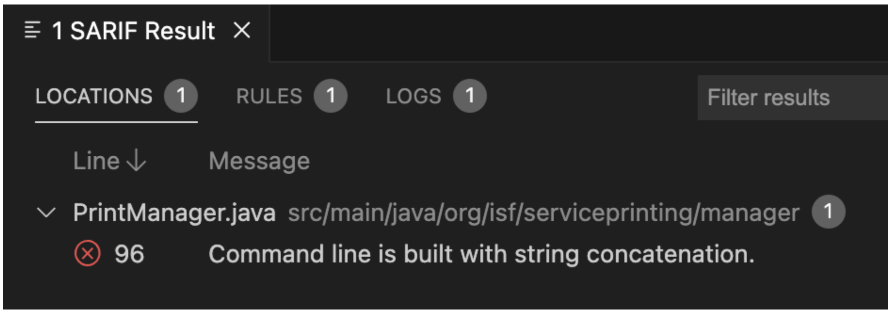


Figure 4.5: Standard scan result for Open Hospital.

The scan results for DHIS2, as shown in Figure 4.4, highlights various security warnings across multiple components of the application. Each listed vulnerability includes a reference to the specific part of the codebase, allowing developers to quickly locate and assess the potential risks.

These automated scans are particularly valuable as they enable continuous monitoring of the codebase, ensuring that any new code changes are evaluated against the same rigorous security standards. The standard query suite is actively maintained by Github, which ensures that the security coverage of the standard scan is constantly evolving along with the security landscape.

As a result of scanning the targets, numerous types of vulnerabilities the team lacked knowledge on were displayed. Therefore, a large bulk of the results gained was knowledge about the encountered vulnerabilities, including examples of good and bad implementations.

4.1.1 Vulnerabilities Investigated

SQL Injection

When running the standard scan on DHIS2, warnings were issued in 40 locations as seen in Figure 4.4. In the suite of queries, the predefined query in Figure 4.6 generated the warnings.

```
/**
 * @name Query built from user-controlled sources
 * @description Building a SQL or Java Persistence query from user-controlled sources
 *              is vulnerable to insertion of malicious code by the user.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 8.8
 * @precision high
 * @id java/sql-injection
 * @tags security
 *       external/cwe/cwe-089
 *       external/cwe/cwe-564
 */

import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.security.SqlInjectionQuery
import QueryInjectionFlow::PathGraph

from
  QueryInjectionSink query, QueryInjectionFlow::PathNode source, QueryInjectionFlow::PathNode sink
where queryIsTaintedBy(query, source, sink)
select query, source, sink, "This query depends on a $@.", source.getNode(), "user-provided value"
```

Figure 4.6: Query detecting SQL sinks tainted by user input, from the standard package of CodeQL queries (‘CWE coverage for Java’, n.d.).

Running this single query against DHIS2 produces 278 individual results. Similarly the first iteration of the team’s custom query, shown in Figure 4.7 produces 286 results.

```
module PossibleSqlInjectionConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  predicate isSink(DataFlow::Node sink) {
    sink instanceof QueryInjectionSink
  }
}

module PossibleSqlInjectionTaint = TaintTracking::Global<PossibleSqlInjectionConfig>;

from DataFlow::Node source, DataFlow::Node sink
where PossibleSqlInjectionTaint::flow(source, sink)
select sink, source
```

Figure 4.7: Customized query detecting SQL sinks tainted by user input.

The custom query tracks tainted data flowing from user controlled input to arguments in a SQL-statement execution. The end nodes of this flow is captured by the predefined classes ”RemoteFlowSource” and ”QueryInjectionSink”. To remove these false positives the query was extended as shown in Figure 4.8.

```
class QueryParams extends RefType {
  QueryParams() {
    /**
     *Param logic to go here
     */
  }
}

class QueryParamsInstanceExpr extends Expr {
  QueryParamsInstanceExpr() {
    this.getType() instanceof QueryParams
  }
}

module PossibleSqlInjectionConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  predicate isSink(DataFlow::Node sink) {
    sink instanceof QueryInjectionSink
  }
}

module PossibleSqlInjectionTaint = TaintTracking::Global<PossibleSqlInjectionConfig>;

from DataFlow::Node source, DataFlow::Node sink
where PossibleSqlInjectionTaint::flow(source, sink)
select sink, source
```

Figure 4.8: Customized query detecting SQL sinks tainted by user input.

By using the predicate ”isBarrier()” in the data flow module, the results are restricted to dataflow paths not flowing through the node in the predicate. The class ”QueryParams” will hold the logic for what is deemed a safe implementation and should thereby be excluded from the results. Doing this essentially boils down to the logic: *Find all instances of an SQL query being executed, where an argument can be controlled by a remote user, and where the tainted data does not flow through a node that is an instance of ”QueryParams”*. An example of a false positive that was removed is instances of SQL arguments being parameterized. From reading source code in DHIS2 with the help of the results from the query in Figure 4.7, a class called ”DataQueryParams” was found, in which the arguments of a SQL query is parameterized, thereby making it safe from SQL injections. To exclude this false-positive from the results, the following logic was added.

```
class QueryParams extends RefType {
  QueryParams() {
    this.hasQualifiedName("org.hisp.dhis.analytics", "DataQueryParams")
    or exists(RefType r | r.hasQualifiedName("org.hisp.dhis.analytics", "DataQueryParams")
    | this.hasSupertype(r))
    or this.hasQualifiedName("org.hisp.dhis.analytics.outlier.data", "OutlierQueryParams")
  }
}
```

Figure 4.9: Customized query detecting SQL sinks tainted by user input.

This class now captures the following logic: *This is a reference type that has the package ”org.hisp.dhis.analytics” and is of class ”DataQueryParams”. Or there exists a reference type that has the package ”org.hisp.dhis.analytics” and is of class ”DataQueryParams” and this is a subclass of this reference type*. This means that if an intermediary node in the data flow corresponds to the defined class, i.e the tainted data is parameterized, it is not included in the results.

Log Injection

The standard log injection query seen in Figure 4.10, ran on DHIS2s, gave nearly 300 results.

```
/**
 * name Log Injection
 * description Building log entries from user-controlled data may allow
 *              insertion of forged log entries by malicious users.
 * kind path-problem
 * problem.severity error
 * security-severity 7.8
 * precision medium
 * id java/log-injection
 * tags security
 *       external/cwe/cwe-117
 */

import java
import semmle.code.java.security.LogInjectionQuery
import LogInjectionFlow::PathGraph

from LogInjectionFlow::PathNode source, LogInjectionFlow::PathNode sink
where LogInjectionFlow::flowPath(source, sink)
select sink.getNode(), source, sink, "This log entry depends on a $@.", source.getNode(),
  "user-provided value"
```

Figure 4.10: Log Injection standard query (‘CWE coverage for Java’, n.d.).

The standard query was used as a starting point for expanding the query. To try and remove possible false positives, as seen in Figure 4.11, all results surrounded in try-catch blocks were removed. Furthermore, all results using expressions such as ”sql”, ”msg”, ”format()”, ”logstring” and ”tostring” were removed. Reducing the number of results down to 70.

```
import java
import semmle.code.java.security.LogInjectionQuery
import LogInjectionFlow::PathGraph
import semmle.code.java.dataflow.DataFlow

from LogInjectionFlow::PathNode source, LogInjectionFlow::PathNode sink
where
    LogInjectionFlow::flowPath(source, sink) and not
    (
        exists(Expr sinkExpr |
            sinkExpr = sink.getNode().asExpr() and
            (
                // Check if the sink expression is directly inside the try block
                sinkExpr.getEnclosingStmt().getEnclosingStmt*() instanceof TryStmt or
                // Check if the sink expression is inside any of the catch clauses
                exists(CatchClause cc |
                    cc.getParent() = sinkExpr.getEnclosingStmt().getEnclosingStmt*()
                )
            )
        )
    ) and not (
        exists(Expr sinkExpr |
            sinkExpr = sink.getNode().asExpr() and
            (
            sinkExpr.toString().toLowerCase().matches("%sql") or
            sinkExpr.toString().toLowerCase().matches("%msg") or
            sinkExpr.toString().toLowerCase().matches("%format(...)") or
            sinkExpr.toString().toLowerCase().matches("%logstring") or
            sinkExpr.toString().toLowerCase().matches("%tostring(...)") or
            sinkExpr.toString().toLowerCase().matches("%...+...")
            )
    ))
select sink.getNode(), source, sink, "This log entry depends on a $@.", source.getNode(),
       "user-provided value", sink.getNode().asExpr(), "expression"
```

Figure 4.11: Refined log injection query narrowing down possible false positives.

HTTP Response Splitting

As a result from the scan, a potential HTTP Response Splitting vulnerability was found as seen in Figure 4.12.

```
if (StringUtils.isEmpty(eventCriteria.getAttachment())) {
    response.addHeader(
        "Content-Disposition", "attachment; filename=" +
        eventCriteria.getAttachment());
}
```

Figure 4.12: Potential HTTP Response Splitting vulnerability in the DHIS2 codebase.

The potential vulnerability here is that ”criteria.getAttachment()” is directly concatenated into the filename parameter of the ”Content-Disposition” header. If an attacker can control the value of ”criteria.getAttachment()”, they could potentially inject malicious values that include characters like \n or \r, causing the HTTP response to be split and additional headers or content to be added.

After research on this issue, this proves to be an example of a false positive, and in turn a demonstration of a CodeQL limitation. This is due to its inability to detect security measures handled in external packages. In this instance, DHIS2 uses Tomcat, which already has a built in protection from these kinds of attacks (‘False positive: HTTP Response Splitting’, 2024).

Unsafe Deserialization

The query seen in Figure 4.13 was created from scratch to try and find user controlled serialization. The logic captures user controlled input that flows to a serialization call. The query did not produce any results.

```
predicate isOOSMethod(Method serialization) {
  serialization.hasName("writeBoolean")
  or serialization.hasName("writeByte")
  or serialization.hasName("writeChar")
  or serialization.hasName("writeDouble")
  or serialization.hasName("writeFloat")
  or serialization.hasName("writeInt")
  or serialization.hasName("writeLong")
  or serialization.hasName("writeShort")
  or serialization.hasName("writeUTF")
  and serialization.getDeclaringType().hasQualifiedName("java.io", "ObjectOutputStream")
}

predicate isCalledFromTest(MethodCall call) {
  call.getLocation().getFile().getAbsolutePath().regexpMatch(".*[\\\\/]src[\\\\/]test[\\\\/]java.*")
}

class SerializationWriteCall extends MethodCall {
  SerializationWriteCall() {
    not isCalledFromTest(this)
    and exists(Method serialization
    this.getMethod() = serialization
    isOOSMethod(serialization))
  }
}

module UserInputToWriteConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof UserInput
  }

  predicate isSink(DataFlow::Node sink) {
    sink.asExpr() instanceof SerializationWriteCall
  }
}


module UserInputToWrite = DataFlow::Global<UserInputToWriteConfig>;

from DataFlow::Node src, DataFlow::Node sink
where UserInputToWrite::flow(src,sink)
select src, sink
```

Figure 4.13: Query created for finding user controlled serialization calls.

ReDoS

In DHIS2 the standard scan highlighted a potential ReDoS vulnerability as seen in Figure 4.14.

```java
/**
* regex to detect delimiter, ignores spaces, allows delimiter in comment,
* allows an equals-sign
*/
public static final Pattern delimP =
    Pattern.compile(
            "^\\s*+(--)?\\s*+delimiter\\s*+=?\\s*+([^\\s]+)+\\s*+.*+$",
            Pattern.CASE_INSENSITIVE);
```

Figure 4.14: Potential ReDoS vulnerability in the DHIS2 codebase.

However, the inputs that are used in this regex do not seem to cause any problems, as they are not user-provided values, as well as it being used in a limited context.

It is worth noting that Java versions 9 and above have some mitigation against ReDoS, but they are not perfect and complex regex can still be affected ('Inefficient Regex Documentation', 2024).

Regex Injection

The standard scan highlights the code in Figure 4.1.1 from DHIS2 as a possible regex injection vulnerability.

```java
for (String name : names) {
      if (name.toUpperCase().equals(name) && name.indexOf('_') < 0) {
        // assume it is a code
        checksByName.values().stream()
            .filter(check -> check.getCode().equals(name))
            .map(DataIntegrityCheck::getName)
            .forEach(expanded::add);
      } else if (name.contains("*")) {
        String pattern =
            name.toLowerCase()
                .replace('-', '_') // make uniform
                .replaceAll("[^*_a-z0-9]+", "") // sanitise against regex attacks
                .replace("*", ".*"); // expand regex wildcard match
        for (DataIntegrityCheck check : checksByName.values()) {
          if (check.getName().matches(pattern)) {
            expanded.add(check.getName());
          }
        }
      } else {
        expanded.add(name.toLowerCase().replace('-', '_'));
      }
    }
```

Figure 4.15: Potential regex
injection vulnerability in the DHIS2 codebase.

This function attempts to sanitize the input by removing characters that do not match the pattern `[^*_a-z0-9]+`, which aims to prevent the inclusion of regex special characters that might lead to complex or unsafe regex patterns.

Although this approach attempts to limit the potential for regex injection by sanitizing input, it relies on a blacklist approach rather than a whitelist approach. The existing sanitization is effective in most cases but may not completely mitigate the risk if not all dangerous characters or combinations are accounted for.

Disabled CSRF Protection

The scan reveals that CSRF protection is disabled in the config file as seen in Figure 4.16. However, in the context of DHIS2, this configuration can be justified because the platform integrates strong authentication measures via OAuth2. This security protocol ensures that only authorized clients can access resources and perform sensitive operations.

```
@Bean
  protected SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf().disable();

    http.requestCache().requestCache(requestCache);

    configureMatchers(http);
    configureFormLogin(http);
    configureCspFilter(http, dhisConfig, configurationService);
    configureCorsFilter(http);
    configureMobileAuthFilter(http);
    configureApiTokenAuthorizationFilter(http);
    configureOAuthTokenFilters(http);

    setHttpHeaders(http);

    return http.build();
  }
```

Figure 4.16: Potential CSRF vulnerability in the DHIS2 codebase

4.1.2 Identifying Challenges

In alignment with the method framework outlined in Chapter 3.4, this section presents the specific challenges encountered during the practical implementation of CodeQL.

Research to CTF transition

Transitioning from theoretical learning to practical CTF challenges highlighted multiple challenges. These challenges revealed a steep learning curve and considerable knowledge gaps in both vulnerability detection and CodeQL application. According to the GitHub-provided CTF challenge descriptions, these activities are intended to enhance vulnerability hunting skills and facilitate rapid acquisition of CodeQL knowledge ('Capture The Flag', 2024).

The team found the practical application via CTF challenges demanding due to the advanced use of CodeQL and a pronounced gap in practical security analysis skills. Figures 4.17 and 4.18 illustrate the disparity in query complexity between a tutorial puzzle query and a CTF challenge query:

```
import tutorial

Person relativeOf(Person p) { parentOf*(result) = parentOf*(p) }

predicate hasCriminalRecord(Person p) {
  p = "Hester" or
  p = "Hugh" or
  p = "Charlie"
}

from Person p
where
  not p.isDeceased() and
  p = relativeOf("King Basil") and
  not hasCriminalRecord(p)
select p
```

Figure 4.17: Query used to solve the "Crown the rightful heir" puzzle from CodeQL tutorials.

```
import go
import semmle.go.dataflow.DataFlow


module IsReqAuthToEqTestConfig implements DataFlow::ConfigSig {
        predicate isSource(DataFlow::Node source) {
        exists(DataFlow::CallNode cn |
        source = cn.getResult() and
        cn.getTarget().hasQualifiedName(_, "isReqAuthenticated")
    )
}

        predicate isSink(DataFlow::Node sink) {
        exists(DataFlow::EqualityTestNode etn |
        sink = etn.getAnOperand())}
}

module IsRecAuthToEqTest = DataFlow::Global<IsReqAuthToEqTestConfig>;

predicate returnStmtInThenBlock(BlockStmt block) {
        exists(Stmt stmt |
          block.getAChildStmt() = stmt |
        stmt instanceof ReturnStmt)
}

from DataFlow::Node source, DataFlow::Node sink, IfStmt ifs, EqualityTestExpr expr, Ident errn
where IsRecAuthToEqTest::flow(source, sink)
  and expr.getAnOperand() = sink.asExpr()
  and not returnStmtInThenBlock(ifs.getThen())
        and ifs.getCond() = expr
        and expr.getAnOperand() = errn
        and errn.getName() = "ErrNone"
select source, sink, ifs
```

Figure 4.18: Query used to solve a section of the ”Go and don’t return” CTF challenge.

The complexity of the queries used in the CTF challenges, employing library classes such as ”IfStmt” and ”ReturnStmt”, reflects a significant leap from only working on strings in the tutorial puzzles. This transition tested the team’s understanding of CodeQL and highlighted deficiencies in the ability to detect vulnerabilities.

Application to Targets

The initial approach of applying CodeQL to the targets was to develop custom queries, with guidance primarily derived from the OWASP Top Ten list. This approach quickly revealed the limitations of the team’s capabilities in formulating effective queries from scratch. The custom queries often resulted in ambiguous outcomes, where it was unclear whether their lack of results was due to the absence of vulnerabilities or the inadequacy of the queries themselves.

As the team transitioned to using GitHub's pre-written standard queries, the attempts to modify these queries to uncover novel vulnerabilities were unsuccessful. This led to the modifications of the standard queries being restricted to reducing the noise in the results, focusing on eliminating false positives rather than extending the detection capabilities to new vulnerabilities within the targets.

In every instance where the results of the custom queries were compared to those of the standard scans, they did not uncover any additional results. This can be seen in the results from the queries shown in Figure 4.6 and Figure 4.7. The efforts were confined to adjusting the sensitivity and specificity of existing queries, which, while valuable for understanding the mechanics of CodeQL, did not extend the scope of vulnerability detection as originally hoped. This limitation highlighted a significant gap in the team's practical skills and emphasized the competence needed to develop effective security queries beyond the standard queries.

Lack of Cyber Competence

While the team had a foundational understanding of software security principles, applying these principles to effectively identify vulnerabilities in practical scenarios proved challenging. The initial encounters with real-world applications highlighted significant knowledge gaps, particularly in the areas of static analysis and the nuances of common software vulnerabilities. Substantial difficulties in dealing with specific types of vulnerabilities were encountered, for example when examining ReDoS attacks. Due to limited familiarity with this vulnerability, significant time was invested to discern whether this was a legitimate threat or a false positive.

The team faced considerable challenges in effectively utilizing CodeQL to identify and manage data sanitization points between sources of tainted data and potential sinks. For example, early attempts at tracking taint propagation often failed to correctly identify or account for data sanitization practices within the data flow, leading to numerous false positives or possibly overlooked vulnerabilities. A particular difficulty was understanding how to incorporate intermediary nodes within queries. In the initial attempts, such as the one shown in Figure 4.7, the aim was to track taint flow through specific classes without considering intermediary sanitization points.

The project also exposed a significant gap in understanding of the security mechanisms employed within the analyzed applications. The lack of deep knowledge about specific security implementations, such as authentication checks, encryption methods, and error handling procedures, frequently impeded the team's ability to accurately assess security vulnerabilities. This often resulted in unnecessary investigations into what appeared to be security flaws, but were in fact safe practices due to other security measures.

An example of this can be seen in Section 4.1.1, where it was noted that CSRF protection was disabled. The initial assessment considered this a potential vulnerability, prompting a detailed examination of the associated risks. However, DHIS2 employs OAuth2 for its authentication processes, which mitigates CSRF risks in its context. The lack of familiarity with how OAuth2's token-based authentication effectively circumvents CSRF attacks led to misdirected efforts.

Lack of Repository Understanding

The complexity and extensive scale of the DHIS2 codebase posed substantial challenges. Team members, despite having experience with Java and RESTful applications, spent considerable time understanding the specific implementations within DHIS2. This lack of deep contextual repository understanding significantly slowed down the vulnerability assessment process, highlighting the importance of familiarity with the target repositories.

Adaptations

To navigate the challenges encountered during the project, significant adaptations were necessary, primarily revolving around the utilization of CodeQL. As detailed in Section **??**, writing queries from scratch proved ineffective.

Recognizing these limitations, the team shifted focus towards leveraging GitHub's pre-written standard queries. This pivot aimed not at enhancing the specificity but at reducing the noise of the results provided by these standard queries. An example of this adaptation is found in Figure 4.9. The modified query did not uncover additional vulnerabilities beyond those identified by the standard scans. Instead, they were fine-tuned to decrease the incidence of false positives.

Additionally, the reliance on established resources like CWE from MITRE and OWASP was intensified. These resources became instrumental in interpreting the vulnerability warnings generated by the standard queries, thereby enhancing the team's learning and understanding of each identified issue. This approach allowed for a better comprehension of the vulnerabilities, assessment of their impact, and understanding of the security mechanisms at play, which in turn informed the query refinement process.

4.1.3 Learning with CodeQL

The outcomes of the standard scans were systematically categorized by vulnerability classes. To determine the validity of these results, a comprehensive understanding of each vulnerability was essential. Utilizing OWASP and MITRE as resources facilitated the learning of various security issues. The detailed examples provided by OWASP on safe and unsafe implementations improved comprehension, even for vulnerabilities that were previously unfamiliar to the team.

The process of refining queries in CodeQL necessitated a deep engagement with the behavior patterns within the code. The learning outcomes came from identifying harmful behavior to be flagged, and recognizing safe implementations to be excluded. Each iteration of reviewing and adjusting queries sharpened the understanding of how vulnerabilities manifest in real-world applications, and the security measures necessary to mitigate them.

4.1.4 Learning to Use Custom Queries

Custom queries in CodeQL allow for the development of tailored security assessments specific to the unique aspects of a target codebase. Learning to create and refine custom queries was one of the outcomes from the application of CodeQL.

Initially, executing a CodeQL query generates a wide range of results, often including false positives. This initial output provides a basis for refining the queries to improve the accuracy.

The refinement process involves an evaluation of the query results, identifying those leading to consistent false positives. For instance, a query might initially flag many data paths as security risks, which, upon analysis, are found to have effective custom sanitization routines. The query is adjusted manually to exclude such secure implementations. An example of this is seen in Section 4.1.1. This iterative process of refining and re-evaluating helps reduce irrelevant findings and enhances the focus on true vulnerabilities. Custom queries can also be compiled into a custom pack that can be ran together with the standard scan.

Using GPT-4

The usage of GPT-4 into this project, as outlined in Chapter 3.2.2, significantly facilitated both information gathering and deeper learning about CodeQL's application. For example, GPT-4 was used to condense the DHIS2 developers manual as seen in Figure 4.19, to make it easier to find information relevant to a specific query.

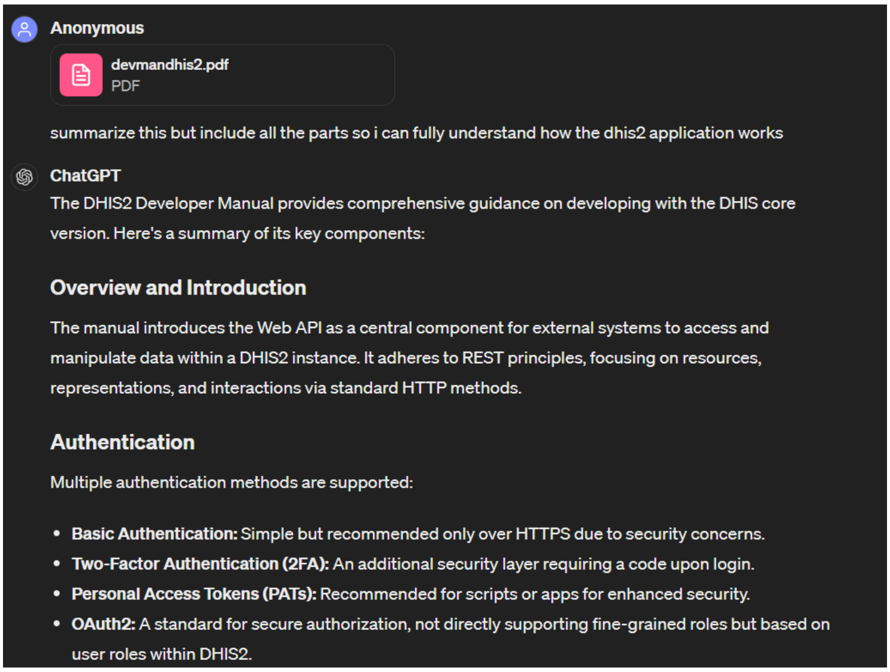
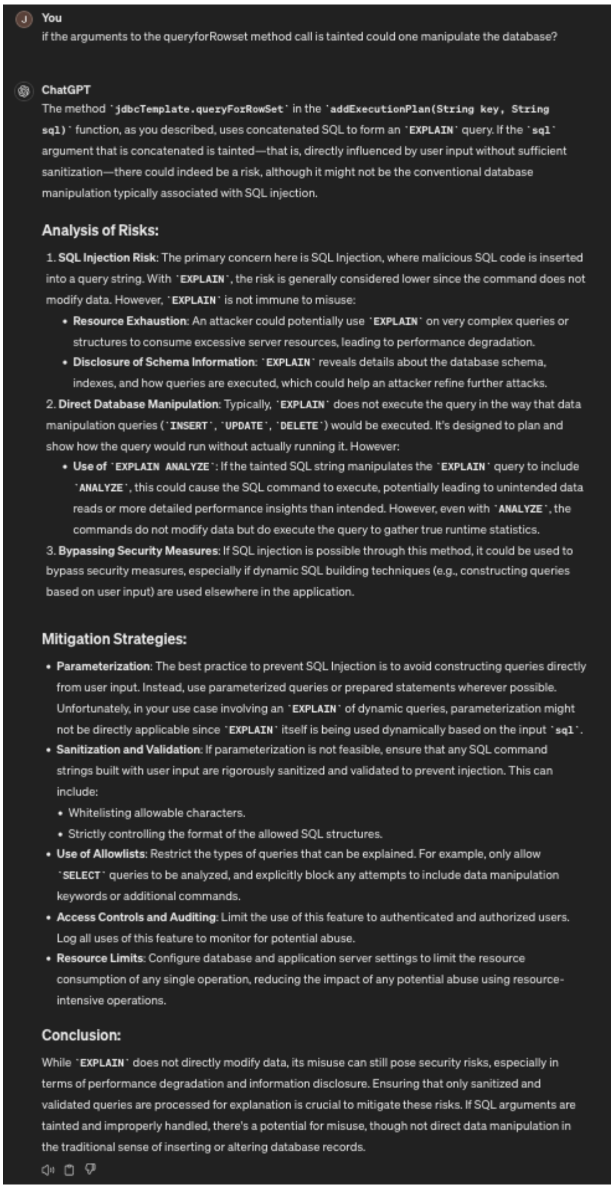


Figure 4.19: GPT-4 summarization example, reducing the developer manual to essential elements.

Figure 4.20 captures a discussion where GPT-4 was queried to explain a code-snippet and discuss the implications of certain security mechanisms. This interaction not only clarified advanced topics but also aided in visualizing how these concepts apply within CodeQL tasks.



You
if the arguments to the queryforRowset method call is tainted could one manipulate the database?

ChatGPT
The method `jdbcTemplate.queryForRowSet` in the `addExecutionPlan(String key, String sql)` function, as you described, uses concatenated SQL to form an `EXPLAIN` query. If the `sql` argument that is concatenated is tainted—that is, directly influenced by user input without sufficient sanitization—there could indeed be a risk, although it might not be the conventional database manipulation typically associated with SQL injection.

Analysis of Risks:

1. **SQL Injection Risk**: The primary concern here is SQL Injection, where malicious SQL code is inserted into a query string. With `EXPLAIN`, the risk is generally considered lower since the command does not modify data. However, `EXPLAIN` is not immune to misuse:
   - **Resource Exhaustion**: An attacker could potentially use `EXPLAIN` on very complex queries or structures to consume excessive server resources, leading to performance degradation.
   - **Disclosure of Schema Information**: `EXPLAIN` reveals details about the database schema, indexes, and how queries are executed, which could help an attacker refine further attacks.
2. **Direct Database Manipulation**: Typically, `EXPLAIN` does not execute the query in the way that data manipulation queries (`INSERT`, `UPDATE`, `DELETE`) would be executed. It's designed to plan and show how the query would run without actually running it. However:
   - **Use of `EXPLAIN ANALYZE`**: If the tainted SQL string manipulates the `EXPLAIN` query to include `ANALYZE`, this could cause the SQL command to execute, potentially leading to unintended data reads or more detailed performance insights than intended. However, even with `ANALYZE`, the commands do not modify data but do execute the query to gather true runtime statistics.
3. **Bypassing Security Measures**: If SQL injection is possible through this method, it could be used to bypass security measures, especially if dynamic SQL building techniques (e.g., constructing queries based on user input) are used elsewhere in the application.

Mitigation Strategies:

- **Parameterization**: The best practice to prevent SQL Injection is to avoid constructing queries directly from user input. Instead, use parameterized queries or prepared statements wherever possible. Unfortunately, in your use case involving an `EXPLAIN` of dynamic queries, parameterization might not be directly applicable since `EXPLAIN` itself is being used dynamically based on the input `sql`.
- **Sanitization and Validation**: If parameterization is not feasible, ensure that any SQL command strings built with user input are rigorously sanitized and validated to prevent injection. This can include:
  - Whitelisting allowable characters.
  - Strictly controlling the format of the allowed SQL structures.
- **Use of Allowlists**: Restrict the types of queries that can be explained. For example, only allow `SELECT` queries to be analyzed, and explicitly block any attempts to include data manipulation keywords or additional commands.
- **Access Controls and Auditing**: Limit the use of this feature to authenticated and authorized users. Log all uses of this feature to monitor for potential abuse.
- **Resource Limits**: Configure database and application server settings to limit the resource consumption of any single operation, reducing the impact of any potential abuse using resource-intensive operations.

Conclusion:

While `EXPLAIN` does not directly modify data, its misuse can still pose security risks, especially in terms of performance degradation and information disclosure. Ensuring that only sanitized and validated queries are processed for explanation is crucial to mitigate these risks. If SQL arguments are tainted and improperly handled, there's a potential for misuse, though not direct data manipulation in the traditional sense of inserting or altering database records.

Figure 4.20: GPT-4's role in explaining complex logic and its implications for cybersecurity.

Moreover, GPT-4 was helpful in debugging and refining CodeQL commands as seen in Figure 4.21. Here the method ”getCatchClause()” can not be resolved for the class ”TryStmt”, the real method is called ”getACatchClause()”.

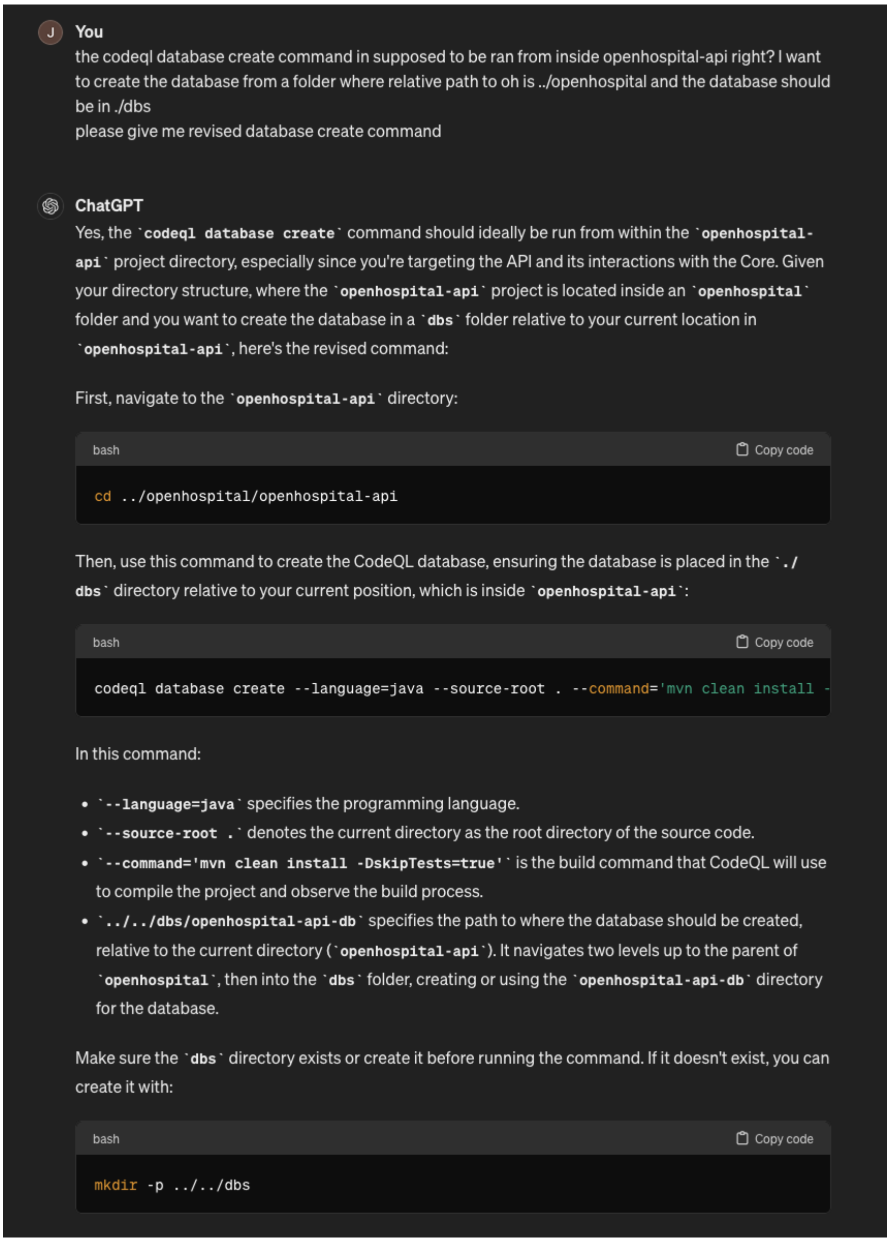
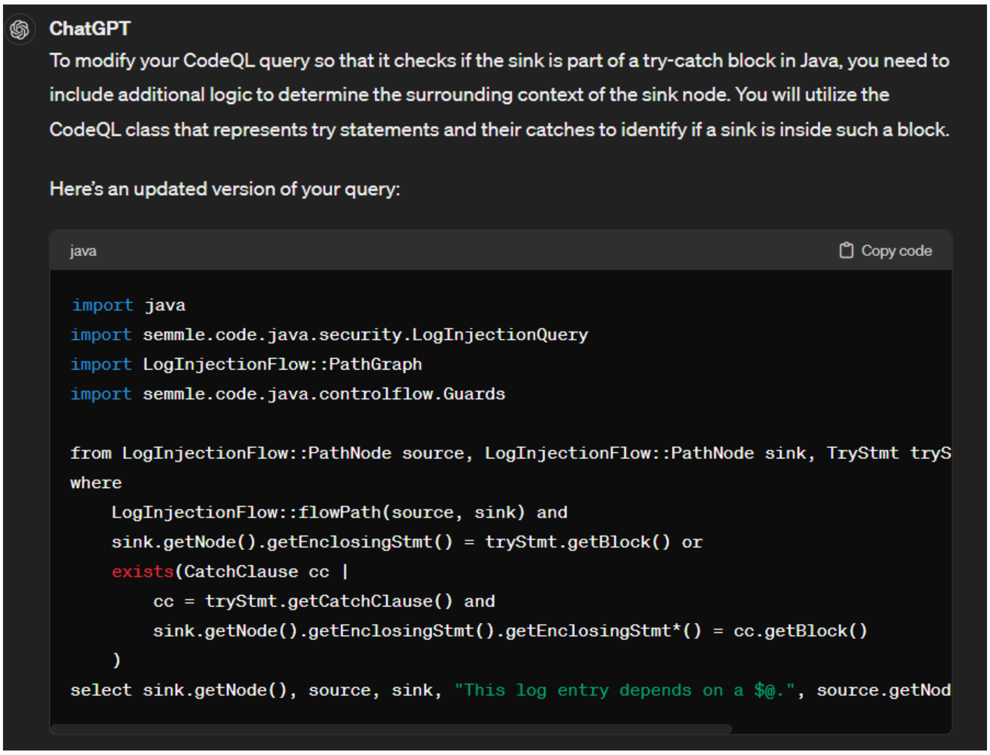


Figure 4.21: GPT-4 assisting in debugging and refining CodeQL command syntax.

GPT-4 occasionally provided misleading or incorrect information. An illustrative example is depicted in Figure 4.22, where GPT-4 suggests a method not applicable to the context of the query.



ChatGPT

To modify your CodeQL query so that it checks if the sink is part of a try-catch block in Java, you need to include additional logic to determine the surrounding context of the sink node. You will utilize the CodeQL class that represents try statements and their catches to identify if a sink is inside such a block.

Here's an updated version of your query:

java Copy code

```
import java
import semmle.code.java.security.LogInjectionQuery
import LogInjectionFlow::PathGraph
import semmle.code.java.controlflow.Guards

from LogInjectionFlow::PathNode source, LogInjectionFlow::PathNode sink, TryStmt tryS
where
    LogInjectionFlow::flowPath(source, sink) and
    sink.getNode().getEnclosingStmt() = tryStmt.getBlock() or
    exists(CatchClause cc |
        cc = tryStmt.getCatchClause() and
        sink.getNode().getEnclosingStmt().getEnclosingStmt*() = cc.getBlock()
    )
select sink.getNode(), source, sink, "This log entry depends on a $@.", source.getNod
```

Figure 4.22: Example of an incorrect suggestion provided by GPT-4.

4.1.5 Survey results

A survey to gather additional data about the current state of cybersecurity knowledge within different Information Technology (IT) educations was sent out, as outlined in Chapter 3.4.7, and recieved 62 answers. Figures 4.23 and 4.24 show how the answers to this survey is distributed across year of study and line of study, with the distribution among years of study being very even, and line of study more leaned towards computer science and digital business development.

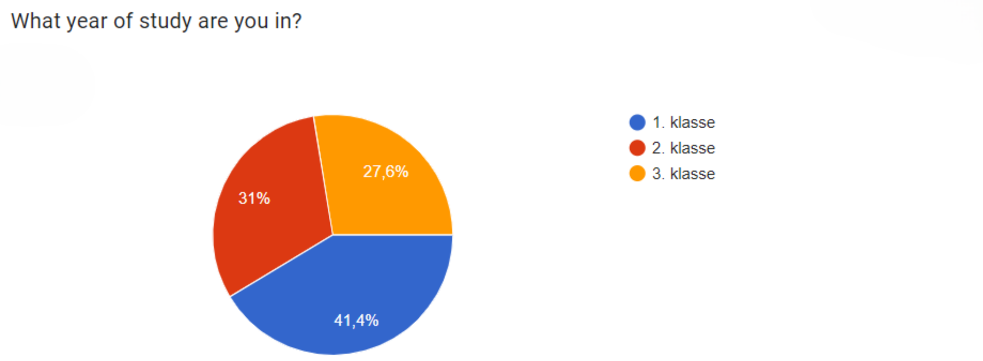


Figure 4.23: Answer distribution across year of study. Question asks "What year of study are you in?" and answers reads 1st grade, 2nd grade and 3rd grade.

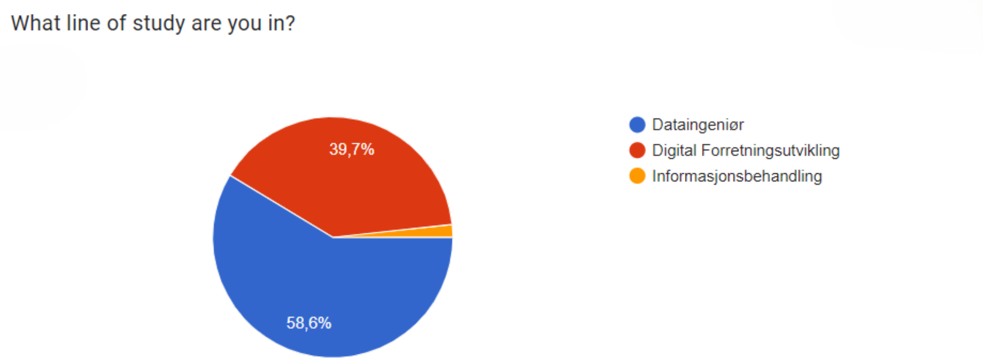


Figure 4.24: Answer distribution across line of study. Question asks "What line of study are you in?" and answers reads Computer Science, Digital Business Development and Information Management.

Figures 4.25 and 4.26 indicate the lack of cybersecurity knowledge the team encountered. This is shown by under 50% of students rating their knowledge on cybersecurity above a 5. The lack of knowledge is made even more apparent when considering the trend in the answer distribution of the question in Figure 4.26. Over 50% answered 5 or under, and 23% do not keep security practices in mind at all during development, answering with a 1.

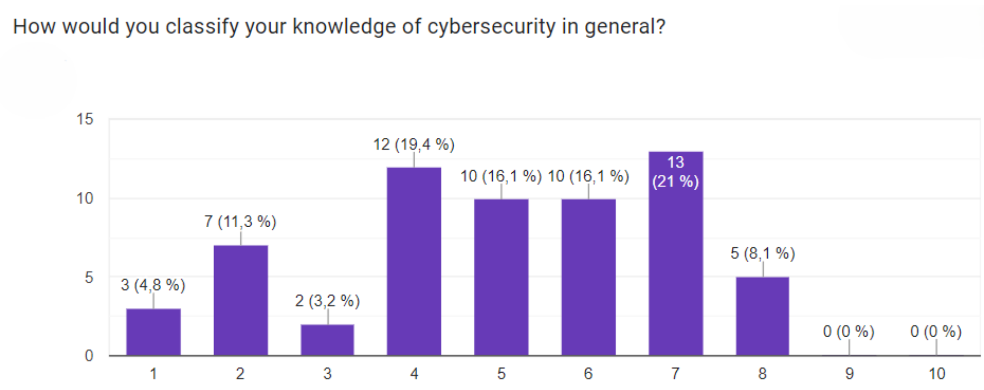


Figure 4.25: Answer distribution to question about rating general cybersecurity knowledge.

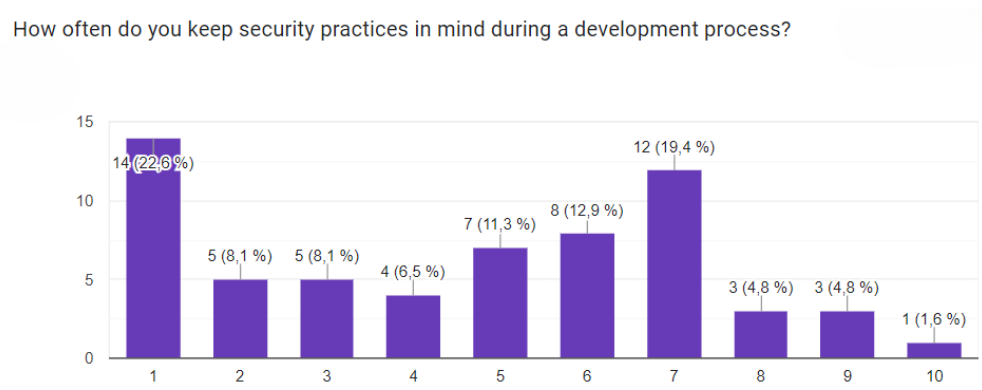


Figure 4.26: Answer distribution to question about how often security practices are kept in mind during a development process.

Figures 4.27 and 4.28 figures illustrate the general lack of knowledge on SAST tools.

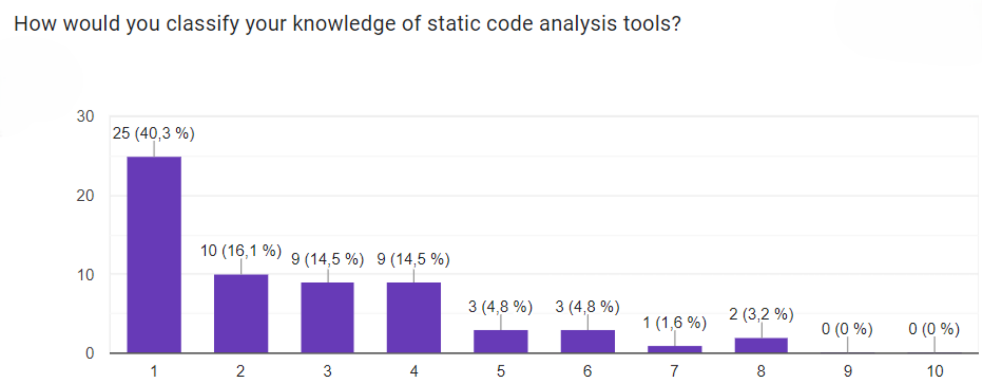


Figure 4.27: Answer distribution to question about rating SAST-tool knowledge.

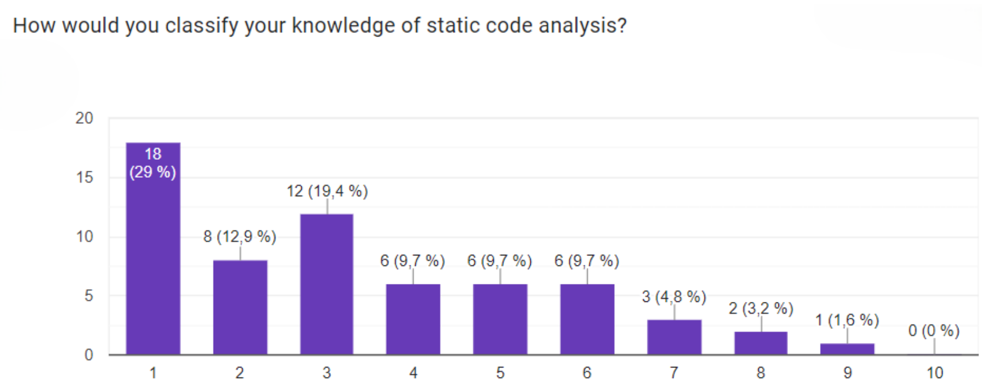


Figure 4.28: Answer distribution to question about rating static code analysis knowledge.

4.2 Administrative results

The following section outlines the results of the administrative effort. It provides a clear overview of the administrative work that supported this project's plan.

4.2.1 Time Accounting

Due to the exploratory nature of this research, the project was divided into research, implementation, and documentation phases. This division facilitated time management, with Trello used for tracking tasks and milestones. Additionally, a Gantt chart provided an overview of major milestones. There were no discrepancies in the plan.

Research

Through this phase, the team managed to decide on a topic of research, as well as picking out repositories which could be used in the implementation phase. Approximately 450-550 hours was spent in total during this phase, attributed both to research, as well as other administrative tasks.

Implementation

The implementation phase resulted in the team being able to explore the codebases of the target repositories, and relate the takeaways from this towards the documentation phase. Approximately 550-600 hours went to this phase.

Documentation

From the moment the team started solely focusing on the documentation phase, significant progress was made. Setting aside time to solely focus on this phase ended up being an integral part of this project. Approximately 480 hours were allotted to this phase.

Meetings

The consistent and frequent internal team meetings resulted in the process becoming more fluent. Being able to know where the other team members are at with their work, as well as what they are working on, makes it easier to plan for what to do. Furthermore, it made it easier to comply with internal deadlines.

There were a total of six meetings with the supervisor. Meetings consisted of a presentation about the current project status, as well as a discussion around the research topic.

5. Discussion

5.1 Scientific

This section examines our scientific results within the context of the thesis. We assess the impact of applying CodeQL on the targets and the educational implications. The discussion unifies our findings to suggest improvements in CodeQL documentation, as well as in software engineering education.

5.1.1 Methodology

Here we discuss the strengths and weaknesses of our systematic approach of employing CodeQL, evaluating how our methodological choices impacted the depth and thoroughness of the vulnerability analysis and learning outcomes.

Systematic Application of CodeQL

As detailed in Chapter 3.2, we characterize our approach of employing CodeQL by its open-ended and exploratory nature. This type of approach helped us uncover a broad range of issues and knowledge gaps in domains like cybersecurity, static analysis tools, and CodeQL's functionalities.

It also facilitated broad learning outcomes, encompassing a general understanding of software security, the capabilities and use of SAST tools like CodeQL, and vulnerability detection. The broad scope of our exploration enhanced our general cybersecurity knowledge, but at the expense of developing deep, specialized expertise in any single area. This breadth over depth approach inevitably impacted the effectiveness of our vulnerability assessments.

Target Selection

Choosing complex targets like DHIS2, along with our limited repository knowledge, significantly hindered our query development. Having to focus extensively on understanding the codebase rather than applying CodeQL was a misstep.

The choice of targets impacted the efficacy of our vulnerability analysis. For instance, while the scan on DHIS2 did result in several warnings, the ones investigated were benign. Open Hospital yielded only one warning, highlighting a possible mismatch between the chosen target and the types of vulnerabilities CodeQL is most effective at detecting. This highlights CodeQL's limitations as a penetration testing tool for computer science students with limited cybersecurity expertise.

Efficacy of the Learning Modules

The documentation was crucial for our foundational learning, offering essential insights into CodeQL's concepts and syntax. This theoretical grounding was vital as we began exploring the tool's capabilities. The shift from theoretical knowledge to practical application was done through CTF challenges, which provided a hands-on opportunity to apply what we had learned. These exercises were important in developing our query-writing skills. However, they also exposed the limits of our learning, particularly when it came to the complexities of real-world vulnerabilities. The learning modules succeeded in teaching us CodeQL, but did not sufficiently prepare us to identify novel vulnerabilities outside of the standard scans, highlighting a significant gap in our ability to leverage the tool for advanced security analysis.

Use of GPT-4

GPT-4 was a valuable tool in our project, helping us quickly understand and apply complex security concepts, as well as summarizing lengthy documents. This was essential for connecting theoretical knowledge with practical security applications. However, we had to be cautious as GPT-4 occasionally provided incorrect information, specifically for writing CodeQL queries. As a result of this, answers provided by GPT-4 was always verified by other sources while researching.

5.1.2 CodeQL use in Development versus Pentesting

CodeQL offers use to both developers and penetration testers, but its application can yield different outcomes depending on the context. In our study, we primarily utilized CodeQL for penetration testing. This approach did not yield significant results. We attribute this to our lack of experience with penetration testing and a limited understanding of complex security mechanisms.

In the project, while we did not integrate CodeQL within a development environment, the process of setting up and running queries on pre-existing applications provided us with valuable insights. We learned that even without integration into development, running CodeQL's standard queries could highlight potential security issues. By only using two commands showcased in Chapter 4.1, ”database create”, and ”database analyze”, you can already get results from the standard scan.

A developer in a CI/CD environment could encounter these CodeQL warnings when they push changes to the repository, prompting them to review and reconsider their implementations. This aspect of CodeQL usage not only aids in catching security flaws but also fosters a continual learning process for developers, enhancing their ability to think critically about security as an integral part of their coding practice.

5.1.3 Vulnerability Findings

Our CodeQL analysis did not identify any definite vulnerabilities. This highlighted the complexities of vulnerability detection and the challenges encountered during its application.

Challenges Impacting Detection

The main factor contributing to the non-detection of vulnerabilities was the foundational gap in our understanding of complex security mechanisms and static analysis techniques. This deficiency was exacerbated by our limited experience with CodeQL's advanced functionalities. These challenges highlight the need for in-depth knowledge of both the tool and the security landscape it is employed in.

Educational Gaps

The absence of detected vulnerabilities in our project should not only be interpreted as an indication of security in the targets. More importantly, it reflects the limitations on the effectiveness of our CodeQL application. These limitations do not solely reflect the team's own lacking knowledge, but our education as a whole. With this thesis being the conclusion of our three year education in computer science we question the lack of cybersecurity knowledge integrated in the three year education span. Specifically we believe that the lack of software security, and secure development practices in our education exacerbated these results and should be addressed at a higher level than the individual student.

5.1.4 CodeQL as an Educational Tool

Using CodeQL revealed the educational potential of the tool. Using the standard scan generates a categorized file of potential security issues that prompted further investigation and learning to confirm whether these were genuine threats.

Bridging Theoretical and Practical Knowledge

Throughout the project, CodeQL served as a vital tool in applying theoretical security concepts practically.

For example, during our analysis of the DHIS2 codebase, we utilized CodeQL to trace how user input was processed and potentially misused. The tool highlighted several points in the application where user input was tainting arguments used in a SQL expression, posing a risk of SQL injection attacks. This practical application allowed us to learn and investigate data flow in an application, and the concept of taint tracking.

The process of iteratively refining our queries based on the initial results, enhanced our understanding of both the tool's capabilities and the vulnerabilities. Each iteration provided deeper insights into safe and possibly unsafe implementations in the code.

Insights Gained from Specific Vulnerabilities

The automatic scanning feature of CodeQL introduced us to a variety of vulnerabilities, enriching our theoretical understanding and prompting further research into each potential issue. For example, during our analysis of log injection vulnerabilities, CodeQL identified several suspicious logging practices. Although, none were exploitable in the code specific context, which led us to delve deeper into understanding what makes a logging practice secure versus insecure. We compared these findings against both secure and insecure code examples found in external resources, enhancing our practical understanding of the vulnerability.

The structured presentation of results by CodeQL, categorized by vulnerability types that correspond to common weaknesses in OWASP Top Ten and the CWE list from MITRE, significantly aided this learning process. This categorization not only made the results more accessible, but also provided clear starting points for further research into each vulnerability type.

Practical Application and Learning Facilitation

The low barrier to entry for running the standard CodeQL queries on any given repository underscores its utility as a learning tool. Even users with minimal experience can initiate scans and get immediate feedback on potential security issues, which can also be used as a springboard for learning. Furthermore, the necessity to refine queries to reduce noise inherently required a deep dive into understanding each flagged vulnerability. This not only forced the team to learn about the vulnerabilities but also to think critically about their manifestation in code.

Our experience with CodeQL highlighted its strength not only as a security tool but also as an educational tool. This dual capability of CodeQL emphasizes the importance of integrating such tools into regular development practices.

5.1.5 Survey

The survey conducted among IT students at Norwegian University of Science and Technology (NTNU) offered insights into the current state of cybersecurity knowledge and the integration of security practices in software development for our fellow students. The findings help to contextualize the challenges in embedding robust cybersecurity measures within the software development lifecycle.

The survey uncovered self-assessed cybersecurity knowledge among students, with many indicating only moderate understanding and a significant portion acknowledging minimal knowledge. This suggests potential inconsistencies in the integration of cybersecurity education across IT curricula, which may leave some graduates unprepared for security challenges in their careers.

Responses also show limited familiarity with static code analysis and SAST tools, which are important for secure software development. This gap highlights a critical area for educational enhancement to improve the security competence of future software developers.

The distribution of responses regarding security considerations during development leans towards infrequent consideration. This highlights a dangerous implication of security measures being employed after the fact, even though as the use of CodeQL shows, measures can in fact be integrated throughout developments and into deployment.

The variability in cybersecurity awareness and the infrequent application of security practices advocate for a structured inclusion of cybersecurity topics within IT programs. Strengthening the curriculum with practical tools like SAST could narrow the existing knowledge gap and cultivate a generation more versed in security.

When interpreting the survey results, it is worth noting that many responses came from first- and second-year students, potentially skewing the data negatively. However, this thesis emphasizes the need to integrate cybersecurity practices throughout the education span. Thus, we believe self-assessed cybersecurity awareness should be higher regardless of the year of study.

The survey's scope, limited to IT students at NTNU, and the modest number of respondents, may not fully represent the global state of cybersecurity knowledge among all software developers. Nevertheless, as students of NTNU ourselves, the survey's results align with our own observed deficiencies in security education, making the findings of the survey more credible within the context of our thesis.

5.1.6 Improvement of Education

The experience documented in this thesis has highlighted significant gaps in the cybersecurity knowledge of computer science students, particularly evident in the integration and effective utilization of tools like CodeQL. This underscores the urgent need for IT curricula to adapt and evolve by embedding practical security tools training within their programs. As our familiarity with CodeQL increased over time, it became evident that early integration of such tools in educational settings could significantly enhance learning outcomes and improve the security practices among developers. This thesis advocates for a curriculum that intertwines development concepts with corresponding security concerns, ensuring that cybersecurity becomes an integral part of the learning process. Such an approach not only equips students with the theoretical knowledge of security concepts but also the practical skills necessary to apply these concepts in real-world scenarios.

Incorporating rigorous cybersecurity training into the curriculum may complicate the learning process and increase its difficulty. However, this challenge is outweighed by the necessity for developers to have a deep understanding of security measures.

By integrating security into the core curriculum, educational institutions can equip future developers with the ability to create more secure and robust software. This emphasizes the educators' crucial role in preparing a workforce proficient in both development and cybersecurity.

5.1.7 Improvement of Documentation

The challenges faced by new users of CodeQL, as noted in this research, suggest that while the existing documentation is extensive, it could be improved to better cater to novices in cybersecurity. This thesis proposes that documentation should be extended to gradually guide new users from basic to complex cyber concepts, incorporating comprehensive examples that illustrate common vulnerabilities and how to detect them. Enhancements could include visual aids, step-by-step tutorials, and a more intuitive organizing that allows users to easily navigate through information. Such improvements could make a significant difference in demystifying the initial learning phase of using CodeQL, thereby making it more accessible to a broader audience.

5.2 Administrative

The decision to schedule meetings with our supervisor on an as-needed basis proved effective. During periods where intensive work was required, scheduling a meeting would have been counterproductive. Meetings were scheduled when there was significant work to present or specific questions to address.

Internal communication was maintained through daily stand-ups, which were highly effective in keeping everyone informed. These meetings made it easy for everyone to stay on the same page and seek help when needed.

Good communication ensured that any scheduling issues were minor inconveniences rather than major disruptions.

Utilizing Trello to track tasks across all phases provided a structured way to monitor progress and ensure accountability. This made working in parallel easier.

Setting more specific deadlines could have provided additional structure. However, this being an exploratory project made it challenging to establish fixed deadlines.

5.3 Summary

This discussion has explored our application of CodeQL during this project, highlighting both the successes and limitations encountered. The findings illuminate significant insights into the integration of theoretical knowledge and practical security tool utilization.

Our experience has underscored the necessity of a robust understanding of security tools and their effective application in real-world scenarios. Despite the absence of detected vulnerabilities, the project provided an understanding of the complexities involved in automated vulnerability detection and the importance of comprehensive security training.

The insights gained align with the overarching goals of this thesis, emphasizing the critical need for enhancing cybersecurity education within software development curricula. Our findings advocate for an educational approach that balances theoretical understanding with practical tool proficiency, highlighting the essential role of static analysis tools like CodeQL in contemporary cybersecurity practices.

The project's outcome contribute to a broader understanding of how early and continuous integration of security practices and tools into the software development lifecycle can significantly enhance the security posture of developed applications. This aligns with the thesis's objective to bridge the gap between software development and operational security, confirming the need for educational reforms that incorporate practical security tool training alongside traditional computer science education.

6. Conclusion and Future Work

This study investigated the potential of CodeQL to bridge the observed cybersecurity knowledge gap among software developers with minimal experience. Set against the backdrop of increasing digital threats, this research aimed to provide ways to enhance the cybersecurity competence of developers, thereby strengthening the integrity of software solutions.

Utilizing a mixed-method approach that emphasized individual experimentation throughout the research and implementation phases, the case study resulted in substantial individual learning in cybersecurity and proficiency in CodeQL. Furthermore, it revealed significant knowledge gaps among the participants, highlighting the critical need for integrating tools like CodeQL into the software development lifecycle to enhance software security and educational growth simultaneously.

The research successfully demonstrated that CodeQL can be used as an educational tool that help new developers understand and implement cybersecurity practices effectively. While the project did not uncover concrete security vulnerabilities and therefore not meeting the result goal of uncovering real vulnerabilities in the targets. The efforts, however, led to a comprehensive understanding of the challenges associated with implementing CodeQL in a real-world development setting. This insight has fulfilled all the other goals as outlined in Section 1.3

The steep learning curve encountered when starting with CodeQL suggests the need for more structured educational materials to support beginners in cybersecurity. Future research could explore a broader range of SAST tools and other cybersecurity tools, as well as their impact on improving security awareness for software developers. Additionally, there is a significant opportunity to integrate these tools into software development curricula, potentially through both traditional university courses and free online resources.

Integrating tools like CodeQL into educational settings could profoundly transform software development practices by embedding essential security measures within the development lifecycle. This approach not only prepares developers to handle emerging security threats but also enhances their overall programming knowledge.

This research journey has revealed the transformative potential of integrating SAST tools like CodeQL into the toolkit of software developers. By systematically applying CodeQL, we enhanced our capability to understand software vulnerabilities and comprehension of software security principles. This integration of security practices into software development training aligns with the thesis' goals, confirming that educational gaps in cybersecurity among developers can be effectively bridged through strategic tool integration.

Educators and industry leaders should consider the strategic incorporation of tools like CodeQL in curriculum designs and development practices to prepare the next generation of software developers for the security challenges of the future.

7. Societal Impact

In an era marked by increasing emphasis on sustainability, it is crucial to understand the societal implications of emerging research. This chapter delves into the potential impacts of this thesis on society, focusing on how enhanced cybersecurity practices can strengthen digital infrastructure. We will explore how integrating robust security measures from the outset of software development not only mitigates the need for subsequent corrections but also significantly enhances public trust in digital services. By doing so, this research contributes to a broader understanding of how proactive cybersecurity practises can safeguard essential digital assets and support sustainable development.

7.1 Environmental Impact

Integrating advanced security practices like CodeQL within the software development lifecycle significantly streamlines the process of securing code, reducing the need for frequent updates and patches. This proactive security implementation not only lessens server load but also curtails energy consumption, aligning with the sustainability goals of minimizing the carbon footprint of digital operations. Moreover, CodeQL's alignment with GitHub's green initiatives furthers our contribution to Sustainable Development Goals (SDG) 13: Climate Action, by promoting more sustainable and energy-efficient software development practices across the industry (Julien, 2023; Brescia, 2021).

7.2 Health Impact

Integrating security measures in the development of health-related software ensures the safety of critical devices like diabetic sensors and pacemakers. By improving cybersecurity knowledge among developers, we reduce risks of malfunctions that could endanger lives, aligning with SDG 3: Good Health and Well-being. The 2017 recall of pacemakers due to security flaws underscores the necessity for secure software, highlighting our thesis's role in enhancing public health by advocating for robust cybersecurity protocols (hern, 2017).

7.3 Economic Impact

Enhancing cybersecurity knowledge among developers can lead to significant economic savings by preventing costly data breaches, estimated at an average of $4.45 million USD per incident in 2023 (‘Cost of a data breach report 2023’, 2023). This proactive approach supports SDG 8: Decent Work and Economic Growth, by mitigating financial losses and enabling more sustainable economic allocations.

7.4 Societal Impact

By enhancing the cybersecurity proficiency of software developers, we bolster public trust in digital services. This improvement in cybersecurity measures helps protect sensitive information and fosters greater consumer confidence in digital platforms, addressing SDG 16: Peace, Justice, and Strong Institutions.

7.5 Ethical Considerations

Software developers bear significant ethical responsibilities. They serve as the custodians of user data, tasked with creating robust and secure software from the start, rather than patching the software only after vulnerabilities are exploited.

Society’s reliance on the internet means vast amounts of sensitive data, such as banking details, social security numbers, and personal passwords, are constantly being transferred and stored online. This reality imposes a substantial ethical duty on developers to protect such information from unauthorized access. Developers must have a comprehensive understanding of cybersecurity to effectively implement the necessary safeguards, ensuring the security of personal data.

Incorporating tools like CodeQL into the software development process supports these ethical considerations. By enabling developers to identify and fix vulnerabilities before software release, CodeQL minimizes the chances of security breaches. This proactive stance not only bolsters the security of digital services but also strengthens user trust, firmly anchoring these services in ethical practices that prioritize the safety and integrity of user data.

7.6 Conclusion

Our research enhances the capabilities of developers to identify and mitigate security vulnerabilities early in the development process, contributing to several key areas of sustainability and societal well-being. By reducing server loads and energy consumption, our work supports environmental sustainability (SDG 13: Climate Action) and aids in building a resilient digital infrastructure (SDG 9: Industry, Innovation, and Infrastructure). Furthermore, by securing critical health infrastructure, we enhance public health and safety (SDG 3: Good Health and Well-being), and by mitigating financial losses from data breaches, we promote economic stability (SDG 8: Decent Work and Economic Growth).

Additionally, our efforts in bolstering cybersecurity knowledge and practices help to protect sensitive information, thereby fostering greater public trust in digital services and addressing ethical considerations critical to today's digital era. This supports SDG 16: Peace, Justice, and Strong Institutions.

In summary, this chapter consolidates the significant impacts of our thesis across environmental, health, economic, societal, and ethical dimensions, underscoring how integrating advanced cybersecurity practices like CodeQL can enhance both the security of our digital world, and thereby society as a whole.

Bibliography

About cwe. (2024). MITRE. Retrieved April 29, 2024, from https://cwe.mitre.org/about/index.html

About dataflow analysis. (n.d.). GitHub. Retrieved May 18, 2024, from https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/

About DHIS2. (n.d.). DHIS2. Retrieved April 16, 2024, from https://dhis2.org/about/

About open hospital. (2024). Open Hospital. Retrieved April 16, 2024, from https://www.open-hospital.org/about-open-hospital/]

Att&ck matrix for enterprise. (2024). MITRE. Retrieved May 18, 2024, from https://attack.mitre.org/

Avgustinov, P., de Moor, O., Jones, M. P., & Schäfer, M. (2016). Ql: Object-oriented queries on relational data. *ECOOP*. Retrieved May 1, 2024, from https://drops.dagstuhl.de/storage/00lipics/lipics-vol056-ecoop2016/LIPIcs.ECOOP.2016.2/LIPIcs.ECOOP.2016.2.pdf

Brescia, E. (2021, April 22). *Environmental sustainability at github*. The GitHub Blog. Retrieved May 9, 2024, from https://github.blog/2021-04-22-environmental-sustainability-github/

Burt, T. (2020). *Microsoft report shows increasing sophistication of cyber threats*. Microsoft. Retrieved April 30, 2024, from https://blogs.microsoft.com/on-the-issues/2020/09/29/microsoft-digital-defense-report-cyber-threats/

Capture the flag 101 overview. (2024). CTF101. Retrieved May 18, 2024, from https://ctf101.org/

CodeQL. (n.d.). Github. Retrieved April 18, 2024, from https://codeql.github.com/

Codeql documentation [Accessed: 2024-04-16]. (2024). *GitHub*.

contributors, W. (2024). *Mitre corporation*. Retrieved May 18, 2024, from https://en.wikipedia.org/w/index.php?title=Mitre_Corporation&oldid=1224660973

Cost of a data breach report 2023. (2023). IBM. Retrieved April 29, 2024, from https://www.ibm.com/reports/data-breach

Cross site scripting. (n.d.). OWASP. Retrieved May 16, 2024, from https://owasp.org/www-community/attacks/xss/

Csrf. (n.d.). OWASP. Retrieved May 16, 2024, from https://owasp.org/www-community/attacks/csrf

Cve. (n.d.). MITRE. Retrieved May 18, 2024, from https://cve.mitre.org/

CWE coverage for Java and Kotlin. (n.d.). Github. Retrieved April 30, 2024, from https://codeql.github.com/codeql-query-help/java-cwe/

Dewhurst, R. (2020). Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis#

False positive: HTTP Response Splitting. (2024). Github. Retrieved April 30, 2024, from https://github.com/github/codeql/issues/15056

Friedman, N. (2019). *Semmle to GitHub*. Github. Retrieved May 7, 2024, from https://github.blog/2019-09-18-github-welcomes-semmle/

Gillis, A. S. (2020). *What is static analysis (static code analysis?* Retrieved May 13, 2024, from https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis

Github security lab: Capture the flag. (2024). *Github*. Retrieved April 18, 2024, from https://securitylab.github.com/ctf/

Hansen, R., & Kern, C. (2024, April). *Tackling cybersecurity vulnerabilities through secure by design*. Retrieved May 11, 2024, from https://blog.google/technology/safety-security/tackling-cybersecurity-vulnerabilities-through-secure-by-design/

hern, A. (2017). Hacking risk leads to recall of 500,000 pacemakers due to patient death fears [newspaper]. *The Guardian: Technology*. Retrieved May 10, 2024, from https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update

Hossain, S. (2018). Rework and reuse effects in software economy. *Computer Research*. Retrieved May 7, 2024, from https://computerresearch.org/index.php/computer/article/view/1780/1764

HTTP response splitting. (n.d.). Github. Retrieved April 30, 2024, from https://codeql.github.com/codeql-query-help/java/java-http-response-splitting/

Inefficient Regex Documentation. (2024). Github. Retrieved April 30, 2024, from https://codeql.github.com/codeql-query-help/java/java-redos/

Julien. (2023, June 28). *Software development for sustainable development: Unlocking the potential of the environmental sector*. Bocasay. Retrieved May 9, 2024, from https://www.bocasay.com/software-development-environmental-sector/

Log injection. (n.d.). OWASP. Retrieved April 30, 2024, from https://owasp.org/www-community/attacks/Log_Injection

Lutkevich, B. (2023). *What is gpt-4? everything you need to know | techtarget*. Retrieved April 25, 2024, from https://www.techtarget.com/whatis/definition/GPT-4

Matke, M. (2023). *Understanding cybersecurity frameworks: Stride, owasp top 10, and mitre att&ck*. Retrieved April 18, 2024, from https:

//medium.com/@mangeshmatke/understanding-cybersecurity-frameworks-stride-owasp-top-10-and-mitre-att-ck-8803a2b0cfa5

Metadata for CodeQL queries. (n.d.). Github. Retrieved May 7, 2024, from https://codeql.github.com/docs/writing-codeql-queries/metadata-for-codeql-queries/

Ntnu curriculum. (2024). NTNU. Retrieved May 18, 2024, from www.ntnu.no/studier/studieplan#programmeCode=BIDATA&year=2021

Owasp top ten. (2023). *OWASP*. Retrieved April 18, 2024, from https://owasp.org/www-project-top-ten/

Path Traversal. (n.d.). OWASP. Retrieved April 30, 2024, from https://owasp.org/www-community/attacks/Path_Traversal

Redos. (n.d.). OWASP. Retrieved April 30, 2024, from https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS

Sql injection. (n.d.). OWASP. Retrieved May 6, 2024, from https://owasp.org/www-community/attacks/SQL_Injection

Ssrf. (n.d.). OWASP. Retrieved May 5, 2024, from https://owasp.org/www-community/attacks/Server_Side_Request_Forgery

Statistics results. (2024). NIST. Retrieved April 25, 2024, from https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all&isCpeNameSearch=false

Unsafe deserialization. (n.d.). OWASP. Retrieved May 16, 2024, from https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data

What is datalog. (n.d.). Oxford Semantic Technologies. Retrieved May 18, 2024, from https://www.oxfordsemantic.tech/faqs/what-is-datalog

Wolford, B. (2024, April). *What is gdpr?* Retrieved May 4, 2024, from https://gdpr.eu/what-is-gdpr/

NTNU

Norwegian University of
Science and Technology