

BABEȘ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER
SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Advanced Threat Detection: SQL and XSS Auditing

Supervisor
Asist. Dr. Florentin Bota

Author
Moldovan Daniel-Gelu

2024

**UNIVERSITATEA BABEȘ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ**

LUCRARE DE LICENȚĂ

Detectarea Avansată a Amenințărilor: Auditarea SQL și XSS

**Conducător științific
Asist. Dr. Florentin Bota**

*Absolvent
Moldovan Daniel-Gelu*

2024

ABSTRACT

In this bachelor thesis, I present the importance of web security. It focuses on two types of cyber attacks: SQL injection and XSS. For both of them, I implemented a Bidirectional Long Short-Term Memory model, a Random Forest model and also a Regular Expression. The web application features a dashboard with four panels, two for each attack. These panels display charts showcasing the performance of the models and comparisons between them. Moreover, this thesis introduces an experimentation option, empowering users to conduct meticulous analyses using labeled datasets. From precision and accuracy to execution time, providing invaluable insights into the performance of our defense mechanisms. The originality of this paper lies in its ability to facilitate the comparison between Bi-LSTM, Random Forests, and Regular Expressions through interactive panels and an intuitive interface, allowing users to effortlessly distinguish the differences between the presented models. Additionally, logs can be supplied to the application directly from the database or from cloud platforms such as Google Cloud Storage. Alternatively, users can upload logs directly to the application. This research study shows the efficacy of machine learning models over the traditional method Regex, however Regex is significantly faster in detection times, making it a viable option where computational efficiency is critical.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	1
1.3	Structure	2
2	Background	3
2.1	Overview of Web Security Threats	3
2.2	Techniques for Detecting SQL Injection	4
2.3	Techniques for Detecting XSS Attacks	6
3	Threat Detection	11
3.1	The Problem	11
3.2	SQL Injection Detection	11
3.3	XSS Detection	12
3.4	Common AI Model for detecting SQL injection and XSS attacks . . .	14
3.4.1	Dataset Preparation	14
3.4.2	Long Short-Term Memory Model	14
3.4.3	Random Forest Model	19
4	Architecture	23
4.1	Analysis	23
4.1.1	Use Case Diagrams	23
4.2	Technology Stack	25
4.3	Database Design and Management	28
5	Cyblo	30
5.1	User Manual	30
5.2	Dashboard Implementation	35
5.3	User Authentication	36
5.4	Manual Testing	37
5.4.1	Realtime Audit	37
5.4.2	Audit	37
5.4.3	Manual Scanning	37

5.4.4	User Interface (UI) Testing	38
5.4.5	Functional Testing	38
5.4.6	Model Selection and Analysis	38
6	Results	39
6.1	Dataset	39
6.2	Performance Evaluation for Detection Algorithms	39
6.2.1	Accuracy	40
6.2.2	Precision	40
6.2.3	Recall	40
6.2.4	F1-Score	41
6.2.5	Time	41
6.2.6	Confusion Matrix	42
7	Conclusion	43
	Bibliography	44

Chapter 1

Introduction

1.1 Motivation

From a young age, I have been deeply fascinated by the field of cybersecurity. The rise in cyber threats and the need to keep sensitive information safe have always made me realize how important good cybersecurity is. Despite the advancements in technology, there remains a significant gap in platforms that allow users to select multiple models and run comprehensive experiments on audit logs for detecting security threats. This realization drove me to develop a unique platform that addresses this gap. My platform provides users with the capability to choose from various models and execute experiments on audit logs, enhancing the detection and prevention of cyber attacks such as SQL Injection and XSS attacks. This project is a step towards creating more secure web environments by offering a versatile and user-friendly tool for cybersecurity professionals.

1.2 Objectives

In today's digital landscape, web applications have become indispensable tools for businesses of all sizes. From large corporations to small startups, reliance on web applications has grown exponentially, serving various purposes such as customer communication, operational efficiency, and sales. Even traditional businesses, initially hesitant to adopt technology, have been compelled to embrace it, particularly in the wake of the COVID-19 pandemic.

However, with increased reliance on web applications comes the heightened risk of cyber attacks, particularly SQL Injection and XSS Injection, which exploit vulnerabilities and developer errors. These attacks rank among the top 10 most common threats to web applications [She10]. Recognizing the critical need for robust security measures, this Bachelor Thesis sets out to explore methods for detecting and

mitigating such attacks.

The thesis focuses on developing a web application featuring a dashboard equipped with various graphs and charts. Its primary objective is to detect and prevent attempts to compromise a company's web application, thereby safeguarding against potential risks. To achieve this, the thesis employs a comparative analysis between three detection methodologies: Regular Expressions, Random Forest model and Bidirectional Long Short-Term Memory models.

One of the key contributions of this thesis lies in its unique approach to evaluating the efficacy of Artificial Intelligence models against the traditional Regular Expressions. By visually presenting the differences between these approaches and providing comparative metrics. The primary reason behind the success of injection attacks lies in the absence of robust input validation mechanisms and the excessive granting of permissions beyond necessary limits.

1.3 Structure

The paper encompasses all steps from idea to implementation and results. It consists of several chapters. First chapter, Introduction, outlines the motivation and importance of detecting cyber attacks in modern businesses, presenting my own contribution. The second chapter, Background, presents related research from other researches and various approaches. Chapter 3, Threat Detection, explains the problem and its solution, utilizing Artificial Intelligence models and parsers for SQL injection and XSS attacks. Chapter 4, Architecture, covers the application's architecture, including chosen programming languages, database design and diagrams. The fifth chapter, Cyblo, contains details about the User Interface and User Experience aspects along with relevant code snippets. Chapter 6, Testing and Evaluation, demonstrates the application's testing process and obtained results. Lastly, Chapter 7, Conclusion, offers conclusions and recommendations for future work and improvements.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work OpenAI's GPT-4 was used in order to improve the readability of the thesis. After using this service, the author reviewed and edited the content as needed and takes full responsibility for the content of the thesis.

Chapter 2

Background

2.1 Overview of Web Security Threats

Web security threats are a major cybersecurity concern, with the potential to cause significant harm and adverse consequences for users. These threats endanger both businesses and individuals, encompassing issues such as computer viruses, phishing attacks and data theft. The impact of these threats goes beyond the digital realm, as cybercriminals use the internet to harm their targets. The consequences can include denial of access to computers and networks, unauthorized access and use of corporate systems, theft and exposure of sensitive information, and unauthorized changes to computer systems and networks.

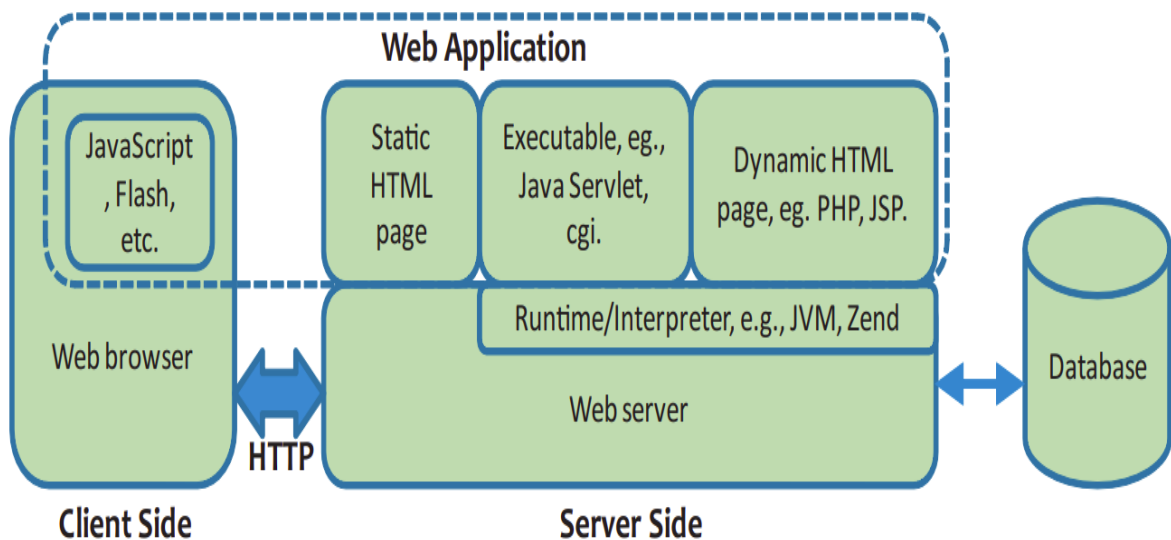


Figure 2.1: Web Application Components [LX11]

To understand the weaknesses of the web ecosystem, it's crucial to examine the parts of web applications. These applications generally include three fundamental elements as shown in Figure 2.1. Initially, programming languages are used for the

client-side interface and formulate database queries. Subsequently, the Hypertext Transfer Protocol (HTTP) acts as the conduit connecting the client-side and server-side elements. Lastly, there's the business process, which is the aspect of any web application [AS12].

There are multiple ways to protect web applications, including Web application firewall (WAF), HTTPS instead of HTTP, long and random passwords, usage of environment variables, secure authorization using JWT token.

2.2 Techniques for Detecting SQL Injection

SQL (structured query language) is a powerful programming language used to represent queries to database management systems, such as PostgreSQL, SQL Server and others. SQL injection attacks are the process where the user is able to insert code that later will be executed in SQL. In order to mitigate to be a victim of this kind of attack, the programmer should always validate the input from the user, sanitize it and check for any special characters.

```
1 $user_var = $_POST['user_from_client'];
2 $pass_var = $_POST['pass_from_client'];
3 $sql = "SELECT * FROM Users_Table WHERE username = '" . $user_var . "'
      AND pass = '" . $pass_var . "'";
4 $result = mysqli_query($connection, $sql);
```

Listing 2.1: Backend vulnerable to SQL Injection

```
1 SELECT * FROM Users_Table WHERE username = 'admin-- AND pass = ''
```

Listing 2.2: Example of SQL Injection

As shown in Listing 2.2, on a login page, the attacker doesn't even have to try to guess the password, it is enough to put the username 'admin-- and the code from Listing 2.1 will login the attacker.

There are several types of SQL injection attacks, including **tautologies**, **incorrect queries**, **piggy-backed queries**, **inference**, **alternate encodings** and **union queries**, among others [HVO⁺06]. The following Table 2.1 contain the above mentioned types, usage and an example related to Listing 2.1 In the Table 2.1, on "alternate encodings" type, '0x73687574646f7776e' is the ASCII code for SHUTDOWN.

Researchers and programmers have identified multiple approaches for detecting and mitigating SQL injection attacks [AY17]. However, none can guarantee 100% protection, particularly against modern SQL attacks. SQL injection attacks can be mitigated by ensuring input validation and sanitization.

Type	Usage	Username Injection
Tautologies	Bypass authentication	<pre>1 'OR 1=1 -- 2</pre>
Incorrect Queries	Gain information about database management system	<pre>1 admin' 2</pre>
Piggy-backed Queries	Execute remote commands	<pre>1 '; DROP TABLE Accounts -- 2</pre>
Inference	Determine Database schema, blind injection	<pre>1 admin' AND ASCII(SUBSTRING((SELECT 2 TOP 1 name from SYSOBJECTS),1,1)) >X WAITFOR 5 --</pre>
Alternate Encodings	Avoid detection	<pre>1 a d m i n ; e x e c (0 x 7 3 6 8 7 5 7 4 6 4 6 f 7 7 6 e) 2 --</pre>
Union Query	Extract data	<pre>1 'UNION SELECT owner FROM Accounts 2 --</pre>

Table 2.1: SQL Injection Types

One notable approach is **AMNESIA** (Analysis and Monitoring for Neutralizing SQLIA), which combines static analysis to automatically create a model for the legitimate queries generated by the web application and a dynamic component that monitors runtime queries and compares them with the pre-generated static model [HO].

Another significant approach is **R-WASP** (Real Time-Web Application SQL Injection Detector and Preventer) [Med13], which employs positive tainting to identify and mark trusted data sources, track these trust markings at the character level during runtime, and evaluate query syntax before execution. Although this method effectively prevents most SQLIAs, it falls short of fully protecting against modern SQLIA variants.

2.3 Techniques for Detecting XSS Attacks

JavaScript is one of the most used programming language in the world, usually used for web application, for the client-side, but it can also be used for the backend. It was created in 1995, dynamic language, and according to a Stack Overflow survey (from 2019), it is used by 71.5% of developers over the world [WBE20]. **Cross-Site Scripting** (XSS) is a type of attack where malicious code is injected into a user's browser without their knowledge [SBK18]. In order to mitigate this kind of attack, the developers should always validate the input of the user and check it against any special characters.

```
1 <script>
2 const parameters = new URLSearchParams(window.location.search);
3 const message = parameters.get('message');
4 document.getElementById('output').innerHTML = message;
5 </script>
```

Listing 2.3: Frontend vulnerable to XSS Attacks

```
1 http://examplesite.com/vulnerablepage.html?message=<script>alert('XSS
  Attack!');</script>
```

Listing 2.4: Example of XSS Attack

As shown in Listing 2.3, the message value is taken from the URL of the user and the result is directly put into the element of output without getting encoded or sanitized. In Listing 2.4, the user sends a script instead of a simple message and because the input is not validated, the script will run on client-side and it will show the alert meaning that the attack was successful.

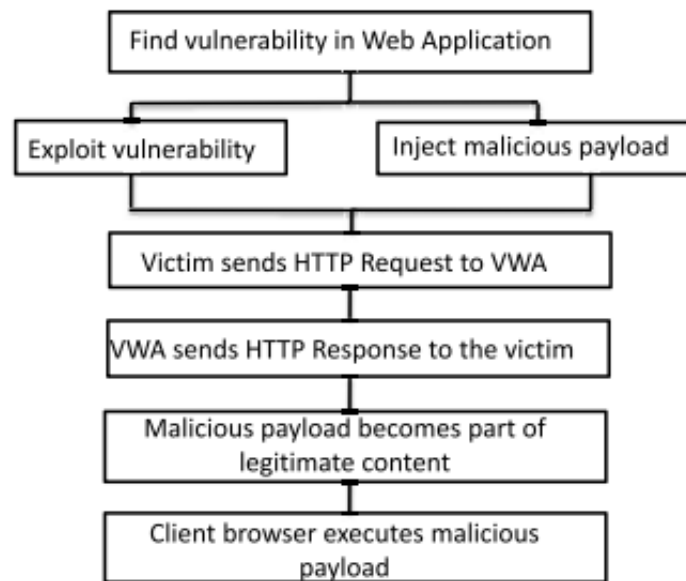


Figure 2.2: Steps in an XSS Attack [SBK18]

In Figure 2.2, we can see the steps involved in an XSS attack. It begins with identifying a vulnerability in the web application. Next, the attacker takes advantage of this weakness by injecting harmful code into the application. The attacker sends an HTTP request to the application and if the attack is successful, the harmful code becomes part of the legitimate content. The client browser then runs the harmful code as if it were normal content.

XSS attacks can be categorized into three types: persistent, non-persistent and Document Object Model (DOM).

Persistent or **stored** attacks are the ones that allow the user to inject malicious code directly into the web page and it will be saved on the server-side, that means everyone who will try to access the specific path where the code was injected, it will be affected by it. For example, if there is a simple form with a single field called Message and the attacker puts a malicious code and it is not encoded, but it is directly inserted into the database, everytime the users will see that message, the injected code will run on their browser. The injected code can be anything including stealing session information, cookies and then send them to the attacker as demonstrated in Figure 2.3.

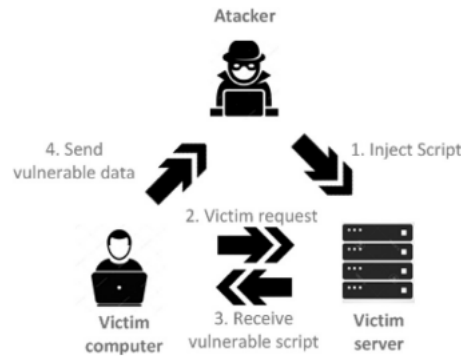


Figure 2.3: Scenario for XSS Persistent/Stored Attacks [SBK18]

In **Non-Persistent** or **reflected** attacks, users can steal victims' session cookies, enabling them to execute actions with the victim's permissions without requiring a password. To execute this attack successfully, the victim must be deceived into clicking on a malicious link, a tactic known as spoofing. This action redirects traffic to the attacker, allowing them to execute any malicious code using the victim's data, as illustrated in Figure 2.4.



Figure 2.4: Scenario for XSS Non-Persistent or Reflected Attacks [SBK18]

DOM-based attacks are a type of attack where the malicious payload is executed entirely on client-side within the Document Object Model of the web page. The victim accesses a malicious web link, which triggers a request to the "trusted" page. Subsequently, the response may alter the victim's DOM, enabling the attacker to obtain the cookies information and potentially impersonate the victim, thereby stealing their identity. This allows the attacker to perform actions with the victim's permissions. In Figure 2.5 can be seen the flow, how the hacker sends the malicious link, the user clicks on it, it sends a HTTP request to the website and then the response changes the victim's DOM and the attacker is able to steal all the important information it needs, in order to finalize his attack.

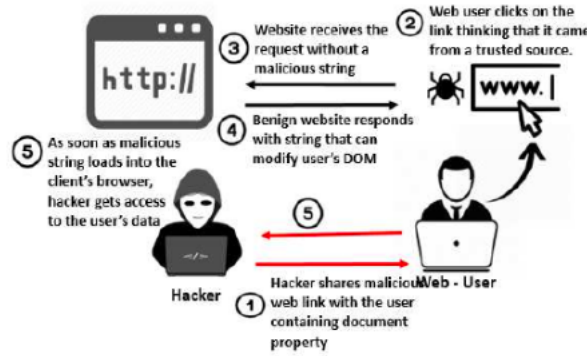


Figure 2.5: Scenario for XSS DOM-based Attacks [KG21]

Researchers and programmers have developed three main approaches: **Client-Side detection**, **Server-Side detection** and **Client-Server detection** [SBK18] in order to identify and mitigate XSS injection attacks, but neither one can guarantee 100% protection especially from modern XSS attacks. Most XSS attacks can be prevented by programmers by validating and sanitizing input. In figure 2.6 we can see how these three big approaches go more detailed.

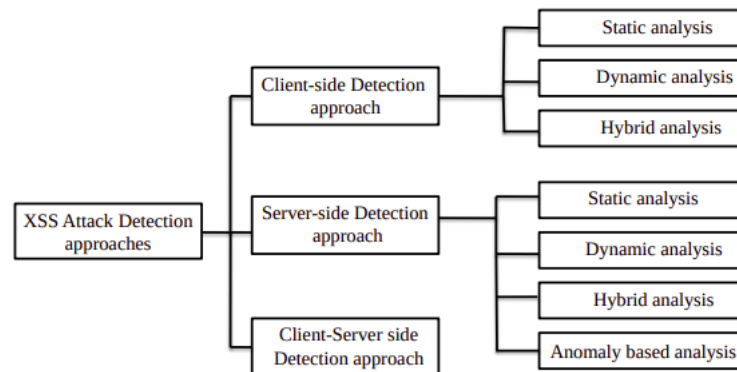


Figure 2.6: Main approaches [SBK18]

In the article [IEKY04], it is proposed a proxy server based method that uses two modes: request change mode and response change mode. It is a **Client-Side** approach and also it uses Dynamic analysis. In order to detect the XSS vulnerabilities, the response change mode contains special tags and it matches the characters from the collection, it will encode the message and will forward it safe to the client. In the request change mode, if there is a multi-parameter HTTP request, after the first parameter it generates a random number that will be inserted after the special characters and then it keeps increasing it by one for each parameter. After the response from the server is received it detects if the system has XSS scripts and also which parameter is vulnerable. It saves data about the requests, path, timestamp, parameters into a database for logging and future reference.

In a separate investigation [HYH⁺04], researchers introduce a **Server-Side** strategy utilizing a Hybrid analysis that combines the benefits of static and dynamic mechanism analysis [SBK18]. They introduce a system called WebSSARI (Web application Security by Static Analysis and Runtime Inspection), which functions as a framework by extending existing script languages, with a focus on supporting **PHP**, one of the most prevalent programming languages in 2004. This framework statically examines web applications for vulnerable code segments. If any such sections are identified, the framework provides runtime protection to secure them.

In the article [NSS09], the authors introduce an alternative perspective, employing a **Client-Server** approach. They argue that XSS attacks should not be viewed solely as input validation issues but rather as privilege escalation vulnerabilities. The system presented is called Document Structure Integrity (DSI) and it has four main components, two on Server-Side: Separation of trusted and user-generated data, serialization of the entire data with additional markups; two on Client-Side: Deserialization (which will reconstruct the document structure, without the untrusted data) and dynamic Parser Level Isolation (PLI) which ensures that when the pages are dynamically updated, DSI will track the untrusted data. There were no false positives for stored XSS attacks experiments, but there were a few for reflected XSS attacks.

Chapter 3

Threat Detection

3.1 The Problem

Many web applications neglect cybersecurity measures until they fall victim to attacks. This application prioritizes proactive detection by thoroughly examining all logs to identify potential threats before they escalate. While various methods like Artificial Intelligence or Regular Expressions can aid in detection, input validation remains the most robust defense mechanism. Today's attacks are increasingly sophisticated, often mimicking typical user behavior, making early detection more challenging.

3.2 SQL Injection Detection

SQL injection attacks were detected by using three different methods. First method was using multiple **Regular Expressions**, that was checking for comments and specific keywords in the queries or simple tautologies and the other two approaches are based on machine learning where I used two models: **Long Short Term Memory** and **Random Forests**, the last two will be detailed in subsections 3.4.

For Regular Expressions, I have three checks in the following order. The first check presented in Listing 3.1 is for comments or unfinished quotes, the second check, can be seen in Listing 3.2 is for tautologies, logical expressions that are always true and the third check shown in Listing 3.3 is for keywords (in PostgreSQL) that are usually used in order to extract sensitive data about the version of database or schema regarding the tables.

```
1 ( -- | \/ | * | \* | \/ | [ " ] { 2 , } \s* | \S ; \s* | S | ^ " [ ^ " ] * \ " $ | # | ^ 1 [ " \' ] | ^ 1 \s + [ \" \' ] )
```

Listing 3.1: Regex that searches for comments and quotes

Listing 3.1 is a regular expression for searching all types of comments, consecutive quotes, queries that end with 1 and then followed by a quotation mark and for

queries that contain ; in the middle of the query which means it executes multiple queries in same log.

```
1 \b(\w+)\s*=\s*\1\b|([\\"']\w+[\\"']\s+[\\"']\w+[\\"'])|(\w+)\s+LIKE\s+\3\b
```

Listing 3.2: Regex that searches for tautologies

Listing 3.2 is a regular expression that searches for tautologies. The first part `\b(\w+)\s*=\s*\1\b` searches if the sequence of characters before the equal sign is the same as the one after the equal sign. The second part `([\\"']\w+[\\"']\s+[\\"']\w+[\\"'])` searches if there are white spaces between two strings and the quotation marks. The third part `\w+\s+LIKE\s+\3\b` checks if the sequence of characters before the keyword **LIKE** (it is case insensitive) is the same as after the keyword.

```
1 \b(sleep|version|postgres|postgresql|schema|table|database|
  information_schema|pg_catalog|sysusers|systables|utl_inaddr|dbms_pipe|
  pg_sleep|rdb$\w_*)|waitfor|delay)\b
```

Listing 3.3: Regex that searches for database schema and sensitive information

Listing 3.3 is a regular expression that searches for most of the specific keywords regarding PostgreSQL and structure of the tables, getting the IP address of the server, version of database management system used.

3.3 XSS Detection

XSS Injection attacks were detected using three different methods. The first approach utilized multiple **Regular Expressions** to check for specific HTML keywords, non-ASCII characters, and suspected links in the source of images. The second method employed a **Long Short Term Memory (LSTM)** model for detection, with more details available in section 3.4. The third approach used a **Random Forest** model, also detailed in section 3.4.

For Regular Expressions method, there are 4 checks in the following order: non-ASCII characters after decoding (Listing 3.4), searching for specific JavaScript keywords that should not be found in the url (Listing 3.5), looking if the url/site/host doesn't start with http or https (Listing 3.7) and looking if the value of "src" tag from HTML ends with specific extension for photos/videos (Listing 3.8). All the presented regular expressions are case insensitive. In addition, there is also a check that counts the number of quotes (" or ') and if the number of quotes is not even, it indicates a missing payload so it will be predicted as an attack (Listing 3.6).

```
1 [^\x00-\x7F]
```

Listing 3.4: Regex for non-ASCII characters

The first regular expression that is used in the algorithm after the url was decoded, it's looking for any non-ASCII characters. ASCII table, is a table that contains all the letters, digits, special characters, it stands for American Standard Code and each character is on a specific position. The table starts from 0 and it contains 128 characters. In the Listing 3.4, I checked if the result of the decoding, is not between 00 and 7F in hexadecimal $7F_{(16)} = 127_{(10)}$.

```
1 alert|<(\/)?script(>)?|<marquee>|<br(\/)?>
```

Listing 3.5: Regex for specific keywords

Listing 3.5 is a regular expression that is looking for keywords that should not be found into the path of a request. Alert is a HTML tag that is usually used to notify the user about something special. Also it looks for the keyword 'script' which is the HTML tag that introduces JavaScript code into the page, 'marquee' is a non-standard HTML element and it is also deprecated and the last check is for 'br' element that introduces a new line into HTML document.

```
1 single_quotes_count = xss_query.count("'")
2 double_quotes_count = xss_query.count('"')
3 if single_quotes_count % 2 != 0 or double_quotes_count % 2 != 0:
4     detect_xss_attack += 1
```

Listing 3.6: Snippet of Python Code that checks for parity of quotes

After these 2 regular expressions, I check if the number of ' and " is odd, because that means the request is missing some payload or it was commented out or a string started with a single quote and ended with double quote which defines a possible attack. The Listing 3.6 shows the algorithm used in order to count and check if there is a possible attack.

```
1 (url|host|site)=(?! (http|https)) .*\. [a-zA-Z]{2,}'
```

Listing 3.7: Regex for non-standard URLs

The third regular expression, Listing 3.7, checks if the path contains one of the following keywords: "url", "host" or "site". Additionally, it verifies that after the equal sign, the path does not contain "http" or "https", thereby identifying non-standard URLs and potential XSS vulnerabilities.

```
1 src="[ ^"]*(?<!\. (mp4|jpg|png|gif|svg|web|ogg|mp3|wav|pdf)) (?<!\. (docx|
    xlsx|pptx|jpeg)) "
```

Listing 3.8: Regex for suspicious source of an image/video

The last regular expression used for detection, presented in Listing 3.8, searches for the content of attribute "src" which is usually used together with the HTML element "img" and it does not allow the user to inject a JavaScript code inside the source of an image. It checks for the most popular file extensions for images, videos and documents.

3.4 Common AI Model for detecting SQL injection and XSS attacks

3.4.1 Dataset Preparation

The datasets used for testing were obtained from Kaggle. For XSS, the dataset comprises 13686 logs [Sha20], with each log accompanied by a label (1 for XSS attack, 0 for safe logs). Regarding SQL Injection, the dataset [SAJ21] contains 148326 queries, each labeled as either 1 or 0 depending on whether it represents a SQL injection. The distribution of the queries is presented in Table 3.1.

	Attack	Not attack	Total
SQL dataset	77750	70576	148326
XSS dataset	7373	6313	13686

Table 3.1: Distribution of attacks and non-attacks in each dataset

3.4.2 Long Short-Term Memory Model

Convolutional Neural Networks (CNNs), which were introduced in 1988 [ZRSC15], represent a class of deep learning models predominantly employed for tasks like image recognition and classification. CNN structures are constructed with convolutional layers, pooling layers and fully connected layers. Convolutional layers function by applying filters to input data, allowing them to capture and recognize local patterns and features. Following this, pooling layers downsample the feature maps produced by the convolutional layers, thereby reducing their spatial dimensions. Figure 3.1 shows the architecture of CNN for images, making it easier to understand. Finally, fully connected layers process the flattened feature vectors to make predictions. While CNNs are traditionally associated with computer vision tasks, their principles can be extended to other domains, including natural language processing (NLP). By treating text as a sequence of tokens and leveraging one-dimensional convolutions, CNNs have demonstrated promising results for text classification [ZZL15], sentiment analysis and document summarization tasks.

Recurrent Neural Networks (RNN) are like a chain of blocks where each block remembers things from before [Sch19]. Unlike regular networks (CNNs for example) that only look at the current data, RNNs can also consider what happened earlier. This helps them understand sequences, like words in a sentence or steps in a process. In Figure 3.2, you can see how RNNs take into account not just the current input, but also what came before it. This shows how RNNs are able to remember and use past information to make sense of the current input. However, sometimes

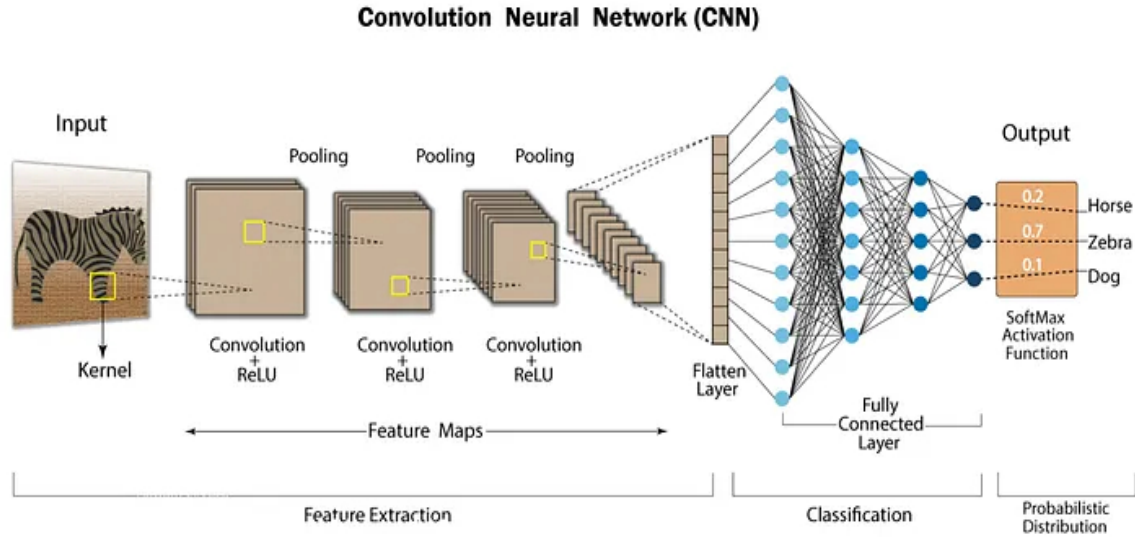
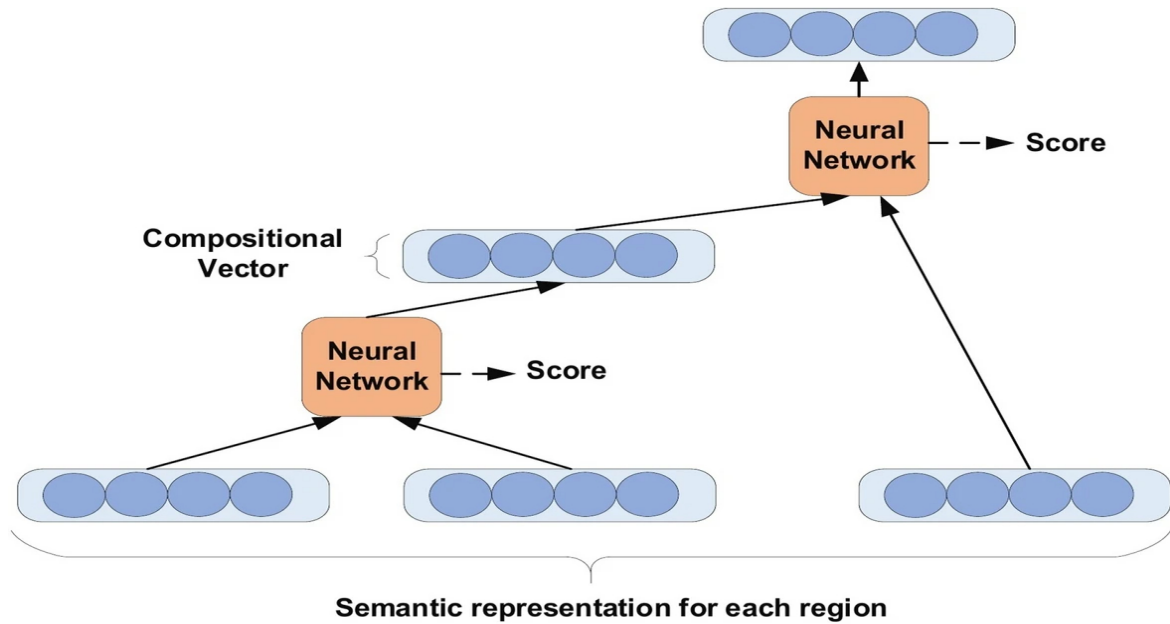


Figure 3.1: Architecture of CNN [Sha23]

RNNs forget important stuff if the sequence is too long. To fix this, we have special versions called Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs). These versions can remember important things for longer.

Figure 3.2: Architecture of RNN [AZH⁺21]

Long Short-Term Memory (LSTM) is a variant of RNN that includes an internal state and multiple gates. Initially proposed in 1997 [SJ19], the original LSTM lacked a forget gate. However, subsequent improvements, detailed in [GSC99], introduced the forget gate, ensuring the network retains vital information during longer sequences. Figure 3.3 depicts the LSTM architecture, showcasing the input, output and forget gates.

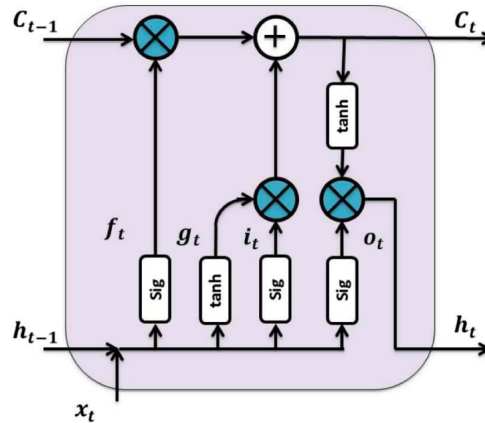


Figure 3.3: Representation of LSTM with forget gate[SJ19]

My approach involved utilizing **Bidirectional Long Short-Term Memory** (BI-LSTM), a variant of LSTM designed to process sequential data in both forward and backward directions. The model was developed using the open-source deep learning framework TensorFlow and Python's Keras library. I applied the same model to both XSS and SQL Injection detection tasks, as they share similar principles and the datasets exhibited significant overlap, eliminating the need for model modifications.

```

1 data_training = pd.read_csv("sample_data/XSS_dataset.csv")
2
3 # Data description
4 logging.info("Dataset Description:")
5 logging.info(data.describe())
6
7 logging.info("\nLabel Distribution:")
8 logging.info(data['Label'].value_counts())

```

Listing 3.9: Dataset Preparation

In Listing 3.9, first I load the dataset using Pandas library, then I print information about dataset and the distribution of labels, the number of attacks and the number of total queries.

```

1 # Preprocessing
2 X = data['Sentence'].str.lower()
3 y = data['Label']
4
5 # Tokenization
6 tokenizer_training = Tokenizer()
7 tokenizer_training.fit_on_texts(X)
8 X_seq = tokenizer_training.texts_to_sequences(X)
9
10 with open('tokenizer_training.pickle', 'wb') as handle:
11     pickle.dump(tokenizer_training, handle, protocol=pickle.

```

```

HIGHEST_PROTOCOL)
12
13 # Download the tokenizer file (using google colab)
14 files.download('tokenizer_training.pickle')
15
16 # Padding sequences
17 max_len = max([len(x) for x in X_seq])
18 X_pad = pad_sequences(X_seq, maxlen=max_len, padding='post')

```

Listing 3.10: Prepare the dataset and the tokenizer

The next step, described by Listing 3.10, it was to preprocess the data, by transforming all the characters from Sentence/Query into lowercase letters, then tokenize the data which transforms the text data into numerical data making it suitable for machine learning models and also I download the tokenizer in order to use it for loading the model in the backend. In the end, I add padding in order to make sure that all the input sequences are of the same length, requirement for batch processing in neural networks.

```

1 # Encoding labels
2 label_encoder_training = LabelEncoder()
3 y = label_encoder_training.fit_transform(y)
4
5 # Train-test split
6 X_train_set, X_test_set, y_train_set, y_test_set = train_test_split(X_pad
    , y, test_size=0.2, random_state=42)

```

Listing 3.11: Encoding labels and split the train-test dataset

In Listing 3.11, is presented the encoder for the label, which is essential for binary classification tasks and then, the dataset is split into train and test, the test set being 20% of the total logs of dataset. The random state parameter is the seed of randomness.

```

1 # Model definition
2 model_training = Sequential()
3 model_training.add(Embedding(input_dim=len(tokenizer_training.word_index)
    + 1, output_dim=100, input_length=max_len))

```

Listing 3.12: Model initialization

Listing 3.12 marks the initiation of the model. It begins by initializing a linear stack of layers within the neural network. Following this, it converts input tokens into dense vectors of a fixed size, designated as 100 (output_dim=100), with input_dim representing the vocabulary size plus one for padding.

```

1 model_training.add(Bidirectional(LSTM(128, return_sequences=True)))
2 model_training.add(Dropout(0.5))
3 model_training.add(Bidirectional(LSTM(64)))

```

```

4
5 model_training.add(Dense(64, activation='relu'))
6 model_training.add(Dropout(0.5))
7 model_training.add(Dense(1, activation='sigmoid'))

```

Listing 3.13: Adding layers to the model using BI-LSTM

In Listing 3.13 is presented the type of neural network that will be used: BI-LSTM. First, it adds BI-LSTM layer that will process the sequence both forwards and also backwards and it will return the sequence in order to be used for the further LSTM layer. After that, it performs a Dropout of 0.5 which means that it will randomly set 50% of inputs to zero at each update during training in order to prevent overfitting. The second layer does not return sequences, it just produces a single output vector for each input sequence. In the end, it adds a fully connected layer with ReLU activation, performs another dropout of 0.5 for overfitting and then it uses Sigmoid activation function for binary classification.

```

1 optimizer = Adam(learning_rate=0.001)
2 model_training.compile(loss='binary_crossentropy', optimizer=optimizer,
   metrics=['accuracy'])

```

Listing 3.14: Optimizer and Loss functions using BI-LSTM

The optimizer utilized is Adam, favored for NLP tasks due to potential variations in word occurrences across batches. This choice is evident in Listing 3.14, where Adam dynamically adjusts the learning rate during training. Additionally, binary cross-entropy serves as the loss function, well-suited for binary classification tasks. Performance metrics include accuracy.

```

1 callbacks = [
2     EarlyStopping(patience=3, monitor='val_loss', restore_best_weights=
   True),
3     ModelCheckpoint('best_model_bi_lstm.h5', monitor='val_loss',
   save_best_only=True)
4 ]

```

Listing 3.15: Callbacks

The purpose of callbacks, as illustrated in Listing 3.15, is to pause the training process if the validation loss fails to improve for three consecutive epochs. It also ensures the restoration of the best weights and saves the optimal model based on validation loss throughout the training phase.

```

1 history = model_training.fit(X_train_set, y_train_set, epochs=10,
   batch_size=32, validation_split=0.1, callbacks=callbacks)

```

Listing 3.16: Training of the model

In the Listing 3.16, it is presented the number of epochs, the batch size and also the validation dataset that is 10% of the training set. In order to monitor and save the best model, the earlier defined callbacks are used.

```

1 loss, accuracy = model.evaluate(X_test_set, y_test_set)
2 logging.info("Test Loss: %f", loss)
3 logging.info("Test Accuracy: %f", accuracy)
4
5 y_pred = (model.predict(X_test_set) > 0.5).astype("int32")
6
7 logging.info("\nClassification Report:")
8 logging.info(classification_report(y_test_set, y_pred))
9
10 logging.info("\nConfusion Matrix:")
11 logging.info(confusion_matrix(y_test_set, y_pred))

```

Listing 3.17: Model evaluation

The Listing 3.17 is about the evaluation of the model, by showing the loss and accuracy of the test set, a classification report and the confusion matrix of the predictions.

```

1 model.save('best_model_bi_lstm.h5')
2 files.download('best_model_bi_lstm.h5')

```

Listing 3.18: Save the model

In the end, Listing 3.18 presents the saving of the model and it downloads it using Python library: Google Colab.

3.4.3 Random Forest Model

Random Forest (RF) is a versatile **ensemble learning model** utilized in machine learning for various tasks including classification and regression. Its operation involves the construction of multiple decision trees during training, with the final prediction being the mode of the classes determined by individual trees [Bre01]. RF utilizes a method known as **bagging**, where every decision tree is trained on a randomized subset of the training data, with the possibility of replacement. This introduces variability into the training process, enhancing performance [Bia12] and mitigating overfitting risks. Additionally, Random Forests introduce further randomness by selecting a random subset of features for each split in the tree, thereby improving the ensemble's robustness. As shown in Figure 3.4, the algorithm generates multiple decision trees via bootstrap sampling and introduces randomness at each split to reduce correlation between trees. The final prediction is determined by aggregating the outputs of all trees, resulting in improved accuracy and robustness.

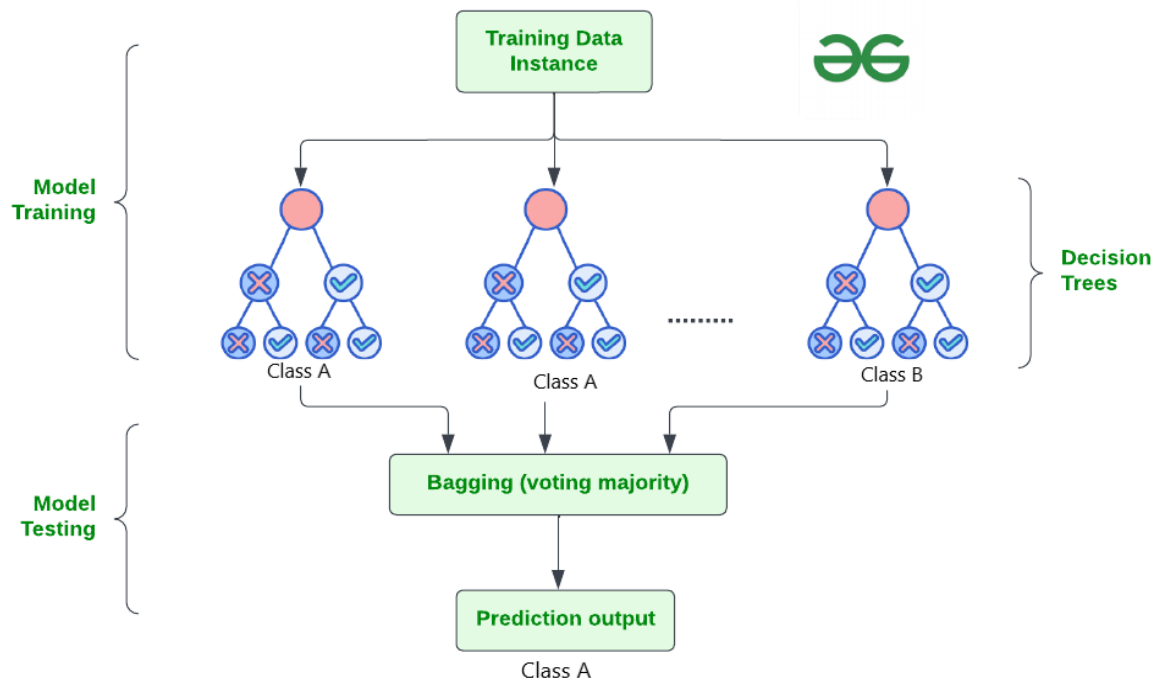


Figure 3.4: Architecture of Random Forest [Bha24]

The proposed algorithm was used to detect both SQL injection and Cross-Site Scripting (XSS) attacks. It is designed to identify these common web security threats and help protect against them. The algorithm works by analyzing patterns in the data to accurately detect instances of SQL injection and XSS attacks, using Random Forest Classifier with 100 decision trees.

```

1 data = pd.read_csv("sample_data/XSS_dataset.csv")
2
3 logging.info("Dataset Description:")
4 logging.info(data.describe())
5
6 logging.info("\nLabel Distribution:")
7 logging.info(data['Label'].value_counts())

```

Listing 3.19: Dataset Preparation

In Listing 3.19, first I load the dataset using Pandas library, then I print information about dataset and the distribution of labels. The used dataset is the same as the one from Bi-LSTM.

```

1 X = data['Query'].str.lower()
2 y = data['Label']
3
4 # Vectorization
5 vectorizer_rf = TfidfVectorizer()
6 X_tfidf = vectorizer_rf.fit_transform(X)

```

```

7
8 with open('vectorizer_sqli.pickle', 'wb') as handle:
9     pickle.dump(vectorizer_rf, handle, protocol=pickle.HIGHEST_PROTOCOL)

```

Listing 3.20: Preprocessing and Vectorization

After I read the data, I am starting to preprocess it, Listing 3.20, by transforming all the characters from Sentence/Query into lowercase letters and then I declare the vectorizer that will transform the data into numerical vectors. TF-IDF, short for Term Frequency-Inverse Document Frequency, is a numerical metric that indicates the significance of a word within a document compared to a corpus of documents. Eventually, I store the vectorizer for future use when loading the model into my web application.

```

1 # Encoding labels
2 label_encoder_train = LabelEncoder()
3 y = label_encoder_train.fit_transform(y)
4
5 # Train-test split
6 X_train_set, X_test_set, y_train_set, y_test_set = train_test_split(
    X_tfidf, y, test_size=0.2, random_state=42)

```

Listing 3.21: Encoding Labels and Train-Test split

The following step, as outlined in Listing 3.21, involves encoding the labels and dividing the dataset into training and testing sets. In this process, 80% of the data is allocated for training the model, while the remaining 20% is reserved for testing its performance.

```

1 # For SQL
2 model_train = RandomForestClassifier(bootstrap=False,
3                                     min_samples_leaf=2,
4                                     min_samples_split=5,
5                                     n_estimators=100,
6                                     random_state=42)
7 # For XSS
8 model_train = RandomForestClassifier(n_estimators=100, random_state=42)
9 # Training
10 model_train.fit(X_train_set, y_train_set)

```

Listing 3.22: Model definition and training

In Listing 3.22, a Random Forest Classifier model is defined for detecting SQL and XSS attacks. The model is instantiated with specific hyperparameters for SQL injection detection, while default parameters are used for XSS attack detection. Further details are provided below.

- **Model Definition for SQL:** The Random Forest Classifier is initialized with the following hyperparameters:

- `bootstrap=False`: The entire dataset is used for building each tree, instead of bootstrap samples.
 - `min_samples_leaf=2`: Ensures that each leaf node has at least 2 samples, which can help in preventing overfitting.
 - `min_samples_split=5`: Specifies that a node must have at least 5 samples to be considered for splitting.
 - `n_estimators=100`: Builds a forest of 100 decision trees.
 - `random_state=42`: Sets the seed for the random number generator, ensuring reproducibility of results.
- **Model Definition for XSS**: A simpler Random Forest Classifier is used with default parameters, except for:
 - `n_estimators=100`: Builds a forest of 100 decision trees.
 - `random_state=42`: Sets the seed for the random number generator, ensuring reproducibility of results.

```
1 y_pred = model.predict(X_test_set)
2 accuracy = model.score(X_test_set, y_test_set)
3 logging.info("Test Accuracy: %f", accuracy)
4
5 logging.info("\nClassification Report:")
6 logging.info(classification_report(y_test_set, y_pred))
7
8 logging.info("\nConfusion Matrix:")
9 logging.info(confusion_matrix(y_test_set, y_pred))
10
11 with open('random_forest_model.pickle', 'wb') as handle:
12     pickle.dump(model, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Listing 3.23: Run the test set and save the model

In the end, in Listing 3.23, I run the test set, print few metrics in order to see the accuracy and how well the model performed and then I save the model to use it later in my web application.

Chapter 4

Architecture

The architecture is the foundation of every application, shaping the performance, scalability and security. In the next subsections, I provide insights into the technologies chosen for their ability to address specific requirements, along with the logic behind their selection. Additionally, detailed diagrams illustrate key scenarios, offering a clear view of the application's functionality.

4.1 Analysis

4.1.1 Use Case Diagrams

In software engineering, use case diagrams are fundamental for understanding, visualizing and communicating the requirements of an application. They provide a view of the interactions between actors and the system. In Figure 4.1, I created a diagram with all the functionalities of the application, it contains 2 actors: a user who does not have an account and is able only to register; a registered user that is able to manage the audit file or manage the projects and also manage the connections in order to be able to see the charts, the graphs and the statistics regarding detection of the XSS attacks and SQL Injection. The purpose of this diagram is to illustrate the core functionalities of the application and provide a better overview of its capabilities.

The scenario for viewing the charts and run experiments is shown in Figure 4.2, it contains a single actor, the user; after the user selected the project on which it wants to run the experiments, it has to select the type (Realtime, Audit, Manual). For example, for Realtime type, it has to select multiple options like selecting the connection to the database, the table which contains the logs and also the model (Bi-LSTM, Random Forest or Regex) that wants to be used and after everything it was selected, the user will be able to see the charts and the comparison between the models.

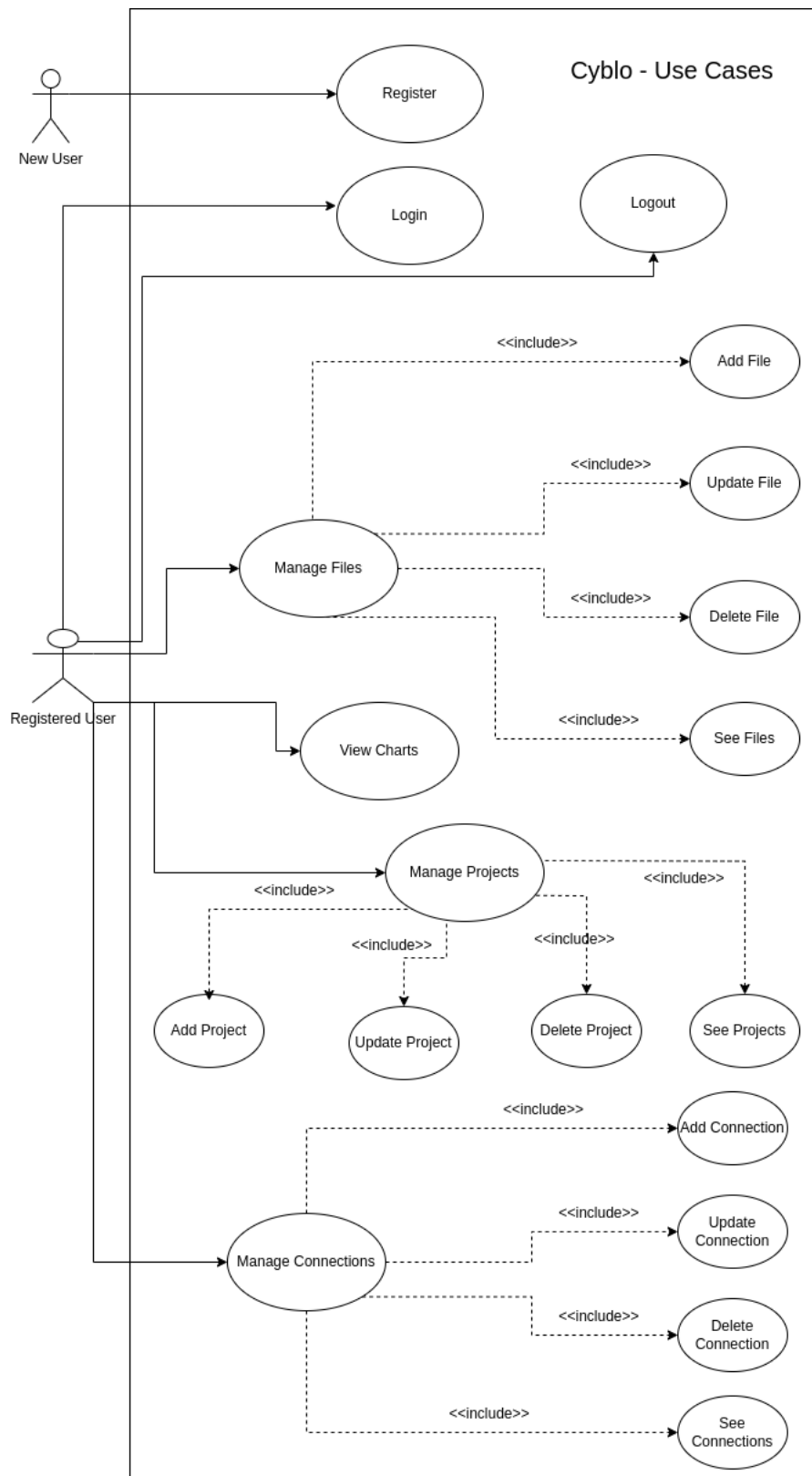


Figure 4.1: Use Case Diagram

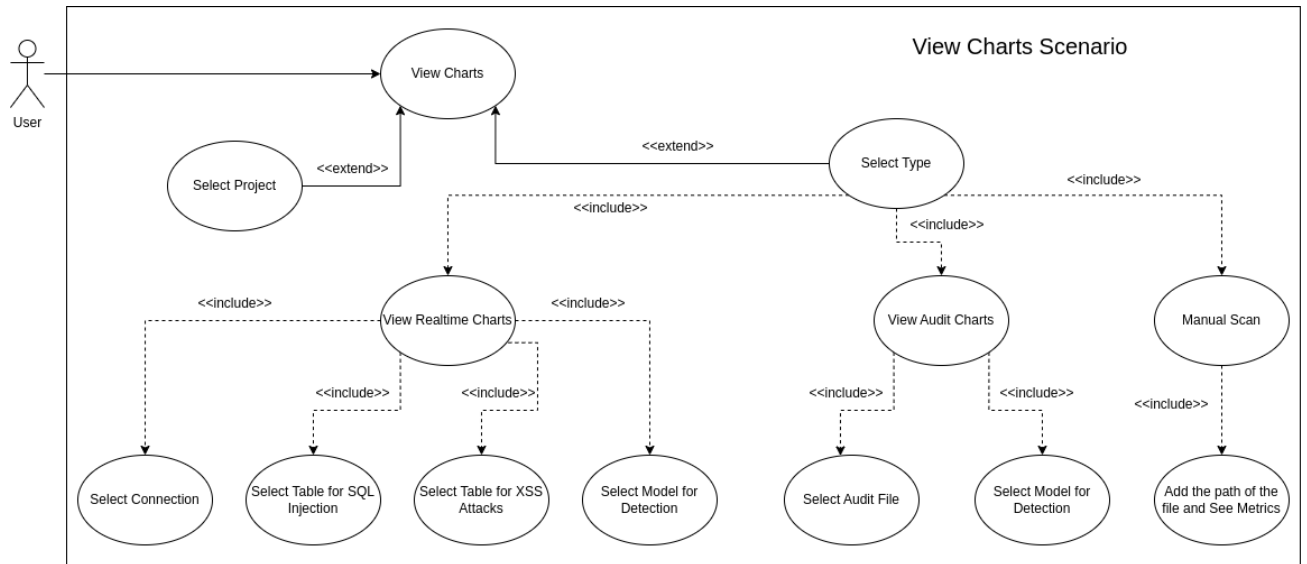


Figure 4.2: View Charts Scenario

4.2 Technology Stack

There are multiple types of web applications. The one I created is a **Dynamic Web Application**, this type contains the following concepts: **client-side** or **frontend**; **server-side** or **backend** and **database**. Each part will be explained in the below sections. The chosen technologies have been considered for scalability, security and also performance. Additionally, their active communities ensure access to a wealth of best practices, mitigating the risk of potential vulnerabilities.

Backend

Backend, also referred to as the **server-side**, forms the foundational layer of the application, used for important functions such as data processing, security; it acts as an intermediary between the frontend and the database, facilitating communication by processing JSON data received from the client-side. For instance, CORS (Cross-Origin Resource Sharing) is a security feature that restricts access to the backend from other servers unless explicitly specified. The backend contains sensitive data, processes information received from the client-side and returns responses along with status codes.

Python is a high-level programming language that was invented in the late 1980s by Guido Van Rossum which was a programmer that coded in ABC and he saw that ABC has a lot of disadvantages and it was pretty hard to debug such a code. Python is used for all types of applications, few of them are: Machine Learning, Web Applications, Game Development. A survey created by Stack Overflow, shows that Python is the third most used programming language in 2023 (Figure 4.3).

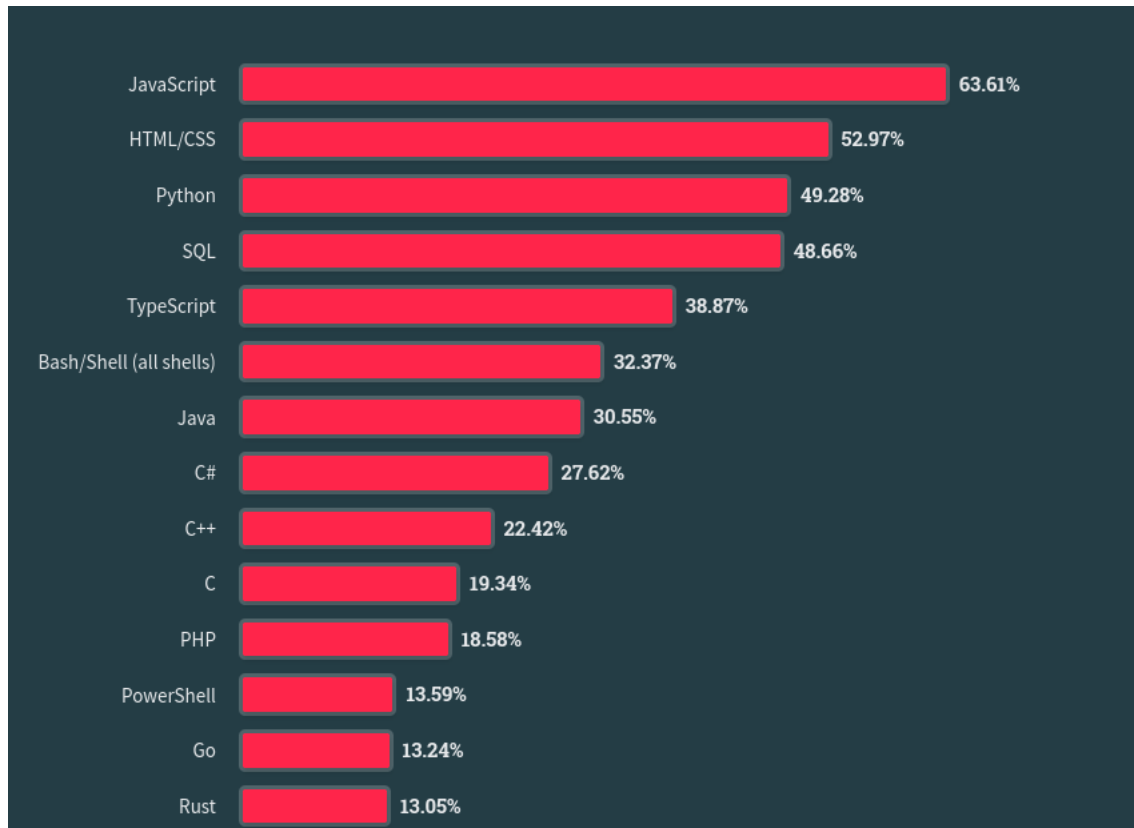


Figure 4.3: Top programming languages 2023 [Ove23]

Django Rest Framework is a framework in Python used for web development. The purpose of this framework is to make "life easier" for the programmers by giving a simple interface where coders don't have to "reinvent the wheel" in order to create a simple backend and they focus more on development. It was invented in 2003 and it is the most popular web framework for Python developers. [Ghi20]. In this thesis, I chose Django because it is easier to integrate the AI models using Python code and also it is easier to parse the logs in order to detect the attacks.

Frontend

Frontend, also known as the client-side, represents the user interface of the application that users interact with directly. It contains various components such as web pages, UI elements and interactive features that facilitate user engagement and interaction. It is responsible for presenting data to users in a nice way, enhancing the user experience and usability of the application. It leverages technologies such as HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript to create dynamic and interactive interfaces.

JavaScript is a high-level programming language, developed by NetScape and the first version of JavaScript was released in 1995 [Kei05]. The initial purpose of

this scripting language was to enhance static web applications by creating dynamic and interactive content [Wil04]. It is the most popular programming language used for client-side as we can see in the Figure 4.3.

AngularJS is a JavaScript framework developed by Google and released in 2016, it uses the Model-View-Controller architecture and it uses TypeScript which is a statically-typed superset of JavaScript [RVTS16]. This framework was created in order to simplify the development process for client-side and also scale web applications. The core blocks of an Angular application are components. A component defines a view, each component contains a **template**, which is written in HTML, a **style** for CSS and a **class** which contains the business logic and properties, written in TypeScript. In order to communicate with the backend for data retrieval, it uses **services** which with the help of HttpClient it makes requests to the backend and gets the necessary information which will be displayed or used later for the user. I chose this framework instead of **ReactJS** (JavaScript library) because Angular comes with a complete solution for everything I needed, including: form handling, built-in state management, routing, HTTP Client and also because it has TypeScript integration which makes development easier by catching errors/bugs earlier in the development process. Also, Angular's Ahead-Of-Time (AOT) compiles the application during the build process, resulting in faster rendering and improved load times.

Database

Database serves as the repository for persistent data storage within the application architecture. It plays an important role in storing and organizing various types of data required by the application. The database ensures data integrity, consistency and accessibility. It facilitates structured storage and retrieval of data, allowing the application to efficiently manage and manipulate information. It supports various operations such as insertion, retrieval, updating, and deletion of data. The scripting language used to write queries into the database is called SQL for traditional databases. There are also NoSQL databases which are used for "Big Data" are easier to be scaled [Cat11] and also performs better and faster for searching [LM13]. After careful consideration and comparison between SQL and NoSQL databases, I've decided to utilize a SQL database, specifically PostgreSQL, as the database management system (DBMS) for this web application.

PostgreSQL is a database management system, the successor to the INGRES relational database system [SR86] and there are multiple reasons why I chose to use it. It supports a relational data model, it is open-source, it has an active community of developers which helped me to follow the best practices, it is easy to scale and it performs well under heavy workloads. Compared to Microsoft SQL Server,

PostgreSQL supports user-defined data types, also it recognizes Ipv4 and Ipv6 data types. Also, PostgreSQL uses a system named Write Ahead Logging which ensures integrity and consistency in case of a crash [Con06].

4.3 Database Design and Management

In creating the architecture for the application, the foundational step was the meticulous design of the database structure using Django. This step laid the groundwork for storing and managing data and also establish the blueprint for the entire system's functionality and interactions. The database schema acts as the foundation, organizing how information moves and setting up connections between different parts of the system. In Figure 4.4, I created 4 entities: Project, ExternalDBConnection, File and Log. In order to make the application fully functionally and to connect all the data from a specific user and ensure the integrity of the data I also use the User entity that it was provided by Django framework. A user can have multiple projects, each project can contain at least an external database connection or at least a file and each file has multiple logs. The project entity contains only a reference to the user table and contains information about the project: name and description. An external database connection, contains information regarding the connection of the backend to a new instance of database and also it references a specific user and a specific project. The file entity is used for the audits of the application and contains the path, a service account key used for authentication on Google Cloud Storage, a type (XSS or SQLi) and few attributes for optimizing and speeding up the process of parsing the file, it can reference only a project. The Log entity, holds data about the query, the timestamp it was created, the prediction (level attribute), type (XSS or SQLi) and also it references a database connection or a file, it can't reference both.

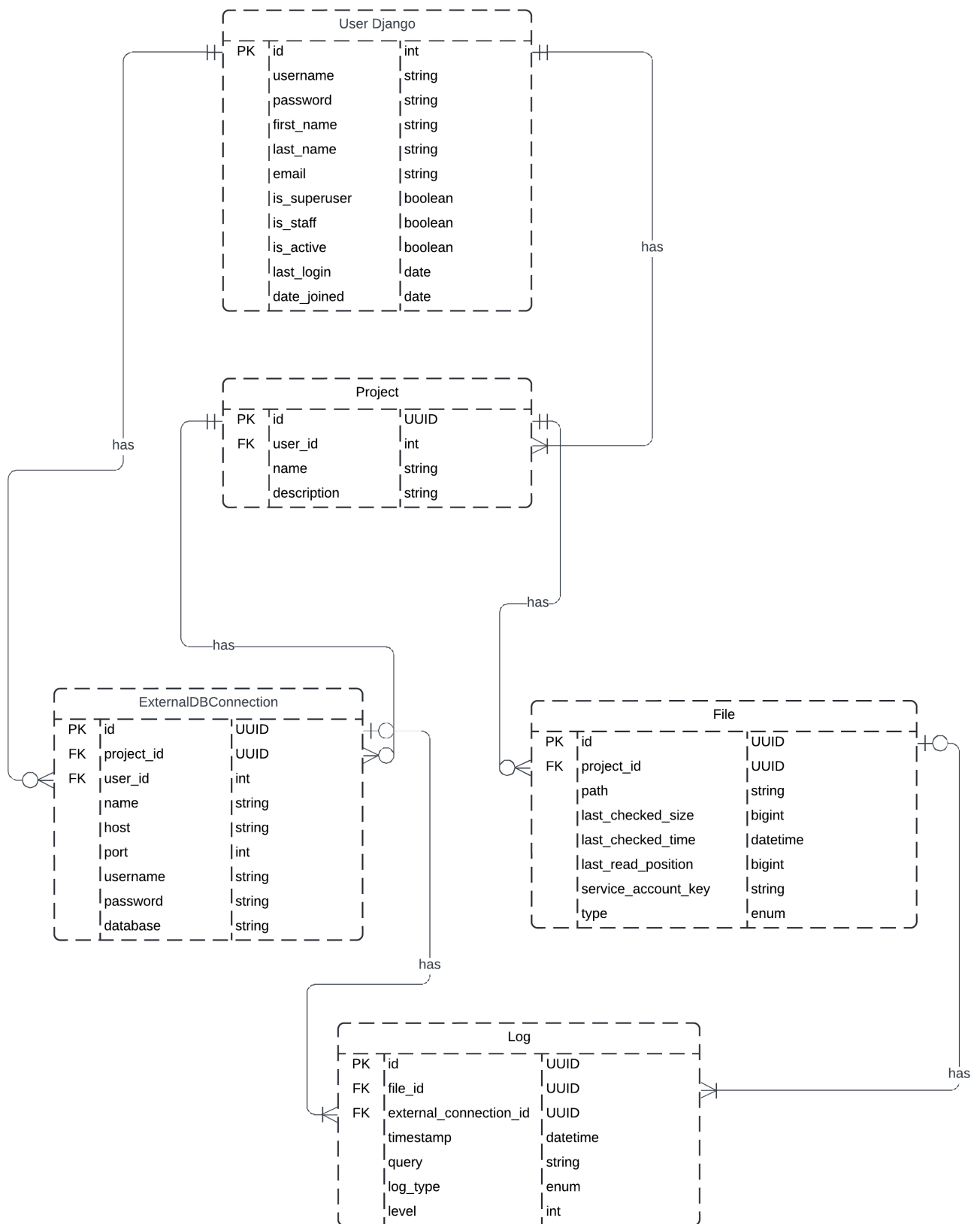


Figure 4.4: Database Diagram

Chapter 5

Cyblo

5.1 User Manual

There are 2 important pages involved in the user authentication: **register** and **login**. The user is able to register on the platform by accessing the web application, it will be redirected to /register and it has to complete the following fields considering the constraints shown in Table 5.1.

Field	Constraint
Username	Unique, at least 8 characters
Email	Must contain the following characters "@" and ".", at least 8 characters
Password	At least 8 characters long, at least one uppercase letter, lowercase letter, number and symbol
Confirm Password	Same as Password

Table 5.1: Fields and Constraints

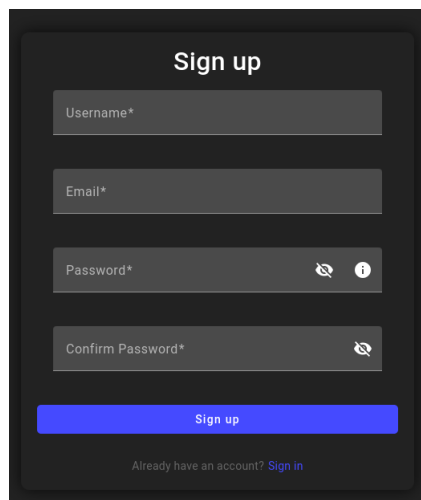


Figure 5.1: Register Component

In Picture 5.1, there are four fields, icons accompanying password fields to display password requirements and allow visibility without concealment, a Sign Up button, and an option to access the login page by clicking the Sign In button.

The user can access the login page at /login and is required to fill in two fields: Username and Password with the credentials set during registration. After completing the fields, the user must click the Sign In button, as depicted in Figure 5.2. If the user does not already have an account, they can navigate to the Sign up button at the bottom of the page, which will redirect them to /register. In case of incorrect credentials, an error message will be displayed.

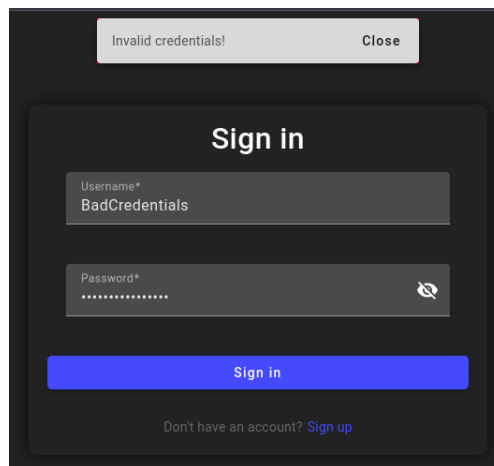


Figure 5.2: Login Component with error message

A user is able to create a new project after it signed in on the application. It has to click on button **Manage Projects**, Figure 5.3 on the left-side and then on the plus sign, a dialog will open, where the user has to insert a name and a description for the project.

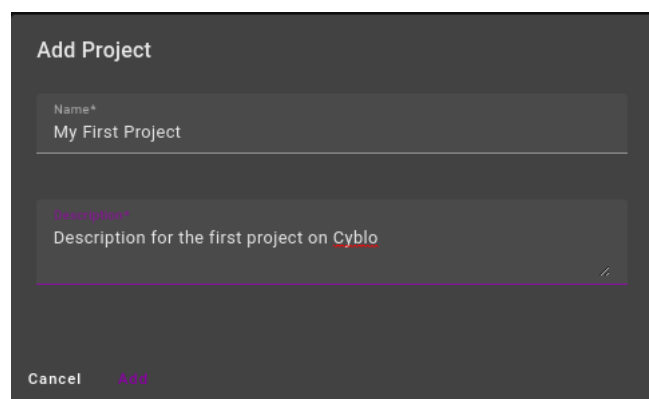


Figure 5.3: Add Project Component

After the project was added, the user can create a connection, by clicking on Manage Connections button and fill all the necessary fields, in order to connect to

an external instance of PostgreSQL or it can create a Google Cloud Audit. If the user wants to create a connection, it has to make sure that the username used in PostgreSQL has enough permissions to perform a select query over the interested tables and also to get all the tables from the database. If the user wants to create a Google Cloud Audit, it needs to provide the path of the file from Google Cloud Storage, a service account key that will be used to authenticate the system to Google Cloud in order to get the file from the bucket and the type of audit (SQL or XSS).

In order to modify a connection or add a new one, the user can click on Manage Connection button that is on the left-side of the screen. After that, it can see all the connections for all the projects. If it clicks on the plus button, it will be able to add a new connection, if it clicks on the pen button, it will be able to modify the existing connection and if it clicks on the trash button, a confirmation dialog will open and the user will have to write 'delete me' in order to confirm that it does not want that connection anymore as is shown in Figure 5.4. The flow is the same also for files or projects.

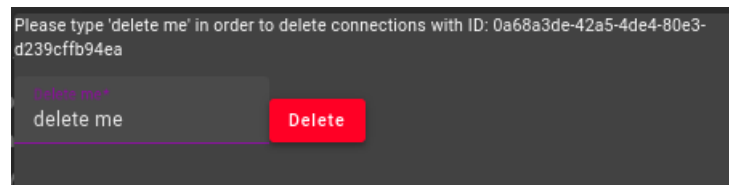


Figure 5.4: Delete confirmation for Connection

If the user wants to run the models on Realtime logs, it has to select the project, the Realtime option, the connection of the database, the table that contains SQL queries and the table that contains XSS queries and choose the model that wants to run (Bi-LSTM, Random Forest or Regex). After it selected all the options, 4 charts will be displayed, 2 for SQL injections and 2 for XSS Attacks. The first chart will depict the occurrence rate of detected SQL injection attacks relative to the total number of queries, based on the timestamp. The second chart will provide a comparative analysis of predictions made by all three algorithms (Bi-LSTM, Random Forest, and Regex), showcasing the number of XSS attacks detected by utilizing the selected option. As shown in Figure 5.5, both charts update in real time, for the first chart, with red line are the queries detected and with green line is the total number of queries and for the second chart, with red line are the attacks detected with Bi-LSTM, with blue line are the attacks detected with Random Forest and with green line detected by Regular Expressions. On the OY axis, is the count of queries and on OX axis is the timestamp. Also the charts are responsive, it can zoom in/out, it can hover over the points to see more precise data.

The user can also run the models on Audit files, by selecting the project, the

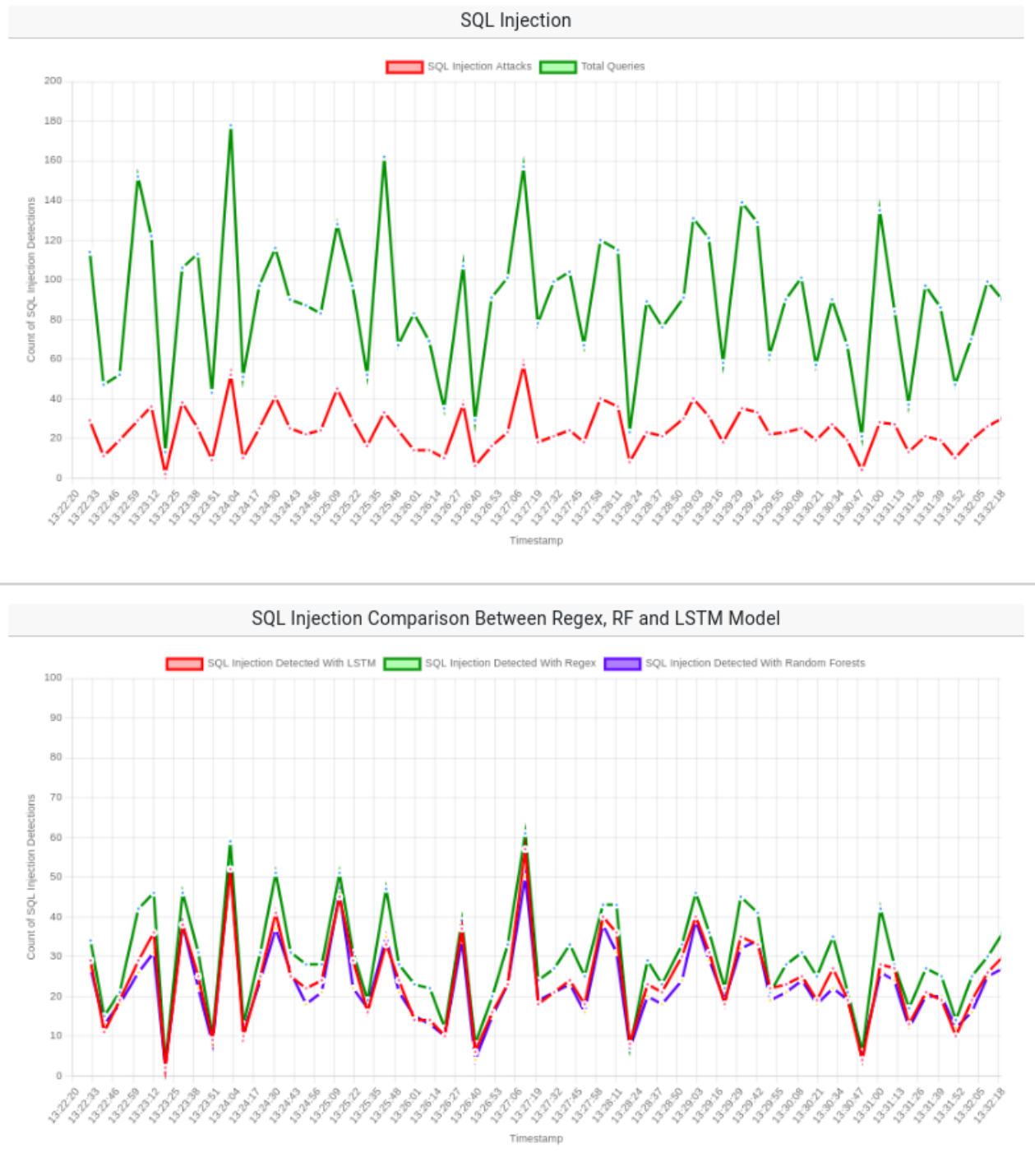


Figure 5.5: Dashboard that contains the two SQL Injection Charts

Audit option, the file and the model. After that, the system will create the two charts based on the type of Audit (XSS or SQL injection) and it will provide insight data about the detected attacks, the charts are the same as the ones in the Realtime option.

The last option to check the logs, without having to use Google Cloud Storage, is by selecting Manual Scanning. The user is able to add a file, from his machine and it will be parsed by all three algorithms (Regex, Bi-LSTM and RF) and the result will be in a table. If the file, is labeled, containing the query and then 0 or 1 if it is an attack, the table will contain also information about accuracy, precision, f1-score, recall, confusion matrix for both models as in Figure 5.6.

Results for "merged_output_with_flags.csv"			
Number of Records: 64833			
File Size: 4.578 MB			
Metric	LSTM	Regex	Random Forests
Total LSTM predicted as 1	32790	32393	33260
Total LSTM predicted as 0	32043	32440	33260
Accuracy	98.92%	98.02%	99.66%
F1 Score	98.94%	98.05%	99.67%
Recall	98.00%	96.53%	99.42%
Precision	99.90%	99.61%	99.92%
Time	346.923 seconds	0.292 seconds	6.930 seconds
Confusion Matrix			
LSTM			
Actual/Predicted	Predicted 0	Predicted 1	
Actual 0	31374	33	
Actual 1	669	32757	
Random Forests			
Actual/Predicted	Predicted 0	Predicted 1	
Actual 0	31379	28	
Actual 1	194	33232	
Regex			
Actual/Predicted	Predicted 0	Predicted 1	
Actual 0	31281	126	
Actual 1	1159	32267	

Figure 5.6: Table with metrics for manual selection

5.2 Dashboard Implementation

In order to generate charts and populate them I used on frontend the library chart.js and ng2-charts. Chart.js is a popular JavaScript library used for dashboards and ng2-charts is a wrapper to integrate Chart.js into the Angular application. First I defined a component called "sql-injection-chart", in the HTML I use the tag "canvas" in order to define the chart of type line. In the Typescript file, I define the zoom plugin and also the configuration for the chart like label names, datasets, the colors of the lines. After that, I implemented the method OnInit which gets trigger when the component is created and I make a HTTP request in order to get the data from the backend. It makes this request every 10 seconds, to make sure it gets always the Realtime data from the database. The data from the backend consist of queries labeled 0 or 1 depending if it is an attack or not and also a timestamp for that specific log. After I get the data from the backend, I update the chart, by concatenate the old data with the new data and set it for the datasets. Listing 5.1 describes the method updateChartData which is used after fetching new data from backend; on lines 3-4 I filter the records based on prediction and count them, then I create the points to draw them later on the chart, after that it concatenates the old data with the new one, on lines 9-10 it updates the chart for OX axis and then it force to detect the changes of the dataset and update the chart.

```

1  updateChartData(records: any[]): void {
2      const currentTime = moment();
3      const sqliCount = records.filter(record => record.prediction === 1).
        length;
4      const totalQueriesCount = records.length;
5      const pointDataSqli = { x: currentTime.valueOf(), y: sqliCount };
6      const pointDataTotalQueries = { x: currentTime.valueOf(), y:
        totalQueriesCount };
7      this.lineChartData.datasets[0].data = [...this.lineChartData.datasets
        [0].data, pointDataSqli];
8      this.lineChartData.datasets[1].data = [...this.lineChartData.datasets
        [1].data, pointDataTotalQueries];
9      this.lineChartOptions.scales!['x']!.min = moment().valueOf();
10     this.lineChartOptions.scales!['x']!.max = moment().add(10, 'minutes')
        .valueOf();
11     this.cdr.detectChanges(); // Force change detection
12     if (this.chart) {
13         this.chart.update();
14     }
15 }

```

Listing 5.1: Update Chart Data for SQL Injection Chart

5.3 User Authentication

The user authentication mechanism revolves around JWT tokens for secure authentication. These tokens are stored in HTTP-only cookies on the client side, ensuring they cannot be accessed or modified by client-side scripts, enhancing security. On the backend, I have implemented a middleware component that automatically retrieves the JWT token from the cookie and adds it to the Authorization Header of each HTTP request. This allows to utilize the [IsAuthenticated] decorator in Django app's views, restricting access to authenticated users only. Additionally, on the frontend, I have an authentication guard that safeguards routes requiring authentication. It validates the expiration time of the JWT token stored in the cookie by making a HTTP request to the backend and redirects users back to the login page if the token has expired. This dual-layered approach ensures that the application maintains robust security measures while providing a seamless and protected user experience. In the Listing 5.2, is presented a snippet from the described flow, that checks if the token exists, it decodes it and checks looks for the 'exp' key which contains the expiration time. If it finds an expiration time, it sends back the response with the date and on frontend I check in my AuthGuard if it expired or not.

```

1 def get_token_expiration(request):
2     token = request.COOKIES.get('jwt')
3     if not token:
4         return JsonResponse({'error': 'Token not found'}, status=status.
HTTP_401_UNAUTHORIZED)
5
6     try:
7         decoded_token = jwt.decode(token)
8         expiration_timestamp = decoded_token.get('exp', None)
9         if not expiration_timestamp:
10            return JsonResponse({'error': 'Expiration time not found in
token'}, status=status.HTTP_400_BAD_REQUEST)
11
12         expiration_date = datetime.fromtimestamp(expiration_timestamp)
13         return JsonResponse({'expiration_date': expiration_date}, status=
status.HTTP_200_OK)
14
15     except jwt.InvalidTokenError:
16         return JsonResponse({'error': 'Invalid token'}, status=status.
HTTP_401_UNAUTHORIZED)

```

Listing 5.2: Implementation of getting the JWT expiration date

5.4 Manual Testing

Manual testing is an essential component of the validation process for the Cyblo application, ensuring its functionality and reliability. This section outlines the procedures and steps taken to manually test the web application.

The Cyblo application allows users to select a project and specify the type of audit they wish to perform: Realtime, Audit, or Manual Scanning. Depending on the selected type, users will follow different workflows:

5.4.1 Realtime Audit

- Users select the project, the connection and the specific table for Cross-Site Scripting (XSS) and SQL Injection (SQLi) checks.
- For real-time scanning, users choose among different detection models: Regular Expressions, Bidirectional Long Short-Term Memory (Bi-LSTM) and Random Forest (RF).
- The application displays two charts per attack type (XSS and SQLi), showing real-time metrics.

5.4.2 Audit

- Users upload a file for audit analysis in Google Cloud Storage and also provide the service account key.
- Similar to real-time scanning, users can select from the detection models (Regex, Bi-LSTM, RF).
- The application processes the file and presents the findings through two detailed charts for the selected type of attack.

5.4.3 Manual Scanning

- Users input the path to the specific data or logs they wish to analyze manually.
- The application scans the input data and displays a table with relevant metrics, summarizing the security assessment findings.

To ensure the application performs as expected, the following manual testing procedures were implemented:

5.4.4 User Interface (UI) Testing

- Verify that all UI elements (buttons, charts, tables) are present and function correctly.
- Ensure that the navigation between different sections (project selection, audit types) is smooth and intuitive.

5.4.5 Functional Testing

- Test the process of selecting projects and audit types, ensuring that the appropriate options and models are available for selection.
- Validate the accuracy and responsiveness of the real-time charts for both XSS and SQLi detections.
- Confirm that file uploads for audits are processed correctly and that the resulting analysis is accurately reflected in the charts.
- Check the manual scanning process by inputting various paths and verifying that the output metrics are correctly generated and displayed.

5.4.6 Model Selection and Analysis

- Assess the functionality of selecting different models (Regex, Bi-LSTM, RF) and ensure that the application correctly applies these models during real-time and audit processes.
- Evaluate the performance and accuracy of the models by comparing the application's results with expected outcomes from known datasets.

Chapter 6

Results

6.1 Dataset

The datasets used for testing were sourced from Kaggle and GitHub. For XSS, the dataset consist of 64833 logs, each log contains a label (1 for XSS attack, 0 for safe logs). I used the dataset that the authors of the article [FLLH18] created it. There are 2 csv files, one called **xssed.csv** that contains only XSS attacks and the other one is called **dmzo_normal.csv** that contains simple logs. I merged those two datasets. For SQL Injection the dataset was obtained from Kaggle [SAJ21] it contains 30905 queries and each query is labeled 1 or 0 depending if it s a SQL injection or not, 38% of the dataset includes SQL injection queries. The distribution of the queries, can be seen in the Table 6.1.

	Attack	Not attack	Total
SQL dataset	11382	19537	30919
XSS dataset	33426	31407	64833

Table 6.1: Distribution of attacks and non-attacks in each dataset

6.2 Performance Evaluation for Detection Algorithms

In the following subsections, I present metrics about the detection algorithms for XSS Attacks and also for SQL injection, for all three proposed models: Bi-LSTM, RF and Regex. It can be seen that in order to detect SQL injection, Bidirectional Long Short-Term Memory model is more accurate than the other two approaches, but for XSS attacks detection, Random Forests model is the best model in terms of detection and also time comparing to Bi-LSTM.

6.2.1 Accuracy

Accuracy quantifies the overall precision of the model's predictions. It is calculated using Equation 6.1. In Table 6.2 is presented the comparison between Bi-LSTM, RF and Regex for detecting SQL injection and XSS attacks. For both types of attacks, Bi-LSTM and RF show higher accuracy compared to Regex, with Bi-LSTM outperforming RF for SQL injections, and RF slightly outperforming Bi-LSTM for XSS attacks.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (6.1)$$

	Bi-LSTM	RF	Regex
SQLi	99.57%	95.80%	87.44%
XSS	98.92%	99.66%	98.02%

Table 6.2: Accuracy comparison for SQL and XSS using Bi-LSTM, RF and Regex

6.2.2 Precision

Precision denotes the ratio of true positive predictions to all positive predictions made by the model. It is calculated using Equation 6.2. Table 6.3 shows the precision comparison for SQL and XSS detection. The Bi-LSTM model achieved higher precision for SQL injection detection compared to RF and Regex, while RF showed a slight edge over Bi-LSTM for XSS detection.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (6.2)$$

	Bi-LSTM	RF	Regex
SQLi	99.89%	99.77%	75.78%
XSS	99.90%	99.92%	99.61%

Table 6.3: Precision comparison for SQL and XSS using Bi-LSTM, RF and Regex

6.2.3 Recall

Recall quantifies the ratio of true positive predictions to all actual positive instances in the dataset. It is calculated using Equation 6.3. In Table 6.4, the recall comparison for SQL and XSS detection is presented. The Bi-LSTM model has higher recall for SQL injection detection compared to RF and Regex, while RF achieved the highest recall for XSS detection.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (6.3)$$

	Bi-LSTM	RF	Regex
SQLi	98.94%	88.79%	96.84%
XSS	98.00%	99.42%	96.53%

Table 6.4: Recall comparison for SQL and XSS detection using Bi-LSTM, RF and Regex

6.2.4 F1-Score

The F1-Score, being the harmonic mean of precision and recall, offers a balanced assessment of a model's performance. It is calculated using Equation 6.4. Table 6.5 shows the F1-Score comparison for SQL and XSS detection. The Bi-LSTM model achieved the highest F1-Score for SQL injection detection, while RF had the highest F1-Score for XSS detection, indicating a better balance between precision and recall.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (6.4)$$

	Bi-LSTM	RF	Regex
SQL	99.41%	93.96%	85.02%
XSS	98.94%	99.67%	98.05%

Table 6.5: F1-Score comparison for SQL and XSS detection using Bi-LSTM, RF and Regex

6.2.5 Time

Time is an important metric that measures the computational efficiency of the detection algorithms. In Table 6.6, the time comparison for SQL and XSS detection using Bi-LSTM and Regex models is presented. The Regex model demonstrated significantly faster detection times for SQL and XSS, highlighting its computational efficiency compared to the Bi-LSTM and RF models, but also the RF was much significantly faster than the Bi-LSTM.

	Bi-LSTM	RF	Regex
SQL	118.46 seconds	2.38 seconds	0.137 seconds
XSS	309.832 seconds	6.93 seconds	0.267 seconds

Table 6.6: Time comparison for SQL and XSS detection using Bi-LSTM, RF and Regex

6.2.6 Confusion Matrix

The confusion matrix offers detailed insights into the performance of detection algorithms, providing information on true positives, true negatives, false positives, and false negatives. Table 6.7 illustrates the confusion matrix for SQL and XSS detection utilizing Bi-LSTM, RF, and Regex models. For SQL injection detection, the Bi-LSTM model demonstrated higher counts of true positives and true negatives compared to the Regex model, whereas RF exhibited the best balance for XSS detection.

	TN	FP	FN	TP
Bi-LSTM SQL	19525	121	12	11261
RF SQL	19514	1276	23	10106
Regex SQL	16014	360	3523	11022
Bi-LSTM XSS	31374	669	33	32757
RF XSS	31379	194	28	33232
Regex XSS	31281	1159	126	32267

Table 6.7: Confusion matrix for SQL and XSS detection using Bi-LSTM, RF and Regex

Chapter 7

Conclusion

In conclusion, the evaluation of detection algorithms for SQL Injection and XSS Attacks using Bi-LSTM, RF, and Regex models revealed valuable insights into their effectiveness in mitigating cyber threats. The consistent superiority of AI models over Regex in accuracy, precision, and recall for SQL injections and XSS attacks underscores the potential of advanced machine learning techniques in enhancing web security. Notably, for XSS attacks, the RF model slightly outperformed Bi-LSTM, demonstrating its efficacy in this context.

The results indicate that while Bi-LSTM is highly effective for SQL injections, RF offers a robust alternative, particularly for XSS attacks. This suggests that different models may be better suited for different types of cyber threats, highlighting the importance of a tailored approach in cybersecurity.

Moreover, the visual representations of these results in the web application through the created charts provided an intuitive understanding of the performance between the models. This user-friendly presentation enhances accessibility, making it easier for users to interpret and utilize the findings.

Looking ahead, there is significant room for further improvement in detection capabilities. Future research can focus on leveraging larger datasets to train more robust models and exploring faster machine learning architectures to enhance computational efficiency without compromising accuracy. Additionally, combining multiple models in a hybrid approach could potentially offer even greater accuracy and reliability in detecting and mitigating various types of cyber attacks.

Overall, the findings highlight the practical effectiveness of advanced machine learning techniques, especially Bi-LSTM and RF, in identifying and mitigating cyber threats, thus enhancing web application security.

Bibliography

- [AS12] Mohammad Shkoukani Hesham Abusaimh and Mohammad Shkoukani. Survey of web application and internet security threats. *International journal of computer science and network security*, 12(12):67–76, 2012.
- [AY17] Zainab S Alwan and Manal F Younis. Detection and prevention of sql injection attack: a survey. *International Journal of Computer Science and Mobile Computing*, 6(8):5–17, 2017.
- [AZH⁺21] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-dujaili, Ye Duan, Omran Al-Shamma, José I. Santamaría, Mohammed Abdurraheem Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8, 2021.
- [Bha24] Susmit Sekhar Bhakta. Random forest algorithm in machine learning. <https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/>, February 22, 2024. Accessed: June 7, 2024.
- [Bia12] Gérard Biau. Analysis of a random forests model. *The Journal of Machine Learning Research*, 13(1):1063–1095, 2012.
- [Bre01] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.
- [Cat11] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.
- [Con06] Tim Conrad. Postgresql vs. mysql vs. commercial databases: It’s all about what you need, 2006.
- [FLLH18] Yong Fang, Yang Li, Liang Liu, and Cheng Huang. Deepxss: Cross site scripting detection based on deep learning. In *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, ICCAI ’18*,

- page 47–51, New York, NY, USA, 2018. Association for Computing Machinery.
- [Ghi20] Devendra Ghimire. Comparative study on python web frameworks: Flask and django. 2020.
- [GSC99] F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2, 1999.
- [HO] William GJ Halfond and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing sql injection attacks.
- [HVO⁺06] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1, pages 13–15. IEEE Piscataway, NJ, 2006.
- [HYH⁺04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web, WWW '04*, page 40–52, New York, NY, USA, 2004. Association for Computing Machinery.
- [IEKY04] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *18th International Conference on Advanced Information Networking and Applications, 2004. AINA 2004.*, volume 1, pages 145–151 Vol.1, 2004.
- [Kei05] Jeremy Keith. A brief history of javascript. *DOM Scripting: Web Design with JavaScript and the Document Object Model*, pages 3–10, 2005.
- [KG21] Jasleen Kaur and Urvashi Garg. A detailed survey on recent xss web-attacks machine learning detection techniques. pages 1–6, 10 2021.
- [LM13] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of sql and nosql databases. In *2013 IEEE Pacific Rim conference on communications, computers and signal processing (PACRIM)*, pages 15–19. IEEE, 2013.
- [LX11] Xiaowei Li and Yuan Xue. A survey on web application security, 2011.

- [Med13] MHasP Medhane. R-wasp: real time-web application sql injection detector and preventer. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 2(5):327–330, 2013.
- [NSS09] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. 01 2009.
- [Ove23] Stack Overflow. 2023 developer survey. <https://survey.stackoverflow.co/2023/>, June 13, 2023. Accessed: April 12, 2024.
- [RVTS16] Miguel Ramos, Marco Tulio Valente, Ricardo Terra, and Gustavo Santos. Angularjs in the wild: A survey with 460 developers. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 9–16, 2016.
- [SAJ21] SAJID576. Sql injection dataset. <https://www.kaggle.com/datasets/sajid576/sql-injection-dataset>, 2021. Accessed: April 4, 2024.
- [SBK18] Upasana Sarmah, DK Bhattacharyya, and Jugal K Kalita. A survey of detection methods for xss attacks. *Journal of Network and Computer Applications*, 118:113–143, 2018.
- [Sch19] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *ArXiv*, abs/1912.05911, 2019.
- [Sha20] Syed Saqlain Hussain Shah. Cross site scripting xss dataset for deep learning. <https://www.kaggle.com/datasets/syedsaqlainhussain/cross-site-scripting-xss-dataset-for-deep-learning>, 2020. Accessed: March 17, 2024.
- [Sha23] Nafiz Shahriar. What is convolutional neural network — cnn (deep learning). <https://nafizshahriar.medium.com/what-is-convolutional-neural-network-cnn-deep-learning-b3921bd> February 1, 2023. Accessed: May 31, 2024.
- [She10] Mike Shema. *Seven deadliest web application attacks*. Syngress, 2010.
- [SJ19] Kamilya Smagulova and Alex Pappachen James. A survey on lstm memristive neural network architectures and applications. *The European Physical Journal Special Topics*, 228(10):2313–2324, 2019.
- [SR86] Michael Stonebraker and Lawrence A Rowe. The design of postgres. *ACM Sigmod Record*, 15(2):340–355, 1986.

- [WBE20] Allen Wirfs-Brock and Brendan Eich. Javascript: the first 20 years. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–189, 2020.
- [Wil04] Paul Wilton. *Beginning JavaScript*. John Wiley & Sons, 2004.
- [ZRSC15] Ákos Zarándy, Csaba Rekeczky, Peter Szolgay, and Leon O. Chua. Overview of cnn research: 25 years history and the current trends. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 401–404, 2015.
- [ZZL15] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1, NIPS’15*, page 649–657, Cambridge, MA, USA, 2015. MIT Press.