UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Development of a Website for Creation of Vulnerability Datasets

Miguel Pinto da Silva Ferreira

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves
Profª. Doutora Ibéria Vitória de Sousa Medeiros

2023

Acknowledgments

I would like to express my profound gratitude to both my supervisors, Prof. Doutor Nuno Neves and Profª. Doutora Ibéria Medeiros, for all the invaluable support and availability offered throughout the development of this dissertation. I can safely admit that I was fortunate in having the chance to work with both, as without their expertise and guidance, this thesis would not have been possible to accomplish. It truly was a pleasure.

I would also like to deeply thank my father. The advice and motivation he gave me kept me going in the long run, giving me the necessary strength to always try to achieve the best that my potential allows. His unconditional friendship and support belong to the group of fundamental reasons that keep me striving and motivate me to do better. I also owe deep thanks my uncle and aunt, who are very dear to me, made me feel at home no matter the circumstances, and always treated me like a son of their own. I also thank my cousins, who always treated me like a brother, in the good and bad times. A mention to my cousins' grandmother, Maria, who I also consider a grandmother of my own and to Isabel, my cousins' aunt, who I see as an aunt of mine. My gratitude extends to my grandmother, who never forgets about me and wishes me nothing but the best.

Finally, a special dedication to my mother, who although is no longer present, she was, she is, and she will always be my source of inspiration in life, whether professionally or personally.

This work was partially supported by the national funds through P2020 with reference to the XIVT project (LISBOA-01-0247, FEDER-039238, an ITEA3 European project (I3C4-17039)), and through FCT with reference to LASIGE Research Unit (UIDB/00408/ 2020 and UIDP/00408/2020).

Resumo

Com a evolução da era digital, garantir que o software seja robusto e seguro contra ameaças tornou-se um desafio. Para abordar este assunto, é importante não só detetar, mas também mitigar eficazmente as vulnerabilidades que este possa conter.

As ferramentas de análise estática, ou *Static Analysis Tools* (SATs) em inglês, podem ser consideradas uma solução viável, sendo capazes de realizar uma análise de baixo custo e também de rápida execução, mas trazem consigo um incoveninente: produzem, de uma forma frequente, uma elevada percentagem de falsos positivos e falsos negativos, o que leva a que os utilizadores não tenham muita confiança nestas.

As técnicas tradicionais de análise estática baseiam-se num conjunto de regras pré-definidas elaboradas por peritos de segurança onde são incluídas características ou padrões associados às classes de vulnerabilidades a detetar. Estas regras podem ser vistas como um conjunto de diretrizes que a ferramenta utiliza para examinar o código. Por exemplo, uma regra pode ser uma diretriz que obriga todas as entradas que recebem dados de um utilizador a serem sanitizadas, exigindo assim que a ferramenta notifique sempre que esta regra for violada. Apesar do elevado grau de conhecimento que um programador ou profissional de cibersegurança possa ter sobre linguagens de programação, a tarefa de elaborar uma ferramenta de análise estática não se torna menos propícia a erros ou menos complexa, o que faz com que esta técnica possa ter limitações práticas. Para além disto, tal tarefa envolve um trabalho exaustivo que normalmente resulta num elevado número de falsos negativos e falsos positivos.

Estudos recentes sugerem que as técnicas de Aprendizagem Automática (AA), ou, em inglês, Machine Learning (ML) podem ser utilizadas aquando da construção destas ferramentas, aumentando a sua eficácia. Estas, ao contrário da análise estática tradicional, não dependem inteiramente de regras e especificações pré-definidas pelos programadores. Em vez disso, a AA torna possível uma aprendizagem mais autónoma dos padões contidos nos dados e associados a vulnerabilidades, que é um processo conseguido através do uso de algoritmos que podem identificar e aprender esses padrões. Estes padrões são frequentemente extraídos de grandes bases de dados que incluem blocos de código potencialmente vulneráveis, sendo utilizados pelo modelo de AA durante a sua fase de treino. Em vez de ser o perito de segurança a criar as regras, a aprendizagem automática, juntamente com os dados, encarrega-se da tarefa. Isto não só simplifica o processo, como também elimina a necessidade de os programadores criarem ou atualizarem constantemente as regras.

A Aprendizagem Profunda (AP), ou em inglês, Deep learning (DL), um subconjunto da AA, consegue ainda aumentar o nível de automatização da aprendizagem dos dados. Esta metodologia também tem vindo a crescer e baseia-se na utilização de várias camadas de redes neuronais artificiais, que consistem numa estrutura de nós ou neurónios interligados que se assemelham ao cérebro humano. Esta estrutura é

responsável pelo reconhecimento de padrões e pela interpretação de dados sem qualquer trabalho prévio, incluíndo texto, imagem e, evidentemente, blocos de código. Em contraste com os métodos tradicionais e abordagens de AA, a AP é capaz de eliminar completamente a necessidade de trabalho manual efetuado pelos especialistas, uma vez que esta metodologia é capaz de extrair características mais significativas dos dados quando comparada com a AA. De maneira a conseguir obter o máximo proveito destas estratégias, são necessárias bases de dados fiáveis para treinar os modelos de AA.

Esta dissertação tem por objetivo fornecer uma metodologia que permita obter estas bases de dados robustas de maneira a que, por sua vez, possam ajudar no desenvolvimento de modelos de AA capazes de identificar vulnerabilidades em programas. Para tal, criámos uma nova abordagem para a construção destas bases de dados, que consiste na construção de uma aplicação web – o *BugSpotting*. Através desta plataforma, conseguimos reunir classificações de excertos de código efetuadas por um elevado número de pessoas. Esta metodologia de contribuição coletiva, chamada em inglês de *crowdsourcing*, consiste num processo de resolução de um dado problema ou obtenção de serviços recorrendo ao uso da inteligência humana para executar uma dada tarefa. Após a conclusão da tarefa em causa, a pessoa que a completou pode receber algum tipo de compensação, como por exemplo, um pagamento monetário. Isto nem sempre acontece, visto que os motivos que estão por detrás da execução de tais tarefas podem ir para além da compensação monetária. Outras motivações podem ser a obtenção de reconhecimento social, a contribuição para grandes projectos ou mero entertenimento. No caso da nossa solução, o objetivo é obter classificações de excertos de código que possam conter vulnerabilidades web no contexto da linguagem de programação PHP.

A razão para o uso desta metodologia prende-se com o facto de as SATs, como já mencionado anteriormente, gerarem muito falsos positivos e negativos. Através da opinião de várias pessoas, pretendemos mitigá-los de forma a que os resultados obtidos sejam de confiança. A plataforma do *BugSpotting* permite aos os utilizadores classificar blocos de código, indicando se estes são (ou não) vulneráveis para um dado conjunto de classes de vulnerabilidades. Estes ainda podem confirmar as suas classificações efetuadas no passado, assim como consultar a sua reputação dentro do sistema.

Visto que esta plataforma se trata de uma aplicação web, esta pode ser dividida em dois componentes principais: o *frontend* e o *backend*. Enquanto o primeiro pode ser entendido como a parte visual de uma aplicação, a segunda, pelo contrário, não é visível para o utilizador e é responsável pelo processamento e armazenamento de dados, bem como pela comunicação com sistemas externos. Para a construção do BugSpotting utilizámos diversas tecnologias tanto de *frontend* como de *backend*. A nível de prototipagem das páginas e secções do website, foi escolhido o Figma, que consiste numa ferramenta que permite o *design* e teste de interfaces gráficas de aplicações; o React foi selecionado como biblioteca de frontend, permitindo a implementação a nível de código do design criado através do Figma. Utilizámos também o .NET como framework de backend. Esta permitiu a implementação de uma API que é responsável por responder aos pedidos provenientes do frontend. Por fim, optámos por utilizar a base de dados relacional MySQL para a gestão dos dados.

O *BugSpotting* foi construído tendo em consideração várias boas práticas e princípios de desenvolvimento web. Uma das caraterísticas centrais nesta aplicação foi o desenvolvimento de dois algoritmos: primeiro, um algoritmo dedicado à filtragem de um bloco de código (com base num conjunto de critérios). Uma vez obtidas classificações suficientes (de acordo com um limiar pré-definido), um se-

gundo algoritmo é executado, responsável por determinar, através de consenso entre as classificações, a classificação final do excerto de código.

Para avaliar a qualidade do *BugSpotting*, foi elaborado um questionário para recolher informação sobre os vários aspectos da aplicação Web relacionados com a UI e a UX. A avaliação foi dividida em duas partes: uma quantitativa e outra qualitativa. A primeira consistiu numa série de questões sobre a aplicação em termos de UI e UX, as quais foram avaliadas pelos utilizadores numa escala de um a dez, onde um indicava que os utilizadores discordavam veemente da afirmação e dez significava concordância plena para com a afirmação. O feedback geral dos utilizadores tanto a nível de UX como de UI foi amplamente positivo, confirmando a usabilidade da plataforma, apesar de haver espaço para melhorias futuras em certas áreas. Um exemplo de uma possível melhoria que foi referida com alguma regularidade tanto a nível de UI como de UX foi a tabela de reputação do utilizador. Os participantes tinham dúvidas sobre o funcionamento da reputação, alguns referindo que não conseguiram perceber bem como funcionava o sistema de pontos.

Embora o nosso estudo não tenha sido capaz de chegar a um consenso sobre a classificação final de cada fragmento de código devido a várias limitações, como a falta de participação dos utilizadores e o atraso na disponibilização da aplicação ao público, é essencial referir que o objetivo principal de construir uma plataforma funcional foi atingido com sucesso. O sistema está a funcionar corretamente, com potencial para melhorias futuras. Ainda assim, com os dados que obtemos até ao momento, tiramos algumas conclusões sobre as classificações dos excertos de código que se revelaram bastante promissoras.

Palavras-chave: vulnerabilidades em aplicações web, deteção de vulnerabilidades, análise estática, aprendizagem automática, contribuição coletiva

Abstract

With the evolution of the digital era, guaranteeing the robustness and security of software has become a major concern. In order to address this subject, it is important to effectively not only detect, but also mitigate software vulnerabilities. Static Analysis Tools (SATs) present a cost-effective solution to this, being able to achieve a cheap and fast analysis, but often incur in a high percentage of false positives and negatives. Recent studies suggest that machine learning (ML) techniques could enhance the effectiveness of these tools, but this requires trustworthy and reliable datasets to train the ML models.

This dissertation aims to provide a way of create the aforesaid datasets that can help with the development of ML models capable of identifying vulnerabilities in computer programs. To achieve this, we propose a novel approach to construct these datasets, which consists in collecting inputs from the crowd as a way of mitigating the false positives and negatives generated by the SATs, but at the same time leverage from their deterministic classifications. This approach is applied within the context of web vulnerabilities that appear in applications built with the PHP programming language. To facilitate crowdsourcing, we developed a user-friendly website called BugSpotting where users can classify PHP code snippets, indicating whether these are vulnerable (or not vulnerable) to a set of vulnerability classes. With the results obtained both from the crowd and the SATs, we are able to obtain a reliable and trustworthy dataset comprised of accurately classified PHP code snippets.

We evaluated BugSpotting in terms of UI and UX and the results obtained were very satisfactory. Moreover, although we were not able to reach a consensus about the code snippet's final label, we still manage to analyse the data we have collected until the moment, showing promising results.

Keywords: web application vulnerabilities, vulnerability detecion, static analysis, machine learning, crowdsourcing

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

AA Aprendizagem Automática

AP Aprendizagem Profunda

API Application Programming Interface

CLR Common Language Runtime

CSS Cascading Style Sheets

DBMS Database Management System

DFD Data Flow Diagram

DL Deep learning

DOM Document Object Model

DT Directory Traversal

ER Entity-Relationship

HATEOAS Hypermedia as the Engine of Application State

HMAC Hash-based Message Authentication Code

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IoC Inversion of Control

JSON JavaScript Object Notation

JSX JavaScript Syntax Extension

JVM Java Virtual Machine

JWT JSON Web Token

KAVe Knowledge-based Agent-system Vulnerability-detector

LFI	Local File Inclusion
MAC	Message Authentication Code
MIME	Multipurpose Internet Mail Extensions
ML	Machine Learning
MPA	Multi Page Application
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
NIST	National Institute of Standards and Technology
NPM	Node Package Manager
NVD	National Vulnerability Database
OS CI	OS Command Injection
OWASP	Open Web Application Security Project
PHP	PHP Hypertext Preprocessor
PHP CI	PHP Code injection
RDBMS	Relational Database Management System
REST	Representational State Transfer
RFI	Remote File Inclusion
SAMATE	Software Assurance Metrics and Tool Evaluation
SARD	Software Assurance Reference Dataset
SAT	Static Analysis Tool
SCD	Source Code Disclosure
SDK	Software Development Kit
SoC	Separation of Concerns
SPA	Single Page Application
SQLi	SQL Injection
TLS	Transport Layer Security

UI	User Interface
URI	Uniform Resource Identifier
UX	User Experience
VM	Virtual Machine
WAP	Web Application Protection
XSS	Cross-site Scripting

Chapter 1

Introduction

Over the last few years, society has become more dependent on the digital world than ever. It is a known fact that the reliance on the Internet and technology has immensely grown due to their ease of access and usefulness. Along with this increase, cyberattacks have seen their numbers rise [34], and therefore, there is an urgent need to prevent and detect these malicious actions. A dominant cause of this issue is the poorly designed software that is often released for production environments without being tested properly for security, leaving various kinds of vulnerabilities in their code. These vulnerabilities can have different flavours, such as coding errors, flaws in the implemented logic or failure to comply with good programming practices. All the aforementioned issues can contribute to an attacker conducting malicious activities that can lead to serious consequences for the victim. For instance, an individual can face identity theft and fraud, and companies can incur major financial losses. Taking it even further, it can lead to a public lack of trust in digital systems. Hence, it is safe to admit that addressing software vulnerabilities is a crucial aspect of cybersecurity and by identifying and emending these, it becomes possible to significantly reduce the chances of successful cyberattacks.

In this sense, it is possible to detect and prevent vulnerable code by resorting to several techniques. These can be divided into three main categories: static analysis, dynamic analysis, and hybrid analysis [42]. The first one mentioned is a type of analysis that consists in the inspection of the source code or bytecode without the actual execution of the program. Dynamic analysis is the testing and evaluation of a program by executing the latter in real time. It can be beneficial since it is able to find vulnerabilities that might not be visible in a static analysis but can be detected when a specific input is used to run the program. Finally, the hybrid analysis technique is a combination of the both already mentioned. It leverages the strengths of each one to provide a more comprehensive analysis of the code.

Furthermore, recent studies show that it is also possible to locate vulnerabilities in programs through the use of machine learning (ML) strategies [26]. These ML techniques learn from patterns in data and based on these, they are capable of predicting vulnerabilities in new or unseen code snippets (i.e., small programs) – without the need of being explicitly programmed to. Moreover, these ML techniques often have an algorithm behind: it is the algorithm that will learn from these patterns and will be able to discern whether a given code snippet is indeed vulnerable or not.

In order to achieve the above stated, ML strategies typically use a dataset containing code snippets to train the model [12]. The quality and reliability of these datasets significantly impact the ML model's performance. For instance, if the dataset contains a high percentage of bias in the data and the labeling is mostly inaccurate, it becomes obvious that the quality of the derived ML model will be deteriorated, resulting in imprecise predictions. Therefore, the reliability and accuracy of a dataset are crucial for achieving accurate vulnerability detection results. The more reliable and unbiased the dataset, the more likely the ML model will effectively identify vulnerabilities in code snippets.

Furthermore, a current problem is the lack of well founded and dependable datasets, since the existing ones often suffer from wrong labels and poor vulnerability descriptions [31]. Despite the aforementioned, it is still possible to take advantage of the available datasets, since this contain code snippets that can be analysed in order to derive new datasets. In this regard, one widely used dataset in this domain is the Software Assurance Reference Dataset (SARD) [14], a project supported by the National Institute of Standards and Technology (NIST). This dataset ranges from simple code snippets to production software and contains more than 450,000 test cases. It supports many programming languages, including JAVA, C, C++, C# and PHP, being able to cover more than 150 classes of weaknesses. As such, it has been widely used as a benchmark to test vulnerability detection methods.

Although there are some real-world examples of code snippets in SARD, it is worth mentioning that the majority of these programs are synthetic [15], i.e., they do not correspond to real-world software. Such a synthetic dataset can become less valuable in terms of data representation. For instance, there is the risk that the ML model will be less general as it will be restricted to basic and impracticable code snippets which often do not occur in real-world software, ending up being trained for those patterns only [19]. Although, at a first glance, this may pose a potential disadvantage, there are several benefits that SARD offers. SARD ensures that the vulnerabilities are self-contained within the code snippets and each one of these only possesses relevant lines of code, thereby making the learning process more efficient. Moreover, the examples provided by SARD are often specifically tailored for evaluating the effectiveness of analysis tools. It provides both vulnerable and non-vulnerable code snippets in order to preserve a balanced dataset. As already mentioned before, with a vast universe of code examples from many programming languages, SARD supplies researchers and developers with the opportunity to select and adapt their testing to meet the intended requirements.

1.1 Motivation

The aforementioned observations converge to the necessity of a novel strategy for building reliable and accurate datasets. Hence, this project proposes the construction of an infrastructure (website) that will support the creation of vulnerability datasets with code snippets, which will be labeled as vulnerable (or not vulnerable) to a certain class of vulnerability. The code snippet labeling will focus on vulnerabilities in web applications written in PHP, such as SQL Injection (SQLi) and Cross-site Scripting (XSS). The decision to focus on web vulnerabilities in the PHP programming language is a consequence of its major prevalence in the field of web applications. More precisely, it supports more than 80% of the top ten million websites [22], which include globally recognized platforms such as Tesla, Wikipedia, Tumblr, among others. Despite its magnitude, PHP is far from being flawless in terms of security design. The language is prone to numerous forms of attacks that can potentially lead to significant security breaches, data compromise and potential manipulation of server functionality. The problems described can be explained by PHP's poor security design, which allows numerous entry points that attackers can leverage from in order to inflict harm to the web application. In fact, the language's inconsistent and poorly implemented APIs make it prone to programming errors that can lead to vulnerabilities like the ones previously mentioned. Given its extensive use throughout the web and the problems above stated, these make PHP a suitable candidate for security improvements.

Considering the problems highlighted above, it becomes clear that a fast and cost-efficient method for identifying vulnerabilities in programs is necessary. One such approach is resorting to static analysis tools (SATs). These tools can be a feasible and valuable option when it comes to improving security. They have the capacity to identify potential issues early in the development process. This early detection helps preventing costly and time-consuming debugging efforts, offering the possibility of fixing the issues before the software goes into production. As a result, possible security breaches can be mitigated since the developer is aware of the vulnerabilities. Also, they can be used to enforce certain policies, such as coding standards and practices within a development team. This allows for a better control of quality and security of the software.

Although the security improvements offered by these tools are of great value, one must be aware of the drawbacks that arise. One of the disadvantages in the employment of these tools is the possibility of the analysis becoming time-consuming and cumbersome. Despite the fact that the analysis itself is automated and can quickly scan a vast code base, it still requires human interpretation of the results. This can be time-consuming and may delay the development process. Moreover, these tools are also limited in the sense that they can only identify possible vulnerabilities without executing the programs. The inability of detecting runtime errors can be seen as an incomplete analysis to the application, which can leave some undiscovered vulnerabilities. Finally, these tools often produce false positives, which can be misleading and require extra time and effort. In addition, these tools can also produce false negatives, meaning they fail to detect actual vulnerabilities in the program.

As previously mentioned, ML techniques offer a viable way when it comes to the discovering of vulnerabilities. Their ability to make predictions on new or unseen code snippets based on trained algorithms makes them a powerful option in terms of static analysis. However, as we already noted, the effectiveness of ML techniques mostly depends on the quality of the datasets their models are trained on. The reason for this is quite straightforward: ML algorithm's prediction are based on the patterns identified in the training data. As a result, any biases or inaccuracies that the dataset may contain will be reflected directly in the prediction model. A simple example would be a dataset that is composed of code snippets that are incorrectly labeled. The trained model could fail to identify real vulnerabilities (false negatives) or even recognize harmless code as vulnerable (false positive). Furthermore, if the dataset does not include a wide range of possible scenarios (in this case, multiple code snippets that are both vulnerable and not vulnerable to various vulnerability types) will result in an algorithm that performs well on some data but poorly on other. For instance, if the dataset is mostly composed of code snippets containing XSS vulnerabilities, it is natural that the algorithm will under perform when it comes to other vulnerability types. Hence, the necessity of accurate, unbiased and sound datasets is of critical importance for an ML model predict in a precise and trustworthy manner.

As earlier discussed, the PHP code samples will have to be labeled. However, due to the frequency of false positives and negatives generated by SATs, the task of producing accurate and trustworthy labels becomes less feasible, requiring an alternative strategy. Therefore, to reduce this potential errors and enhance the accuracy of vulnerability detection, the classification process could be complemented with crowdsourcing. This methodology consists in using volunteers to perform a task which, in this case, would be the labeling of PHP code snippets as vulnerable or not vulnerable for a given vulnerability class.

To achieve such complementing and contribution of SATs, a web application will be built in order to allow the users to classify the code snippets present in the dataset (i.e., the web application database), by identifying the class and location of possible vulnerabilities these may (or may not) contain. Our approach will leverage the fact that SATs can, in a quick and inexpensive way, identify the class and location of a possible vulnerability in a code snippet, while combining the efforts of a large group of volunteers to further label the code snippets in order to ensure that the false positives and negatives produced by the SATs are mitigated.

1.2 Proposed Goal

As described in the previous section, the need for a robust, trustworthy and accurate dataset is crucial to train the ML models so that vulnerabilities present in PHP code are discovered in an accurate way. Thus, the primary goal of this project is to develop a novel strategy to create reliable datasets, with accurately labeled code snippets.

In order to accomplish the latter, we will base ourselves in two methodologies: the use of SATs and crowdsourcing. More specifically, we aim to create a new strategy for building datasets which will be further used to train ML models, with the ultimate goal of identifying vulnerabilities in PHP code more accurately. We leverage the fact that SATs are able to provide, in a rapid and cost-effective way, classifications for the code snippets present in the database (labeling them as vulnerable or not vulnerable for a given vulnerability class) and, to mitigate the possible false positives and false negatives that these tools may generate, we resort to crowdsourcing, where we take advantage of the collective intelligence to also classify the code snippets. By following this procedure, we expect to reduce the percentage of errors produced by the SATs.

The incorporation of these two strategies will not only enhance the trustworthiness of the dataset but will also allow a more effective machine learning model capable of pinpointing vulnerabilities with higher precision.

The construction of a web application – BugSpotting – allows the employment of both aforementioned strategies. On the one hand, users will be able provide their classifications about the code snippets. On the other hand, the platform itself will also be responsible for executing an SAT and combine the results from both methodologies, being able to achieve more precise labels.

The results obtained about the labels, although not concrete, show promising indicators that the platform is working correctly. Also, most of the opinions obtained from the users about the website's UI and UX components were very positive, verifying the effectiveness and credibility of BugSpotting.

1.3 Contributions

Taking into consideration what has been stated, this project offers the following contributions:

1. A web application that lets the user classify PHP code snippets, indicating whether they are vulnerable or not. The user will have a reputation associated that reflects the level of his expertise when performing the classification, which will be taken into consideration and have more or less an impact on the code snippet's final labeling result.

2. An algorithm that will choose the code slice to be sent to the user based on a set of criteria (e.g., number of total classifications). The algorithm will also be able to reach a consensus in order to determine the PHP code snippet's final label by merging the classifications obtained from the tools and the workers.

3. A data model, representing the entities that constitute the system's database (and how they relate to one another), which will record all the classifications and information of the code snippets.

1.4 Document Structure

The document is organized in the following structure:

- **Chapter 2 – Background and Related Work:** This chapter provides necessary background information and key concepts related to vulnerabilities in code, static analysis tools, datasets, machine learning and crowdsourcing. It also presents related works in the field of vulnerability detection, static analysis tools and existing datasets.

- **Chapter 3 – Bugspotting:** This chapter describes the proposed web application – BugSpotting – focusing on its requisites and both system's modelling and data model.

- **Chapter 4 – Implementation:** This chapter delves into the details of the implementation process of BugSpotting, discussing the technical aspects of the web application.

- **Chapter 5 – Evaluation:** This chapter presents the methods and metrics used to evaluate the performance and reliability of BugSpotting.

- **Chapter 6 – Conclusion:** This chapter concludes the dissertation by summarizing the findings, discussing the limitations, and suggesting future improvements and features.

Chapter 2

Background and Related Work

This chapter provides the concepts required to understand the problem addressed in this thesis and thus, the proposed solution. It starts by presenting the definition of vulnerability along with the comprised vulnerability classes in Section 2.1. Next, in Section 2.2, static analysis is defined followed by an explanation of crowdsourcing in Section 2.3. In Section 2.4, we tackle the dataset construction topic, presenting the use case of the SARD dataset. Finally, in Section 2.5, we present the related work.

2.1 Vulnerability Classes

In accordance with NIST [38], a vulnerability is a *weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source*. From this definition, it is possible to derive that if left unattended, vulnerabilities can open many doors to potential successful attacks.

The impact that may arise from vulnerabilities can lead to catastrophic consequences. For instance, vulnerabilities can grant an intruder unauthorized access to sensitive information, disrupt the system's usual operations and, in extreme situations, concede full control of the system to the attacker. Consequently, problems like financial loss, reputation damage, and even psychological effects can escalate, posing a serious threat to the organizations [23].

Hence, it becomes critical to take action and identify the vulnerabilities in the system in order to prevent and mitigate them. To develop effective solutions, it is first necessary to understand in what the vulnerabilities consist of. In the next subsections, we will focus on eight particular types of vulnerabilities that our solution targets. These are specifically related to web vulnerabilities that are present in web applications developed in the PHP programming language.

2.1.1 SQL Injection

An SQL Injection (SQLi) is described as a type of attack where the opponent manipulates SQL queries by placing malicious input in the statements (by using SQL keywords, operators or metacharacters). This type of attack can have serious consequences if well executed by the attacker. It provides the capability to access sensitive information, manipulate this data through insertion, modification, or removal and even allow the execution of administrative tasks on the database, such as shutting it down. In some cases, it might enable the attacker to read specific files on the Database Management System (DBMS) file system or issue commands to the operating system.

It is worth mentioning that this type of attack is particularly common in the PHP programming language. This is justified by the fact the applications powered by this language use older interfaces that often lack protection against this type of exploit [11], as for example, the use of the *mysql_query* function.

In order to prevent against this type of attack, one can resort to several strategies. The most common and effective one is the use of parameterized queries (also known as prepared statements). This approach requires the developer to first define all SQL code, then pass each parameter to the query separately. This separation process enables the database to discern between the actual code (the query) and the data (inputs), making it less likely for an attacker to successfully damage the system. Another well known and commonly used way of preventing SQL Injections is by using escaping. This strategy is quite straightforward: it consists in adding an escape character before any special characters in the SQL statements, preventing them from being interpreted in a dangerous manner. This strategy is employed by sanitization functions offered by PHP, e.g., *mysqli_escape_string* function.

```php
<?php
// Connect to database
$conn = new mysqli("localhost", "username", "password", "database");

// Unsafe SQL query
$user_id = $_GET['user_id'];
$sql = "SELECT * FROM users WHERE id = $user_id";

$result = $conn->query($sql);
?>
```

Figure 2.1: Example of a PHP code snippet vulnerable to SQL

The code snippet in Figure 2.1 depicts the scenario of an SQL injection attack. The main issue here is the direct use of a value supplied by the user (*$_GET['user_id']*) in the SQL query. This allows the manipulation of the *user_id* parameter in the URL to execute malicious SQL commands.

2.1.2 Cross-Site Scripting

Cross-site Scripting (XSS) is depicted as a type of web attack where malicious code is sent to the victim's computer and executed in the browser (in the form of a script), as part of an interaction with a vulnerable web application. This type of attack can have severe consequences, allowing the attacker to steal sensitive data such as session cookies, perform actions on behalf of the victim and even rewrite contents of the webpage. There are three variants of XSS attacks: reflected XSS (or non-persistent), stored XSS (or persistent), and finally, DOM-based XSS.

The first one mentioned can occur when the malicious script is part of an HTTP request that is reflected back to the user, triggering the script to execute. This happens when a user is tricked into clicking a malicious link. To prevent reflected XSS, one of the most sound techniques is to validate the user inputs. User input should be treated as untrusted no matter the source and a mechanism that checks against grammatical and semantic requirements should be implemented. Another good technique is output encoding or escaping, which ensures that any user input data is safely rendered in the browser, preventing it from being interpreted as malicious code. Additionally, the formulation of a content security policy can be beneficial. These policies are defined in the HTTP response headers and instruct the browser on trusted content sources, for instance, to block the execution of scripts for specific domains.

Stored XSS is a type of XSS vulnerability where the malicious script is permanently stored on the target server (e.g., blog, database). This means that the server, at certain points, delivers the malicious script to any user who accesses the relevant application page or section (e.g., a blog post application). The script is not injected directly by the attacker into the victim's browser but instead stored on the server and delivered to users in the course of normal browsing, and thus executed in the victim's browser. The techniques used to prevent this type of attack are similar to the ones used in reflected XSS: it is crucial to inspect the user's input. User input should always be treated as untrusted and should be validated or sanitized. Moreover, output encoding and escaping also helps guaranteeing malicious content (in this case, the scripts stored in the server) is not executed in the victim's browser. Finally, the implementation of content security policies is also useful: they can prevent the attacker from injecting malicious scripts from external sources.

Finally, there is DOM-based XSS, a type of attack in which the malicious script is part of a web page's client-side script and is executed due to improper user input validation. In this attack, the payload, or malicious code, is executed as a result of modifications to the Document Object Model (DOM) environment in the victim's browser. A common form of DOM-based XSS involves manipulating a URL parameter in such a way that the DOM is deceived into executing a malicious script injected as part of the URL. Counter measures include the ones previously mentioned for other XSS types, with additional focus on the use of safer JavaScript APIs.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome Page</title>
  </head>
  <body>
  <form method="post" action="">
    <label for="username">Enter your username:</label>
    <input type="text" id="username" name="username">
    <input type="submit" value="Submit">
    </form>
    <?php
      $username = $_GET['username'];
      echo "Welcome, " . $username;
    ?>
  </body>
</html>
```

Figure 2.2: Example of a PHP code snippet vulnerable to XSS

The code snippet in Figure 2.2 is vulnerable to a XSS attack. This example shows that the web page includes a form that will be completed by the user. The issue is the lack of sanitization of the user input. The PHP script dynamically outputs a welcome message that is dependent on the user's input, without proper validation. A malicious user could inject JavaScript code in the place of a valid username, which would end up being executed in the browser of someone visualizing the page.

2.1.3 Remote File Inclusion

A Remote File Inclusion (RFI) is an attack in which the attacker takes advantage of *dynamic file include* mechanisms in web applications. The objective is to deceive the vulnerable web application into including malicious remote files, which often take the form of harmful code. For instance, the adversary could include, in the URL of the vulnerable application, the address where the malicious code is located and since there will be no input sanitization, the vulnerable application will end up fetching (including) the file and consequently executing it.

To prevent this type of vulnerability, one can resort to various techniques. The most common and effective one is the sanitization and validation of input. This ensures that the data received conforms with the expected format. Moreover, it is also possible to apply a simple strategy which consists of disabling file inclusions. This can be done at language level. In addition, permission restrictions can be enforced in the application itself. By employing these, the attacker can not force some operations, such as the downloading and executing remote files.

2.1.4 Local File Inclusion

A Local File Inclusion (LFI) is very similar to the RFI vulnerability explained above. The logic is the same but the difference resides in the fact that the attacker can only include files in the local machine. For example, instead of relying in a URL (the remote location of the malicious file), this technique will leverage on the path to the harmful file as input. This attack can potentially enable an attacker to execute code, access sensitive information, or manipulate files on the server, depending on the permissions and type of file that is included.

In order to prevent this exploit, one can apply the same counter measures that were listed in the RFI case.

2.1.5 Directory Traversal

A Directory Traversal (DT), also known as Path Traversal, has its basis on the Remote and Local File Inclusion attacks. Although there will be no execution of malicious code, the attacker can still compromise the system. The attacker will manipulate the URL of the request by inserting the path of some critical local file, tricking the vulnerable application into returning its contents. The sensitive content that could be exposed by such an attack can vary, including source code files, logs, the server's list of users, amongst others. This type of attack, although it may seem less dangerous due to the absence of code execution, can still cause serious damage by leaking confidential information.

To prevent this threat, it is crucial to validate the user input. It should conform to the expected format and, most importantly, length. Another technique that can be use to mitigate this attack is the use of safe APIs. Some programming languages help the developer with the management of file paths and names in a safer manner. Another strategy would be the sanitization of user input. This includes the removing or encoding of special characters that could be used to traverse directories (e.g., "..", "/" and ".").

2.1.6 Source Code Disclosure

A Source Code Disclosure (SCD) attack consists in obtaining the vulnerable web application's source code. This type of vulnerability allows the adversary to inspect the source code and thus, get a better understanding of the server's logic. This allows the preparation of a more dangerous posterior attack. As mentioned before, one of the techniques for obtaining the code is through a Directory Traversal attack but this exploit can also occur due to server misconfigurations or vulnerabilities that result in the source code being returned as plain text rather than being processed by the server.

There are several ways for one to prevent this type of vulnerability. A common practice is proper server configuration. This means that the developer needs to ensure that it does not send back source code files. To achieve this, it is often necessary to configure the server in terms of which files should be processed. Another common technique is the limitation of permissions. This allows to specify which files can be accessed and which should not be disclosed. And, similarly to the vulnerability classes referred until this point, input validation also plays an important role to prevent this threat. This can include the escaping or encoding of special characters, preventing the adversary from traversing directories.

2.1.7 PHP Code Injection

A PHP Code injection (PHP CI) is a type of attack where the adversary injects custom code into the vulnerable web application. In the case of PHP, this often occurs with the execution of an *eval* function call, in which the attacker controls the input string that is fed to the function. It is worth noting that the impact of this type of attack can be severe, potentially allowing the adversary to steal sensitive data, manipulate data, or even gain full control of the web application.

To avoid this threat, there are various counter measures that can be employed. The most straightforward one, and in the context of the given definition above, is to avoid the use of the *eval* function, since it can execute PHP code from a string, which can be dangerous if the string is controlled by the opponent. Moreover, it is important to validate the user input to guarantee that the data received is in conformity with the expected format, length, type and range. Escaping the data from the user is also worth having into consideration. As already mentioned, XSS can be seen as the injection of malicious code in the victim's browser. In the context of this vulnerability class, this type of attack be leveraged to induce a PHP Code injection. If the attacker can modify the user's input through a XSS exploit, he can create a payload that when evaluated by the server, will execute arbitrary PHP code (by means of the infamous *eval* function). The latter description shows the importance of properly validating and sanitizing user inputs.

2.1.8 OS Command Injection

An OS Command Injection (OS CI), also known as Shell Injection, is a type of web vulnerability which lets an adversary to execute Operating System commands on the server that hosts the web application. This can be achieved by the fact that the application relies on the execution of a shell command to, for instance, retrieve some information the user asked. The attacker leverages on this, injecting shell commands to compromise the operation of the server.

To prevent this attack, it is no surprise that proper input validation is a must. More precisely, one should guarantee that the inputs received should not be interpreted as commands. Also, the use of shell commands should be avoided. Since this is the core of the vulnerability, the developer should find alternatives to this, such as resorting to safer, language-specific functionalities. In the case of shell commands being mandatory to perform some action, then one should escape special characters. This will help mitigating potential user commands that can be interpreted as shell commands.

2.2 Static Analysis

As the name suggests, static analysis consists in the analysis of the application's code without its execution [37]. Often, this technique is applied in two major phases: (i) the transformation of the code into an internal representation that facilitates the analysis (e.g., an Abstract Syntax Tree (AST)) and (ii) the checking of rules to find potential issues.

In the first phase, the code will be initially divided or broken into individual parts, also called tokens. This task is conducted by a program that is called a lexer (or tokenizer). These tokens correspond to the lexical elements of the input text, which, in practical terms, represent the programming language's syntax components, such as variable names, operators, keywords, among others. At this moment, the produced stream of tokens will be processed by a parser. The parsing process consists in rearranging the tokens into a hierarchical structure according to the rules of the programming language's grammar (it is the grammar that dictates how the statements in the language should be structured in order to make them valid). At the end of this first phase, the output is typically an AST, which consists of a tree-like representation of the program's source code where each node represents an element, such as a function, loop or conditional statement.

At this moment, the second phase takes place. After the AST is created, an analysis of this tree is carried out. The analysis may vary, including several techniques such as dataflow analysis, path-sensitive dataflow analysis, among others, in order to find potential vulnerabilities within the code. It is worth noting that the type of analysis employed may vary between tools. This second phase is determining for discovering and identifying possible exploits in the code that an attacker may leverage from. Thus, it allows developers to become aware of these vulnerabilities and secure them before the code is deployed.

It is commonly understood that when developing software, despite the best intentions and practices, developers can (and frequently) make mistakes. These mistakes often take the form of bugs which, under certain circumstances, can be manifested as security vulnerabilities in the code. The severity of the situation increases when these faults pass unnoticed and software goes into production. Hence, the system may be attacked and then compromised, and such exploits often lead to data breaches, disclosure of sensitive data and a loss of performance. Static analysis tools (SATs) are able to provide, in a timely and cost-effective fashion, a way of discovering these vulnerabilities in code, finding potential problems in an early software development stage. In addition, they also contribute to enhance the quality of the code in several ways [43], such as eliminating redundant code, supporting refactoring tasks and even optimize the code. It is thus possible to conclude that the examination of either the source code or the binary code of a program before its deployment is a great form of finding errors and maintaining quality, allowing the software to become more trustworthy and less prone to vulnerabilities (since the developer is able to detect and correct the vulnerability by obtaining the location and nature of the bug by the tool).

Another great advantage that SATs bring can be observed at an industry level, more precisely, a software management cost reduction: according to a study conducted by Baca et al. [13], with the use of a static analysis tool, a decrease of 17% in software cost is possible in terms of security code issues. The authors even go further and affirm that the sooner the issue is discovered, the lesser will be the cost of the correction.

Although, until this point, the static analysis methodology may look quite captivating, it also comes with some drawbacks: static analysis tools can produce many false positives and false negatives [25]. Even when complementing it with manual code analysis [25], this methodology by itself does not suffice for a secure and highly trustworthy code. Moreover, although SATs are automated tools, the results they provide must still be interpreted by the developers. This human labour task can become time consuming if the analysis is executed in a large code base. Besides this, since, as mentioned above, they often produce false positives and negatives, the developer must decide if the flagged issues are in fact a vulnerability or not. Furthermore, the employment of SATs do not concede a total analysis of the code in the sense that this type of technique does not detect any runtime issues that may arise, since the analysis is made without the execution of the code. The aforementioned can be seen as some of the reasons that development teams opt for not using static analysis tools. Often, the lack of trust and the time consuming task of interpreting the results obtained by the SATs may not seem appealing or worthwhile.

2.3 Crowdsourcing

In 2006, the term crowdsourcing was introduced by Jeff Howe in his article ”The rise of Crowdsourcing” [27]. According to the article, crowdsourcing could be understood as *the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call.* In other words, this methodology can be seen as the process of solving a problem or obtaining needed services by relying on the crowd, which after completing the task at hand, some kind of compensation is often received, such as a small monetary payment. Actually, this may not always be true, in the sense that the motives behind executing such tasks could go beyond monetary compensation. Other motivations could be to obtain social recognition, contribute to large projects, or merely entertainment.

The proliferation of this methodology can be explained by the growth of the Internet and advances in technology. The birth of Web 2.0 and the social media revolution allowed the development and increase of online platforms which would then serve as the place where organizations and the crowd would interact with one another. These platforms are often divided into two classes of users: the requesters and the workers [47]. A crowdsourcing website usually presents a set of tasks that are posted by the requesters. A worker will execute the task at hand and, as already mentioned before, may get rewarded for completing it. Besides the payment, the worker also increases his credibility and reputation if the task was correctly executed.

One thing that must be taken into consideration is the fact that crowdsourcing differs from open-sourcing. The difference between them relies on the fact that in crowdsourcing, although the crowd performs the tasks willingly, the product that originates from their conclusion is not free. On the other hand, in open-sourcing, the crowd contributes to a product that is already free and stays like that in the future, so the community is mainly enhancing it [45].

By employing crowdsourcing, the amount of benefits that an organization can gather is quite visible. In fact, it allows an organization that has a problem at hand to scale its resources by letting the problem be shared with an online community. The problem often takes the form of designing a product, solving a scientific issue, achieving consensus in some theme or process data by resorting to human intelligence [17]. Instead of solving these problems internally, the organization exposes them to the crowd, avoiding the costs of hiring, training, and supervising employees, and thus, a pay-per-task model is employed. This means that a company will only pay for the required task that it demanded. Besides reducing costs, it brings, at the same time, a whole new set of skills, ideas, and tools that effectively help to reach a solution for the problems. In spite of the benefits, it is important to know when this methodology should be employed. Crowdsourcing brings advantages when tasks require humans. More precisely, when machines can not complete or execute them in a more efficient way. For instance, it should be used when the problem is divisible into smaller tasks, the solution is objective and verifiable or the problem at hand does not require a vast knowledge about the theme [45].

The data collected from the crowd can be useful in many situations. One particularly relevant example is related to Machine Learning. Computers have difficulty when it comes to perceiving or identifying, for instance, the content of images or videos. By contrast, humans can easily conceptualize, discern and apply common sense to a certain labeling task. In this sense, by recruiting data annotators at a large scale, a dataset can be obtained as output and then be used to train ML models for various tasks [24]. A study conducted by Buecheler et al. [18] foresaw a 97.7% probability of obtaining an accurate answer within a universe of a million persons. Although in practice such a crowd is hard to achieve, it offers more trust and entices the use of this technique.

Despite the fact that crowdsourcing offers numerous benefits, it is important to also acknowledge its challenges and caveats. One of the main concerns is the quality and accuracy of the results obtained. A thorough research carried out by Daniel et al. [21] revealed that the quality of a crowdsourced task can have several factors on which its quality depends. These include the expertise of the workers that execute the tasks, the quality of the tasks itself (which is thereby dictated by the quality of the process that generates them), amongst others. The authors also affirm that previous empirical studies have reached the conclusion that a vast majority of the existing crowdsourcing platforms do not possess an effective strategy to control the quality of the crowdsourced tasks. For example, to protect themselves against cases of results manipulation, cheating or extracting sensitive information from these systems. Moreover, there are other factors that have a considerable weight in the quality of the crowdsourcing process. A user-friendly and intuitive UI can substantially make a difference and increase the quality of the results obtained, as well as improve the learnability of the task, making it easier to execute in the long run [46]. In addition, the usability of the crowdsourcing platform also plays a significant role in obtaining more accurate results from the tasks, meaning that more skilfully elaborated tasks also generate better output quality.

It is worth mention that there might be cases where the tasks do not possess an objective answer or, as referred earlier, the worker has malicious intention. This can be reflected in the task's results, which may incur in the risk of becoming manipulated or incorrect. One strategy to reduce uncertainty is by increasing the number of humans that perform the task and reach the final result by the majority of responses [24]. This way, it is possible to have more feasible and reliable control over the results, since the users with malevolent intent become outnumbered. Furthermore, algorithms have already been created with the objective of minimizing the impact of malicious workers by assigning them a reputation [28], meaning that the less reputation a worker has, the less of an impact he will have on the on the task and vice-versa.

2.4 Dataset Construction

The importance of having reliable and trustworthy datasets is central in many areas of computer science, including ML and software testing. In the case of ML, datasets are the basis for training ML models. The robustness of the models substantially relies on the quality of the data they are trained on. It is safe to affirm that the more sound and dependable the dataset is, the more accurate and valid the model will be when predicting results on unseen data. The same can be applied to software testing, more precisely, when employing static analysis tools. The validity of the tool can increase (or decrease) after being tested on a dataset, since these, by being composed of several test cases, allow a thorough examination of the tools, serving as a benchmark and indicating whether it needs further refinement or is accurately discovering vulnerabilities in the code.

In the following subsection, we present the Software Assurance Reference Dataset (SARD) dataset construction method in the context of the PHP programming language. The code examples extracted from SARD constitute the foundation of our work, as they are used to populate our database, being subsequently evaluated by the users and tool.

2.4.1 SARD Use case

SARD is a repository that encompasses thousands of test cases of different programming languages, consisting in code snippets that may contain vulnerabilities. It is managed by the National Institute of Technologies (NIST), more precisely, it is an output of a project known as the Software Assurance Metrics and Tool Evaluation (SAMATE). This project aims to enhance and improve software assurance by developing several materials, standards, and methodologies. SAMATE's key goal is to evaluate and measure the effectiveness of various tools and techniques used in software assurance.

In this document, we focus on web vulnerabilities associated with the PHP programming language. Given the several programming languages covered by SARD, we will particularly emphasize on its PHP test case generator. It is a subset within SARD that is used in our work, so it becomes important to have a solid grasp of how the code snippets were generated.

The method for generating such test cases of SARD is thoroughly explained in the scientific article ”Large Scale Generation of Complex and Faulty PHP Test Cases” [41]. The reason for the authors to develop this test case generator was self-evident: at the moment they started the project, there were only 15 known vulnerable PHP test cases. It was clear that more were needed to properly evaluate the effectiveness of PHP analyzers. Moreover, for the specific situation of PHP, most of the tests conducted on web applications were focused on the dynamic aspect, that is, the static analysis counterpart did not receive enough attention.

Their solution was designed to target three different types of audience: the SATs developers, so they could test their products and (potentially) enhance them; secondly, so that users of these tools could run the trial versions, making it possible to ensure that they were buying quality software; and lastly, security researches could use the test suite to check how their tools were performing, also allowing them to compare with the tools that already existed. According to the authors, the testing and further comparison between different static analyzers would result in a more competitive environment, which subsequently would end up in an improvement of the tools.

A major concern was to create a dataset that was both versatile and would contain most common weaknesses. In order to accomplish this, the authors decided to follow the Open Web Application Security Project (OWASP) Top 10 most common weaknesses [8]. To achieve versatility, the PHP vulnerability test suite generator, had a modular and reusable design, which allowed the addition of custom rules and classes of weaknesses. The interface was user-friendly – the user only had to insert the desired command with the available options to generate the entire test suite.

In order to represent common vulnerabilities, the generator is composed of 3 XML files: Input.xml, which stores functions that receive some input; Filtering.xml, which lists the methods used for filtering the inputs; and finally Sink.xml, which contains the source code where the vulnerabilities can be exploited. When the user executes the generator, thousands of test cases are created in a short time period. These test cases are stored in categorized directories, being sorted by their vulnerability category and CWE identifier (which consists in a unique identifier assigned to each entry of the Common Weakness Enumeration (CWE), a list of common software and hardware weakness types).

There are several benefits that result from the employment of this tool, such as the increase of productivity, better coverage, less probability of human error, amongst others. Despite these positives aspects aforementioned, it is important to be aware that these are synthetic test cases that only target one weakness each and do not correspond to real world case scenarios. It is also worth noting that not all vulnerability categories present in OWASP Top 10 could be addressed. Nonetheless, in the overall, the PHP test case generator is able to produce thousands of test cases with various complexities and vulnerabilities in a short time, making it a significant resource for enhancing the reliability and robustness of static analysis tools for PHP.

2.5 Related Work

It is our objective to use static analysis tools as an automatic way to label a piece of PHP code and since these can produce a high rate of false positives and false negatives, we intend to supplement their results with feedback from the crowd, more precisely, with the classifications that workers provide about the code slices. Our expectation is that this approach leads to a more accurate and reliable classification.

In this section, we will mainly focus on static analysis tools that are based on machine learning mechanisms, more specifically, deep learning (DL) algorithms that allow the detection of software vulnerabilities, although some examples of SATs based on traditional approaches will also be presented. We will also center our study on the datasets used for their model training phase.

2.5.1 Traditional Static Analysis

Traditional static code analysis is based on a set of pre-defined rules elaborated by experts that comprise features or patterns associated with vulnerability classes. These rules can be seen as a set of guidelines that the tool uses to examine the code. For instance, a rule could be a guideline that requires all inputs to be sanitized, thereby requiring the tool to notify whenever this rule is violated.

As state above, these rules are elaborated by cybersecurity experts or developers who possess the knowledge to identify common pitfalls and weaknesses that a certain programming language may have. This constitutes one of the reasons why these rules are hard to create, since the amount of knowledge about both the vulnerabilities and programming language must be quite extensive in order to produce a quality tool.

Despite the amount of knowledge a developer or cybersecurity professional may have about programming languages and its concepts, the task of elaborating a traditional static analysis tool does not become less error-prone or complex, which causes this technique to have limitations. In fact, the formulation of these rules leads to exhaustive and cumbersome work that often results in a high number of false negatives and false positives. Having into account that this pre-defined specifications are designed to catch specific patterns, they may incur in the situation of not being able to identify or detect more complicated vulnerabilities that do not match the established principles. This is the core reason why SATs often produce erroneous reports, since some weaknesses are hidden and subsequently missed or in fact the weakness may not be a genuine vulnerability, giving rise to a false positive.

Moreover, it is relevant to mention that these rules must be constantly enhanced and updated. The cybersecurity field is an ever-changing realm. New cyberattacks are frequently conducted which often give birth to new vulnerabilities and attack vectors. Thus, it becomes easy for such rules and guidelines to become obsolete if not updated systematically.

Given that our research's focus is centered on addressing issues in web vulnerabilities present in the PHP programming language, the traditional static analysis tools discussed below have been chosen due to their ability to examine and analyze PHP codebases.

Pixy [29] is a static analysis tool developed in Java with the purpose of detecting web vulnerabilities in the PHP programming language. It makes use of taint analysis, which consists of examining the user's input in the application by tracing the flow of the data until a sensitive sink is reached. This process helps to find a possible vulnerability. To enhance the accuracy of its taint analysis, Pixy integrates supplementary alias analysis (when two different variable names might refer to the same location in memory) and literal analysis (the literal values in the code). In addition, Pixy offers a robust vulnerability detection, thus maintaining a manageable number of false positives. In its experimental phase, Pixy was able to identify 15 previously unknown vulnerabilities and could reconstruct 36 known ones. Nonetheless, Pixy has some drawbacks. It only supports two vulnerability classes, SQL injection and Cross-Site Scripting. Moreover, it does not support PHP's object oriented features and does not scan files.

Another tool worth mentioning is WAP [36]. Similarly to Pixy, WAP also employs taint analysis and combines it with data mining. The combination of these two techniques, although not common, enables WAP to initially detect potential vulnerabilities via taint analysis and subsequently, by resorting to data mining, it is able to predict the likelihood of the detected vulnerabilities being false positives. Furthermore, WAP operates by analyzing the source code for input validation weaknesses. When detected, its code corrector proceeds to adjust the code so it is no longer vulnerable. Generally, the corrections employed by this tool consist in invoking a function that sanitizes the data before it arrives to the sensitive sink, ensuring the application's intended execution remains unchanged. It is capable of supporting 8 different types of vulnerability classes, which are the ones addressed in this document.

KAVe [39] is another SAT able to detect web vulnerabilities in the PHP programming language. The approach KAVe employs is quite effective: it creates several "code property graphs" which in turn provide information about the behaviour and flow of the code. Then, these graphs are converted into a knowledge graph. The latter consists in a structure that represents the application's code. In order to optimize the analysis, KAVe prunes the knowledge graph, obtaining only the essential details, enhancing the vulnerability discovery process. Finally, similarly to the aforementioned tools, it also resorts to tainted analysis: a multi-agent system capable of analyzing the knowledge graph making use of the mentioned technique, thus being able to discover vulnerabilities in PHP programs. KAVe has been able to find 169 vulnerabilities in 12 open-source web applications with an accuracy of 98.81%. It is able to detect two types of vulnerabilities: XSS and SQLI.

2.5.2 Machine Learning based Static Analysis

Over the last years, the field of ML has been growing at a fast pace. Through these advances and improvements, the previously stated intense manual labour that researches had to go through to create such rules and principles (in order to identify weaknesses in code) has been decreasing, since it is no longer required to heavily define the features of the ML models that will detect the vulnerabilities.

ML techniques, unlike traditional static analysis, do not rely heavily on pre-defined rules and specifications set by the developers. Instead, ML makes possible a more autonomous learning of patterns in data associated with vulnerabilities, which is a process achieved through the employment of algorithms that can identify and learn from these patterns. This data often comes in the form of large datasets that contain both secure and insecure code snippets. The datasets are thus used by the ML model during the training phase. Rather than being the expert creating the rules, machine learning, together with the data that is fed from the dataset, will take care of the task. This not only simplifies the process but also removes the need for developers to constantly create or update rules.

Deep Learning (DL), a subset of machine learning, takes it a step further. This methodology has also been growing rapidly and is even more automated. It works by using multiple layers of artificial neural networks, which consist in a structure of interconnected nodes. This structure is responsible for pattern recognition and raw data interpretation, including text, image and, evidently, code snippets. In contrast with traditional methods and ML approaches, DL is capable of completely eliminate the need for manual work carried out by experts, as this methodology should be capable of extracting more meaningful features from data when compared to ML.

It is important to mention that for both methodologies, DL and ML, the quality and amount of data they are trained on are crucial factors for the accuracy and performance of the models. Higher quality data ensures that the model is learning the patterns correctly, while a greater amount of data guarantees that more scenarios can be tackled. Together, these factors directly influence the model's capability in detecting vulnerabilities in an accurate way.

Tools like Vuldeepecker [33] and μVuldeepecker [48] follow the approach of Deep Learning described above. Vuldeepecker uses a specific type of neural network that allows it to understand short and long term dependencies in code, which becomes crucial in identifying vulnerabilities. This can be exaplained by the fact that, in order to understand the context of a certain line of code, it becomes useful to have information from both preceding and following lines. This way, Vuldeepecker is thus able to recognise patterns in source code associated with software weaknesses, more specifically, and based on what was already mentioned, it employs code gadgets to represent programs which are subsequently transformed so the neural network can process them.

Vuldeepcker was able to identify 4 vulnerabilities that were not listed in the National Vulnerability Database (NVD) [16] (which is considered an industry standard dataset). However, this tool possesses some downsides. For instance, it can only run C/C++ programs. Moreover, the dataset where it was trained only comprises a limited amount of vulnerabilities.

Based on what was mentioned about Vuldeepecker, a new (and enhanced) version of it was developed: the µVuldeepecker. Similarly to Vuldeepcker, this tool is also based on neural networks. Amongst several improvements when compared to its prior version, one of the most significant is the ability of not only classifying the code as vulnerable or not (as Vuldeepecker was capable of) but it also pinpoints the type of vulnerability.

µVuldeepecker was evaluated in a dataset containing thousands of code gadgets from different programs (including both vulnerable and not vulnerable code gadgets that comprised 40 types of vulnerabilities). According to the authors, the results obtained were promising, as they were able to conclude that µVuldeepecker was, in the overall, better than the Vuldeepecker. Nevertheless, this tool still possesses some drawbacks. While it is able to detect the type of vulnerability, it is not able to output the exact location of the weakness. Moreover, and similarly to its older version, it can only be applied to C/C++ programs. In addition, in terms of performance, it is not as performant as the Vuldeepecker, as it was concluded that it took slightly more time to execute.

Table 2.1 shows a set of state-of-the-art works in Deep Learning solutions along with the programming language, the vulnerability classes they focused on, and the datasets used. As it is possible to observe, the majority of the programming languages tackled by researchers are C and C++. Moreover, the most used datasets for either training or evaluation of the developed frameworks are either SARD or National Vulnerability Database (NVD). The vulnerabilities assessed do not comprise, in their vast majority, web vulnerabilities such as SQL Injection and Cross Site Scripting. The major part of these is related to buffer errors, like buffer overflows. This shows that there is a lack of study and research when it comes to web vulnerabilities. Furthermore, the widely datasets aforementioned possess a number of issues and limitations.

As referred before, SARD is currently one of the most used datasets and, undoubtedly, a starting point in vulnerability detection research, containing an extensive amount of code snippets (for several languages, including secure and unsafe samples). However, in its vast majority, it is composed of synthetic code examples. Although these synthetic examples can be useful in creating controlled testing environments for testing static analysis tools and training machine learning models, they lack the capacity of fully capture the complexity of realistic scenarios. This constitutes a drawback when training the models, since the diversity and authenticity of these code snippets will impose limitations on their capacity for vulnerability detection when it comes to real-world applications.

Table 2.1: A set of State-Of-The-Art Publications of Deep Learning based Vulnerability Detection solutions

Publication	**Programming Language**	**Vulnerabilities Assessed**	**Dataset/Collection**
Automated vulnerability detection in source code using deep representation learning [40]	C/C++	Buffer Overflow, Improper Restriction of Operations within the Bounds of a Memory Buffer, NULL Pointer Dereference, Use of Pointer Subtraction to Determine Size, Improper Input Validation, Use of Uninitialized Variable, Buffer Access with Incorrect Value, amongst others	SATE IV Juliet, Debian Linux distribution, Public Git repositories
Vuldeepecker: A deep learning-based system for vulnerability detection [33]	C/C++	Buffer error, Resource Management Error	NVD, SARD
Sysevr: A framework for using deep learning to detect software vulnerabilities [32]	C/C++	Vulnerabilities, such as Buffer errors. CWE [1] IDs available at https://github.com/SySeVR/SySeVR	NVD, SARD
Combining graph-based learning with automated data collection for code vulnerability detection [44]	C, Java, Php and Swift	Top five to top thirty vulnerabilities in CWE 2019	SARD, NVD and Github Projects
Software vulnerability discovery via learning multi-domain knowledge bases [35]	C/C++	Buffer error, Numeric Error	SARD, Real-world Open source projects (FFmpeg, LibTIFF, LibPNG, Pidgin, VLC media player)

Another important consideration in using SARD is the potential wrongly labeled data. This problem has been proved in certain instances where the code snippets were inaccurately classified. In the context of our research, while we have made use of code snippets from the SARD dataset, we mitigate these inconsistencies. Our approach makes use of the output from static analysis tools along with the feedback from the crowd. It is expected, with our novel approach, to enhance and improve the classifications of such labels, providing more accurate and reliable results, particularly tackling the detection of web vulnerabilities in PHP, an area that has not been the main focus of many research efforts to date. Future researches in the topic can thus use and derive our dataset.

Concerning the NVD, although this latter comprises real-world code excerpts, it does not contain the code examples directly. To obtain them, the user must access the link provided by the NVD's platform in order to access the vulnerable code from the application repository. In addition, these links often incur the need of downloading the application and, in most cases, it does not pinpoint the exact location of the vulnerable code. This process of searching for the intended vulnerability becomes tedious and time-consuming.

Regarding PHP code snippets presented in these two datasets, the same issues above-stated can be applied. Taking into account that PHP is a widely known and used programming language for web development, the lack of resources in terms of security flaws and vulnerabilities can be daunting. This represents the motivation of our proposed solution: the construction of a novel dataset that comprises unique, correctly classified PHP code excerpts that will allow the training and evaluation of Deep Learning models in order to detect software vulnerabilities. It is worth mentioning that, to the best of our knowledge, none of the datasets aforementioned were constructed using the combination of static analysis tools and crowdsourcing methodologies.

Chapter 3

BugSpotting

In this Chapter, we present BugSpotting, a platform with the objective of creating a dataset composed of well classified PHP code snippets. First, in Section 3.1, an overview of BugSpotting is presented, where we provide the motivation for its development as well as its main purpose. Next, in Section 3.2 a set of requisites is described, detailing the solution's requirements that are important. Then, in Section 3.3, we explain the system modeling with three diagrams, each one displaying the solution in more detail. Finally, in Section 3.4, we present the system data model, which characterizes the logical representation of the solution's database.

3.1 Overview

The quality of ML models is fundamentally related to the datasets they are trained on. In fact, without high-quality datasets, even the most efficient ML algorithms will fail to perform regardless of the context they are applied, since these models are as effective as the datasets they learn from. To mitigate these challenges, we propose a platform for creating robust datasets consisting of PHP code snippets – BugSpotting. We focus on PHP due to its frequent use in web development.

This platform employs two different methodologies for classifying code snippets: first, we resort to the use of static analysis tools, which, as already referred in this document, provide deterministic but potentially erroneous classifications (due to the amount of false positives and false negatives they produce). Second, we make use of crowdsourcing, which contributes with human judgment. Although the latter can be valuable, due to its nature, it can lead to biased classifications. Furthermore, each of these classifiers – tools and humans – possess an associated weight or reputation that influences the final classification of the snippet. A specially tailored algorithm integrates the inputs from the aforementioned classifiers to arrive at a consensus label for each code snippet (which can be vulnerable or not vulnerable to a certain vulnerability class). The end goal is a comprehensive dataset of well-classified PHP code snippets, from which other datasets can be derived.

Although the employment of static analysis tools is a good practice in the context of software development, by itself it is not enough to guarantee highly accurate labels when it comes to the detection of vulnerabilities in code [20], hence the need for resorting to crowdsourcing. The integration with crowdsourcing will mitigate this issue and thus, provide more reliable and accurate classifications. Our expectation is that this hybrid approach will yield a more sound system for vulnerability detection.

3.2 Requisits of BugSpotting

Developing a robust and efficient platform is not an easy task. The system should be able to meet certain requisits in order to function correctly. A core component is the User Interface (UI). The UI is not just about the visual components of the website – it is also about allowing a simple and intuitive experience. Moreover, the User Experience (UX) also plays a central role – it enables the user to interact with the system effectively and understand its functionalities. In the following subsections, we emphasize the core requisits for the proposed solution, giving special importance to factors like the UI, UX, accuracy, extensability and specificity.

Easy to Use

Both UI and UX are important when it comes to having an intuitive and uncomplicated system. On one hand, the UI will require, in the context of the proposed solution, a website visually appealing and easy to navigate. On the other hand, when it comes to UX, the website must be fast, accessible, and functional. A good UI/UX is a mandatory aspect for obtaining the desired results since part of the BugSpotting success is based on the feedback obtained from the users. This means that not only providing the code snippets to the user but also the classification processes must be straightforward.

Precision and Sensitivity

One of the main goals of the proposed solution is to have precision in the results. We accomplish this by combining the use of static analysis tools with the feedback given by the users. The website's backend server will take care of the business logic: an algorithm should be implemented to take into account the users' and tools' classifications. Then, with the associated weight of each one, a consensus is reached so a final label can be given to a code snippet. Our expectation is that the precision of the system will increase with the number of classifications obtained: even in the case of the tool generating, for instance, a false positive or a false negative, the multiple opinions from the crowd (and their weight) will contribute to diminishing the divergence, allowing the system to end up with a much more accurate and reliable label.

Extensible

The BugSpotting will be built to facilitate the inclusion of novel features and functionalities throughout time. For instance, at its initial phase, it will not be operating with many static analysis tools so a more obvious decision about the code snippet's label can be formulated. The number of users will increase from time to time, but in the beginning, it is natural that only a few will exist in the system. Nonetheless, we intend that our solution can accommodate growth. This means that in a later development phase or after the core implementation is concluded, it should be relatively uncomplicated to add a new tool for future classifications or even make changes to some components or include other features. We intend to implement a solution that can be compatible with new adjustments and thus be easily extensible.

Real and Synthetic Code Snippets Support

While real-world code snippets offer use-cases similar to authentic applications, the use of synthetic examples aims to provide a more controlled dataset. In other words, the employment of synthetic code snippets can bring a lot of advantages, for instance, each code snippet can be created to represent a specific vulnerability class, which in turn ensures that the ML models that will use this dataset are tested against properly defined examples. Furthermore, this type of test cases avoid incurring in noisy data. The reason is straightforward: real-world code samples often include comments or functionalities that are not relevant for the vulnerability being tested. On the other hand, synthetic examples are cleaner and allow a more focused evaluation. Nonetheless, real-world code snippets can still provide sound results, since this are extracted from real software and thus, are more representative of the complexities that ML models may encounter when detecting vulnerabilities. BugSpotting is designed to accommodate both synthetic and real-world code snippets.

3.3 System Modelling

Figure 3.1 depicts the Data Flow Diagram (DFD) level 0 (or context diagram) of the proposed solution. It contains a high-level overview of how the system is modeled in the form of multiple processes represented by a single double circle, along with its relationships with external actors.

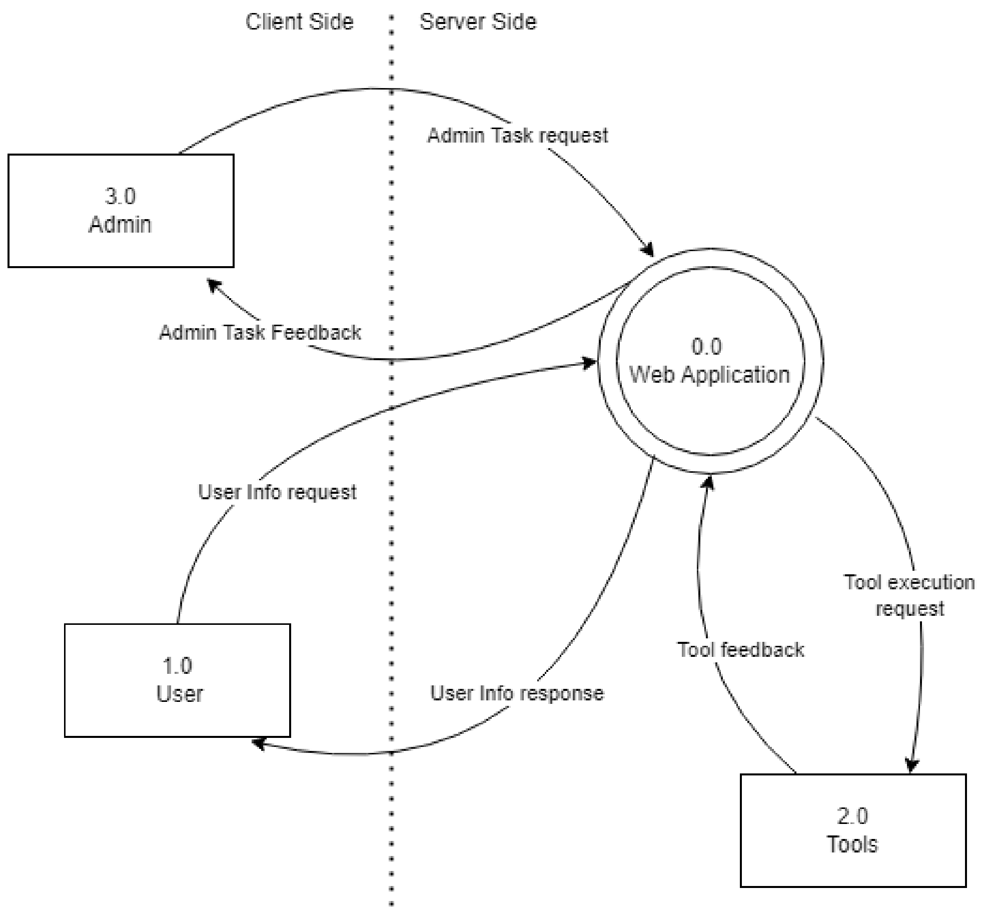


Figure 3.1: Representation of the system's context diagram

The system can be partitioned into two parts: the client side and the server side. The server side consists in the web application process, which contains a database where the code snippets are stored (which is not represented in the context diagram). The web application accommodates the tools that will classify these code examples as well as the business logic that will be responsible for determining the associated label based on the feedback from the crowd and tools.

The client side is composed of three entities: administrator, users, and tools. Although the tools will be included in the web application, in the system modeling they are viewed and act as actors. The administrator of the system has the purpose of manually populating the system's database with code snippets. Besides this, he also possesses privileges for adding and removing users and tools to/from the system. The users of the system are responsible for classifying the code snippets. They have a reputation associated which will be translated in a weight when the algorithm reaches a consensus about the final label of the code snippet: a more experienced user who has a history of correct labels will have a higher weight than a user that lacks accuracy in labeling. Finally, the tools have similar functionality to users, i.e., they analyze code snippets (inserted in the database by the administrator) and output their labeling. As we previously mentioned, the results of the tools are deterministic and may be incorrect. Therefore, as users, they will have a weight associated based on their results' precision.

As represented in Figure 3.1, the interactions between the actors and the system are straightforward: the administrator will send requests to the system to either authenticate himself, populate the database with new code excerpts, add/remove users/tools or execute any other privileged task. The users will be able to perform authentication, classify code snippets (by identifying possible vulnerabilities and their location in the code samples presented to them), track their own history of classifications and consult their reputation. Finally, the tools interact with the system by giving feedback about a code snippet the same way users do – by providing the class of vulnerability and the corresponding lines where it occurs.

With the notions given above, one is able to dig deeper into the system and understand some more complex data flows that it contains. Figure 3.2 depicts the system in more detail by representing the DFD level 1. As it is noticeable, the main difference of this one when compared to the context diagram is the decomposition of the single multi-process into the main processes that together form the system. These are represented by the circles in the figure. Furthermore, the inter-process relationships are also visible, being more explicit and meaningful. Although there are other actions that can be executed by the system, since it is a data flow diagram level 1, only the most important ones are illustrated. As we can observe, process 4.0 is responsible for the insertion of code snippets in the system's database (illustrated by entity 9.0), which is an action triggered by the system's administrator. The administrator also has the privilege of adding or removing classifiers (i.e., tools and users), represented by process 5.0.

Moreover, there is the process of filtering the slices according to a set of criteria (process 6.0), before sending them to be classified by the user or the tool. After the classifications from both actors are obtained, they are merged – this is performed by the Slice classification process (process 7.0). This classification process, when finished, will result in the update of the classified code snippets' label in the system's database. A more detailed explanation of how these processes work altogether along with the actors is illustrated in Figure 3.3.

Figure 3.3 represents the flowchart of the system, depicting the whole process of labeling a code snippet: from being stored in the database until a final label is assigned to it. The data flow is described

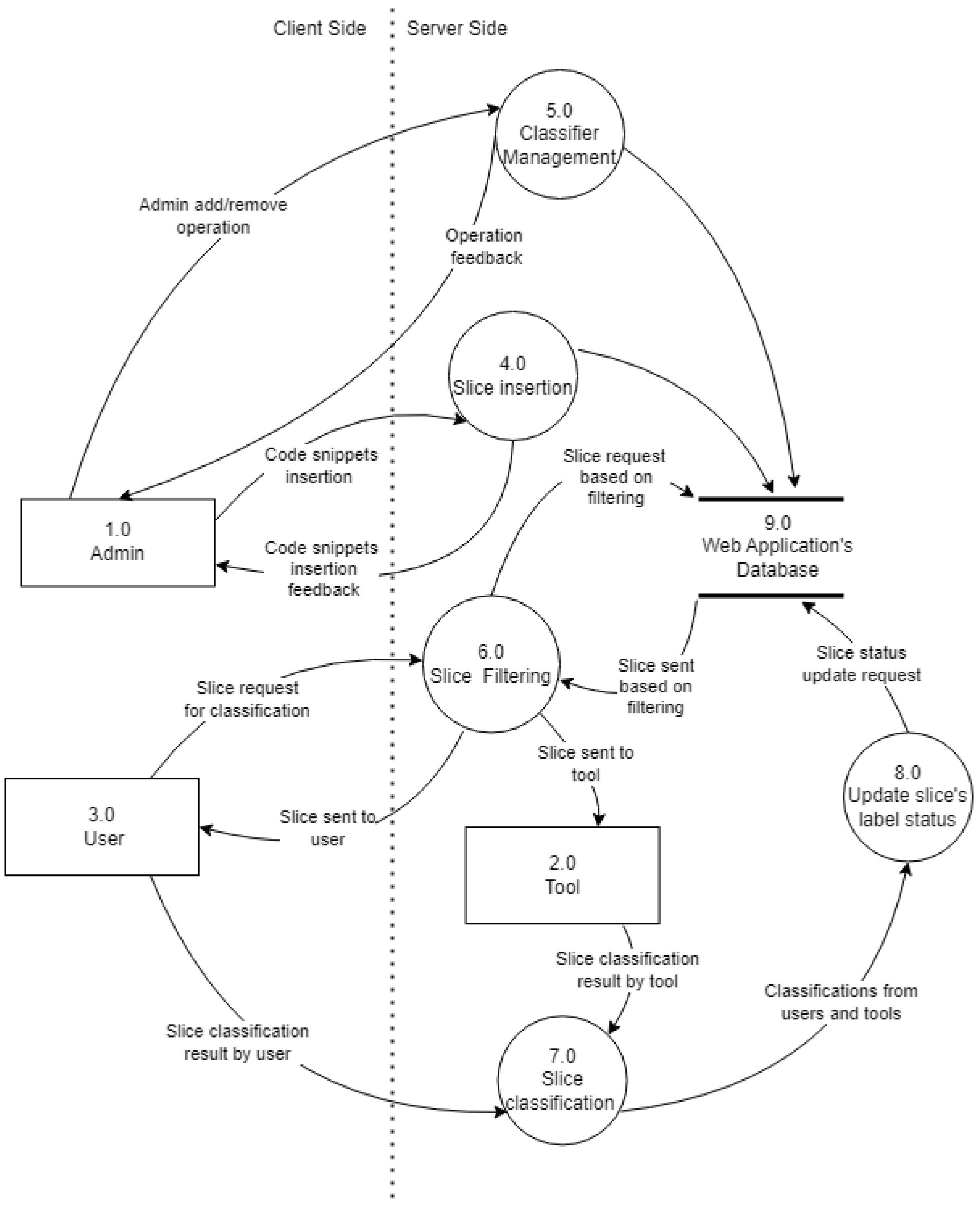

Client Side
Server Side
5.0 Classifier Management
Admin add/remove operation
Operation feedback
4.0 Slice insertion
Code snippets insertion
Code snippets insertion feedback
Slice request based on filtering
9.0 Web Application's Database
1.0 Admin
6.0 Slice Filtering
Slice request for classification
Slice sent based on filtering
Slice status update request
Slice sent to tool
3.0 User
Slice sent to user
8.0 Update slice's label status
2.0 Tool
Slice classification result by tool
Classifications from users and tools
Slice classification result by user
7.0 Slice classification

Figure 3.2: Representation of the system's data flow diagram level 1

as follows: initially, the application's database is empty. The administrator of the system will populate the database with the code snippets to be classified. When all the code slices in the database are already classified, there is the possibility to add more. Next, the user will ask for a slice to be classified according to one of the vulnerability classes addressed by the solution. For instance, a user could ask the system for slices to be classified for the SQL injection vulnerability class, regardless they contain or not such vulnerability. Before the slice is handed over to the user, two events occur: a filtering process and a tool classification.

Regarding the first event, a filtering of code slices is employed based on some criteria: first, the algorithm will choose a slice based on the fact of whether it has been classified yet or not. Slices that have not yet been classified have priority. If all the slices of the database have already been classified, then more criteria are applied: first, a check on the number of classifications will be conducted in order to prioritize and choose those slices that have fewer classifications. The tie-breaker criterion in the case

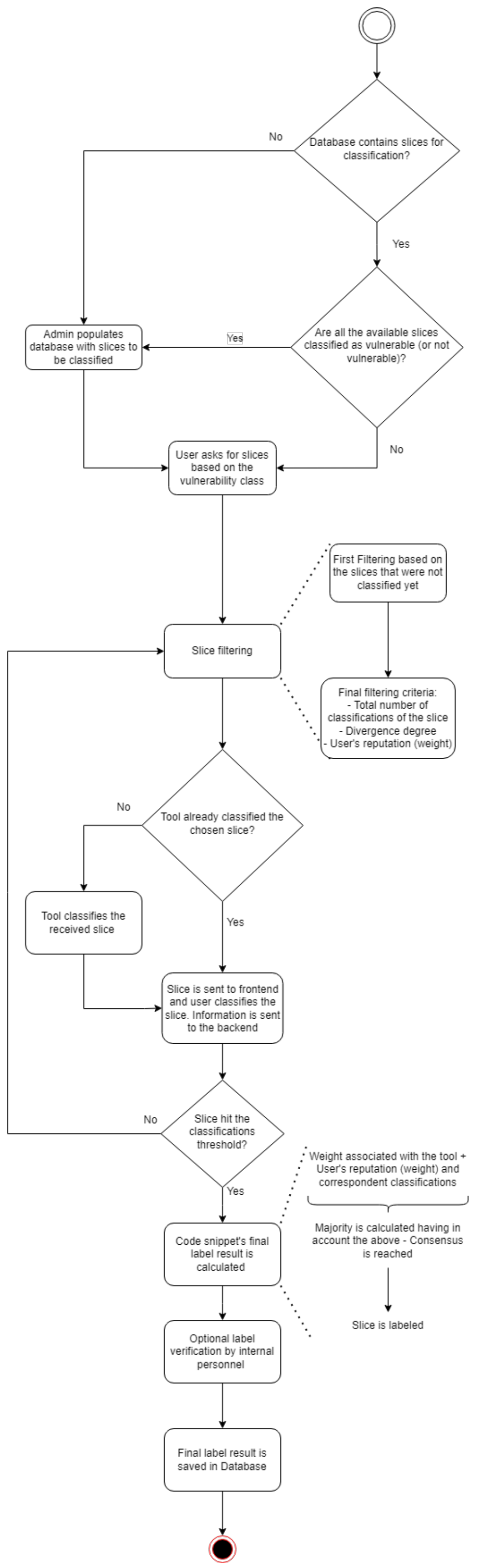

Database contains slices for classification?
No
Yes
Are all the available slices classified as vulnerable (or not vulnerable)?
Yes
No
Admin populates database with slices to be classified
User asks for slices based on the vulnerability class
First Filtering based on the slices that were not classified yet
Slice filtering
Final filtering criteria:
- Total number of classifications of the slice
- Divergence degree
- User's reputation (weight)
Tool already classified the chosen slice?
No
Yes
Tool classifies the received slice
Slice is sent to frontend and user classifies the slice. Information is sent to the backend
Slice hit the classifications threshold?
No
Yes
Weight associated with the tool + User's reputation (weight) and correspondent classifications
Majority is calculated having in account the above - Consensus is reached
Slice is labeled
Code snippet's final label result is calculated
Optional label verification by internal personnel
Final label result is saved in Database

Figure 3.3: Flowchart of the system

that all slices have the same number of classifications is their divergence degree. The divergence degree of a slice represents the level of disagreement among the classifications made for the slice towards a vulnerability class. This means that the higher the degree, the more disagreement in the classifications (either by the tools or the users). Hence, code examples that have a higher divergence degree have priority and will be sent to users that have a higher reputation (revealing more expertise). This way, the degree can be lowered and it is easier for a labeling consensus to be achieved.

After the slice filtering event and getting the slice for classification, the tool event takes place (the second event). Right before the code slice is sent to the user, a verification is executed: if the code slice chosen has not yet been evaluated by the tools, it is immediately evaluated by them. Here, a pseudo-label is assigned to the code snippet, in which the slice is labeled as vulnerable (or not) towards the initial vulnerability class the user chose to classify. Otherwise, in the case of the slice has already been classified by the tools, it is just sent to the user.

After the user submits the classification, a verification is performed to determine whether the slice must continue being classified or not. For this, the system is (previously) configured with a threshold value that represents the total number of classifications (by users and tools). If this threshold is reached, then the algorithm will calculate the final label of the code snippet towards a certain vulnerability class based on all the classifications made by tools and users and their respective weights. In the exceptional case of a tie after the algorithm calculations, a label verification by software security experts is conducted in order to reach a final decision.

3.4 System Data Model

A data model can be understood as the logical representation of a system's database, which includes not only the entities that are part of it but also the relationships they have with each other. Figure 3.4 depicts the proposed solution's data model, more specifically, in the form of an Entity-Relationship (ER) model. This type of data model maps the entities that form the database, including their attributes that form their domains, cardinality, and relations with one another.

The ER model is composed of nine entities. An explanation of what they represent and how they relate to each other can be understood below.

Classifier, Tool Classifier, and Manual Classifier

The *Classifier* entity can represent either a *Tool Classifier*, which is a static analysis tool that will execute a static code analysis on the code snippets; or a *Manual Classifier* that is a user in the system that will classify a code excerpt. It is worth mentioning that the *Classifier* entity itself can not exist without these two (children) entities. They establish a one-to-one relationship where the obligation belongs to the children entities and thus, records of both children will be presented in the *Classifier* entity. More specifically, a record will only exist in the *Classifier* entity if a record of one of the child entities is created.

VulnerabilityClass

The *VulnerabilityClass* entity represents the vulnerability classes considered in the solution. Besides its primary key, the field *vuln_class_name* stands for the name of the vulnerability class itself, and the field *vuln_class_acronym* stands for its acronym, for example, SQLi.

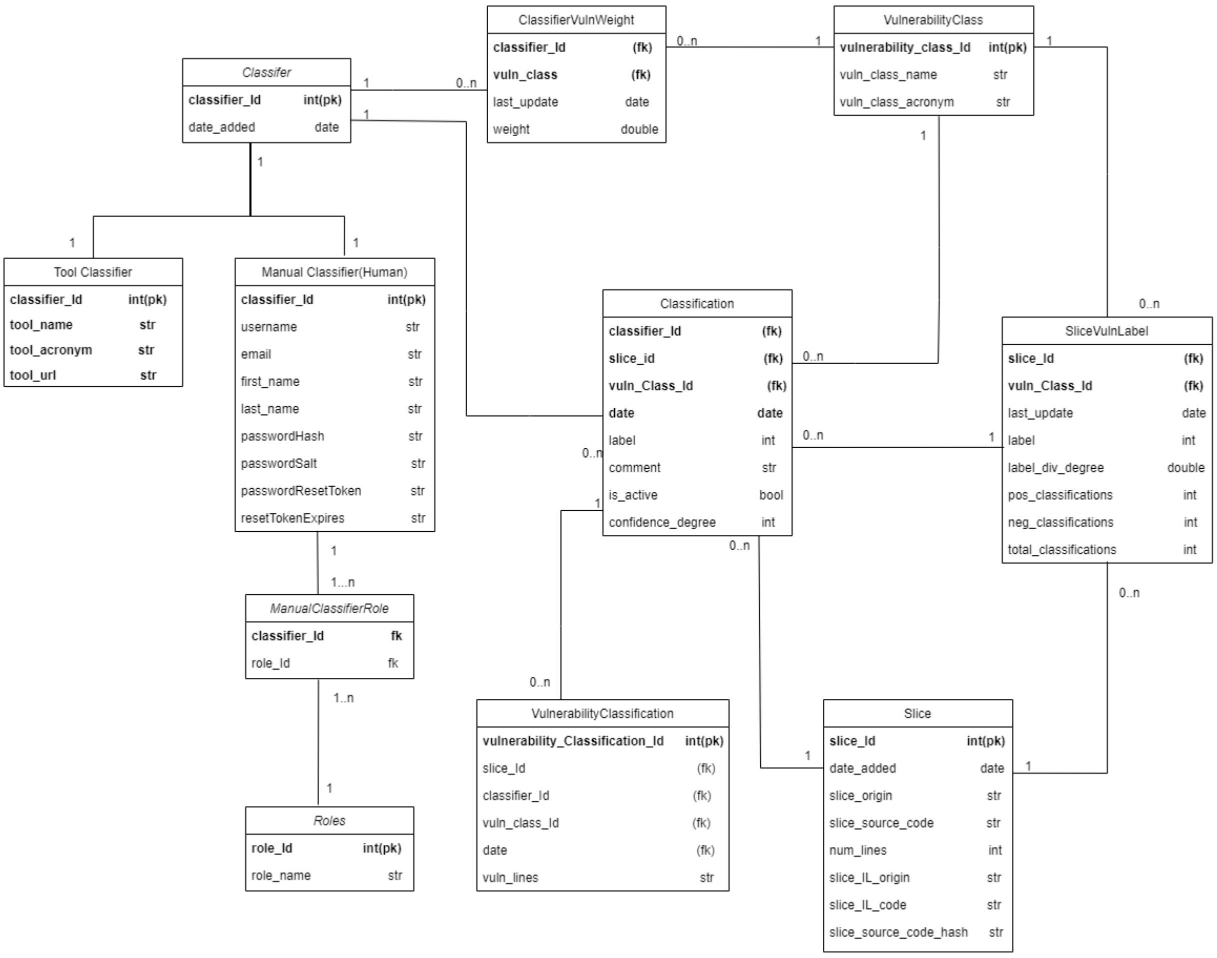


Figure 3.4: Representation of the system's data model

ClassifierVulnWeight

With the notions given above about the *Classifier* and *VulnerabilityClass* entities, one is able to understand the *ClassVulnWeight* entity. On one hand, a classifier can be associated with many vulnerability classes. For instance, a user (*Manual Classifier*) can classify a code snippet and assign it to more than one vulnerability class. On the other hand, a vulnerability class can be associated with many classifiers. For example, a SQL injection can be a possible vulnerability class associated with more than one tool classification and be shared among various users' classifications as well. From this many-to-many relationship, the *ClassifierVulnWeight* is created. Besides keeping a record of these mappings, it also contains a relevant field: the *weight*. The latter represents the impact that each user and tool will have in the final labeling of the code snippet towards a certain vulnerability class. For instance, a user may have higher weight (which translates into more expertise) classifying SQL injections than Cross-Site Scripting vulnerabilities. It is worth noting that this weight has a default initial value of fifty points (which is assigned the moment the user classifies a slice to a certain vulnerability class) and is updated according to the user's correct classifications. If, after consensus was reached, the classification (label) provided by the user matches the label reached by the consensus algorithm, the user's weight for that specific vulnerability class is increased five points. Otherwise, it is decreased five points.

Slice

It represents a PHP code snippet. It establishes a one-to-many relationship with the *Classification* entity, meaning that a slice can have multiple classifications, either from the users or the tools. Furthermore, the Slice entity has some relevant fields, namely, the *slice_origin*, *slice_source_code* and *num_lines* that represent the path (or location) of the slice in the application, the slice's code itself and the total number of lines of the slice, respectively.

SliceVulnLabel

The *SliceVulnLabel* derives from a many-to-many relationship between the *VulnerabilityClass* and *Slice* entities. A vulnerability class can be associated with many or none slices and the opposite also can also occur, i.e., a slice can be associated with many or none vulnerability class. In other words, a slice with no vulnerability class means that it is not vulnerable to any vulnerability class assessed in the solution. It can also happen, for instance, that a slice associated with the vulnerability classes of RFI (Remote File Inclusion) and SQLi (SQL Injection) is vulnerable to both of these. The *SliceVulnLabel* table, besides keeping a record of the mappings between the two aforementioned entities, is also an aggregation table. More precisely, some of its fields get updated from time to time (being those the ones that do not belong to the composite primary key in bold). This update depends on the classification records that will be added to the classifications Table. As already mentioned, the *label* is an updatable field that represents the label of a certain slice with a certain vulnerability class. The value of a label is represented by an integer that may have two values: non-vulnerable (0) or vulnerable (1). Next, there is the divergence degree field that measures the level of disagreement between classifiers towards a slice with a certain label. The last three attributes of the table represent the number of classifications of each type and the total number of classifications a given slice already had. The field *pos_classifications* is incremented when a positive classification occurs or, in other words, when a classifier assigns the label of vulnerable to a slice with a given vulnerability class. The same logic applies to *neg_classifications*, but in this case, the increment occurs when the code excerpt is classified as not vulnerable to a certain vulnerability class. Finally, the last field is a sum of the two aforementioned, representing the total number of classifications of that specific slice for a particular vulnerability class. It is also related to the threshold, which is a variable in the system that indicates whether or not the slice with the vulnerability class in question will or will not keep being classified – if the threshold is reached, no more classifications can occur for that slice with the specified vulnerability class.

Classification

The *Classification* entity represents a classification done by a classifier. It has a composed primary key, which consists of the classifier, slice, vulnerability class and the date when it was added. There is a date in the primary key to ensure the possibility of a classifier to classify the same slice more than once, and hence, without it, there would be no uniqueness in the primary key. It could also happen that, for the same classifier and slice, the same class of vulnerability could appear more than once in the code excerpt. These records would be persisted in the *VulnerabilityClassification* entity. Besides the composed primary key, other important fields are worth mentioning. The entity possesses a *label* field, which consists of the evaluation or feedback obtained by the user or the tool and, unlike the *label* field in the *SliceVulnLabel* entity, this one is not updatable. Besides this, the field *comment* allows the user to provide a commentary on the code excerpt. The *is_active* field indicates to the system if the slice is still being classified or has reached the threshold and it no longer evaluated. Finally, the *confidence_degree* field indicates how confident the user is when he classified the slice as vulnerable or not vulnerable.

VulnerabilityClassification

The *VulnerabilityClassification* entity keeps all the records of the classifications provided by the users and the tools. It contains the slice, the classifier, the vulnerability class, the date of classification, and the lines where the user or tool found the vulnerability. It establishes a one-to-many relationship with the *Classification* table. This is due to the fact that a slice under classification can have one or more vulnerabilities of the same class and, consequently, in different lines within the code snippet. On the other hand, the VulnerabilityClassification can only be associated with one *Classification* entity record since each classification is unique (a condition granted by the *date* field).

Chapter 4

Implementation

In this chapter, a thorough explanation of the BugSpotting's implementation details will be presented. Throughout Sections 4.1 to 4.3, we will introduce the technologies used to build the solution, its architecture and core functionalities and features. Lastly, in Section 4.4 we present the deployment process.

4.1 Technologies Used

When building new software applications, a central theme that must be addressed is the technologies that should be employed in their construction. They can influence the development speed, the system's performance and even the security of the final product. In addition, it is also important to consider the aspects of both visual design and user experience, since these factors will set the user's first impression about the application.

When referring to web applications, it is common to break them in two main components: the frontend and the backend. While the former can be understood as the visual part of an application, the latter, on the contrary, is not visible to the user and is responsible for the processing and storing data, as well as communicating with external systems. In the context of our solution, for prototyping the pages and sections of the website, Figma [3] was the chosen; React [9] was selected as the frontend library and .NET [6] was the employed backend framework. Lastly, we opted to use the MySQL [7] relational database for data management.

The rest of this section is divided between the frontend and backend technologies used in BugSpotting where we provide a more in-depth description of both, which can be understood below.

4.1.1 Frontend

In simple terms, the frontend of a web application is also a synonym of client-side, or, more precisely, the part which the user interacts with. Thus, it comprises everything the client can see, ranging from the design, buttons, navigation menus, etc. In order to better grasp and conceptualize the term (and subsequently, the technologies used in the implementation of the solution), it is essential for one to be familiarized with the foundational technologies it is built upon. These are discussed next.

Frontend Foundations: HTML, CSS and JavaScript

HTML [4] stands for ”HyperText Markup Language”. It is considered a standard markup language in the context of web development, more specifically, for the creation of web pages. HTML’s main responsibility is the structuring of the page. It accomplishes the latter by using predefined tags and elements which tell the browser how to properly display the content. For instance, to make text bold, this can be simply achieved by wrapping the desired text between the `<b>` and `<\b>` tags. With more recent versions of HTML, it is also possible to add meaning (semantic tags) to the page, allowing a better content organization and accessibility. These semantic tags, such as `<footer>` and `<article>`, provide context to both developers and browsers, ensuring that content is organized and easier to interpret.

Although HTML can be viewed as the ”skeleton” of web pages, on its own, it can not accomplish more than structuring the content included in them. In order to make web pages visually appealing, one must employ Cascading Style Sheets (CSS) [2]. CSS consists in a language that allows the developer to specify how web pages are presented to the user in terms of graphical design, i.e., how they are styled. This language is defined by rules and goes hand-in-hand with HTML: the developer creates the rules, which can be seen as a set of styles (for example, defining the background color of the page) that will be applied to the aforementioned HTML elements.

Despite the last two technologies already allow to have an enhanced UI, the content of the page is still static. In other words, there is no behaviour when a user interacts with it (for instance, when a button is clicked). In order to have interactive web pages, another technology must come into play: JavaScript [5]. With JavaScript, developers can create dynamic content that responds to user inputs, fetch and display new data, show animated 2D/3D graphics, amongst many other features and functionalities. It makes web applications feel responsive and alive.

The three technologies above stated — HTML, CSS, and JavaScript — constitute the foundation to web development and as such, we employ them in our implementation.

JavaScript Library: React

With the aforementioned concepts understood, one can now understand what is React and how it works. React.js or ReactJS, often referred as just React, is a JavaScript library developed by Facebook. The goal of React is to build user interfaces as Single Page Applications (SPAs). SPAs dynamically update and render content without the need of a full page reload by the browser. On the contrary, when it comes to traditional websites or Multi Page Applications (MPAs), the browser typically reloads the entire page when content changes.

A fundamental trait of this library is its component-based architecture. When building web applications with React, these are based on reusable components that can have their own state and logic. This strategy of ”divide and conquer” not only helps to maintain a clean and understandable code base but also allows developers to build more complex UIs (from small building blocks) and avoid code duplication. Another positive feature that React offers is its fast UI re-renders and updates. Traditional web applications often stall or become less performant because whenever a small change occurs, the browser needs to reload the entire page, making them less efficient. More precisely, this happens because the

Document Object Model (DOM) [1] needs to be updated or rebuilt whenever a change occurs, which often incurs in the page lagging or slowing down. React is built in way that it only updates or re-renders the parts that change, making applications faster and smoother.

Furthermore, React also offers the possibility to use a special syntax – JavaScript Syntax Extension (JSX). This extension allows writing JavaScript code that resembles HTML, providing developers a more intuitive and readable UI code structure. This JSX code is then transpiled into real JavaScript. When the JavaScript runs, React converts it into real HTML (more specifically, DOM nodes) so the browser can interpret the transformed code. React is very flexible, allowing a seamless integration with other frontend libraries as well as backend systems. This becomes a major advantage, since developers can incorporate React with already existing projects and even new ones, regardless of their technologies.

In the context of our solution, it becomes evident that this library was chosen due to its efficiency, flexibility and ease in the creation of user-friendly interfaces, making the development process faster and cleaner.

Figma: creating the design

While having robust and efficient frontend technologies like the ones mentioned above is crucial to build functional and performant web applications, to maximize their potential, it is also of major importance having a visually and user-friendly interface. An application that is appealing to the eye not only makes users more engaged but further refines the overall experience, making it more enjoyable.

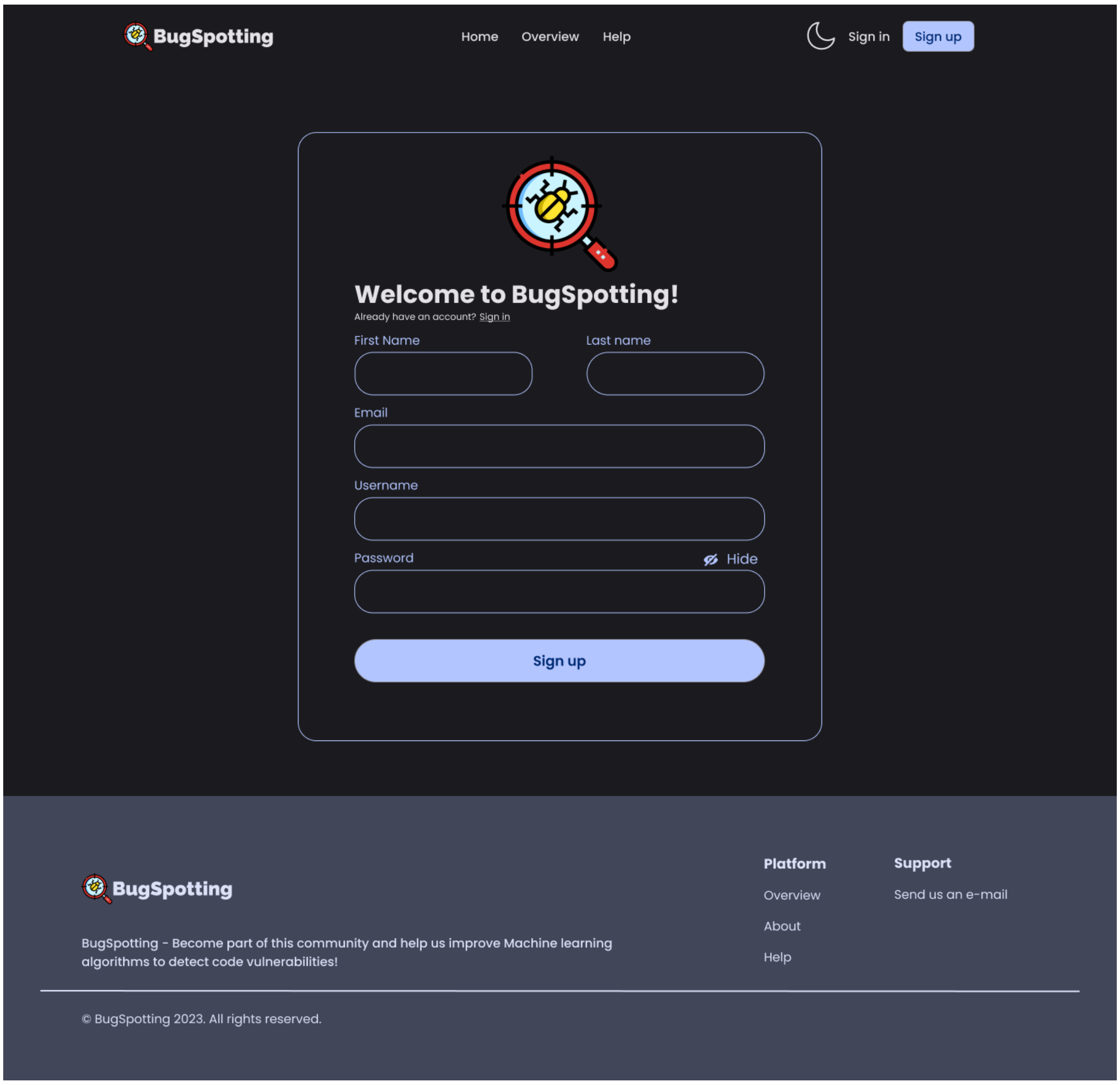
Figma is a tool that has this exact purpose: to design and test interfaces for applications and other digital products. It comes as no surprise that is vastly used by UI/UX designers. Due to its ease of use, this tool is able to easily convey the ideas of the designers into the digital form in a rapid and effortless way. Moreover, by allowing the creation of prototypes, the developers of the web application's frontend are fully aware of how the pages should look like and behave.

The same approach was used in the initial development phase of our web application: we resorted to Figma to create the web pages so a more smoother and more organized implementation could be followed. By adopting this methodology, not only the implementation process was indeed faster as it was less error prone. Figure 4.1 illustrates one of the pages of the web application. The example shown corresponds to the "Sign Up" page created with Figma, responsible for the user registration in the platform.

4.1.2 Backend

The backend of a web application refers to the server-side, or more precisely, the part of the application that runs "under the hood". This means that the users do not interact directly with the backend, but it does not imply that it is of less importance. It is the backend that manages and stores user data in the database, executes the business logic and communicates with other systems, such as the frontend, so that data can be exchanged. The database is also a crucial part of the backend, as it serves of repository for

[1]The DOM can be seen as a blueprint or map of a web page (a structure containing all HTML elements)



Welcome to BugSpotting!

Already have an account? Sign in

BugSpotting - Become part of this community and help us improve Machine learning algorithms to detect code vulnerabilities!

© BugSpotting 2023. All rights reserved.

Figure 4.1: Sign up page of the web application

any information the application may need, ranging from sensitive data such as user credentials to less important content, for instance, application settings or content logs.

Below, we give an explanation of the technologies used in our web application's backend.

.NET Framework

Before delving into the perks and details of the .NET framework, it is important to, in a first instance, understand what is a framework. In software development, a framework can be seen as a structure of a project or system. It is used to help with the implementation of the software and provides a template that allows the developers to modify and add additional features to create a complex solution. It worth noting a fundamental aspect of a framework: while in traditional programming the code invokes functionalities and controls a library, with a framework, it is the other way around. This is called Inversion of Control (IoC) and it is the actual framework that controls the developer's code. In addition, a framework may also contain libraries.

Having into account the above stated, the .NET framework is a cross-platform software development framework for building and running applications on Windows, Linux and macOS operating systems. This framework is composed of two central parts: the Common Language Runtime (CLR) and the Class Library. The first one is the execution engine. More specifically, it provides a runtime environment where .NET programs can be executed and offers a variety of services such as thread management, garbage collection, exception handling, amongst others. It is the CLR that also handles the compilation to machine code in order to run the .NET programs. The Class Library is responsible for providing a set of APIs and built-in classes for the various development tasks. The functionalities are quite extensive and include, for example, reading and writing to files, connecting to databases, security features, amongst many more. These enrich the .NET framework, making it a very powerful development tool.

Furthermore, the .NET framework supports multiple languages such as C#, VB.NET and F#. This constitutes a major advantage since the developer can choose the language that he is more comfortable with. Finally, the framework is supported by Microsoft which ensures that .NET always stays updated and receives the latest bug fixes and improvements, thereby offering organizations and developers trustworthiness.

Backend's Database System: MySQL

MySQL is a widely used, open-source Relational Database Management System (RDBMS). In other words, it is a piece of software that enables users to persist, organize and retrieve data. It is considered a very powerful and flexible database, being able to support almost any amount of data, be run on cloud platforms or on-premises servers and is known for its superior performance. Moreover, MySQL is able to provide a higher level of security by using encrypted passwords. In addition, since it is an open-source project, it lets a large community to contribute and enhance its efficiency.

4.1.3 An enhanced Web Application

The combination of these technologies greatly benefits the implementation of BugSpotting. The user's interface built with React, aligned with the design crafted with Figma, provides a dynamic and visually appealing experience. Additionally, due to React's ease of use and flexibility, it enables a seamless integration with the .NET framework. The latter enforces a greater level of performance and security with always up-to-date software. Finally, the MySQL database, known for its speed and capacity to handle large amounts of information, provides the backbone for the application's data storage.

4.2 BugSpotting Architecture

In this section, we present the architecture. To better understand the design of the solution's architecture, it is important to be aware of the design principles that constitute its foundation.

When designing the architecture of a web application, one should ideally strive for separation and modularity of the components that together form the software. The concept that lies underneath is called Separation of Concerns (SoC). SoC enforces the developer to think in terms of modules or individual units that have minimum overlapping between them. This methodology allows a decoupling of these modules, ensuring that each one of them has a distinct responsibility. This subsequently translates into a more controlled environment, where changes in some part of the application do not interfere with others. In addition, it concedes a more readable and cleaner code, where the organization and structure are made easy to understand.

This approach has been vastly used in enterprise applications and has been conceptualized into the three-tier architecture. A description of the latter can be understood below.

4.2.1 Three-Tier Architecture

The three-tier architecture, widely used in the software industry, divides an application into three logical layers: presentation layer, business logic layer and the data access layer. Figure 4.2 depicts the scheme of this approach. The presentation layer, also referred as the user interface layer, is the tier where the

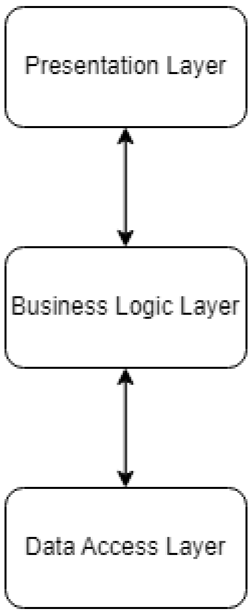


Figure 4.2: Three-tier architecture

user interacts with the application. The main goal of this layer is not only displaying but also collecting information from the user. Often, this top layer runs on a web browser, being developed with the technologies already mentioned before, such as HTML, CSS and JavaScript. The middle tier is the business logic layer, also known as the domain logic layer. This layer constitutes the heart of the application and it is where all the application's core processing happens. It is enriched with the business rules, data transformations and other logic that is required for the application to work as expected. The data that is used in this layer often comes from the top tier (presentation layer). Lastly, the bottom tier, commonly called the data access layer, focuses on interacting with the underlying storage mechanism. It provides an interface to persist and retrieve data, ensuring that it is served in a consistent manner.

Having into account the three-tier architecture, one can dive deeper into the existent patterns that are responsible for the interaction between the user interface and the application's logic and data. In the next section, we describe two of the most employed patterns: the Model-View-Controller (MVC) and Model-View-ViewModel (MVVM) and relate them to our solution, which, as it will be discussed, employs an hybrid approach.

4.2.2 MVC pattern

The MVC pattern [30], belonging to the MV* family, has the primary goal of developing user interfaces by dividing the programming logic into three distinct parts: the Model, the View, and the Controller. The first one fulfills the tasks associated with the application's data (such as managing its state), logic and rules of the application. The View is essentially the UI of the application and displays the data that the Model contains. The Controller has the responsibility of handling the events from the View and conveys them to the Model.

The interaction between the three components follows a predefined flow. It starts by the occurrence of an event (a user interaction in the View). The View will then forward the user event (usually some input) to the Controller, which interprets it and communicates it to the Model. As previously explained, it is within the model that the data is processed in the context of the business logic. After the processing occurs, the state may be updated and the View is notified, being synchronized with the Model and displaying the changes based on the Model's data.

Within an application that follows the MVC pattern, multiple Views and Controllers can exist. There are some cases where a specific controller is bound to one or multiple Views. It is this association that introduces MVC's main challenge: the tight coupling between the View and the Controller. Despite the fact that this pattern has been a pioneer in the industry by going into accordance with the principle of separation of concerns, it is not perfect. Modifications to the user interface frequently lead to changes in the controller (for example, when implementing a new feature). It becomes more severe when we consider a large scale application, which may contain hundreds of views and controllers, since it becomes an exhausting and overwhelming task to make even a minor modification to adapt the components. In addition, there may be Views that are very similar in terms of functionalities, resulting in a lot of repeated logic in multiple controllers.

To mitigate this tight coupling, the MVVM pattern introduced the ViewModel, which can be comprehended in more detail below.

4.2.3 MVVM pattern

Similarly to the MVC pattern, the MVVM pattern [30] also belongs to the MV* family and its primary focus is to develop user interfaces. In the case of MVVM, it allows the development of more complex user interactions and is capable of handling more UI updates. Its three components are the Model, the View and the ViewModel.

The View is responsible for displaying the data that is fed from the ViewModel and is agnostic to the business logic. Besides this, and similarly to MVC, it also issues commands and events that the ViewModel will handle. The Model, although it may possess some business logic, it is not aware of how to display it. It is mainly used for accessing the underlying data source, retrieving the raw data to the ViewModel. It is the ViewModel that is the core of this pattern. The ViewModel acts as a bridge between the View and the Model - this is a key difference between the two patterns. The heavy work is executed in this component, more precisely, the majority of the business rules according to the business logic and the processing of the data. After the processing and transformation of the data, it then can be sent to the View which will simply display it. It is important to mention that the ViewModel is not aware of the

View in the sense that it does not directly manipulate UI components. It simply provides properties or commands to which the View binds to.

The flow and interaction between the three components in the MVVM pattern is based on data binding. The View binds to the ViewModel, which means that any changes that occur in the ViewModel, are automatically reflected in the View. When the users interact with the View, the latter simply delegates the task of handling these events or commands to the ViewModel, which in return, may update the Model if needed. The same happens when the Model changes: it notifies the ViewModel which consequently updates the View.

The major advantage of the ViewModel is the decoupling between the UI (Views) and the business logic. But it comes with a drawback: the ViewModel may become too complex since it will handle all the business logic and data processing as well as the the communication between the Model and the View.

4.2.4 BugSpotting: a hybrid approach

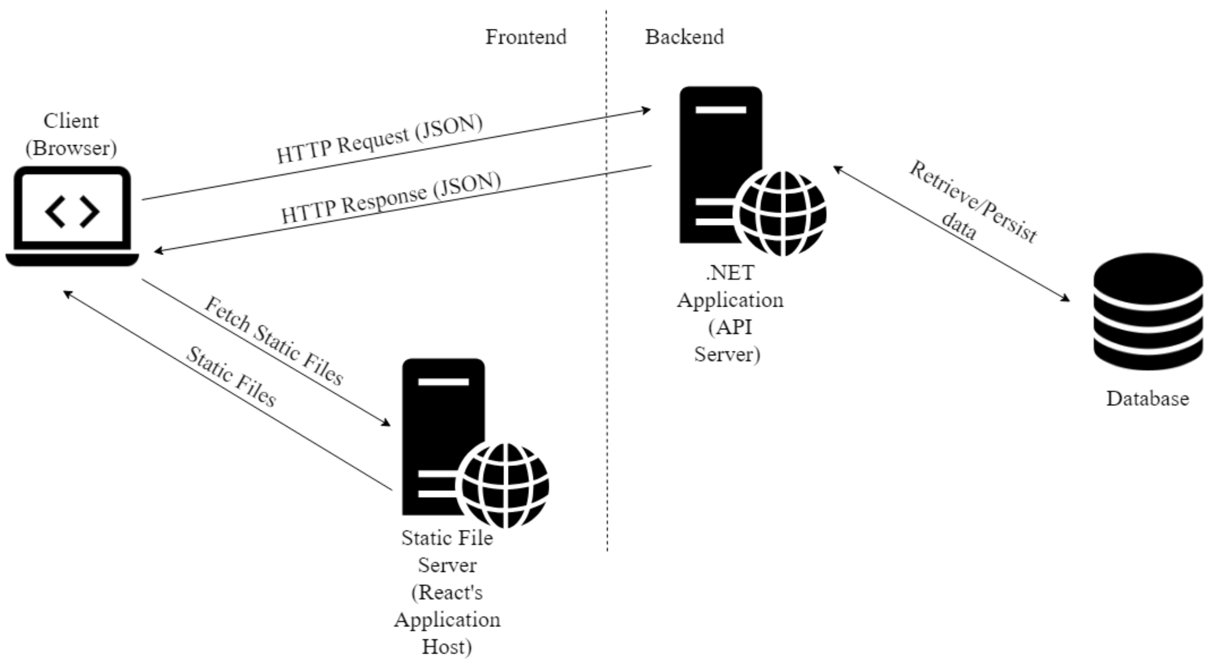


Figure 4.3: BugSpotting's System Architecture

There is no "one-size fits all" when it comes to the architecture of a web application and thus, the aforementioned patterns do not have to be followed strictly. Deviations can and do exist, being driven by the specifics of the application and technologies that support it. This is the scenario of our web application, BugSpotting, which combines both MVC and MVVM patterns.

At the frontend, since the application was developed with React, it operates like a View-ViewModel (VVM). React allows to have complex interfaces and logic supporting them. As it was mentioned in the previous technologies section, React is based on components. The key takeway is that these are not only meant to display the data they receive (like the View in the MV* patterns) but they can also manage state and contain UI logic. These two characteristics are what makes the components align with the VVM part of the MVVM pattern. For clarity, consider the example of displaying the user's classifications in our frontend application. Not only the data is being displayed, but also the user can interact with the UI and instruct it to sort the classifications by, for example, newest entries. This implies that there must be logic running underneath so they can be ordered accordingly to the user's intentions.

At the backend, the system's architecture is more like a Model-Controller (MC). Using the .NET framework technology, the backend does not follow the default MVC architecture because it is not concerned with the visual aspect of the application (the Views), being merely an API. BugSpotting's single controller at the backend processes the requests, and conveys the data to the Model, in which resides the core logic, business rules and where data management and transformation occurs. Besides this, the Model also interacts with the database to either persist or retrieve data. Before this data is returned to the frontend, it is firstly transformed into JSON format so the it be easily interpreted. Evidently, the task of rendering the Views is delegated to the React frontend.

For a more lucid perspective of the interactions and flow of data within the system, Figure 4.3 depicts the main communications between the frontend, backend and the database, emphasizing the aforementioned VVM and MC patterns. As it is possible to observe, there is a dotted line that separates the frontend and backend. The frontend is composed of the client (the browser) and the React's application host, while the backend encompasses the .NET application and the database. The browser represents the end-user interface where the React application will run. It is where the components from the application will be rendered, managing both what the user sees (the View) and the underlying logic (the View-Model). The client firstly communicates with the Static File Server, which consists in a Node.js. It issues a request in order to obtain the static files provided by this host. These static files consist in all the necessary JavaScript, CSS, images, amongst other files that are required to run the React application in the browser. It is worth mentioning that while the server provides the needed files for the frontend, in no way it possesses or relates to the application-specific logic, as the VVM pattern is only employed in the browser, where the application is executed.

After obtaining the desired bundle, the browser is now able to communicate with the backend. The .NET application is where the MC pattern becomes evident. It is typically hosted in a server like Kestrel and its main function is to respond to the incoming requests issued by the client. It can be seen as an API with a set of endpoints that are translated in action methods that the controller implements. The controller will handle the requests and let the model apply the implemented business rules and data processing. The backend also communicates with the database, persisting and retrieving data. The fact that the information returned from the backend is in JSON format and does not deal with any sort information visual display reinforces the MC pattern. Lastly the database is the storage mechanism that is capable of retrieving and storing the application's data based on the requests from the backend application.

In the next section, we go in depth about the capabilities and main functions of the BugSpotting web application.

4.3 Core Functionalities

The heart of BugSpotting's capabilities resides in its backend. Although the frontend is responsible for user interaction and presentation of the data in a visually appealing and intuitive manner, the fundamental actions and operations executed are a mirror of the capabilities offered by the backend. In fact, the essential business logic and data processing tasks lie within the backend. To better clarify what is meant by core functionalities, consider the difference between the frontend's capability of letting a user choose between a light and a dark mode. Although it is indeed a useful feature, it is not core to the system. On the other hand, the backend's crucial task of reaching a consensus about the label of a code snippet is central to the platform's primary objective.

In this section, we will explain in detail the vital functionalities offered by the backend. In other words, we will describe the BugSpotting's API (endpoints), how do they work and some of the main flows of the system that grant the user a satisfactory experience when using the platform.

4.3.1 BugSpotting's API

Before delving into the API's core functionalities per se, it is important to understand the principles and guidelines in the design of the BugSpotting's API as well as some global features.

REST Architectural Style

Representational State Transfer (REST) is an architectural style used in the design of web services. It is based on a set of principles that an API to be considered RESTful has to comply:

1. **Uniform Interface** – This principle ensures that the communication between the client and the server is standardized and consistent. The four main guidelines that define a uniform interface can be understood below.

    (a) **Resource Identification with URIs**: Every resource (for example, a user profile on the BugSpotting website) should have a consistent Uniform Resource Identifier (URI) associated with it. When considering the user profile example, if a user wants to modify some information about the account, the accessed URI is unique (enforced by the user identifier) and remains unchanged.

    (b) **Manipulation of Resources through Representations**: when a resource is requested by the client (such as the information about the user profile), it is provided with a representation of that resource rather than the resource itself. In BugSpotting's case, this representation is a JSON object containing the necessary details to populate the profile page. If the client wants to alter some field, an updated version of that JSON (the representation) is sent back to the server. This helps decoupling the way data is stored in the server from how it is presented to the client.

    (c) **Self-descriptive Messages**: Each message exchanged between the client and the server has enough information to process it correctly. Often, this takes the form of defining the HTTP method (GET, POST, PUT, etc) and the Multipurpose Internet Mail Extensions (MIME) type (such as "application/json").

    (d) **Hypermedia as the Engine of Application State (HATEOAS)**: When the server sends data to the client, it may include links (hypermedia) to related resources or possible actions. The client, by clicking on these, is in fact changing the state of the application. For instance, in the case of BugSpotting, when the client accesses the classifications page, by clicking on the

”Show Code” button, a modal pop-up window is displayed, allowing the visualization of the code snippet classified (hence, changing the state of the application).

2. **Statelessness** – Each request to the server (executed by the client) must be self-contained and have enough information to be processed, meaning that the server should not rely on any previous requests to execute future ones. For example, when accessing the user’s reputation page, the request made to the server is independent. In other words, the server does not ”recall” any other previous client interactions. This helps improving reliability, since even if one request fails, it does not have an impact on further requests because they are not dependent on each other.

3. **Layered System** – This allows the client to communicate with authorized intermediaries between the client layer (BugSpotting’s frontend) and the server layer (BugSpotting’s backend) while still receiving responses from the server. Moreover, in the case of BugSpotting, a proxy server (the intermediary system) redirects the requests for the correct application (both the frontend and backend applications run on a Virtual Machine (VM)). Moreover, it also provides a security layer, since it adds HTTPS to the communication (this ensures the data exchanged is encrypted and its integrity is protected).

4. **Client-Server Architecture** – The client and the server are two separate entities that act independently. As it has already been explained, BugSpotting has a frontend application (responsible for the user interface and user experience) and a backend application (responsible for the core logic and data treatment). This separation enables a more modular approach, providing more room for future changes (for instance, a modification in the backend’s database server technology). This ensures our platform is adaptable and resilient to modifications.

5. **Cacheability** – Caching follows a simple principle: ”If it has been accessed in the past, it is likely to be accessed again in the future”. This technique involves storing temporary copies of data so that the future requests are served faster. While it is not mandatory, caching helps when it comes to enhance performance and reduce server loading. BugSpotting does not employ any caching mechanisms (besides the ones offered by the browser).

BugSpotting’s API was developed having into account the aforementioned principles, making it a RESTful API. This architectural style promotes simplicity and maintainability, while ensuring an efficient communication between the frontend and the backend.

Global API Features

In this section, we present the global features of the BugSpotting’s API. These include the authentication and authorization mechanisms as well as the error handling strategy.

Authentication and Authorization BugSpotting’s authentication and authorization mechanisms rely on the use of JSON Web Tokens (JWTs). JSON Web Token is an open industry standard used to share information between two entities, usually a client (such as the BugSpotting’s application frontend) and a server (like the BugSpotting’s application backend). As the name implies, this information is exchanged in JSON format.

JWTs offer a stateless authentication: the server does not need to store any session data. All the information needed for authentication or authorization lies within the token. The data in the token is divided into three main modules:

1. **Header**, which consists of in two parts: algorithm that is being used to create a Message Authentication Code (MAC), which is a string that is used to ensure the integrity and to verify the origin of the message, and the type of token (which, in most cases, defaults to JWT).

2. **Payload**, that contains the user claims. Claims are statements about an entity (often, the user). In the case of BugSpotting, two important fields are the user identifier and the role.

3. **Signature**, which is a crucial part of the token. To create the MAC, both the header and payload are encoded (often using base64URL encoding) and with the specified algorithm (as already mentioned, from the token's header) plus a secret key, both encoded parts are hashed (in our solution, we used HMAC with SHA-256 hashing algorithm). This MAC is used to prevent any data tampering and, since BugSpotting's backend uses a secret key, to ensure that the sender of the JWT is who it says it is.

Once the client is authenticated, the server generates the token, which has an expiration date (in the case of BugSpotting, it is 7 days from the moment it was created). The client then stores this token in the browser's local storage, and appends it in the header of any subsequent request to the server. This is often placed under the "Authorization" field in the header, prefixed with "Bearer". Upon receiving a request with a JWT, the server interprets the token: it takes the predefined hashing algorithm, the secret key (known only by the server) and creates a MAC of both the header and payload of the received token. It then compares this generated MAC with the one from the token's MAC: if both match, the token is valid; otherwise, it is considered invalid, and responds with an appropriate error message. In addition, the token validity can also depend on other factors, such as its expiration date.

Another important point to mention is the fact that when using JWTs, it is essential to transfer them through a secure channel (with the use of HTTPS with TLS) so man-in-the-middle and eavesdropping attacks can be avoided. The reason lies in the fact that encoding is not a security measure, it is just a form of representing the data (since anyone can decode the token and see its content). To better illustrate the process described, Figure 4.4 shows the flow of interaction between the client and the server as well as the creation of the JWT. The example has into consideration that a user was already registered in the platform but is not in possession of any JWT token.

As it is possible to observe, the process starts with the client sending the authentication data to the server (in this case, and as in BugSpotting's platform, the email and password). Then, the server will validate this information against the database and if correct, generates the JWT and sends it back to the client (browser). The client will store it in its local storage. The further requests will have the token appended in the HTTP header (under "Authorization" followed with the prefix "Bearer"). The server will then check the authenticity of the token. It is worth mention that while the token may be in fact valid, it may not allow the client to access certain resources (for example, in the case of the route accessed be restricted to the administrator of the server and not a regular user). From now on, the token is exchanged in future interactions, being always validated by the server.

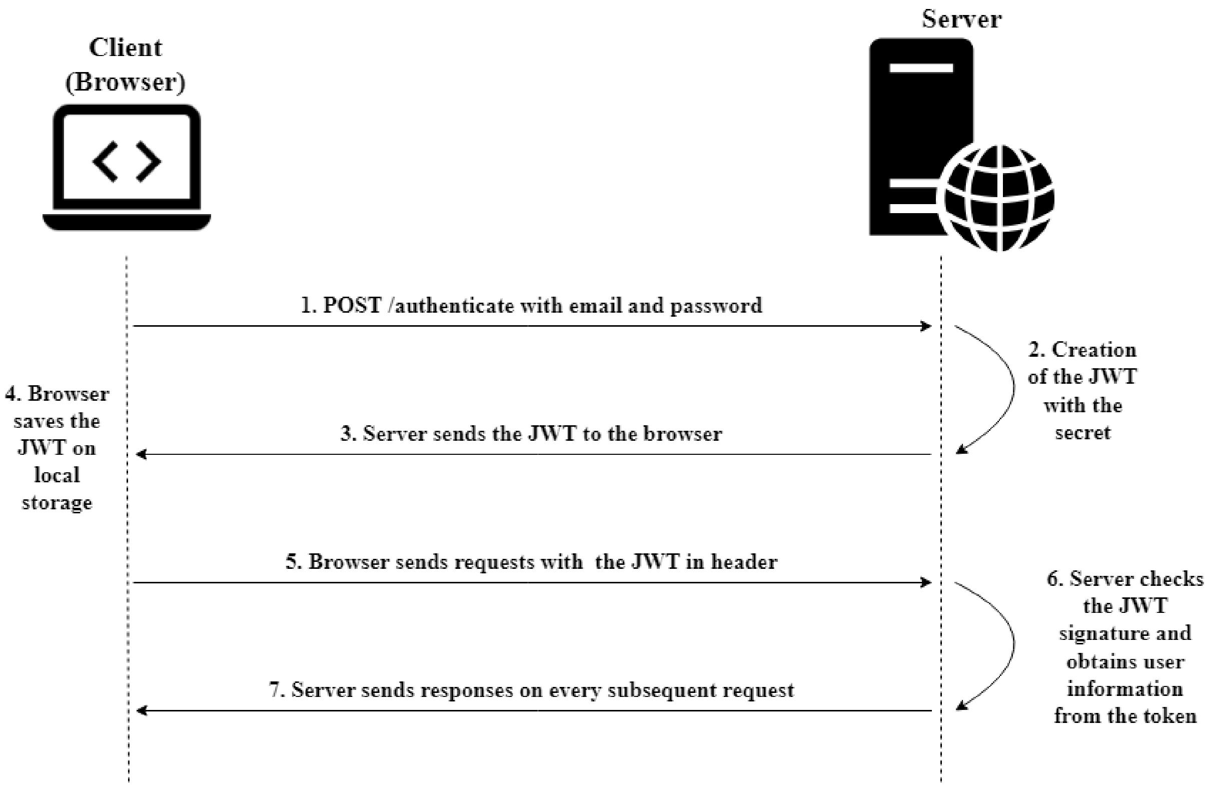

Client (Browser)
Server
1. POST /authenticate with email and password
2. Creation of the JWT with the secret
3. Server sends the JWT to the browser
4. Browser saves the JWT on local storage
5. Browser sends requests with the JWT in header
6. Server checks the JWT signature and obtains user information from the token
7. Server sends responses on every subsequent request

Figure 4.4: JWT authentication process

Error Handling Efficient error handling is crucial for guaranteeing both stability and a user-friendly experience. By being able to anticipate possible errors and handle them in a graceful manner, a system can instruct the user to provide the necessary corrections in order to continue using it correctly.

When it comes to BugSpotting, our strategy for error handling resides primarily in the backend. The server has an internal middleware mechanism whose purpose is to intercept any exceptions that may occur during the processing of the incoming request. This approach ensures that the application, instead of crashing or responding without clear error messages, provides an informative feedback about the issue. For instance, if the user tries to register himself with an already chosen username, the middleware would catch this exception thrown by the application and respond with an error code of 409 (conflict) with a convenient message description of ”The username provided is already in use”. The possible HTTP status codes per endpoint are listed in Table 4.1 along with their descriptions. Moreover, the error messages from the web application often follow a standardized format, consisting in an ”ErrorCode” and an ”ErrorMessage” fields, This standardized format allows an easier interpretation of the information by the frontend. In the explanation of each endpoint, we will evidence the specific scenarios where these can occur in more detail.

Endpoints

Bugspotting’s API was built to be robust, secure and intuitive. It is based on the principles of REST’s architectural style, implements the JWT authentication scheme and ensures a graceful error handling. With the foundation set, one is now able to understand each specific endpoint, their functionalities and

Table 4.1: Server response codes for the endpoints

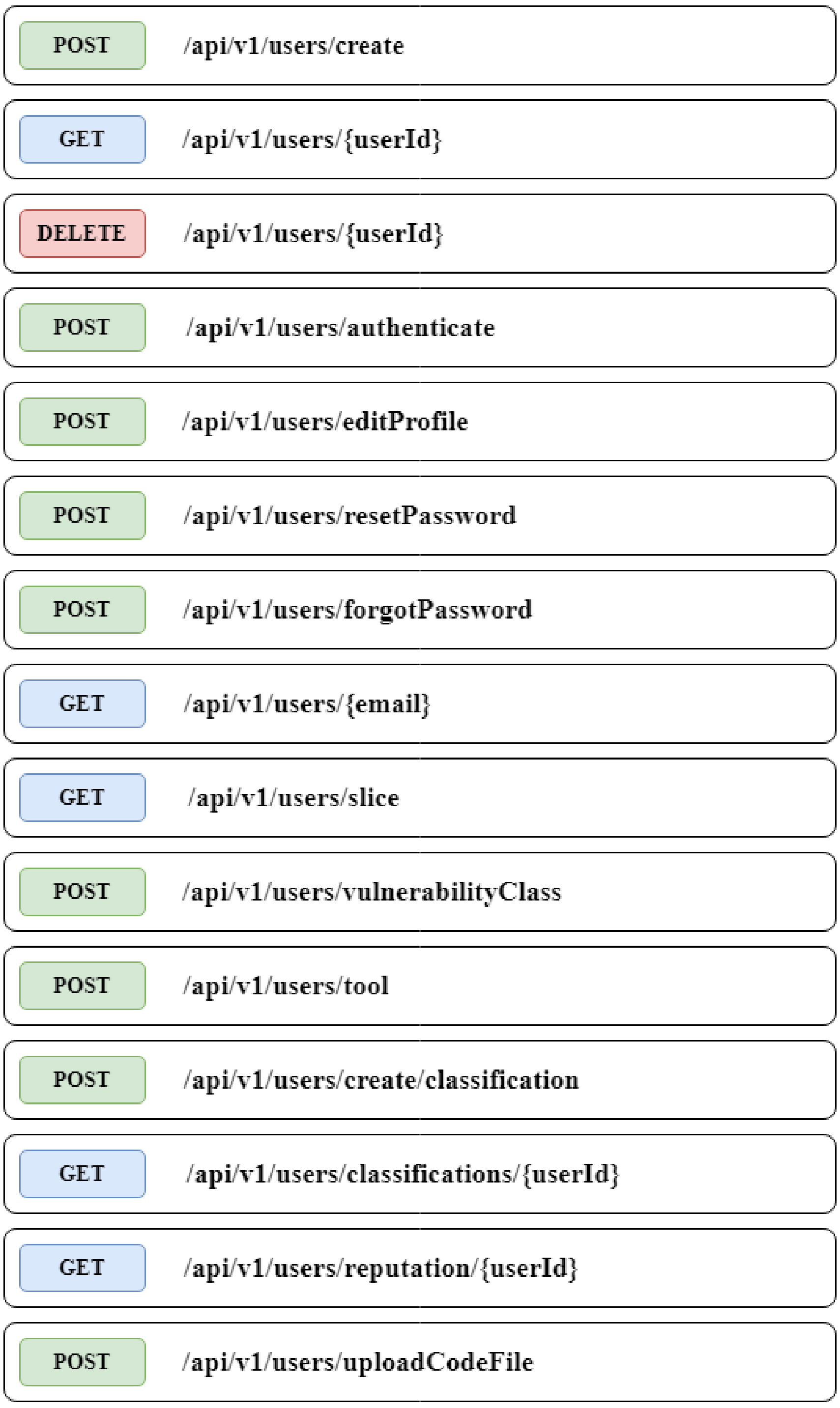
Endpoint (Method)	Possible Response Codes	Description
/api/v1/users/create (POST)	200, 400, 409, 500	OK, Bad Request, Conflict, Internal Server Error
/api/v1/users/userId (GET)	200, 400, 401, 404, 500	OK, Bad Request, Unauthorized, Not Found, Internal Server Error
/api/v1/users/userId (DELETE)	200, 400, 401, 403, 404, 500	OK, Bad Request, Unauthorized, Forbidden, Not Found, Internal Server Error
/api/v1/users/authenticate (POST)	200, 400, 401, 404, 500	OK, Bad Request, Unauthorized, Not Found, Internal Server Error
/api/v1/users/editProfile (POST)	200, 400, 401, 403, 500	OK, Bad Request, Forbidden, Internal Server Error
/api/v1/users/resetPassword (POST)	200, 400, 409, 500	OK, Bad Request, Internal Server Error
/api/v1/users/forgotPassword (GET)	200, 400, 404, 500	OK, Bad Request, Not Found, Internal Server Error
/api/v1/users/email (GET)	200, 400, 401, 404, 500	OK, Bad Request, Unauthorized, Not Found, Internal Server Error
/api/v1/users/slice (GET)	200, 400, 401, 404, 500	OK, Bad Request, Unauthorized, Not Found, Internal Server Error
/api/v1/users/vulnerabilityClass (POST)	200, 400, 401, 403, 409, 500	OK, Bad Request, Unauthorized, Forbidden, Not Found, Internal Server Error
/api/v1/users/tool (POST)	200, 400, 401, 403, 404, 409, 500	OK, Bad Request, Unauthorized, Forbidden, Not Found, Internal Server Error
/api/v1/users/classification (POST)	200, 400, 401, 404, 500	OK, Bad Request, Forbidden, Not Found, Internal Server Error
/api/v1/users/classifications/userId (GET)	200, 400, 401, 404, 500	OK, Bad Request, Unauthorized, Not Found, Internal Server Error
/api/v1/users/reputation/userId (GET)	200, 400, 401, 404, 500	OK, Bad Request, Unauthorized, Not Found, Internal Server Error
/api/v1/users/uploadCodeFile (POST)	200, 400, 401, 403, 500	OK, Bad Request, Unauthorized, Forbidden, Internal Server Error

significance in the system. It is worth noting that all the messages exchanged (between the client and the server) are in JSON format.

Figure 4.5 illustrates the BugSpotting's API in detail, more specifically, the endpoints that it comprises. It is composed of 15 endpoints, each one serving a different purpose and providing a different functionality. As it is possible to observe, the different color in each endpoint highlights the type of HTTP method that is meant to be used for that particular operation. In addition, each of the endpoints' path provides a clear hint about its functionality or resource that it targets.

It is also worth mention that each of these paths start exactly the same way ("/api/v1/..."). The explanation for this design approach is the simple: first, it allows backward compatibility. In other words, it is possible to have a separate "v2" endpoint (introducing new features or making changes) without affecting the logic of API's "v1" version. Clients would still be able to access the "v1" version. In addition, if support will no longer be given to the "v1" endpoint, a smooth transition to the "v2" is granted by notifying the clients that in the future, "v1" will eventually be deprecated.

Each of the endpoints is succinctly described below. A more thorough and detailed description



POST	/api/v1/users/create
GET	/api/v1/users/{userId}
DELETE	/api/v1/users/{userId}
POST	/api/v1/users/authenticate
POST	/api/v1/users/editProfile
POST	/api/v1/users/resetPassword
POST	/api/v1/users/forgotPassword
GET	/api/v1/users/{email}
GET	/api/v1/users/slice
POST	/api/v1/users/vulnerabilityClass
POST	/api/v1/users/tool
POST	/api/v1/users/create/classification
GET	/api/v1/users/classifications/{userId}
GET	/api/v1/users/reputation/{userId}
POST	/api/v1/users/uploadCodeFile

Figure 4.5: BugSpotting's API

(including the message's header, body and responses) can be understood in Appendix B.

POST /api/v1/users/create – This endpoint is responsible for the creation of a new BugSpotting account, allowing new users to register within the platform.

GET /api/v1/users/{userId} – This endpoint retrieves the complete profile information for a user specified by their unique user identifier (*userId*) as well as the role and session token.

DELETE /api/v1/users/delete – This endpoint allows the administrator of the system to delete a certain user based on the user's identifier.

POST /api/v1/users/authenticate – This endpoint is responsible for user's authentication in the BugSpotting's platform, giving him access to protected routes (only accessible in the possession of a session token) such as the creation of a classification, update the profile's information, amongst others.

POST /api/v1/users/editProfile – This endpoint is responsible for allowing the user to update its profile information. The user can change the information in bulk (all the fields changed simultaneously) or independently (with the exception of the password related fields: all or none must be filled). He can change his first name, last name, email and update the current password.

POST /api/v1/users/forgotPassword – This endpoint has the objective of facilitating the password recovery process by triggering an email to the user. The user will issue a request with his email to the solution's backend. Then, the backend initiates another request to an email service provider. Currently, the latter operation is executed through SendGrid [10] in order to send the recovery email.

POST /api/v1/users/resetPassword – This endpoint is responsible for updating the user's password based on a special short duration token (received in the email) and the information from the request's body. It is the follow-up after the request issued to the *POST /api/v1/users/forgotPassword* endpoint, complementing the password recovery process.

GET /api/v1/users/email – This endpoint is very similar to the *GET /api/v1/users/{userId}*. It retrieves the complete profile information for a user, but instead of relying in the user's unique identifier, it uses the email (which is also unique).

GET /api/v1/users/slice – This endpoint is responsible for delivering a code slice to the client.

POST /api/v1/users/vulnerabilityClass – This endpoint allows the admin of the system to insert a (new) vulnerability class in the system.

POST /api/v1/users/tool This endpoint allows the admin to insert information about a static analysis tool in the system. The tool itself (software) still needs to be integrated within the system manually.

POST /api/v1/users/classification – This endpoint constitutes one of the core functionalities offered by the system: the classification of a code snippet. It depends in the *GET /api/v1/users/slice* endpoint, since the user first needs to examine the slice and only then provide feedback. The feedback provided by the user consists in specifying if the code snippet in question is vulnerable or not, providing the vulnerabilities line (in the case of being vulnerable), the confidence degree (how convinced the user is about the classification) and an optional commentary.

GET /api/v1/users/classifications/{userId} – This endpoint retrieves all the user's classifications (from all vulnerability classes).

GET /api/v1/users/reputation/{userId} – This endpoint retrieves all the user's reputation - points per vulnerability class.

POST /api/v1/users/uploadCodeFile – This endpoint is responsible for allowing the system's administrator to insert a file that contains the code snippets *per se* (which will populate the database).

4.3.2 Algorithms

In this section, we tackle the core algorithms that have a major role in the BugSpotting's solution, focusing specifically on two key functions: choosing a code slice from the database to hand to the client (based on a set of criteria) and reaching consensus about the slice's final label (vulnerable or not vulnerable to a certain vulnerability class). The section is organized into two main parts: First, we discuss the various logics and criteria involved in retrieving a code slice from the database. Second, we examine the reasoning behind the achieving of a consensus on whether a given code snippet is vulnerable (or not) to a specific class of vulnerability.

Algorithm 1 Obtain a slice from the database algorithm

```
1: Input: userId, vulnerabilityClassId
2: Output: A Slice based on chosen criteria or an error
3: function GETSLICE(userId, vulnerabilityClassId)
4:     user ← FindUser(userId)
5:     if user is null then
6:         Log ”User with userId not found”
7:         throw UserNotFoundException
8:         exit
9:     end if
10:    slice ← GetUnclassifiedSlice()
11:    if slice is null then
12:        slice ← GetSliceWithFewestClassifications(userId, vulnerabilityClassId)
13:    end if
14:    if slice is null then
15:        slice ← GetSliceWithHighestDivergence(userId, vulnerabilityClassId)
16:    end if
17:    if slice is null then
18:        Log ”Slice does not exist”
19:        throw SliceDoesNotExistException
20:        exit
21:    end if
22:    ExecuteToolAndCreateClassification(slice)
23:    return slice
24: end function
```

Retrieval of a code slice

The main function to retrieve a code snippet of the database is depicted in Algorithm 1. This function receives the user identifier (*userId*) and the vulnerability class identifier (*vulnerabilityClassId*). Initially, the algorithm tries to fetch the user with the identifier provided in the arguments. If this operation returns null (the user is not found), an exception is thrown and the function is no longer executed. Otherwise, the process of obtaining the slice begins. Before the slice is obtained, 3 functions (explained below) may be invoked. If none of the functions returns a slice, an exception is thrown. Otherwise, the algorithm proceeds with the classification of the slice by a tool (WAP) and the results from the latter are saved in the database. After the tool's execution, the slice is returned and handed to the user for further classification.

Phases of Slice Retrieval: The process of obtaining the slice is divided into four phases, each corresponding to the functions illustrated in Algorithm 1 (lines 10 to 12), more specifically:

- **GetUnclassifiedSlice**: Fetches an unclassified slice.
- **GetSliceWithFewestClassifications**: Chooses the slice with the fewest classifications.
- **GetSliceWithHighestDivergenceDegree**: Fetches the slice with the highest divergence.
- **ExecuteToolAndCreateClassification**: Before the slice is sent to the client, the tool classifies it.

A detailed explanation of each function is presented below:

GetUnclassifiedSlice: The function does not receive any input and returns an unclassified slice or null (in the case of all slices have already been classified at least one time). The process is simple: a filtering will occur in order to find a slice that has not been classified yet. If there are no slices matching this criteria, null is returned. Else, the first one of the set (of unclassified slices) is returned.

Algorithm 2 Obtain an unclassified slice from the database algorithm

```
1: Input: None
2: Output: An unclassified slice or null
3: function GETUNCLASSIFIEDSLICE
4:     unclassified ← GetFirstUnclassifiedSlice()
5:     if unclassified is not null then
6:         return unclassified slice
7:     else
8:         return null
9:     end if
10: end function
```

GetSliceWithFewestClassifications: This function is invoked if the previous one, GetUnclassifiedSlice, returns null or, in other words, when all slices have been classified at least once (for the current vulnerability class chosen). The function takes two arguments: the user identifier (*userId*) and the vulnerability class identifier (*vulnerabilityClassId*). Before directly retrieving the slice with the fewest classifications from the database, a selection process takes place. This process involves calling another function, *GetCandidateSlices*, whose details are explained below and illustrated in Algorithm 4. After acquiring the list of candidate slices, if the latter is empty, the function returns null and stops the execution process. Otherwise, it looks for the slice with the fewest number of classifications (from the returned list). This is accomplished by, in a first instance, calculating the maximum number of classifications across the candidate slices. If all the slices have the same number of classifications (in the context of the current vulnerability class) as the calculated maximum, then null is returned. Else, the slice with the fewest classifications is returned (by default, the list is sorted in ascending order by the number of classifications, so the first slice in the ordered list is returned).

Algorithm 3 Obtain the slice with fewest classifications from the database algorithm

```
 1: Input: userId, vulnerabilityClassId
 2: Output: Slice with fewest classifications or null
 3: function GetSliceWithFewestClassifications(userId, vulnerabilityClassId)
 4:     slices ← GetCandidateSlices(userId, vulnerabilityClassId)
 5:     if slices is empty then
 6:         return null
 7:     end if
 8:     maxClassifications ← slice with the highest number of classifications
 9:     if all slices have maxClassifications then
10:         return null
11:     end if
12:     return first slice ordered by the number of classifications
13: end function
```

GetCandidateSlices: This function is illustrated in Algorithm 4 and takes the same arguments as the previous function. The workflow starts by fetching the threshold value from the server's configuration settings. This threshold specifies the maximum number of allowed classifications per vulnerability class for a given slice. The function then retrieves all unique slice identifiers that the user has already classified for the given *vulnerabilityClassId*. Next, it collects all unique slice identifiers that the user has "completely" classified across all vulnerability classes. Here, "completely" means the user has classified the slice for all available vulnerability classes. The function then merges these two lists: *alreadyClassifiedForVulnerabilityClass* and *classifiedSliceIds*. This merged list serves as a filter for acquiring candidate slices, which are further refined by the algorithm. The list ensures that slices already "completely" classified or have been classified for the current vulnerability class by the user are excluded but keeps the ones have fewer classifications than the threshold. This is crucial because it prevents a user from reclassifying a slice for the same vulnerability class. For instance, if a user has already classified a slice under vulnerability class VC1, they will not see that slice again when classifying under VC1. However, that slice could reappear if the user switches to classifying under a different vulnerability class, like VC2, since it has not been classified under VC2 yet.

GetSliceWithHighestDivergence: This function is depicted in Algorithm 5 and, similarly to the last two functions mentioned, it takes as arguments both the user identifier (*userId*) and the vulnerability class identifier (*vulnerabilityClassId*). After invoking the already explained *GetCandidateSlices* function, it verifies if the result from the prior invocation is an empty list: if it is indeed an empty list, then the

Algorithm 4 Obtain candidate slices algorithm

```
1: Input: userId, vulnerabilityClassId
2: Output: List of Candidate Slices
3: function GETCANDIDATESLICES(userId, vulnerabilityClassId)
4:     threshold ← Parse from configuration
5:     alreadyClassifiedForVulnerabilityClass ← Find all Classifications for this user and vulnerabili-
   tyClassId
6:     classifiedSliceIds ← Find all slice IDs classified by this user for all vulnerability classes
7:     Union alreadyClassifiedForVulnerabilityClass and classifiedSliceIds
8:     return All slices from database excluding those in the merged list and below threshold
9: end function
```

Algorithm 5 Obtain the slice with the highest divergence degree algorithm

```
 1: Input: userId, vulnerabilityClassId
 2: Output: Slice with highest divergence or null
 3: function GETSLICEWITHHIGHESTDIVERGENCE(userId, vulnerabilityClassId)
 4:     slices ← GetCandidateSlices(userId, vulnerabilityClassId)
 5:     if slices is empty then
 6:         return null
 7:     end if
 8:     minClassifications ← minimum Classifications among all slices
 9:     Filter slices to only include those with minClassifications
10:     if only one such slice exists then
11:         return that slice
12:     end if
13:     return First slice ordered by maximum divergence degree
14: end function
```

function halts its execution and null is returned. Otherwise, the next step is to get (from the candidate slices list) the slices with the minimum number of classifications (calculated before in line 8). If the result of the filtering is a single slice, that is the one returned. Otherwise, the list is filtered in descendent order, by the highest divergence degree value and the slice associated with it is retrieved.

ExecuteToolAndCreateClassification: This function can be observed in Algorithm 6. The function is executed after the previous three phase mentioned above, hence having as argument the filtered slice. If the tool has already classified the received slice, it immediately halts is execution and that slice is delivered to the user. Otherwise, only after making a classification to the slice (and storing it in the database) the slice is sent to the client.

The core ideas behind retrieving a slice The idea in applying multiple filters and sort the results before handing a slice to the client has multiple logical factors, which are presented below:

- **Uniform distribution**: If there are no unclassified slices available, the algorithms look for slices with the fewest classifications (having into account the current vulnerability class). This can be seen as a strategy to balance the attention each slice receives, thereby achieving a more uniform distribution of classifications across slices.

Algorithm 6 Execute Tool and Create Classification algorithm

```
1: Input: Slice object
2: Output: None
3: function EXECUTETOOLANDCREATECLASSIFICATION(slice)
4:     if tool has already classified slice then
5:         return
6:     end if
7:     classifications ← ExecuteTool(slice)
8:     CreateClassificationInDatabase(classifications)
9: end function
```

- **User-Specific Filtering**: The algorithms take into account the slices that a user has already classified. This avoids redundancy and repetition for individual users, since they are not constantly classifying the same code snippet. This allows the system to offer a less cumbersome and more engaging experience.

- **Divergence as a Tie-breaker strategy**: When the all the slices have been classified at least on time and have the same number of classifications (for the current vulnerability class), the ones with higher divergence have priority. This approach greatly improves the overall process since the slices that possess the higher divergence degree, i.e., the higher discordance amongst classifiers, are the ones that will be selected. This ends up being seen as a ”conflict-solving” strategy that improves the quality of the classifications.

- **Configurability and Scalability**: A threshold parameter allows the system to adjust to different scales. More precisely, it allows the system to adapt to the number of users it has and evolve accordingly. For instance, if the system exponentially grows in terms of number of users, perhaps having the threshold as a lower number will make the system not only less trustworthy (since there are less opinions from the crowd about a slice) but also will end up in exhausting the slices very fast. The fact that it is possible to easily change this configuration in the server’s settings increases the ease to accommodate growth and adapt to different scenarios.

Determine the slice’s final label (consensus)

Algorithm 7 illustrates the process of achieving a code snippet’s final label, which indicates whether the slice in question is indeed vulnerable (or not) to the given vulnerability class. The function receives the slice identifier as argument and returns either 1 (indicating that the slice is vulnerable to the vulnerability class in question) or 0, indicating that it is not vulnerable. The function then retrieves all the classifications associated with the slice and creates two important variables: *positiveWeight*, which will store the positive classifications (classifications marked as vulnerable) and the *negativeWeight*, which, in opposition to *positiveWeight*, stores the negative classifications (classifications marked as not vulnerable). Next, per each one of these classifications, the algorithm invokes another function – *GetWeightOfClassifierForVulnClass* – which returns the reputation (or weight) of the classifier who made that classification. This value is then accumulated to either the *positiveWeight* or *negativeWeight* variables. This means that the higher the reputation of the classifier, the more of an impact it will have on the label’s final result. Finally, after all classifications have been iterated, if the *positiveWeight* is greater than the *negativeWeight*,

Algorithm 7 Label consensus algorithm

```
1: Input: sliceId
2: Output: Consensus Label (1 or 0)
3: function DETERMINELABELCONSENSUS(sliceId)
4:     classifications ← get all classifications of the slice
5:     positiveWeight ← 0
6:     negativeWeight ← 0
7:     for each classification in classifications do
8:         weight ← GetWeightOfClassifierForVulnClass(classification.ClassifierId, classification.VulnerabilityClassId)
9:         if classification.Label == 1 then
10:            positiveWeight ← positiveWeight + weight
11:        else
12:            negativeWeight ← negativeWeight + weight
13:        end if
14:    end for
15:    if positiveWeight > negativeWeight then
16:        return 1
17:    else
18:        return 0
19:    end if
20: end function
```

the slice is marked as vulnerable to that vulnerability class. Otherwise, it is marked as not vulnerable. In the exceptional case of a tie, the admin or security experts that belong to the project will review the code snippet and provide a final label.

GetWeightOfClassifierForVulnClass: Algorithm 8 depicts the process of obtaining the weight (reputation) of a classifier based on its unique identifier and vulnerability class identifier. This helper function will, based on the two aforementioned arguments, obtain the weight (or reputation) of the classifier in question, which will be further used to reach the final label's consensus.

Algorithm 8 Get Weight of Classifier for Vulnerability Class algorithm

```
1: Input: classifierId, vulnerabilityClassId
2: Output: Weight of the classifier for the given vulnerability class
3: function GETWEIGHTOFCLASSIFIERFORVULNCLASS(classifierId, vulnerabilityClassId)
4:     classifierVulnWeight ← Query database for entry with classifierId and vulnerabilityClassId
5:     if classifierVulnWeight is null then
6:         Log ”Could not find the ClassifierVulnWeight for the specified classifier id”
7:         Throw ClassifierVulnWeightNotFoundException
8:     end if
9:     return classifierVulnWeight associated weigth property
10: end function
```

4.4 Deployment

Introduction

In order to make the BugSpotting solution accessible to the world, a set of steps had to be taken. In this section, we delve into the requirements and necessary steps in order to execute the deployment of both the frontend and backend in a virtual machine (VM) supplied by Faculdade de Ciências da Universidade de Lisboa (FCUL).

Requirements

Before deploying the application, the virtual machine had to be prepared with a set of software installations to support both the frontend and backend components. The following software packages were installed:

- **.NET SDK**: The .NET Software Development Kit (SDK) is a mandatory software as it serves as basis for developing, compiling and running .NET applications. It includes multiple libraries that ease the building and running of .NET applications, which is the case of the backend of BugSpotting's solution.
- **Node.js and npm**: Node.js is a JavaScript runtime. It allows the execution of JavaScript code in the server side (no browser required). Node Package Manager (NPM) is the default package manager for Node.js, which has the responsibility of managing the dependencies of the frontend solution. Npm is of major importance for the setup and execution of the React frontend.
- **MySQL database**: The MySQL client was installed in the VM to enable access to the database itself (where the data is persisted). It addition, it is also has the duty of several managing tasks (such as querying and updating the database) so the integration with the .NET backend is seamless.
- **React library**: As already explained in Section 4.1, React is a library meant for build user interfaces. It is thus required by the virtual machine, containing important utilities and libraries crucial for the proper execution of the application's frontend.
- **Java Virtual Machine (JVM)**: The JVM is an important piece of software that allows the execution of Java based programs. While it is not directly related to React or .NET, the JVM plays a crucial role since the tool that is responsible to classify the code snippets (WAP) was developed in Java.

Backend deployment

To deploy a .NET application, the backend was previously prepared for deployment by resorting to the .NET ”Publish” option. This step produces a deployment version of the .NET solution that contains the necessary files and DLLs. Next, this bundle was transferred to the VM and appropriately placed in the application directory so the backend service could function correctly. Finally, to enable a secure communication, a proxy was setup. This allowed the requests to be redirected to HTTPS.

Frontend deployment

Similarly to the backend deployment, a deployable bundle for the React application was generated resorting to npm's build script. This bundle was then transferred to the virtual machine, being placed in the application's directory to be served to the clients.

Chapter 5

Evaluation

5.1 Introduction

In this chapter, we present the evaluation of our solution, BugSpotting, as well as the methodology we used to validate it and conduct the assessment under realistic conditions. In Section 5.2, we refer the methods used for collecting data from the users, outlining the employed methodologies. In the next section, we focus on the results obtained from a custom-made questionnaire given to BugSpotting's users, highlighting both quantitative and qualitative analysis related to the UI/UX. Section 5.4 presents the a summary of the conclusions based on the current classifications. Finally, at the end of the chapter, we mention the limitations of the system and detail the motives behind these.

5.2 Data Collection Methods

5.2.1 Crowdsourced Classifications

As mentioned throughout this document, the classifications of the code snippets were collected through the BugSpotting's platform. Users would provide their opinion about a certain label, indicating whether the code snippet in question was vulnerable (or not vulnerable) to a certain previously chosen vulnerability class. The same would apply to the tool, but in an automated way, where the server would be responsible for its execution.

It is worth mentioning that while BugSpotting's platform supports all the eight vulnerability classes that we proposed to include, the code slices that were inserted in the database for further analysis by users (and the tool) were primarily focused on the SQL Injection and Cross-Site Scripting vulnerability classes. More precisely, the database was populated with 193 code snippets that could only be vulnerable (or not) to the aforementioned vulnerability classes. This means that, although the user could choose another vulnerability class besides the two above mentioned, it would not encounter any slices of that chosen type. The reason for using such code slices lies in the fact that this set was already manually classified. This provided us with a ground truth which in turn allowed to make a comparison with the results of the labels produced by the system, thereby offering a strategy to assess the precision and correctness of BugSpotting.

A total of 695 classifications were obtained for these two vulnerability classes, more precisely, 371 classifications for the class XSS and 324 for the class SQL Injection. This total comprises classifications from both users and the WAP tool. Given that there are 193 unique code snippets, and since each slice and vulnerability class combination is uniquely classified once for each of the two vulnerability classes, the tool sums a total of 386 classifications.

5.2.2 User Questionnaire

In order to thoroughly evaluate both the design and functionality of the BugSpotting's platform, a questionnaire was created. The main goal of this questionnaire was to capture the user's opinions in terms of User Experience(UX), which focus on the ease of use and user satisfaction and User Interface (UI), which evaluates the visual aspect of the website. We also use the results to evaluate if the requisites of Section 3.2 were accomplished. Hence, the questionnaire serves as a pillar to understand the areas where the platform needs improvement and also to measure its effectiveness.

The questionnaire is divided in four main categories:

- **Demographic Questions** This part aimed to gather information about the gender, age group, education level, occupation and experience with coding. The demographic data helps to understand the diversity of the users of the system and if any of the demographic factors could influence the final results. For example, both education level and experience with coding provide valuable insights in terms of classifications obtained, since it is more likely to achieve better results if the person classifying already has some level of expertise in the field of cybersecurity or has some knowledge about programming.

- **UI feedback** The UI statements had the objective to ask the user his opinion about the visual aspect and layout of the platform as well as the navigation and accessibility. This way, it becomes more clear for us what are the areas that are more appealing and well designed or need some improvements. For instance, one of the statements regarding this subject was the color palette used in both light and dark modes (whether it provided ease of reading).

- **UX feedback** This statements served the purpose of obtaining information about the ease of use and functionalities of the website. For example, understanding if the process of classifying a code snippet was easy and intuitive as well as knowing if the user had experienced some bottlenecks or other issues and if the website was quick and responsive.

- **Open-ended questions** These questions were designed to capture other ideas or opinions that could not be gathered by the statements. This allows the users to express their thoughts in a more personalized manner, offering other type of insights that may be useful for future changes or features. For instance, one of the open-ended questions was to know if the user had any other feature that could be added in order to enhance his experience.

To quantify the user's subjective experience in terms of UI/UX in the BugSpotting platform, we employed a 10-point Likert scale that aimed at gathering opinions about several aspects of the system's design and behaviour. Users were asked to rate their agreement to different statements about the website, in which 1 indicated ”strongly disagree” and 10 indicated ”strongly agree”. This scale was chosen with the objective of obtaining a detailed view of the user's perception about the overall features and visual aspect of the system, allowing us to better understand in which areas the platform required enhancements or additional features (or where it was well designed and effective). A total of 13 responses was obtained.

Appendix A contains all the statements and questions that were used in the questionnaire.

5.3 User Feedback and Evaluation

5.3.1 Quantitative Analysis (UI/UX)

The quantitative analysis focuses on the numerical ratings provided by the users (using the already mentioned 10-point Likert scale) to each statement about BugSpotting. In order to represent the obtained results, an average rating of each statement was calculated.

Figures 5.1 and 5.2 illustrate the UI/UX statements present in the questionnaire. Due to space limitations, each one is denoted by an alphanumeric identifier (such as ’S1’, ’S2’, etc.) on the x-axis. It is worth noting that these identifiers are in the same order as the statements provided in Appendix A. The y-axis represents the rating (the numerical values ranging from 1 to 10).

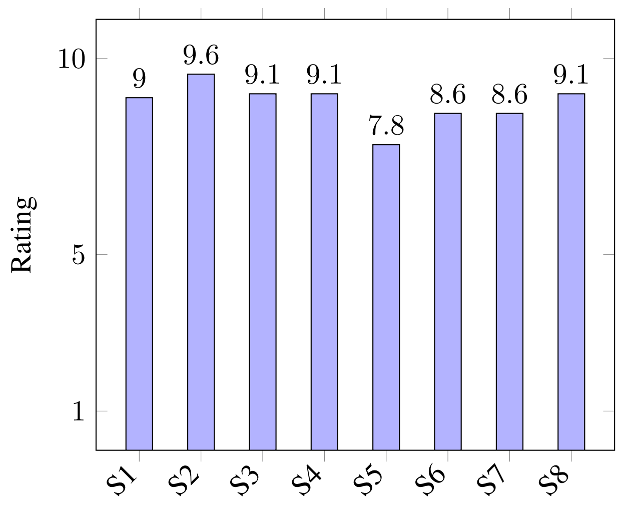


Figure 5.1: Average ratings for UI statements

Figure 5.1 shows the average rating of each statement in terms of UI evaluation, providing a high level overview about the users satisfaction and opinion about the visual aspects of BugSpotting. As it is possible to observe, statements S1, S2, S3, S4, and S8 show particularly high mean ratings, ranging from 9.0 to 9.6, which can be interpreted as a strong level of user satisfaction. In contrast, statements S5, S6, and S7 have relatively lower mean ratings, with 7.8 and 8.6, highlighting areas that may require further enhancement.

S5 stands out by having the lowest rating. This statement refers to the presentation of the reputation table, where the users can visualize their points (reputation) per vulnerability class. This could mean, for instance, that users did not find the table intuitive or of easy perception, which translates in the need for future improvement.

A more broader view and interpretation of such ratings can also be achieved resorting to other statistical metrics, like the median, mode and standard deviation. Table 5.1 depicts the aforementioned statistical metrics per statement. As it is possible to observe, the overall results are quite promising.

Table 5.1: Statistical Summary of UI Ratings

Identifier	Median	Mode	Standard Deviation
S1	9.0	8, 9, 10	1
S2	10.0	10	0.7
S3	10.0	10	1.5
S4	10.0	10	1.2
S5	7.5	10	1.8
S6	9.0	10	1.65
S7	9.0	10	1.1
S8	9.5	10	0.9

Comparing the average rating from Figure 5.1 with the median, it is possible to conclude that there is consistency across these metrics. This emphasizes the user satisfaction about BugSpotting's UI, since the ratings, in their majority, are composed of high values. For example, both S1 and S2 share a high average and median ratings.

Moreover, the mode can also help to reach some interesting conclusion: even though that, for S5 (which, as mentioned before, refers to the user's reputation table feature), the mode is 10 (being the rating that users chose the most), it has an average of 7.8. This is an indicator that the ratings provided by the users are dispersed and can be seen as another evidence that it is a feature worth inspecting and potentially improving, since there is not a clear consensus between users. This is also backed up by the standard deviation, which acts as a measurement of agreement between user ratings. For example, S2 presents a low standard deviation, which, in this case, can be understood that user's had a pleasing experience with the website's color pallet.

Finally, it is also possible to conclude from the table that there are other statements worth taking into account, being those S3 and S6, since they present a high standard deviation, although the mode has a value of 10.

Figure 5.2 depicts the average UX ratings provided by the users. Similarly to the UI case, the overall mean ratings were positive and revealed a high level of satisfaction in terms of the BugSpotting's UX component. As can be seen, statements S6, S8 and S9 show the highest ratings, being an indicator that reveals a considerable user approval. In contrast, statement S11 shows a substantial lower score. Unsurprisingly, this statement is related to the user's reputation table (more specifically, if the reputation system was of easy understanding and motivated the user to keep classifying code snippets), that was already pinpointed as an area needing attention and potential improving in terms of UI as well.

Table 5.2 illustrates other UX statistical metrics per statement, as in the case of UI. The overall consistency of the parameters suggests a positive user experience, but with some nuances. Although the values of the mode are, in their majority, only composed of one value or at most two, for the case of S11 this is not true. S11 has an average of 6.9 and a median of 7.0, which could mean there is not much dispersion on the results. The mode overrides the aforementioned: while the median and average show a relatively low level of contentment, the mode suggests that there does not exist a single dominant rating. Some users rated it as low (5) and others as extremely high (10). This is another strong suggestion that this area needs further refinement (reinforced with the UI results).

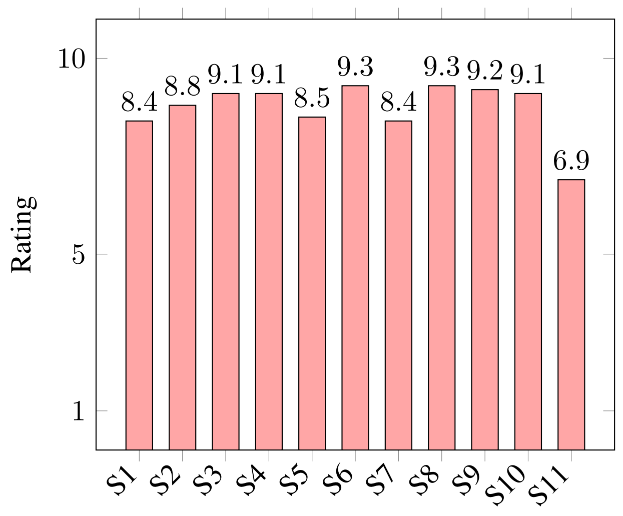


Figure 5.2: Average ratings for UX questions

Table 5.2: Statistical summary of UX ratings

Identifier	Median	Mode	Standard Deviation
S1	8.0	10	1.7
S2	9.0	9, 10	1.3
S3	9.5	10	1.2
S4	9.0	10	1.1
S5	8.0	10	1.9
S6	9.0	10	1.0
S7	8.0	9, 10	1.5
S8	10.0	10	1.0
S9	9.0	10	1.1
S10	10.0	10	1.2
S11	7.0	5, 7, 8, 10	2.0

On the other hand, statements S6, S8 and S9 show high median values accompanied by low standard deviations. This clearly states that these UX elements were highly appreciated by the users and no changes should be required. The areas that may required further enhancements are the ones that show lower median scores. This is even more obvious if the standard deviation presents higher values: this translates in a wider range of opinions, where a consensus is not obvious. This could be the case of statements S1, S5 and S7, which although they do not present particularly low medians (or means), they do in fact show high standard deviations.

The quantitative analysis serves as a valuable tool for not only confirming the overall high levels of user satisfaction, but also for isolating areas that may require additional attention. For example, the standard deviation for questions related to UI components was relatively low, suggesting a consensus among users about the UI's quality. On the other hand, it was possible to observe, in the case of UX, a more wider range of opinions between users. This leads us to conclude that there are some areas related with the UX that need further investigation. While these statistics provide a solid foundation for further improvement, a qualitative analysis can offer a more comprehensive point of view. The users are able to freely express their opinions and provide comments about any feature or functionality that they might have found efficient or needs further enhancement, allowing us to gather a deeper understanding about the BugSpotting's platform. In the next section, we analyse some of the comments and opinions provided by the users.

5.3.2 Qualitative Analysis (User Feedback)

Code Snippet Pool

The overall comments and opinions of the users to this topic were very similar: the code felt repetitive. The explanation of such comments is of easy understanding: the code examples itself were similar. This means that some of the code snippets would change just a few variables or some other minor details of the implementation. The algorithm was still able to balance correctly the code snippets served to the user, as we conducted several tests verifying this. These opinions and comments related to the code snippets used become useful so a more diverse set can be used in the future.

User Interface and Navigation

The user interface and navigation had an unanimous feedback: the ”Make a Classification” button, which is only present in the profile menu drop-down, should also be present in the main navigation bar once a user logs in. The users referred that if it is one of the core functionalities of the system, then it should have more visibility in the platform. Some said that it should be more trivial to access this page, mentioning once again that it ought to be in the main page. We conclude that in fact there should be an easier access to the classifications page, since is one of the most important functionalities of the website and that it requires further refinement.

User Reputation and Scoring Systems

The user reputation feature is the one that had the lowest rating in both UI and UX components. The comments from the crowd are pretty straightforward: some of the users complain that they do not understand what the points are or how they are obtained. Others refer that although they may understand, there should be a scale (with a max limit) so they could better reason about how high or low their points are. We do understand that the lack of a detailed explanation may confuse the users – we only succinctly describe the earning of points in the main page, which may not be clear enough. A better description and explanation must be added so the user easily understands the purpose of the reputation table and how the scoring system works. In addition, a user stated that it was not possible to understand if the 50 points initially received after a providing a classification were a default value or if they change according with the difficulty of the analysed code snippet. This clearly constitutes one of the areas that will need further focus and development.

Ease of the Classification Process

This topic also deserves our attention. The users evidenced that there should be a more clear explanation of how to classify a code snippet. Although the BugSpotting platform contains a dedicated page to explain the classification process, it would benefit from a more detailed one. Some of the users referred the fact that they had doubts about which lines they should pinpoint when they considered the code snippet vulnerable. They also mentioned that the process could be more efficient: the classification process should allow to classify multiple code snippets in a row, instead of returning to the page containing the vulnerability classes grid. These constitute valid points that we intend to enhance in the future, especially the explanation about the vulnerable lines that must be taken into account.

User Experience on Mobile Devices

Some opinions and comments were related to the lack of a mobile version and how the responsiveness in these must be improved. At the moment, the platform does not support mobile devices. This constitutes future work.

Bug Reports and Other Issues

A few bugs and other issues were mentioned by the users. A user stated that in the ”My Classifications” page, the search was slow and that occasionally did not work. Some referred that there are some unclickable buttons in the main page while another user said that the SQL Injection vulnerability classification was not being saved. We conducted further functional tests in the platform and were able to find that indeed there were some elements that were not clickable (for instance, the cards with the succinct explanation of how to classify a code snippet in three steps). We were unable to reproduce the other two issues mentioned. We were able to save every SQLI classification and everytime we accessed the ”My Classifications” page, all our classifications were being loaded correctly and in a timely fashion.

Future Requests

There were several requests in terms of functionalities that the users found interesting for the website to have. A complete list can be understood below:

- **Difficulty filtering**: Some users would find interesting if there was a functionality of filtering by difficulty of code snippet analysis, granting extra points if the code snippet was harder to classify.
- **Check other user’s classifications**: A user suggested that there could be a way to check other user’s classifications.
- **Code sanitization function for extra-points**: A user gave the idea of giving extra points if the sanitization function was mentioned in the comment box.
- **Forum to discuss classifications**: A user referred to the possibility of creating a sort of forum where the classifications could be discussed and thus, reach better results.

5.3.3 Qualitative Feedback Assessment

The qualitative analysis of the feedback provided by the users offered valuable insights about areas that should be enhanced in the future, allowing a deeper comprehension about what the users think about the platform. Some of the key takeaways were the need for a more diversified set of code snippets and a more user-friendly reputation and scoring systems. Users also pointed out that the guidelines for classifying a code slice should be more detailed, particularly what lines should be indicated in the case of the code snippet under classification is considered vulnerable. Finally, a mobile version of the website should also be implemented.

5.3.4 BugSpotting's Requisites

In Section 3.2, we established the requisites that BugSpotting aimed to accomplish:

- **Easy to Use (UI/UX):** We successfully developed a platform with a visually appealing interface that is easy to navigate, aligning with our UI goals. In terms of the UX, we ensured the platform is quick, easily accessible, and functional. With the feedback obtained from the users about both UI and UX, we confirmed that we were able to achieve what was initially intended.

- **Extensible:** The BugSpotting platform has potential to grow. By being developed with good code practices and well-known design patterns, its architecture ensures that is easy to introduce new tools and add new functionalities, as well as improve the ones that already exist.

- **Real and Synthetic Code Snippets Support:** The platform is capable of support both types of code snippets: real and synthetic. Although we used synthetic code snippets to populate the database, the latter is also capable of storing real ones, offering more diversity for future works.

The majority of our requisites for BugSpotting have been successfully met. The only aspect where we faced challenges, and partially met our goals, was in determining the consensus for the code snippet's final label, and thus, having precision and sensitivity in the results. Although we were able implement an algorithm that combined the classifications from both the users and the tool, arriving at a definitive label was not possible due to several factors. Nonetheless, we were still capable of infer conclusions about the final label. In the next section, we detail the challenges encountered and provide insights about the code snippets labels.

5.4 Data Quality Assessment

While our algorithm performed as we initially predicted, it is important to refer that the evaluation faced certain challenges, particularly in achieving the expected number of classifications, which is crucial for providing an objective conclusion about the precision and quality of the code snippet labels. The main reasons for the lack of consensus were the limited number of users on the platform, which in turn led to a diminished count of classifications and some technical delays. Nonetheless, we still present examples of classified code snippets that although do not show the final label, give strong and reliable indicators that prove the algorithm is working correctly and that the final labels tend to converge to the correct value. We also will delve deeper into the reasons and conditioning factors as well as challenges faced that did not made possible the obtaining of the desired label consensus. Illustrated in Figures 5.3 and 5.4, one can observe twenty examples of code snippets that were classified in the context of the XSS and SQLI vulnerability classes. The tables show the database's slice unique identifier, as well as the number of positive, negative and total classifications. In the last column, the real value of the label is shown (in which 0 means not vulnerable and 1 means vulnerable). It is worth mentioning that the total number of classifications for each code snippet in both tables has into consideration the classification generated by the WAP static analysis tool.

Table 5.3: Present code snippets classifications for XSS vulnerability class

Id	Positive Classifications	Negative Classifications	Total Classifications	Real Label
1	1	4	5	0
220	1	3	4	0
264	1	3	4	0
259	3	1	4	0
242	2	2	4	0
231	2	1	3	0
267	2	1	3	0
265	0	3	3	0
276	0	3	3	0
283	0	3	3	0

As it is possible to verify in Table 5.3, although at an initial phase in terms of total number of classifications for XSS class, one can tell that the results look very optimistic. If we focus on the slice with identifier one, if we were to reach a consensus at the moment, chances are that the label produced by BugSpotting would match the real value of the label, providing a precise result. It is also visible that it happens to the other code snippets, where a majority of negative classifications surpasses the positive ones, which match each final label. We even possess cases where classifications are unanimous, represented by the slices with identifiers 265, 276 and 283. The only exceptions are slices with identifiers 259, 242, 231 and 267. Nonetheless, this does not mean the final result produced by the system will not match the real one: after all, not enough classifications were provided. This constitutes the reason why we did not opt for lowering the threshold to a value under seven, as it would produce premature results that could no represent the real label's value.

Table 5.4: Present code snippets classifications for SQLI vulnerability class

Id	Positive Classifications	Negative Classifications	Total Classifications	Real Label
220	4	0	4	1
242	4	0	4	1
259	3	1	4	1
1	0	3	3	0
333	0	3	3	0
261	2	1	3	0
225	2	1	3	1
7	1	2	3	0
8	0	3	3	0
9	0	3	3	0

Table 5.4 also shows good indicators that suggest the system is functioning correctly in order to reach the final label value for SQLi class. Similarly to the XSS case, we also possess a majority of unanimous classifications that tend to the correct real label, which are represented by slices with identifiers 220,

242, 1, 8 and 9. Moreover, we also possess an exception: slice with identifier 261 are deviating from the correct final label. Similarly to what happened with the XSS situation, this is still at a premature phase and the correct result can still be reached.

It is possible to conclude by the interpretation of the results we possess at the moment that, although a consensus was not reached, the precision and correctness of the results can be easily understood and as time passes, we do believe that this indicators strongly suggest that the labels produced by BugSpotting would converge to the correct ones.

5.5 Discussion and Objectives

The code snippets that populated the database employed in this study only comprised two possible types of vulnerability: SQL Injection and Cross-Site Scripting. 193 code excerpts with these two possible vulnerabilities were initially inserted in the database since we already knew in advance the final label of them. More precisely, if each code snippet was vulnerable or not vulnerable to the aforementioned vulnerability classes, providing us a good starting point. When all these code snippets were classified, we would then compare the labels achieved by the system with the real ones, being able to discern if they were correct or if some modifications in the consensus algorithm were needed.

In the end, we were not able to achieve the intended outcome, for fundamentally two reasons: first, there was a lack of user participation to a large number of code snippets to analyse. Second, the way the algorithm itself was implemented also interfered with the gathering of results. More specifically, the number of users providing classifications was not enough to classify every code snippet in the context of the intended vulnerability classes and the algorithm was programmed to provide a distribution of code slices among users in a balanced manner. This algorithm design choice was deliberate and intentional: it aimed at providing a less cumbersome task of classifying code snippets by decreasing the chances of obtaining the same code snippet for classification. On the other hand, it also meant that achieving the final label of a particular code snippet required more time (since the algorithm prioritized serving other code slices that had either not been classified or had received fewer classifications).

Furthermore, we had initially configured a threshold with the value of ten maximum classifications per code snippet (in the context of a certain vulnerability class). We even changed the value to a lower one (seven) in an attempt to get label consensus results. Still, it was not possible to achieve consensus and if a lower value was to be used, there would be a diminished credibility and viability of the final label, which would be of no interest.

Moreover, there were also technical delays: although the application was already implemented and ready to be deployed, due to various circumstances, the system administrator responsible for the VM took some time to configure it (in order for the application to be deployed) and to setup the proxy to redirect the requests to HTTPS. This delay meant that we had to wait until all the configurations were completed, so no classifications could be obtained. While this study faced many obstacles, it is important to remember that the primary focus of this thesis was the construction of a platform that allowed the classification of code snippets and this was achieved: the BugSpotting platform is up and running. In order to achieve a valid consensus, an algorithm was always going to be needed, being it more or less minimalist, for the system to function as a whole.

It is also worth noting that if it was not for the lack of time and factors above stated, the system would produce (potentially) valid and precise labels as demonstrated in this section. We conducted an analysis to the database in order to visualize the classifications and related data we possessed at the moment and were able to notice that the database values were solid and some of the code snippets were already converging to a single label value. In addition, the algorithm responsible for choosing a slice offered good indicators, since the code snippets total number of classifications were balanced (which is in accordance with what we expected).

Chapter 6

Conclusion

This dissertation presents a novel approach to create datasets composed of code snippets that may be labeled as vulnerable (or not vulnerable) to a set of vulnerability classes. The strategy consist in the the construction of a web application – BugSpotting – that aims at gathering classifications from a group of people (crowdsourcing) and also from an SAT.

The BugSpotting platform was built taking into consideration several good practices and well known web development principles. It was implemented resorting to various technologies that not only make the development process easier, but are also efficient and widely supported. Moreover, it is also offers an intuitive and user-friendly interface, which allowed a simplified classification process for the users as well as other features, such as revisiting already classified code snippets and the verification of their reputation. To assess the quality of BugSpotting, a questionnaire was elaborated to gather information about the various aspects of the web application related to the UI and UX. The overall feedback from the users was very positive, confirming the platform's usability.

Furthermore, we developed two algorithms: first, an algorithm that is dedicated to the filtering of a code slice (based on a set of criteria). Once enough classifications are obtained (according to a predefined threshold), a second consensus algorithm comes into play, being responsible for determining the code snippet's final label.

Although our study was not able to reach a consensus about each code snippet's due to several limitations such as the lack of user participation and delayed deployment, it is essential to refer two important aspects: first, the main goal of constructing a functional platform was successfully met. The system is operational and functioning accordingly, with the potential for future improvements. Second, while a consensus was not reached, we still analysed the data we had until this moment, which revealed promising results when it comes to the precision and correctness of the code snippets labels.

6.1 Future work

In terms of future work, BugSpotting could benefit from a few improvements. These improvements range from simple modifications to more complex implementations, including:

- The feature related to the visualization of the user's reputation must be enhanced. This was the platform's area that received the lowest ratings from our users, both in terms of UI and UX. In addition, a more detailed explanation about the reputation should also be provided in the website. This way, there could be more motivation when making classifications.

- The algorithm, although working correctly and in accordance with what was expected, could be optimized in order to make the system faster. Not only in terms of actual implementation, but also enhancements when it comes to complexity and precision, making it more robust.
- As mentioned by our participants, the website could have a mobile version. This would enable more gathering of data, since there would be an extra device supported.
- The main navigation bar possessing a link to the classifications page, since this is one of the core functionalities offered by BugSpotting. This would not only make the classification process more obvious but also also indirectly demonstrate (potentially) the main goal of the system.
- Finally, we would like to prove the accuracy of our system when it comes to labeling code snippets. Due to the reasons already stated in this document, this was not possible to achieve. As time passes, eventually a majority of code slices will be classified and a comparison with the real labels can be made, assuring the system's precision and reliability, as it was implied by the results we had at the moment.

Bibliography

[1] Common weakness enumeration. cwe.mitre.org/data.

[2] Css. https://developer.mozilla.org/en-US/docs/Web/CSS.

[3] Figma plaftorm. www.figma.com.

[4] Html. https://developer.mozilla.org/en-US/docs/Web/HTML.

[5] Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[6] Microsoft's .net plaftorm. dotnet.microsoft.com/en-us/.

[7] Mysql open source database. www.mysql.com.

[8] Owasp top 10 vulnerabilities. https://owasp.org/www-project-top-ten/.

[9] React plaftorm. react.dev.

[10] Sendgrid email plaftorm. sendgrid.com.

[11] Sql injection. owasp.org/www-community/attacks/SQL_Injection.

[12] Mostofa Ahsan, Kendall E Nygard, Rahul Gomes, Md Minhaz Chowdhury, Nafiz Rifat, and Jayden F Connolly. Cybersecurity threats and their mitigation approaches using machine learning—a review. *Journal of Cybersecurity and Privacy*, 2(3):527–555, 2022.

[13] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 79–88, 2008.

[14] Paul Black. Sard: A software assurance reference dataset. https://samate.nist.gov/SARD, 1970.

[15] Paul E Black. A software assurance reference dataset: Thousands of programs with known bugs. *Journal of research of the National Institute of Standards and Technology*, 123:1–3, 2018.

[16] Harold Booth, Doug Rike, Gregory A Witte, et al. The national vulnerability database (nvd): Overview. https://nvd.nist.gov, 2013.

[17] Daren C Brabham. *Crowdsourcing*. Mit Press, 2013.

[18] Thierry Buecheler, Jan Henrik Sieg, Rudolf Marcel Füchslin, and Rolf Pfeifer. Crowdsourcing, open innovation and collective intelligence in the scientific method: a research agenda and operational framework. In *Proceedings of the International Conference on the Synthesis and Simulation of Living Systems*, pages 679–686, 2010.

[19] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(09):3280–3296, 2022.

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

[21] Florian Daniel, Pavel Kucherbaev, Cinzia Cappiello, Boualem Benatallah, and Mohammad Allahbakhsh. Quality control in crowdsourcing: A survey of quality attributes, assessment techniques, and assurance actions. *ACM Comput. Surv.*, 51, 2018.

[22] Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops*, pages 722–727, 2020.

[23] Robin Gandhi, Anup Sharma, William Mahoney, William Sousan, Qiuming Zhu, and Phillip Laplante. Dimensions of cyber-attacks: Cultural, social, economic, and political. *IEEE Technology and Society Magazine*, 30(1):28–38, 2011.

[24] Hector Garcia-Molina, Manas Joglekar, Adam Marcus, Aditya Parameswaran, and Vasilis Verroios. Challenges in data crowdsourcing. *IEEE Transactions on Knowledge and Data Engineering*, 28(4):901–911, 2016.

[25] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the static code analysis approach in software development. Technical report, Faculdade de Engenharia da Universidade do Porto, Portugal, 2009.

[26] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications*, 179, 2021.

[27] Howe and Jeff. The rise of crowdsourcing. *Wired*, 14, 2006.

[28] Srikanth Jagabathula, Lakshminarayanan Subramanian, and Ashwin Venkataraman. Identifying unreliable and adversarial workers in crowdsourced labeling tasks. *The Journal of Machine Learning Research*, 18(1):3233–3299, 2017.

[29] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, pages 27–36, 2006.

[30] John Kouraklis. *MVVM as Design Pattern*. 2016.

[31] Xin Li, Lu Wang, Yang Xin, Yixian Yang, Qifeng Tang, and Yuling Chen. Automated software vulnerability detection based on hybrid neural network. *Applied Sciences*, 11:3201, 2021.

[32] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19:2244–2258, 2021.

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[34] Alexandre K. Ligo, Alexander Kott, and Igor Linkov. How to measure cyber-resilience of a system with autonomous agents: Approaches and challenges. *IEEE Engineering Management Review*, 49(2):89–97, 2021.

[35] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2469–2485, 2021.

[36] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2015.

[37] Anh Nguyen-Duc, Manh Viet Do, Quan Luong Hong, Kiem Nguyen Khac, and Anh Nguyen Quang. On the adoption of static analysis for software security assessment–a case study of an open-source e-government project. *Computers Security*, 111, 2021.

[38] NIST. Minimum security requirement for federal information and information system. Technical Report FIPS PUB 200, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 20899-8930, 2006.

[39] Rafael Francisco Rosa Mesquita Ramires. *Detect Web Vulnerabilities Using Knowledge Graphs*. PhD thesis, Faculdade de Ciências da Universidade de Lisboa, Lisbon, Portugal, 2023.

[40] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the IEEE international conference on machine learning and applications)*, pages 757–762.

[41] Bertrand Stivalet and Elizabeth Fong. Large scale generation of complex and faulty php test cases. In *2016 IEEE International conference on software testing, verification and validation (ICST)*, pages 409–415, 2016.

[42] Rabia Tahir. A study on malware and malware detection techniques. *International Journal of Education and Management Engineering*, 8(2):20–30, 2018.

[43] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25:1419–1457, 2020.

[44] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020.

[45] Kerri Wazny. “crowdsourcing” ten years in: A review. *Journal of Global Health*, 7, 12 2017.

[46] Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. Strategies for crowdsourcing social data analysis. page 227–236, 2012.

[47] Man-Ching Yuen, Irwin King, and Kwong-Sak Leung. A survey of crowdsourcing systems. In *Proceedings of the IEEE International Conference on Privacy, Security, Risk and Trust and IEEE Third International Conference on Social Computing*, pages 766–773, 2011.

[48] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2019.

Appendix A

User Experience Questionnaire

A.1 Demographic Questions

- **Gender:**
  - Male
  - Female
  - Non-Binary
  - Prefer not to say
- **Age group:**
  - 18-24
  - 25-34
  - 18-24
  - 25-34
  - 35-44
  - 45-54
  - 55-64
  - 65 and above
- **Education level:**
  - Below High School
  - High School or Equivalent
  - Bachelor’s Degree
  - Master’s Degree
  - Doctorate or Higher
  - Prefer not to say
- **Occupation:**

  - Student
  - Software Developer/Engineer
  - IT Professional (non-development role)
  - Cybersecurity Expert/Professional
  - Academic/Researcher
  - Unemployed
  - Other (please specify):
  - Prefer not to say

- **Experience with coding:**
  - No experience
  - Basic (e.g., took a few courses, self-taught basics)
  - Intermediate (e.g., regularly code but not a primary job, hobbyist)
  - Advanced (e.g., professional developer, extensive coding experience)
  - Expert (e.g., deep knowledge in multiple languages, frameworks, etc.)

A.2 UI (User Interface) Questions

A.2.1 Visual Design and Layout

- The website's design is visually appealing to me.
- The website's color palette is comfortable for my eyes and promotes ease of reading (both dark and light mode).
- I found the switch between light and dark mode seamless and visually satisfactory.
- The presentation and layout of the code snippets (when making a classification) are clear and easy to understand.
- The presentation of user's reputation (table) is well structured and easy to interpret.

A.2.2 Navigation and Accessibility

- I could easily navigate between different sections and pages of the website.
- The icons and buttons on the website are intuitive and well placed.
- The grid layout for vulnerability classes made it straightforward for me to select a desired vulnerability class.

A.3 UX (User Experience) Questions

A.3.1 Ease of Use

- I found the website to be intuitive.
- I could easily navigate through the website without needing help.
- The process of classifying a code snippet (as vulnerable or not) was straightforward.
- Specifying the vulnerability lines and adding comments (when making a classification) was a simple process.
- The scale for indicating my confidence level was clear and easy to use.
- The help page provided clear and helpful instructions on classifying a code snippet.
- Navigating and deciding where to click on the website was instinctive for me.
- Editing my profile was easy and straightforward.

A.3.2 Functionality

- The website loaded the content quickly and was fast to react to my actions.
- Checking my past classifications was convenient and informative.
- The reputation system (points for each vulnerability class) was clear and motivated me to classify correctly.

A.3.3 Open-ended Questions

- What features or design elements of the website did you particularly like?
- Which areas or functionalities of the website do you believe that need improvement?
- Are there additional features or changes you would suggest for enhancing your experience on the website?
- Do you have any other feedback or comments about the website?

Appendix B

API Endpoints

B.1 POST /api/v1/users/create

HTTP Headers: The request should include the following HTTP header:

- `Content-Type: application/json`

Request Body: The request body should be a JSON object with the following keys:

- `firstName`: The user's first name.
- `lastName`: The user's last name.
- `email`: The email address for the new account.
- `username`: The desired username of the new account.
- `password`: The password for the new account.

Example:

```
{
   "firstName": "John",
   "lastName": "Doe",
   "email": "johnDoe123@example.com",
   "username": "jdoe2023",
   "password": "My5uper5ecretPassw0rd2023!"
}
```

Responses: Possible responses when creating a new user:

- `200 OK`: The user was successfully created.
- `400 Bad Request`: The request body is malformed or missing required fields. There can also be the case where the user does not comply with the required password specifications - the password must be between 8 and 25 characters in size, contain at least on upper case letter, one lower case letter, one digit and one special character.

- `409 Conflict`: A user with the given username or email already exists.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Response Example: Example of a successful user registration's response:

```
{
    "id": 100,
    "firstName": "john",
    "lastName": "doe",
    "email": "johnDoe123@example.com",
    "username": "jdoe2023",
    "roles": [
        "User"
    ],
    "sessionToken": "eyJhbGciOiJIUzI1NitsInR5cCI
    6IkpXVCJ7.eyJ1bmlxdWVfbmFtZSI6IjIiLCJyb2xlIj
    oiVXNlciIsIm5iZiI2MTY5Mjg5MDIxMiwiZXhwgjoxNj
    kzNDk1MDEyLCJpYXQiOjE2OTI4OTAyMFP9.rvPsMHKmz
    XtewZRNh24Sf7Dp115SH7dQ0MKhbDedIug"
}
```

The above response depicts the scenario of a successful user creation, where the data obtained is the following: user's unique identifier, first name, last name, email, username, role and the JWT token.

Error Response: Example of an error response when creating a new user:

```
{
    "ErrorCode": "USERNAME_ALREADY_EXISTS",
    "ErrorMessage": "Username is already taken"
}
```

The above example shows one possible response provided by the BugSpotting's backend (API). In this case, the HTTP status code is a 409, indicating the user tried to register himself under an already existing username.

Additional Notes: When creating a new user, the password is hashed by the backend, granting a more secure and robust database, since in the eventuality of the database is leaked, it is of no use for the attacker (the passwords are not in clear text). The used hashing algorithm is the SHA-256.

B.2 GET /api/v1/users/{userId}

HTTP Headers: The request must contain the following HTTP headers:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Parameters: This endpoint requires the following parameter:

- `userId`: The unique identifier for the user. This should be included in the URL path.

Example:

```
{
   {protocol}://{domain}/api/v1/users/1000
}
```

Responses: Possible responses when creating a new user:

- `200 OK`: The user's data was successfully retrieved.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `404 Not Found`: The user associated with the identifier contained in the request does not exist.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when getting a user by the unique identifier:

```
{
    "ErrorCode": "USER_NOT_FOUND",
    "ErrorMessage": "The account with the user
    1000 does not exist"
}
```

In the example above, a request was executed with a non existing identifier, resulting in an HTTP status code of 404, meaning that the user was not found.

Additional Notes: As previously shown, the header of the request must include the JWT token. It is mandatory in order to access this endpoint. The reason lies in the fact that each token is unique and is with the data contained inside the token (the identifier of the user) that the correct information is retrieved. Moreover, the successful response, although not depicted, follows the same format as the previous endpoint, except it does not possess the ”sessionToken” field.

B.3 DELETE /api/v1/users/delete

HTTP Headers: The request should include the following HTTP headers:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Parameters: The request must contain the following HTTP headers:

`userId`: The unique identifier for the user. This should be included in the URL path.

Example:

```
{
   {protocol}://{domain}/api/v1/users/1000
}
```

Responses: Possible responses when deleting a new user:

- `200 OK`: The user was successfully deleted.
- `400 Bad Request`: The request is malformed, missing or having invalid URL/query parameters, or invalid field types.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `403 Forbidden`: The user, although in the possession of a valid token, does not has the required permissions. Only an administrator (whose token's role value is "Admin") is able to delete a user.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when deleting a user by the unique identifier:

```
{
    "ErrorCode": "USER_NOT_FOUND",
    "ErrorMessage": "The account with the user
    1001 does not exist"
}
```

In the example above, a request was executed with a non-existing identifier, resulting in an HTTP status code of 404, meaning that the user was not found and thus, the system was not able to delete it.

Additional Notes: There is no success message returned by the backend. The operation of deleting a user is successful if the returned HTTP status code equals 200. Moreover, one can notice that the example URL is similar to the previous endpoint. Although this is in fact true, the API is able to respond to the correct operation because the HTTP methods of each operation are different (*GET* and *DELETE*).

B.4 POST /api/v1/users/authenticate

HTTP Headers: The request should include the following HTTP header:

- `Content-Type: application/json`

Request Body: The request body should be a JSON object with the following keys:

- `email`: The email address for the new account.
- `password`: The password for the new account.

Example:

```
{
   "email": "johnDoe123@example.com",
   "password": "My5uper5ecretPassw0rd2023!"
}
```

Responses: Possible responses when creating a new user:

- `200 OK`: The user was successfully authenticated.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the correct credentials.
- `404 Not Found`: The user with the credentials provided does not exist.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Response Example: Example of a successful user registration's response:

```
{
    "sessionToken": "eyJhbGciOiJIUzI1NitsInR5cCI
    6IkpXVCJ7.eyJ1bmlxdWVfbmFtZSI6IjIiLCJyb2xlIj
    oiVXNlciIsIm5iZiI2MTY5Mjg5MDIxMiwiZXhwgjoxNj
    kzNDk1MDEyLCJpYXQiOjE2OTI4OTAyMFP9.rvPsMHKmz
    XtewZRNh24Sf7Dp115SH7dQ0MKhbDedIug"
}
```

The above response depicts the scenario of a successful user authentication, consisting in the session token.

Error Response: Example of an error response when creating a new user:

```
{
    "ErrorCode": "EMAIL_NOT_FOUND",
    "ErrorMessage": "The account with the email
    johnDoe12@example.com does not exist"
}
```

The above example shows one possible response provided by the BugSpotting's backend. In this case, the HTTP status code is a 401, indicating the user tried to login but the credentials were incorrect, more precisely, the email.

B.5 POST /api/v1/users/editProfile

HTTP Headers: The request should include the following HTTP header:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Body: The request body should be a JSON object with the following keys:

- `id`: The user's unique identifier.
- `firstName`: The user's first name.
- `lastName`: The user's last name.
- `email`: The email address for the new account.
- `oldPassword`: The user's old password.
- `newPassword`: The user's new password.
- `confirmNewPassword`: Confirmation of the new user's password.

Example:

```
{
    "id": 100,
    "firstName": "John",
    "lastName": "Doe",
    "email": "jhonDoe123@example.com",
    "updatePassword": {
    "oldPassword": "My5uper5ecretPassw0rd2023!",
    "newPassword": "MyNewPassw0rd2023!",
        "confirmNewPassword": "MyNewPassw0rd2023!"
        }
    }
```

In the above example, the user is updating his current password. He did not alter any of the other fields, but by default, this are sent in the request (although no changes occur except for the password update).

Responses: Possible responses when updating the user's profile information:

- `200 OK`: The user was successfully created.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `404 Not Found`: The user associated with the identifier contained in the request does not exist.
- `409 Conflict`: The user did not provide the correct current password or the fields "newPassword" and "confirmNewPassword" do not match. It can also happen if the user fills only one or two out of the three password fields. Finally, there could also happen that the user is trying to update his email to an existing one.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when updating user's profile information:

```
{
    "ErrorCode": "PASSWORDS_MUST_BE_EQUAL",
    "ErrorMessage": "Passwords must be equal"
}
```

The above example shows an error message provided by BugSpotting's backend. In this case, the user did not fill the "newPassword" and "confirmNewPassword" fields with the same input.

Additional Notes: Although not illustrated, the successful response the same format as the *GET /api/v1/users/{userId}* endpoint.

B.6 POST /api/v1/users/forgotPassword

HTTP Headers: The request should include the following HTTP header:

- `Content-Type: application/json`

Request Body: The request body should be a JSON object with the following keys:

- `email`: The user's email.

Example:

```
{
   "email": "johnDoe123@example.com",
}
```

Responses: Possible responses when creating a new user:

- `200 OK`: The user was successfully created.
- `400 Bad Request`: The request body is malformed or missing the required field.
- `404 Not Found`: A user with the given email does not exist.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Response Example: Example of a successful user registration's response:

```
{
    {protocol}://{domain}/setnewpassword?token=
    32B34F76189014870E7B23581120F9152D1FD01EC
    488C623FAC5B52375669CE0640FADE530170FCC407901D9E
    F171287C2175B0F913365A7301676FB693FF672B6
}
```

The above response depicts the scenario of a successful response. This is an example of an hyperlink that the user will receive in his email and that will further redirect him to the reset password page of the BugSpotting's website. The special token in the response is further explained in the ”resetPassword” endpoint.

Error Response: Example of an error response when trying to recover the user's password:

```
    {
    "ErrorCode": "EMAIL_NOT_FOUND",
    "ErrorMessage": "The account with the email
    johnDoe312@hotmail.com does not exist"
}
```

The above example shows one possible response when the user tries to initiate the process of recovering his password. In this case, the user did not provide an existing email, obtaining an HTTP status code of 404 (the email was not found).

B.7 POST /api/v1/users/resetPassword

Request Parameters: The request must contain the following parameter:

`token`: This is a short duration token (1 hour) that is used to identify the user who requested a password recovery.

HTTP Headers: The request should include the following HTTP header:

- `Content-Type: application/json`

Request Body: The request body should be a JSON object with the following keys:

- `newPassword`: The user's new password.
- `confirmNewPassword`: A security measure so the user does not mistype his new password.

Example:

```
{
    "newPassword": "MyNewPassw0rd2023!",
    "confirmNewPassword": "MyNewPassw0rd2023!"
}
```

Responses: Possible responses when resetting the user's password:

- `200 OK`: The password was successfully updated.
- `400 Bad Request`: The request body is malformed or missing the required field. It can also happen to be the case where the token has expired or is incorrect.
- `409 Conflict`: The user did not fill both fields or they do not match.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when creating the new password:

```
{
    "ErrorCode": "PASSWORDS_MUST_BE_EQUAL",
    "ErrorMessage": "Passwords must be equal"
}
```

The above example shows one possible response when the user tries to input the new password, but they do not match. The backend responds with a HTTP status code of 409, indicating that the passwords must be equal.

Additional Notes: After the user receives an email containing an hyperlink to the password reset page, the latter includes a special token composed of multiple numbers and letters (which makes it unique and hard to forge). This token, with the duration of 1 hour, is meant to uniquely identify the user who initiated the password recovery process. After the request by the user (providing his email) in the previous endpoint, this token is generated and linked to the requesting user's account. The user then includes this token, which they receive via email, as a request parameter in their follow-up request to BugSpotting's backend (the current endpoint being explained), along with *newPassword* and *confirmPassword* fields in the request body. The backend identifies the user associated with the token and uses the information from the request body to update the user's password.

B.8 GET /api/v1/users/email

HTTP Headers: The request must contain the following HTTP headers:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Parameters: This endpoint requires the following parameter:

- `email`: The email of the user. This should be included in the URL path.

Responses: Possible responses when creating a new user:

- `200 OK`: The user's data was successfully retrieved.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `404 Not Found`: The user associated with email contained in the request does not exist.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when getting a user by the unique identifier:

```
{
    "ErrorCode": "EMAIL_NOT_FOUND",
    "ErrorMessage": "The account with the
    email testa@gmail.com does not exist"
}
```

In the example above, a request was executed with a non existing email, resulting in an HTTP status code of 404, meaning that the user with the specified email was not found.

Additional Notes: Similarly to the *GET /api/v1/users/{userId}* endpoint, a valid JWT token is also required in the request header for this endpoint. Additionally, the format of a successful response remains the same as with the aforementioned endpoint.

B.9 GET /api/v1/users/slice

HTTP Headers: The request should include the following HTTP header:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Parameters: This endpoint requires the following parameters:

- `userId`: The user's unique identifier.
- `vulnerabilityClass`: The chosen vulnerability class by the user.

Example:

```
{protocol}://{domain}/api/v1/users
/slice?userId=1000&vulnerabilityClassId=500
```

Responses: Possible responses when requesting a slice:

- `200 OK`: The slice was successfully delivered to the client.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `404 Not Found`: Either the user identifier or the slice were not found.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Response Example: Example of a successful response:

```
{
    "id": 1000,
    "sliceSourceCode": "{Base64 Encoding}"
}
```

The above response depicts the scenario of a successful slice request. It is worth noting that the *sliceSourceCode* field contains the slice encoded in Base64. Due to the size of the Base64 string, it was omitted.

Error Response: Example of an error response when creating a new user:

```
{
    "ErrorCode": "SLICE_NOT_FOUND",
    "ErrorMessage": "Slice does not exist"
}
```

The above example shows one possible error message provided by the server. In this case, the HTTP status code is a 404, indicating that a slice was not found.

Additional Notes: The requesting of a code slice is a complex process, involving a dedicated algorithm in order to choose the slice. In the next section, we will focus in detail in the execution that occurs ”under the hood” when a code snippet is requested.

B.10 POST /api/v1/users/vulnerabilityClass

HTTP Headers: The request should include the following HTTP headers:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Body: The request body should be a JSON object with the following keys:

- `vulnerabilityClassName`: The vulnerability’s class name.
- `vulnerabilityClassAcronym`: The vulnerability’s class acronym.

Example:

```
{
    "vulnerabilityClassName": "example",
    "vulnerabilityClassAcronym": "EXPL"
}
```

Responses: Possible responses when inserting a new vulnerability class:

- `200 OK`: The user was successfully deleted.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `403 Forbidden`: The user, although in the possession of a valid token, does not has the required permissions. Only an administrator (whose token’s role value is ”Admin”) is able to insert a new vulnerability class.
- `409 Conflict`: This occurs when the admin tries to insert an already existing vulnerability class (either with an existing name or acronym).

- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when inserting a new vulnerability class:

```
{
    "ErrorCode": "VULNERABILITY_CLASS_ALREADY_EXISTS",
    "ErrorMessage": "Vulnerability class with given
    acronym or name already exists"
}
```

In the example above, a request was executed providing an already existing vulnerability class, resulting in a response from the server with the HTTP status code of 409 (conflict).

Additional Notes: There is no success message returned by the backend. The operation of inserting a new vulnerability class is successful if the returned HTTP status code equals 200.

B.11 POST /api/v1/users/tool

HTTP Headers: The request should include the following HTTP headers:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Body: The request body should be a JSON object with the following keys:

- `vulnerabilityClassName`: The vulnerability's class name.
- `vulnerabilityClassAcronym`: The vulnerability's class acronym.

Example:

```
{
    "toolName": "Example Tool",
    "toolAcronym": "EXPL",
    "toolURL": "www.exampleTool.com"
}
```

Responses: Possible responses when inserting a new vulnerability class:

- `200 OK`: The user was successfully deleted.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.

- `403 Forbidden`: The user, although in the possession of a valid token, does not has the required permissions. Only an administrator (whose token's role value is "Admin") is able to insert new data about a tool.
- `409 Conflict`: This occurs when the admin tries to insert an already existing tool (either with an existing name, acronym or website).
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when inserting a new vulnerability class:

```
{
    "ErrorCode": "TOOL_CLASSIFIER_ALREADY_EXISTS",
    "errorMessage": "Tool classifier with given name,
    acronym or URL already exists"
}
```

In the example above, a request was executed an already existing tool, resulting in a response from the server with the HTTP status code of 409 (conflict).

Additional Notes: There is no success message returned by the backend. The operation of inserting new information about a tool is successful if the returned HTTP status code equals 200.

B.12 POST /api/v1/users/classification

HTTP Headers: The request should include the following HTTP headers:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Request Body: The request body should be a JSON object with the following keys:

- `vulnerabilityClassName`: The vulnerability's class name.
- `vulnerabilityClassAcronym`: The vulnerability's class acronym.

Example:

```
{
    "classifierId": 500,
    "sliceId": 1000,
    "vulnerabilityClassId": 15,
    "label": 1,
    "comment": "I am positive that this code snippet
    is vulnerable.",
        "confidenceDegree": 10,
        "vulnerabilityLinesViewModel": {
        "lines": "1, 5, 10-12"
        }
    }
```

This example of request shows the case of a classification where the user classified the code snippet as vulnerable (label with value 1) and with the maximum confidence degree (10). The user also indicated the vulnerable lines and provided a comment.

Responses: Possible responses when inserting a new vulnerability class:

- `200 OK`: The user was successfully deleted.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `404 Not Found`: This response occurs when either the user or the vulnerability class were not found.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Error Response: Example of an error response when inserting a new vulnerability class:

```
    {
        "ErrorCode": "USER_NOT_FOUND",
        "ErrorMessage": "The account with the user 500
        does not exist"
    }
```

In the example above, a request was executed with a user's identifier that does not exist, resulting in the response with an HTTP status code of 404.

Additional Notes: Similarly to the *GET /api/v1/users/slice* endpoint, this route also involves the execution of a complex algorithm in order to reach a consensus about the code snippet's label. A detailed explanation can be understood in the next section. In addition, there is no success message returned by the backend. The operation of creating a new classification is successful if the returned HTTP status code equals 200.

B.13 GET /api/v1/users/classifications/{userId}

HTTP Headers: The request must contain the following HTTP headers:

- `Content-Type: application/json`
- `Authorization: Bearer {JWT}`

Example:

```
{protocol}://{domain}/api/v1/users
/classifications/1000?skip=0&top=10
```

Request Parameters: This endpoint accepts the following parameters:

- `userId`: The unique identifier for the user. This should be included in the URL path.
- `skip`: A dynamic query parameter that initially starts at 0 and increments in intervals of 10. It instructs the backend on the starting point for the classifications to be returned, based on their unique identifiers.
- `top`: This parameter complements the `skip` parameter to define the range of classifications to retrieve. For instance, if `skip` is set to 10 and `top` is set to 10, classifications with unique identifiers ranging from 10 to 20 will be returned from the database.

Responses: Possible responses when creating a new user:

- `200 OK`: The user's classifications were successfully retrieved.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `404 Not Found`: The user associated with the identifier contained in the request does not exist.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Response Example: Example of a successful response with the user's classifications:

```
[
    {
        "sliceSourceCode": "{Base64 Enconding}",
        "vulnerabilityClassName": "Example One",
        "vulnerabilityClassAcronym": "EXPL1",
        "date": "2023-07-30T16:48:22.7159290",
        "label": 0,
        "comment": "I'm not very confident in my classification",
        "confidenceDegree": 3,
        "vulnLines": null
    },
        {
            "sliceSourceCode": "{Base64 Enconding}",
            "vulnerabilityClassName": "Example Two",
            "vulnerabilityClassAcronym": "EXPL2",
            "date": "2023-07-30T17:25:22.5060290",
            "label": 0,
            "comment": "I'm sure this code slice is not vulnerable",
            "confidenceDegree": 10,
            "vulnLines": null
        }
    ]
```

The above response depicts the scenario of a successful response when asking for the user's classification. The response is an array of JSON objects that represent each classification. In this case, the user only has classified two code snippets.

Error Response: Example of an error response when getting a user by the unique identifier:

```
    {
        "ErrorCode": "USER_NOT_FOUND",
        "ErrorMessage": "The account with the user
        1000 does not exist"
    }
```

In the example above, a request was executed with a non existing identifier, resulting in an HTTP status code of 404, meaning that the user was not found.

B.14 GET /api/v1/users/reputation/{userId}

HTTP Headers: The request must contain the following HTTP headers:

- `Content-Type: application/json` `Authorization: Bearer {JWT}`

Request Parameters: This endpoint accepts the following parameters:

- `userId`: The unique identifier for the user. This should be included in the URL path.

Example:

```
    {protocol}://{domain}/api/v1/users/reputation/1000
```

Responses: Possible responses when creating a new user:

- `200 OK`: The user's reputation was successfully retrieved.
- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `404 Not Found`: The user associated with the identifier contained in the request does not exist.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Response Example: Example of a successful response with the user's reputation:

```
[
    {
        "vulnerabilityClassName": "Example One",
        "vulnerabilityClassAcronym": "EXPL1",
        "weight": 50
    },
    {
        "vulnerabilityClassName": "Example Two",
        "vulnerabilityClassAcronym": "EXPL2",
        "weight": 50
    }
]
```

The above response depicts the scenario of a successful response when asking for the user's reputation. The response is an array of JSON objects that represent each weight (number of points) that a user has per vulnerability class. In this case, the user only has only classified two types of vulnerability class, so the response only includes two objects in the array.

Additional Notes: The error response example is not depicted because each possible one is the same as the previous endpoint.

B.15 POST /api/v1/users/uploadCodeFile

HTTP Headers: The request must contain the following HTTP headers:

- `Content-Type: multipart/form-data`
- `Authorization: Bearer {JWT}`

Responses: Possible responses when uploading a code file:

- `200 OK`: The user's data was successfully retrieved.

- `400 Bad Request`: The request body is malformed or missing required fields.
- `401 Unauthorized`: The user did not provide the session token or the token itself is malformed.
- `403 Forbidden`: The user, although in the possession of a valid token, does not has the required permissions. Only an administrator (whose token's role value is ”Admin”) is able to populate the database with code snippets.
- `404 Not Found`: The user associated with the identifier contained in the request does not exist.
- `500 Internal Server Error`: An error occurred on the server while processing the request.

Additional Notes: Unlike other endpoints that require a JSON object or URL parameters, this endpoint requires a file. As shown above in the HTTP header specification, the content type of this request is not ”application/json”, but a ”multipart/form-data), indicating that a file should be uploaded within the HTTP request body. Also, there is no success message returned by the backend. The operation of deleting a user is successful if the returned HTTP status code equals 200.