AI-BASED THREAT DETECTION SYSTEM

A PROJECT REPORT

Submitted by

GOWTHAMI K

(2022115026)

GABRIELLA D

(2022115055)

A report for the dissertation-I
submitted to the faculty of

INFORMATION AND COMMUNICATION ENGINEERING

in partial fulfillment

for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE AND TECHNOLOGY

COLLEGE OF ENGINEERING GUINDY

ANNA UNIVERSITY

CHENNAI 600 025

NOVEMBER 2024

ANNA UNIVERSITY

CHENNAI - 600 025

BONAFIDE CERTIFICATE

Certified that this project report titled **“AI-BASED THREAT DETECTION SYSTEM”** is the bonafide work of **GOWTHAMI K (2022115026) and GABRIELLA D (2022115055)** who carried out project work under my supervision. Certified further that to the best of my knowledge and belief, the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or an award was conferred on an earlier occasion on this or any other candidate.

PLACE:CHENNAI
DATE:

Dr. K. INDRA GANDHI
ASSOCIATE PROFESSOR
PROJECT GUIDE
DEPARTMENT OF IST, CEG
ANNA UNIVERSITY
CHENNAI 600025

COUNTERSIGNED

Dr. S. SWAMYNATHAN
HEAD OF THE DEPARTMENT
DEPARTMENT OF INFORMATION SCIENCE AND TECHNOLOGY
COLLEGE OF ENGINEERING GUINDY
ANNA UNIVERSITY
CHENNAI 600025

ABSTRACT

This project addresses two critical web application vulnerabilities—Broken Access Control (BAC) and SQL Injection (SQLi)—both consistently ranked among the top security risks by OWASP. BAC vulnerabilities occur when systems fail to enforce proper permissions, enabling unauthorized users to access restricted resources or perform privileged actions. These vulnerabilities can manifest as Horizontal Access Control (HAC), where users access others' data, and Vertical Access Control (VAC), where lower-privileged users access administrative resources. Additionally, Missing Function-Level Access Control (MFAC) and Insecure Direct Object References (IDOR) expose sensitive functions and identifiers, increasing the potential for data breaches and privilege escalations. By incorporating BAC detection into this tool, the project aims to mitigate a wide range of access control issues, enhancing system security against unauthorized access.

For SQL Injection (SQLi), the project introduces a machine learning-based detection model that utilizes dense neural networks to analyze SQL queries, classifying them as vulnerable or safe. Users can upload SQL files for automated analysis, with the tool generating comprehensive reports that identify vulnerabilities, categorize query types, and provide tailored mitigation strategies. To support risk assessment, the tool integrates graph-based visualizations for intuitive displays of vulnerability distributions. Unlike traditional, resource-intensive methods, this model leverages machine learning to process large datasets quickly, making it scalable and efficient. Initial testing validates the model's effectiveness in detecting SQLi vulnerabilities, offering developers and security professionals a proactive, user-friendly resource to safeguard applications. The tool's adaptability suggests potential for future extensions to address other code vulnerabilities, supporting broader security applications.

ABSTRACT TAMIL

ACKNOWLEDGEMENT

It is our privilege to express our deepest sense of gratitude and sincere thanks to **Dr. K. INDRA GANDHI,** Associate Professor, Project Guide, Department of Information Science and Technology, College of Engineering, Guindy, Anna University, for her constant supervision, encouragement, and support in my project work. We greatly appreciate the constructive advice and motivation that was given to help us advance my project in the right direction.

We are grateful to **Dr. S. SWAMYNATHAN,** Professor and Head, Department of Information Science and Technology, College of Engineering Guindy, Anna University for providing us with the opportunity and necessary resources to do this project.

We would also wish to express our deepest sense of gratitude to the Member of the Project Review Committee: **Dr. M. VIJAYALAKSHMI** Professor, Department of Information Science and Technology,College of Engineering Guindy, Anna University, for their guidance and useful suggestions that were beneficial in helping us improve our project.

We also thank the faculty member and non teaching staff members of the Department of Information Science and Technology, Anna University, Chennai for their valuable support throughout the course of our project work.

GOWTHAMI K
(2022115026)
GABRIELLA D
(2022115055)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

BAC	Broken Access Control
VAC	Vertical Access Contorl
HAC	Horizontal Access Control
MFAC	Missing Function Level Access Control
IDOR	Insecure Direct Object Reference
SQL	Structured Query Language
SQLi	Structured Query Language Injection
NLP	Natural Language Processing
SIEM	Security Information and Event Management

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Broken Access Control (BAC) is a critical security vulnerability that occurs when a system fails to restrict user permissions properly, allowing attackers to access highly sensitive data or functionalities. These types of vulnerabilities are commonly exploited in web applications where attackers manipulate user permissions to gain access to highly sensitive information, modify data or perform functionalities that bypasses their roles.

As per OWASP's top ten vulnerabilities lastly released on 2021, Broken Access Control is at the top of the vulnerabilities listed, having moved from 5th position since 2017. With such an increasing surge of applicants having tested positive for this specific vulnerability, there is a need for automated tools that detects these vulnerabilities effectively. As opposed to traditional tools, a tool developed using Machine Learning techniques would be less prone to error, more adaptable, and faster.

SQL Injection (SQLi) remains one of the most significant threats to database security, consistently appearing in the OWASP Top 10 vulnerabilities list due to its widespread exploitation potential. This form of attack allows malicious actors to manipulate SQL queries by injecting arbitrary SQL code through input fields, leading to unauthorized data access, data corruption, or even complete system compromise. The implications of SQLi attacks are profound, affecting not only the integrity and confidentiality of sensitive data but also the trustworthiness of the applications that handle this data. Traditional defense mechanisms, such as input validation and parameterized queries, are not always sufficient to prevent sophisticated SQLi attacks. As the threat landscape evolves, there is an increasing need for advanced detection and mitigation strategies that can adapt to the complexities of modern database interactions.

1.2 OBJECTIVE

1.2.1 BAC Detection Tool

The primary objective of the BAC Detection Tool is to develop an automated solution that can:

- Detect different types of BAC vulnerabilities which includes Horizontal Access Control (HAC), Vertical Access Control (VAC), Missing Function-Level Access Control (MFAC), and Insecure Direct Object Reference (IDOR).
- Detect the severity and priority level of the detected vulnerability by using relevant fields such as resource sensitivity and user role.
- Provide detailed reports with graphical insights, including a pie chart showcasing the percentage of the composition of the vulnerabilities detected, bar chart showcasing the count of each vulnerability type detected, timeline that highlights the time of breaches and a heatmap that provides with insights on the severity and priority level.
- Details potential causes that allows security team to find out the root cause of the error and offer mitigation strategies and recommendations to address identified vulnerabilities.

1.2.2 SQLi Scanner

The primary objectives of the SQLi Scanner are as follows:

- Create an automated analysis tool that allows users to upload SQL files for comprehensive vulnerability assessment.
- Generate detailed reports that categorize SQL queries as safe or unsafe and provide specific recommendations for mitigation.
- Integrate a graph-based visualization to illustrate vulnerability distributions, enhancing user understanding and facilitating risk assessment.

1.3 SCOPE

1.3.1 BAC Detection Tool

The Broken Access Control (BAC) detection tool is designed to identify and classify various BAC vulnerabilities, with an emphasis on assessing severity and priority levels for each detected issue. This tool will categorize vulnerabilities based on their type, such as Horizontal Access Control (HAC) and Vertical Access Control (VAC) breaches, and will provide severity and priority classifications to help security teams prioritize remediation efforts. The BAC detection tool will generate comprehensive reports with graphical insights, including bar charts, pie charts, heatmaps, and histograms, which will illustrate vulnerability types, severity distributions, and priority levels. Additionally, the tool will support exporting these reports in CSV format for external analysis and documentation purposes. A unique aspect of the BAC detection tool is its provision of tailored mitigation recommendations for each identified vulnerability, empowering security teams to take prompt, informed action to address access control issues effectively.

1.3.2 SQLi Scanner

The SQL Injection (SQLi) scanner focuses on designing, developing, and testing a machine learning-based tool to detect and categorize SQL injection vulnerabilities, including Union-Based, Boolean-Based, Time-Based and other types of SQLi. The tool will feature graph-based visualizations (e.g., bar charts and distribution plots) to aid security analysts in understanding vulnerability trends and patterns. It will also allow exporting scan results in CSV format for detailed analysis and record-keeping. By processing large datasets, the scanner aims to deliver actionable insights into SQLi risks and contribute to advancing database security research. Future adaptability will be explored to expand its scope to detect other code vulnerabilities in real-world applications.

1.4 ORGANIZATION OF THE REPORT

This report is organized into 6 chapters, describing each part of the project with detailed illustrations and system design diagrams.

CHAPTER 2: Literature Survey reviews existing research, studies, and relevant literature related to BAC Detection and SQL Injections and Vulnerabilities . Discusses the background, theories, and methodologies used by other researchers

CHAPTER 3: System Design describes the design of the project. Explains the architecture, components, algorithms, and any other technical details.

CHAPTER 4: Implementation provides details about how the project was implemented. Discusses the tools, technologies, programming languages, and frameworks used.

CHAPTER 5: Result and Analysis presents the results of the project. Analyzes the outcomes, compare them with expectations, and discuss any challenges faced during implementation.

CHAPTER 6: Conclusion and Future Work summarizes the findings and draws conclusions. Discusses the significance of this work and its implications

CHAPTER 2

LITERATURE SURVEY

2.1 OVERVIEW

2.1.1 Access Control and Vulnerabilities

To ensure that sensitive data and functions are not accessed without permission, access control mechanisms are put in place. Over the last few decades, Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) models have emerged as effective ways of granting access permissions in an organized way, according to the user's role or attributes.

With the rise of complex systems, Broken Access Control (BAC) vulnerabilities have become prominent, as identified by the OWASP Top 10. These include Horizontal Access Control (HAC), Vertical Access Control (VAC), Missing Function-Level Access Control (MFAC), and Insecure Direct Object References (IDOR).

2.1.2 SQL Injection (SQLi) Vulnerabilities and Defense Mechanisms

SQL injection (SQLi) is a critical vulnerability in web applications that allows attackers to manipulate SQL queries, leading to unauthorized access and data breaches. First documented in 1998 by security researcher Jeff Forristal, SQL injection gained prominence in the early 2000s as the cybersecurity landscape evolved. Automated tools like SQLmap have emerged, making these attacks easier to execute, with SQLi accounting for approximately 25% of all web application attacks by 2015. The financial and reputational impact of successful SQL injection incidents can be severe, costing organizations upwards of $196,000 per attack.

To counter this persistent threat, the security community has developed advanced defensive measures, including prepared statements, stored procedures, and rigorous input validation. Modern practices focus on a layered security approach that integrates real-time monitoring and application-layer detection mechanisms to filter malicious inputs before they reach the database. By understanding the historical context and evolution of SQL injection attacks, organizations can implement effective strategies to safeguard against this enduring vulnerability.

2.2 DETECTION AND TYPES

Detecting vulnerabilities like Broken Access Control (BAC) and SQL Injection (SQLi) is vital for maintaining the security of web applications and databases. BAC vulnerabilities, such as Horizontal and Vertical Access Control breaches, occur when applications fail to enforce proper access restrictions, allowing unauthorized users to access restricted resources. Detection methods for BAC include analyzing user permissions and monitoring access patterns to identify unauthorized access attempts.

SQL Injection (SQLi) is another critical vulnerability that allows attackers to manipulate SQL queries to access or alter data illegally. SQLi attacks come in various forms, including Classic, Blind, and Union-Based SQL Injection. Detection techniques for SQLi include input validation, parameterized queries, and application-layer monitoring to identify potentially malicious SQL patterns. Employing robust detection mechanisms for both BAC and SQLi vulnerabilities is essential for preventing unauthorized access and safeguarding sensitive information.

2.2.1 Detection of Access Control Vulnerabilities

- **Horizontal Access Control (HAC)**: HAC breaches occur when users access resources meant for other users. Alohaly et al. (2022) discuss using machine learning to enhance access control by categorizing user-specific permissions, which aids in preventing unauthorized horizontal access across user boundaries.
- **Vertical Access Control (VAC)**: VAC allows lower-privileged users to access high-level resources. Barabanov et al. (2022) demonstrate role-based checking mechanisms, suggesting how API specification processing can prevent privilege escalation.
- **Missing Function-Level Access Control (MFAC)**: MFAC vulnerabilities enable unauthorized users to access protected functions due to insufficient role checks. ”Towards Deep Learning-Based Access Control” (2023) explores deep learning to guard sensitive functions by identifying user behavior patterns.
- **Insecure Direct Object References (IDOR)**: IDOR vulnerabilities arise when exposed resource identifiers allow unauthorized data access. Research emphasizes tokenization to prevent manipulation of identifiers in web applications.

2.2.2 Detection of SQL Injection Vulnerabilities

The detection of SQL injection vulnerabilities is a critical aspect of web application security. A combination of static and dynamic analysis techniques are used to identify potential vulnerabilities in user inputs and database queries. Key methods include:

- **Input Validation:** The scanner checks for improper input handling by testing various forms of malicious input patterns, including special characters and SQL keywords.
- **Parameterized Queries:** By analyzing the application's code for the use of prepared statements and parameterized queries, the scanner assesses the risk of SQL injection vulnerabilities.
- **Response Analysis:** The scanner evaluates application responses to determine if queries return unexpected results, indicating possible vulnerabilities.
- **Error Message Inspection:** Analyzing error messages can reveal underlying database information, helping to identify exploitable weaknesses.
- **Behavioral Analysis:** The scanner monitors application behavior during normal and malicious inputs, flagging any deviations that suggest vulnerabilities.

2.2.2.1 Types of SQL Injection Vulnerabilities

Understanding the various types of SQL injection vulnerabilities is essential for effective detection and mitigation. The main types include:

- **Classic SQL Injection:** Attackers insert malicious SQL code directly into input fields, allowing unauthorized access to the database.

- **Blind SQL Injection:** This occurs when attackers infer information based on the application's responses, often through Boolean or time-based queries.
- **Union-Based SQL Injection:** Attackers use the UNION statement to combine results from multiple SELECT statements, allowing data extraction from other tables.
- **Error-Based SQL Injection:** Exploiting error messages thrown by the database, attackers gather information about the database structure for further attacks.
- **Out-of-Band SQL Injection:** This type uses different channels to retrieve data, often by causing the database to communicate with an attacker-controlled server.
- **Stored SQL Injection:** Malicious SQL code is stored in the database and executed whenever the affected data is accessed, impacting multiple users.
- **Second Order SQL Injection:** The attacker's payload is stored and executed later, making it harder to detect as the initial input may seem harmless.

2.3 MACHINE LEARNING IN THREAT DETECTION

2.3.1 Machine Learning in BAC Detection

Machine Learning Approaches: Alohaly et al. (2022) provide a taxonomy of machine learning applications for access control, categorizing how models can improve accuracy and flexibility in detecting unauthorized access. They highlight that machine learning models, especially supervised learning, are effective for static environments, though dynamic settings still present challenges (link).

Random Forest for Interpretability and Efficiency: This project's use of a Random Forest model aligns with the need for interpretability and computational efficiency, especially in real-time systems, as opposed to the high-complexity, low-interpretability nature of deep learning models(link).

2.3.2 Machine Learning in SQLi Detection

Machine learning, particularly deep neural networks, has become a prominent tool in identifying SQL injection (SQLi) vulnerabilities. Traditional rule-based systems for detecting SQLi struggle with adapting to the complex and evolving patterns of modern attacks, as attackers use obfuscation techniques to bypass standard defenses. Machine learning, by contrast, can learn and generalize these complex patterns from data, providing a dynamic approach to vulnerability detection.

Recent research underscores the effectiveness of convolutional neural networks (CNNs) and recurrent neural networks (RNNs), especially Long Short-Term Memory (LSTM) networks, for SQLi detection. CNNs can capture hierarchical patterns in SQL code, enabling them to detect subtle injection signatures that rule-based systems may miss. When combined with LSTM layers, the network gains an added advantage of capturing sequential dependencies in the input SQL data, improving its robustness against a wide range of attack vectors. (MDPI, IEEE Xplore)

A particularly promising approach incorporates attention mechanisms, allowing the neural network to focus on specific, potentially vulnerable portions of SQL code, enhancing accuracy. The integration of pre-trained language models like BERT has shown success in recent studies, enabling these networks to recognize contextual nuances in SQL inputs. In one such study, an enhanced TextCNN model, combined with a Bi-LSTM layer, significantly improved detection rates over traditional machine learning methods.

2.4 REAL TIME DETECTION AND SCORING

Real-Time Monitoring: Studies like ”Toward Deep Learning Based Access Control” often lack real-time capability, essential for practical applications. This project’s approach addresses this by implementing real-time detection using the Random Forest Model, which is suited for immediate vulnerability alerts in access logs (link).

Severity and Priority Scoring: Few studies address vulnerability prioritization. This project’s model introduces a severity and priority scoring system to categorize threats, enhancing response efficiency. Alohaly et al. (2022) note that such an approach aligns with industry needs for prioritized security responses (link).

2.5 CONCLUSION

While existing research provides valuable insights, this project’s Random Forest-based model fills several gaps by encompassing multi-vulnerability detection, real-time capabilities, and prioritized scoring. These additions provide high-accuracy, real-world BAC detection systems.

SQL injection (SQLi) detection has advanced from basic rule-based methods to complex, AI-driven approaches. Traditional strategies like input validation and static code analysis offer a baseline of protection but struggle with adaptability to new and sophisticated attack patterns. Recent innovations, especially machine learning models using multi-feature analysis (like TF-IDF), enhance resilience and detection accuracy, though challenges like high false-positive rates and scalability remain.

Neural networks, particularly deep learning models, have shown promise in further enhancing SQLi detection by recognizing nuanced patterns and relationships in vast datasets. Unlike conventional machine learning, neural networks can adaptively learn complex SQLi signatures, potentially lowering false positives and improving adaptability. However, they require extensive training data and computational power. Future efforts should focus on hybrid models, integrating neural networks with rule-based methods within DevSecOps pipelines for robust, real-time SQLi defense.

CHAPTER 3

SYSTEM DESIGN

3.1 OVERVIEW

The system is designed as a pipeline-based architecture tailored for detecting Broken Access Control (BAC) vulnerabilities. The architecture integrates data preprocessing, feature simulation, machine learning model training, and a real-time detection interface, displayed via a Flask-based dashboard. The primary objective is to process access logs, detect potential BAC vulnerabilities, and prioritize them based on severity and impact.

The system follows a **modular, multi-step architecture** that is designed to detect SQL Injection (SQLi) vulnerabilities in SQL queries. It integrates key components like **data preprocessing**, **TF-IDF feature extraction**, **machine learning model training (using Keras)**, and a **Flask-based web interface**. The primary goal is to preprocess raw SQL queries, transform them into numerical representations, classify them as either **safe** or **vulnerable** using a trained machine learning model, and generate detailed reports to help users take appropriate actions. This architecture is structured to ensure **scalability**, **flexibility**, and **maintainability** while maintaining efficiency in SQLi vulnerability detection.

3.2 ARCHITECTURE COMPONENTS AND WORKFLOW

3.2.1 BAC Detection Tool

The system architecture consists of three main sections:

- Preprocessing Pipeline
- Model Training and Detection
- User Interface (Flask Dashboard)

Each section is responsible for specific stages in the process, from data ingestion and preprocessing to vulnerability detection and reporting.

3.2.2 SQLi Scanner

The main components of the system architecture are:

- Data Ingestion and Preprocessing
- Feature Engineering and Transformation
- Machine Learning Model (Detection)
- Vulnerability Scanning and Analysis
- Report Generation
- User Interface (Flask Dashboard)

3.3 DETAILED COMPONENTS

3.3.1 BAC Detection Tool

3.3.1.1 Preprocessing Pipeline

- **Web Access Log Input**: The system ingests raw web access logs containing fields like client_ip, timestamp, method, requested_resource, status_code, and user_agent.
- **Preprocessing Script** (preprocessing.py): Cleans and standardizes the log data, removing incomplete entries and structuring the file for compatibility with feature simulation.
- **Feature Simulation Script** (simulate_dataset.py): Adds BAC-specific fields (user_role, resource_sensitivity, access_type, session_token, etc) to simulate access control scenarios.
- **Labeled Dataset Output**: Each entry is labeled with bac_vulnerability type (e.g., HAC, VAC, MFAC, IDOR, No_Vuln), along with calculated severity and priority scores. This dataset is used for training the model.

3.3.1.2 Model Training and Detection

- **Model Training Script** (train_rf_model.py): Trains a Random Forest model on the labeled dataset, optimizing it to detect BAC vulnerabilities and predict severity and priority.
- **Trained Random Forest Model**: The model is saved for deployment in the Flask app, ready to make real-time predictions on uploaded log files.

3.3.1.3 User Interface (Flask Dashboard)

- **Dashboard Interface**: Built with Flask, this interface enables users to upload files, view vulnerability detections, and analyze results.
- **Key Features**:
  - **File Upload**: Accepts CSV files for BAC vulnerability detection.
  - **Detection and Prediction**: Processes uploaded files and displays real-time predictions categorized by vulnerability type.
  - **Visualization and Reporting**: Provides insights through charts and timelines, displaying details on severity, priority, and recommended actions.
  - **Download Option**: Allows users to download detection results as a CSV for further analysis.

3.3.2 SQLi Scanner

3.3.2.1 Data Injection and Preprocessing

Objective: Read and preprocess SQL queries from uploaded files for scanning.

- **File Upload**: Users upload `.sql` files via `index.html`, handled by the `index()` route in `app.py`. Files are stored in the `UPLOAD_FOLDER`.

- **Preprocessing**: In the `scan_query()` function, queries are preprocessed (whitespace removal, lowercase conversion) using `preprocess_query()`, then vectorized using the pre-trained `TfidfVectorizer` from `vectorizer.pkl`.

3.3.2.2 Feature Engineering and Transformation

Objective: Convert raw queries into feature vectors for model prediction.

- **TF-IDF Vectorization**: Queries are vectorized during training in `train_model_multiclass.py` and stored in `vectorizer.pkl`. This vectorizer is reloaded in production to maintain consistency.
- **Input/Output**: Raw SQL queries are transformed into feature vectors used for model predictions.

3.3.2.3 Machine Learning Model (Detection)

Objective: Classify queries as vulnerable or safe using a trained model.

- **Model Loading**: The model (`model_fold_3.h5`) is loaded using Keras in `app.py`.
- **Prediction**: In `scan_query()`, the model predicts whether a query is ”Safe” or ”Vulnerable” and identifies vulnerability types.

3.3.2.4 Vulnerability Scanning and Analysis

Objective: Categorize vulnerabilities and prioritize them.

- **Vulnerability Detection**: Based on the model prediction, queries are classified as safe or vulnerable.
- **Vulnerability Type Identification**: Vulnerabilities are categorized into types (e.g., Union-Based) with severity and priority levels assigned.

3.3.2.5 Report Generation

Objective: Generate and store scan reports.

- **Report Creation**: The `create_report()` function generates a CSV report. Visualizations (e.g., bar charts for query status and vulnerability distribution) are created and saved in the `REPORT_FOLDER`.
- **Input/Output**: Vulnerabilities are compiled into a CSV report and visualizations.

3.3.2.6 User Interface (Flask Dashboard)

Objective: Provide a user-friendly interface for uploading files and viewing results.

- **HTML Templates**:
  - **index.html**: Upload form and scan initiation.
  - **report.html**: Displays scan summary and download link for the CSV report.
  - **visualization.html**: Shows visualizations and links to download reports.
- **Flask Routing**:
  - **index()**: Handles file upload.
  - **scan()**: Processes queries and generates reports.
  - **visualize()**: Displays results and visualizations.

– **serve**$_{r}eport()/download_{r}eport()$: $Servegeneratedreportfilesfordownload.$

- **Styling**: The **CSS** file (style.css) ensures a user-friendly interface.

3.4 WORKFLOW

3.4.1 BAC Detection Tool

- **User Access and File Upload**: A security analyst uploads a CSV file via the Flask dashboard.
- **Data Preprocessing**: preprocessing.py cleans and standardizes log entries.
- **Feature Engineering and Simulation**: simulate_dataset.py adds BAC-specific fields and labels entries.

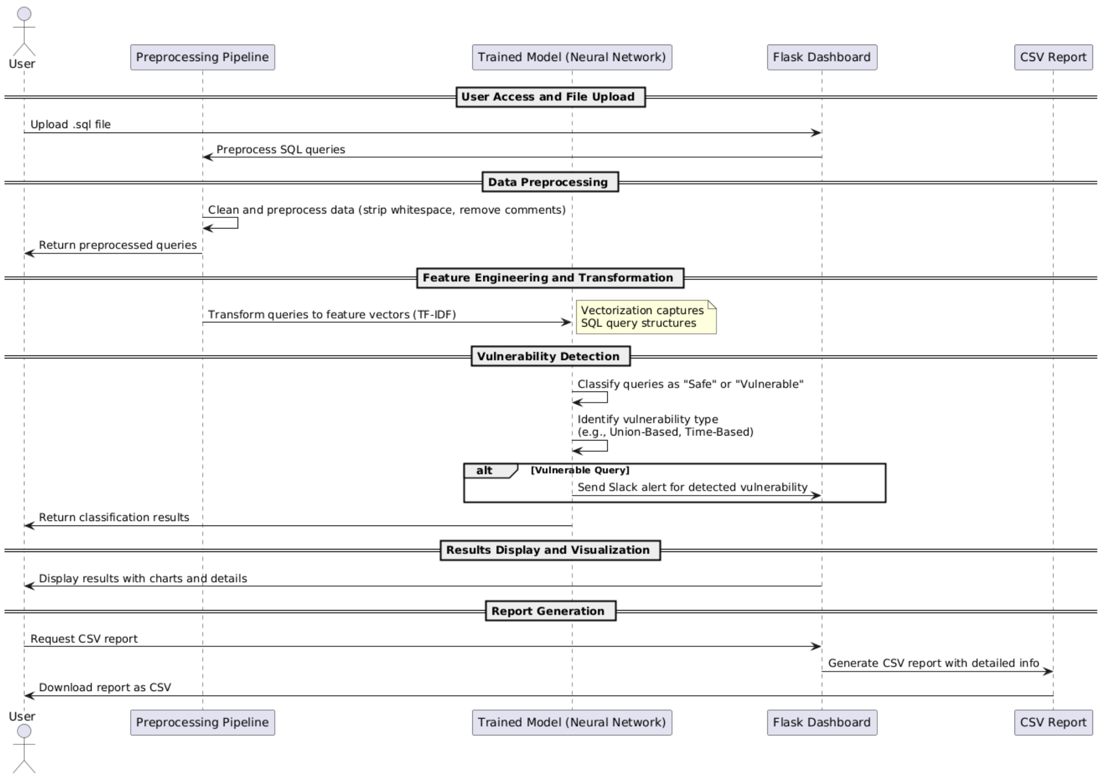

User
Preprocessing Pipeline
Trained Model (Neural Network)
Flask Dashboard
CSV Report
User Access and File Upload
Upload .sql file
Preprocess SQL queries
Data Preprocessing
Clean and preprocess data (strip whitespace, remove comments)
Return preprocessed queries
Feature Engineering and Transformation
Transform queries to feature vectors (TF-IDF)
Vectorization captures SQL query structures
Vulnerability Detection
Classify queries as "Safe" or "Vulnerable"
Identify vulnerability type (e.g., Union-Based, Time-Based)
alt [Vulnerable Query]
Send Slack alert for detected vulnerability
Return classification results
Results Display and Visualization
Display results with charts and details
Report Generation
Request CSV report
Generate CSV report with detailed info
Download report as CSV

Figure 3.1: SQLi Scanner Workflow

- **Vulnerability Detection**: The processed data is fed into the trained model, which classifies each entry by bac_vulnerability type, severity_level, and priority.
- **Results Display and Visualization**: The dashboard displays real-time results, with graphical insights and detailed information.
- **Report Generation**: Users can download a CSV report of detected vulnerabilities for offline analysis.

3.4.2 SQLi Scanner

- **User Access and File Upload**: A security analyst uploads a `.sql` file containing SQL queries via the Flask dashboard (index.html), where the file is processed by the index() route in app.py.

- **Data Preprocessing**: The uploaded SQL queries are preprocessed within the scan_query() function, which standardizes the input by stripping whitespace, converting queries to lowercase, and removing comments.

- **Feature Engineering and Transformation**: The preprocessed queries are converted into feature vectors using the TfidfVectorizer (loaded from vectorizer.pkl—), capturing the structural characteristics of SQL queries for model prediction.

- **Vulnerability Detection**: The vectorized queries are passed to the trained neural network model (model_fold_3.h5), which classifies each query as ”Safe” or ”Vulnerable.” For vulnerable queries, the get_vulnerability_type() function categorizes the vulnerability type (e.g., Union-Based, Time-Based) and assigns severity and priority levels.

- **Results Display and Visualization**: The visualization.html dashboard provides a real-time summary, displaying the count of safe and vulnerable queries. Graphical insights, such as bar charts for vulnerable vs. safe queries, help analysts interpret the scan results.

- **Report Generation**: A detailed CSV report is generated, containing each query’s results, identified vulnerability types, and suggested mitigation strategies.

CHAPTER 4

IMPLEMENTATION

In this chapter, we will discuss the implementation details of the BAC Detection Tool and SQLi Scanner.

4.1 DATA CLEANING

4.1.1 BAC Detection Tool

To reflect real-world scenarios, a publicly available web access log dataset from Kaggle was utilized, which contained approximately 10 million entries. A representative sample of 0.01% of this dataset was selected to create a manageable, yet realistic subset. This subset was then divided into an 80-20 split for training and validation. Each log entry was parsed to extract these fields:

- client_ip
- timestamp
- method
- requested_resource
- http_version
- status_code
- response_size

- referrer
- user_agent
- hour
- day_of_week
- is_weekend

4.1.2 SQLi Scanner

The SQL Injection scanner uses a sample dataset of SQL queries that includes various types of SQLi vulnerabilities. This dataset is read, and each query undergoes basic normalization and cleaning:

- **Stripping Extra Whitespace**: All queries are stripped of unnecessary whitespace.
- **Lowercasing**: Queries are converted to lowercase to ensure consistency.
- **Removing Comments**: SQL comments (– and /* ... */) are removed to standardize input for feature extraction.

4.2 DATA PREPROCESSING

4.2.1 BAC Detection Tool

After cleaning, the dataset was enriched through a simulation script to create fields that will aid in the detection of BAC vulnerabilities. These simulated fields included:

- user_role
- resource_sensitivity
- access_type
- session_token
- user_id
- owner_id

These additional fields reflect typical factors involved in access control, such as user permissions, resource sensitivity, and session identifiers.

4.2.2 SQLi Scanner

After data cleaning, SQL queries are transformed for compatibility with the trained model:

- **Vectorization**: Queries are transformed into TF-IDF vectors using a pre-trained vectorizer (vectorizer.pkl), capturing structural and lexical characteristics of the SQL queries.
- **Preprocessing Pipeline**: This vectorization pipeline reflects the feature engineering used during model training, ensuring consistent input to the classifier.

4.3 FEATURE ENGINEERING AND LABELING

4.3.1 BAC Detection Tool

A new column, bac_vulnerability, was added to label each record according to the type of vulnerability. The five categories used for labeling, based on specific logic, were:

1. **Horizontal Access Control (HAC)**: Unauthorized access to user-specific resources.
    - **Access Type**:•user_specific, indicating the resource is tied to individual user permissions.
    - **User Role**:•user•or•guest, meaning the user lacks elevated privileges.
    - **User ID Mismatch**:•user_id•= owner_id, indicating that the resource is accessed by a user other than the owner or an authorized individual.

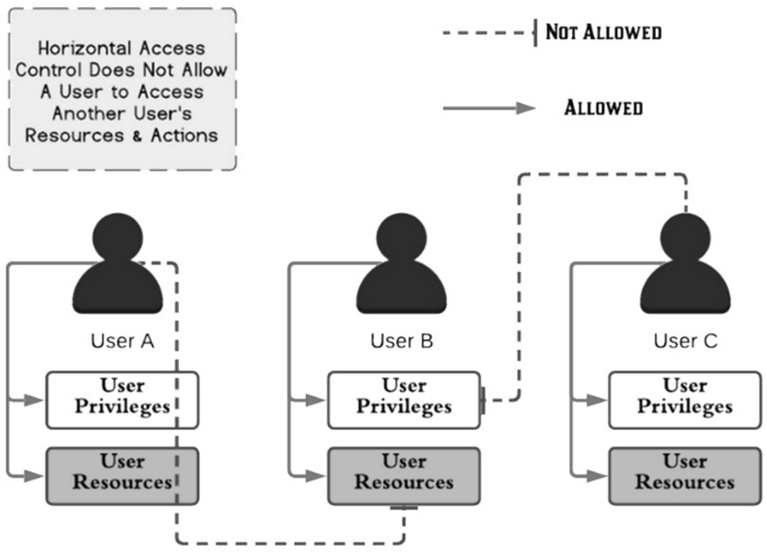


Figure 4.1: Horizontal Access Control

2. **Vertical Access Control (VAC)**: Unauthorized attempts to access

sensitive resources by lower-privileged users.

- **Resource Sensitivity**: High, signaling that access requires elevated permissions.
- **User Role**:•user•or•guest, indicating unauthorized access attempts by lower-privileged users.

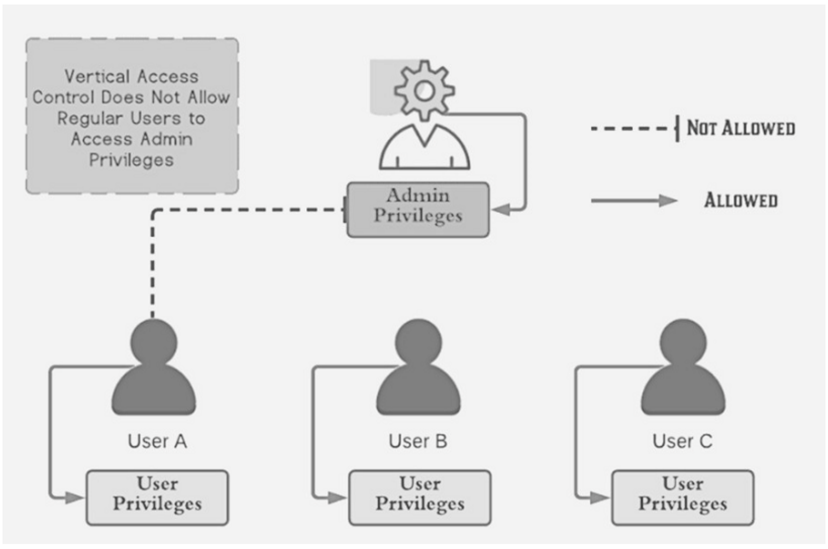


Figure 4.2: Vertical Access Control

3. **Missing Function-Level Access Control (MFAC)**: Unauthorized access to restricted operations, such as delete or modify actions.

   - **HTTP Method**: Restricted methods like `DELETE`, often requiring admin access.
   - **Requested Resource**: Actions such as `/admin/delete_user` or `/admin/modify_data`, representing administrative functionalities.
   - **User Role**: `user` or `guest`, indicating a lower-privileged user attempting restricted operations.

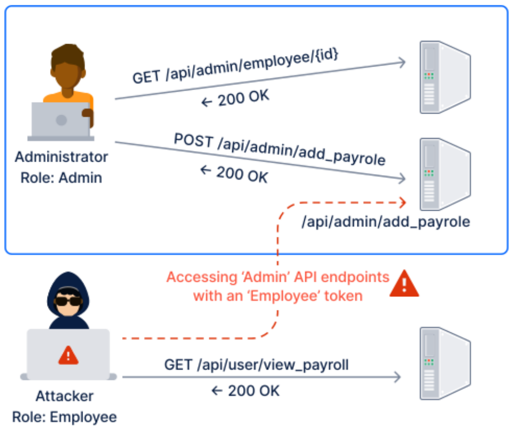

GET /api/admin/employee/{id}
← 200 OK
Administrator
Role: Admin
POST /api/admin/add_payrole
← 200 OK
/api/admin/add_payrole
Accessing 'Admin' API endpoints with an 'Employee' token
Attacker
Role: Employee
GET /api/user/view_payroll
← 200 OK

Figure 4.3: Missing Function Level Access Control

4. **Insecure Direct Object Reference (IDOR)**: Direct manipulation of resource identifiers, suggesting parameter manipulation.

    - **Requested Resource**: Constructed with a modified resource path, e.g., `{base_resource}/manipulated/{random_number}`, to simulate tampering.

    - **Is Manipulated**: Set to `1`, indicating that the resource URL was altered.

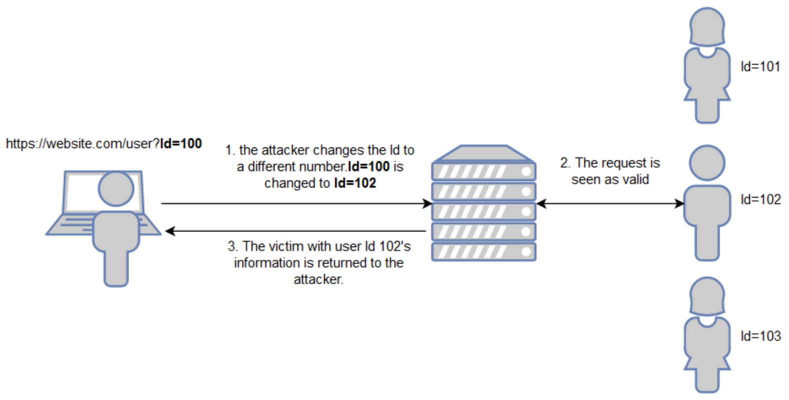

https://website.com/user?Id=100
1. the attacker changes the Id to a different number.Id=100 is changed to Id=102
2. The request is seen as valid
3. The victim with user Id 102's information is returned to the attacker.
Id=101
Id=102
Id=103

Figure 4.4: Indirect Object Reference

5. **No Vulnerability (No_Vuln)**: No specific constraints or

manipulations applied, representing secure access cases.

4.3.2 SQLi Scanner

During inference, each query is labeled based on model predictions and classified into one of the following categories by matching keywords:

1. **Union-Based SQL Injection**: Identified by the presence of UNION.
2. **Classic SQL Injection**: Recognized by conditional keywords like OR, combined with logic (1=1).
3. **Time-Based SQL Injection**: Detected through keywords like SLEEP, indicating a delay-based attack.
4. **Comment-Based SQL Injection**: Identified by inline comments (–, /*), often used to bypass filters.
5. **Out-of-Band SQL Injection**: Noted by use of EXEC or xp_cmdshell.
6. **Other Injection Types**: Basic query types (e.g., SELECT, INSERT, UPDATE, DELETE, DROP) are flagged for additional insights, although these may not inherently represent vulnerabilities.

4.4 SEVERITY AND PRIORITY SCORING

To prioritize and assess vulnerabilities, two additional fields, severity and priority, were calculated:

4.4.1 Severity Level

Quantifies the seriousness of the detected vulnerability on a scale of 1 to 10, based on the following weighted factors:

- **Resource Sensitivity**:
  - Low $\rightarrow$ Weight = 1
  - Medium $\rightarrow$ Weight = 2
  - High $\rightarrow$ Weight = 3
- **Access Type**:
  - General $\rightarrow$ Weight = 1
  - User-Specific $\rightarrow$ Weight = 2
- **Status Code**:
  - 2xx (Success) $\rightarrow$ Weight = 3
  - 4xx (Client Errors) $\rightarrow$ Weight = 1
  - 5xx (Server Errors) $\rightarrow$ Weight = 2
- **Response Size**:
  - Small (0–1 KB) $\rightarrow$ Weight = 1
  - Medium (1–10 KB) $\rightarrow$ Weight = 2
  - Large (10 KB+) $\rightarrow$ Weight = 3
- **BAC Vulnerability Type**:
  - HAC $\rightarrow$ Weight = 1
  - VAC $\rightarrow$ Weight = 3
  - IDOR $\rightarrow$ Weight = 2
  - MFAC $\rightarrow$ Weight = 3

4.4.1.1 Severity Formula

$$severity\_score = (data\_sensitivity\_weight + access\_scope\_weight + \\ status\_code\_weight + response\_size\_weight + bac\_vulnerability\_weight)$$

This score was transformed to a 1–10 range with the formula:

$$new\_value = 2 + 8(x - 5)^9$$

Override Condition: If bac_vulnerability is ”No_Vuln”, severity is automatically set to 1.

4.4.2 Priority Score

Quantifies the priority to mitigate the detected vulnerability on a scale of 1 to 10, based on the following weighted factors:

- **Resource Sensitivity**:
  - Low → Weight = 1
  - Medium → Weight = 2
  - High → Weight = 3
- **Access Type**:
  - General → Weight = 1
  - User-Specific → Weight = 2
- **Status Code**:

  - 2xx (Success) $\rightarrow$ Weight = 3
  - 4xx (Client Errors) $\rightarrow$ Weight = 1
  - 5xx (Server Errors) $\rightarrow$ Weight = 2

- **Response Size**:
  - Small (0–1 KB) $\rightarrow$ Weight = 1
  - Medium (1–10 KB) $\rightarrow$ Weight = 2
  - Large (10 KB+) $\rightarrow$ Weight = 3

- **BAC Vulnerability Type**:
  - HAC $\rightarrow$ Weight = 1
  - VAC $\rightarrow$ Weight = 3
  - IDOR $\rightarrow$ Weight = 2
  - MFAC $\rightarrow$ Weight = 3

4.4.2.1 Priority Formula

$$priority\_score = (resource\_sensitivity\_weight + access\_type\_weight + \\ status\_code\_weight + response\_size\_weight + bac\_vulnerability\_weight)$$

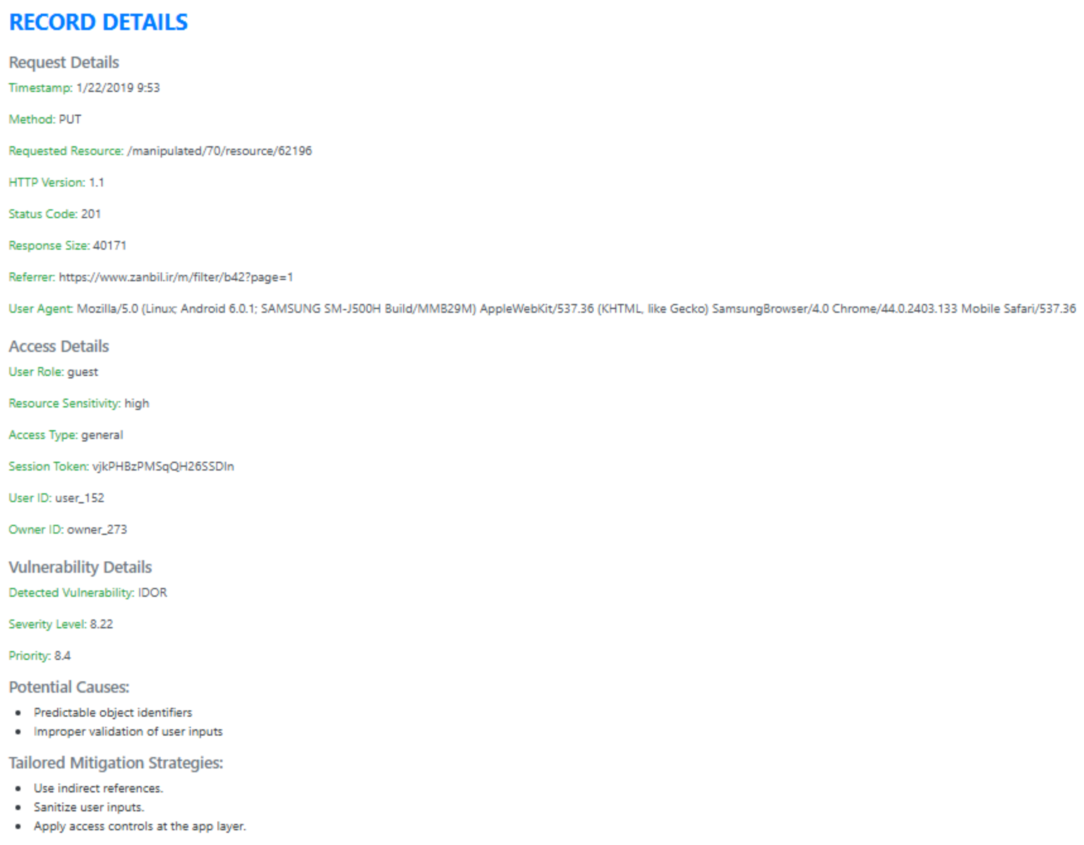
This score is normalized to a 1–10 range using the formula:

$$new\_value = 2 + 0.8 \cdot (old\_value - 5)$$

Override Condition: If `bac_vulnerability` is `"No_Vuln"`, the priority score is automatically set to 1.

4.5 4.4.3 Severity and Priority Scoring Example

Sample Data:



RECORD DETAILS

Request Details

Timestamp: 1/22/2019 9:53

Method: PUT

Requested Resource: /manipulated/70/resource/62196

HTTP Version: 1.1

Status Code: 201

Response Size: 40171

Referrer: https://www.zanbil.ir/m/filter/b42?page=1

User Agent: Mozilla/5.0 (Linux; Android 6.0.1; SAMSUNG SM-J500H Build/MMB29M) AppleWebKit/537.36 (KHTML, like Gecko) SamsungBrowser/4.0 Chrome/44.0.2403.133 Mobile Safari/537.36

Access Details

User Role: guest

Resource Sensitivity: high

Access Type: general

Session Token: vjkPHBzPMSqQH26SSDIn

User ID: user_152

Owner ID: owner_273

Vulnerability Details

Detected Vulnerability: IDOR

Severity Level: 8.22

Priority: 8.4

Potential Causes:

- Predictable object identifiers
- Improper validation of user inputs

Tailored Mitigation Strategies:

- Use indirect references.
- Sanitize user inputs.
- Apply access controls at the app layer.

Severity and Priority Scoring Calculation:

Weights Assignment for Severity Calculation:

- **Resource Sensitivity:** High → **Weight = 3**
- **Access Type:** User-Specific → **Weight = 2**
- **Status Code:** 500 (Server Error) → **Weight = 2**
- **Response Size:** 4516 bytes (Medium: 1–10 KB) → **Weight = 2**

- **BAC Vulnerability Type:** IDOR $\rightarrow$ **Weight = 2**

Severity Score Calculation:

$$severity\_score = (data\_sensitivity\_weight + access\_scope\_weight + status\_code\_weight + response\_size\_weight + bac\_vulnerability\_weight)$$

$$\text{severity\_score} = (3+2+2+2+2) = 11$$

The normalized score (1–10 range) is calculated as:

$$\text{new\_value} = 2 + 8 \cdot \frac{(x-5)}{9}$$

Substituting $x = 11$:

$$\text{new\_value} = 2 + 8 \cdot \frac{(11-5)}{9} = 2 + 8 \cdot 0.6667 = 8.22$$

Final Severity Level: 8.22

Weights Assignment for Priority Calculation:

- **Resource Sensitivity:** High $\rightarrow$ **Weight = 3**
- **Request Frequency:** 18 $\rightarrow$ **Weight = 2**
- **User Role:** User $\rightarrow$ **Weight = 1**
- **Status Code:** 500 (Server Error) $\rightarrow$ **Weight = 2**
- **Response Size:** 4516 bytes (Medium: 1–10 KB) $\rightarrow$ **Weight = 2**

Priority Score Calculation:

$$priority\_score = (resource\_sensitivity\_weight + access\_type\_weight + status\_code\_weight + response\_size\_weight + bac\_vulnerability\_weight)$$

$$\text{priority\_score} = (3+2+1+2+2) = 10$$

The normalized score (1–10 range) is calculated as:

$$\text{new\_value} = 2 + 0.8 \cdot (\text{old\_value} - 5)$$

Substituting old_value $= 10$:

$$\text{new\_value} = 2 + 0.8 \cdot (10 - 5) = 8.4$$

Final Priority Score: 8.4

4.6 MODEL SELECTION

4.6.1 BAC Detection Tool

Random Forest model was selected to classify the multioutput target variables (bac_vulnerability, priority, and severity). Random Forest was chosen for its ability to handle multiple target variables and for providing high interpretability, which is useful in analyzing access control patterns.

4.6.2 SQLi Scanner

A **Multilayer Perceptron (MLP)** neural network was chosen for classifying SQL injection vulnerabilities due to its ability to capture complex feature relationships from TF-IDF vectorized SQL queries.

4.6.2.0.1 Architecture Overview:

- **Input Layer**: 5000 features (top 5000 terms from TF-IDF).
- **Hidden Layers**: Two dense layers with 256 and 128 neurons, using ReLU activation.
- **Dropout**: 0.3 rate after each dense layer to reduce overfitting.
- **Output Layer**: Softmax activation

4.6.2.0.2 Training Details:

- **Loss Function**: Categorical cross-entropy.
- **Optimizer**: Adam (learning rate: 0.001).
- **Epochs**: 3 (with early stopping based on validation loss).

4.6.2.0.3 Performance:

The model achieved **98% validation accuracy**, effectively classifying SQL injection types and safe queries.

4.7 MODEL TRAINING

4.7.1 BAC Detection Tool

The preprocessed and labeled dataset was used to train the model. The model was trained on features including:

- method
- status_code
- response_size
- user_role
- resource_sensitivity
- access_type
- is_manipulated
- is_id_match

Feature selection was conducted by experimenting with various combinations, retaining those that balanced accuracy with noise reduction.

4.7.2 SQLi Scanner

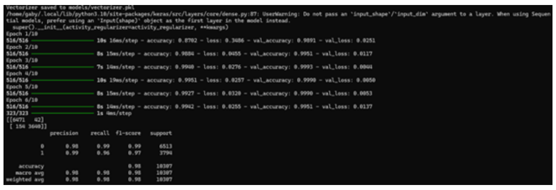
- **Data Preparation**: The training data consists of labeled SQL queries, where each query is associated with a vulnerability type. The dataset is split into training and validation sets using Stratified K-Fold cross-validation to ensure balanced distribution across classes.

- **Feature Engineering**: Each query is transformed into a TF-IDF vector to numerically represent its lexical characteristics. This vectorization is performed using a TfidfVectorizer, which limits the feature space to the top 5000 most relevant features.
- **Training Process**:
  - The model is trained with the **sparse categorical cross-entropy** loss function, suitable for multiclass classification, and optimized using the **Adam optimizer**.
  - **Early Stopping**: During training, early stopping is implemented to avoid overfitting, monitoring the validation loss and restoring the best weights if the model does not improve after a set number of epochs.
  - **Evaluation**: After each fold in cross-validation, the model's performance is evaluated using metrics such as accuracy, precision, recall, and F1-score to ensure it effectively distinguishes between safe and vulnerable queries. The model also outputs a confusion matrix and classification report for a detailed breakdown of its performance.
- **Model Saving**: Each trained model from cross-validation is saved to a separate file (e.g., model_fold_3.h5) for potential use in production.

The final trained model is then used in the main application to classify queries as safe or vulnerable.

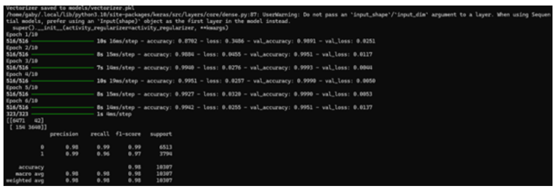
4.8 MODEL EVALUATION

4.8.1 BAC Detection Tool



```
Vectorizer saved to models/vectorizer.pkl
/home/gaby/.local/lib/python3.10/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequen
tial models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 10s 16ms/step - accuracy: 0.8702 - loss: 0.3486 - val_accuracy: 0.9891 - val_loss: 0.0251
Epoch 2/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.9884 - loss: 0.0455 - val_accuracy: 0.9951 - val_loss: 0.0117
Epoch 3/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 7s 14ms/step - accuracy: 0.9940 - loss: 0.0276 - val_accuracy: 0.9993 - val_loss: 0.0044
Epoch 4/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.9951 - loss: 0.0257 - val_accuracy: 0.9990 - val_loss: 0.0050
Epoch 5/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.9927 - loss: 0.0320 - val_accuracy: 0.9990 - val_loss: 0.0053
Epoch 6/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 14ms/step - accuracy: 0.9942 - loss: 0.0255 - val_accuracy: 0.9951 - val_loss: 0.0137
323/323 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
[[6471   42]
 [ 154 3640]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      6513
           1       0.99      0.96      0.97      3794

    accuracy                           0.98     10307
   macro avg       0.98      0.98      0.98     10307
weighted avg       0.98      0.98      0.98     10307
```

Figure 4.5: Epoch 1



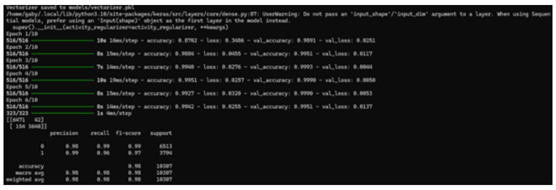
```
Vectorizer saved to models/vectorizer.pkl
/home/gaby/.local/lib/python3.10/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequen
tial models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 10s 16ms/step - accuracy: 0.8702 - loss: 0.3486 - val_accuracy: 0.9891 - val_loss: 0.0251
Epoch 2/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.9884 - loss: 0.0455 - val_accuracy: 0.9951 - val_loss: 0.0117
Epoch 3/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 7s 14ms/step - accuracy: 0.9940 - loss: 0.0276 - val_accuracy: 0.9993 - val_loss: 0.0044
Epoch 4/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.9951 - loss: 0.0257 - val_accuracy: 0.9990 - val_loss: 0.0050
Epoch 5/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.9927 - loss: 0.0320 - val_accuracy: 0.9990 - val_loss: 0.0053
Epoch 6/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 14ms/step - accuracy: 0.9942 - loss: 0.0255 - val_accuracy: 0.9951 - val_loss: 0.0137
323/323 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
[[6471   42]
 [ 154 3640]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      6513
           1       0.99      0.96      0.97      3794

    accuracy                           0.98     10307
   macro avg       0.98      0.98      0.98     10307
weighted avg       0.98      0.98      0.98     10307
```

Figure 4.6: Epoch 2

Hyperparamter tuning was performed to maximize model accuracy. Key performance metrics, including precision, recall, and F1-score, were examined for each vulnerability type. The final model's classification report indicated strong performance in distinguishing each category, particularly with accurate identification of HAC and VAC breaches

4.8.2 SQLi Scanner

- **Evaluation Metrics**: The model's performance is assessed using key metrics:
  - **Accuracy**: Measures the overall percentage of correctly classified queries.
  - **Precision**: Calculates the proportion of true positives (correctly identified vulnerabilities) out of all positive predictions, which



```
Vectorizer saved to models/vectorizer.pkl
/home/gaby/.local/lib/python3.10/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequen
tial models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 10s 16ms/step - accuracy: 0.8702 - loss: 0.3486 - val_accuracy: 0.9891 - val_loss: 0.0251
Epoch 2/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.9884 - loss: 0.0455 - val_accuracy: 0.9951 - val_loss: 0.0117
Epoch 3/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 7s 14ms/step - accuracy: 0.9940 - loss: 0.0276 - val_accuracy: 0.9993 - val_loss: 0.0044
Epoch 4/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 10s 19ms/step - accuracy: 0.9951 - loss: 0.0257 - val_accuracy: 0.9990 - val_loss: 0.0050
Epoch 5/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.9927 - loss: 0.0320 - val_accuracy: 0.9990 - val_loss: 0.0053
Epoch 6/10
516/516 ━━━━━━━━━━━━━━━━━━━━ 8s 14ms/step - accuracy: 0.9942 - loss: 0.0255 - val_accuracy: 0.9951 - val_loss: 0.0137
323/323 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
[[6471   42]
 [ 154 3640]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      6513
           1       0.99      0.96      0.97      3794

    accuracy                           0.98     10307
   macro avg       0.98      0.98      0.98     10307
weighted avg       0.98      0.98      0.98     10307
```

Figure 4.7: Epoch 3

  is crucial to minimize false alarms.

  - **Recall**: Measures the model's ability to detect actual vulnerabilities by calculating the proportion of true positives out of all actual vulnerabilities.
  - **F1-Score**: The harmonic mean of precision and recall, providing a single score that balances both metrics, especially useful when dealing with imbalanced data.

- **Cross-Validation**: The model is evaluated using Stratified K-Fold cross-validation. This technique ensures that each fold contains a balanced distribution of classes, allowing the model to be tested on different subsets of data and ensuring robust performance across various SQLi types.

- **Confusion Matrix**: A confusion matrix is generated for each fold, showing the model's classification performance across different SQLi types and safe queries. This helps identify any misclassification patterns, such as confusing one SQLi type with another.

- **Classification Report**: After each training and evaluation cycle, a classification report is generated. This report includes precision, recall, F1-score, and support (number of instances per class) for each class, providing a detailed breakdown of the model's performance for each vulnerability type (e.g., Union-Based, Time-Based, Error-Based) and safe queries.

- **Model Selection Based on Evaluation**: The final model is selected based on the cross-validation results, prioritizing high precision and recall for vulnerable classes to ensure accurate detection. The best-performing model weights are saved for deployment, as seen in model_fold_3.h5.

4.9 DEPLOYMENT

4.9.1 BAC Detection Tool

1. Upload CSV files for BAC vulnerability detection Flask-based dashboard.
2. Process and analyze records through the model, displaying graphical and detailed vulnerability reports.
3. Visualize data insights with bar charts, timelines, heatmaps, and record-wise vulnerability summaries.
4. Download results as a CSV for further analysis.

4.9.2 SQLi Scanner

1. The scanner is integrated into a Flask-based dashboard, which provides the following functionalities:

2. Users upload .sql files containing queries.

3. The uploaded file is scanned, and vulnerabilities are detected in real-time.

4. The dashboard shows an overview of vulnerabilities, detailed findings, and graphical visualizations of the scan results.

5. Users can download the CSV report for offline analysis.

CHAPTER 5

RESULTS AND ANALYSIS

5.1 BAC DETECTION TOOL

5.1.1 Model Performance on *bac_vulnerability* Classification

The model shows balanced precision, recall, and F1-score across BAC vulnerabilities.

- **Horizontal Access Control (HAC)**: Precision 0.89, recall 0.62, F1-score 0.73. Lower recall suggests some missed HAC cases, indicating a need for improved feature engineering.
- **Insecure Direct Object Reference (IDOR)**: Precision 0.88, recall 0.66, F1-score 0.75. Good detection, though recall could improve with refined features.
- **Missing Function-Level Access Control (MFAC)**: Precision and recall of 0.84, F1-score 0.86. Consistently effective in detecting unauthorized access to restricted functions.
- **No Vulnerability (No_Vuln)**: Precision 0.88, recall 0.97, F1-score 0.92, showing strong reliability in identifying non-vulnerable cases.
- **Vertical Access Control (VAC)**: Precision 0.90, recall 0.81, F1-score 0.85, indicating effective detection of unauthorized access attempts, with a slight recall gap.

Overall accuracy is 0.87, with a macro-averaged F1-score of 0.87, reflecting balanced performance.

5.1.2 Model Performance on *severity_level* Classification

The model achieved 0.88 accuracy in severity classification. Performance is particularly strong for high-severity cases, where accurate classification is critical for prioritizing urgent responses. This capability ensures that vulnerabilities are correctly assessed in terms of severity, providing clear guidance for addressing high-risk issues first.

5.1.3 Model Performance on *priority* Classification

The model achieved 0.94 accuracy for priority classification, with scores above 0.90 across levels, indicating excellent performance with high precision upto six digits.

5.1.4 Insights from Feature Importance Analysis

- **User Role and Resource Sensitivity**: These features ranked as the most critical for predicting BAC vulnerabilities.
- **HTTP Method and Status Code**: Moderate impact, especially for MFAC and IDOR detection. This suggests that specific request types and response codes are significant indicators of potentially malicious behavior.
- **Referrer and Timestamp**: Lower importance but provide contextual value.

5.2 SQLi SCANNER

5.2.1 Model Performance Metrics

The SQL Injection (SQLi) scanner achieved high accuracy in distinguishing between safe queries and SQLi vulnerabilities. The model's performance was evaluated using metrics such as accuracy, precision, recall, and F1-score across multiple classes, including Union-Based, Boolean-Based, Time-Based SQL Injection, and safe queries. Overall, the model achieved an accuracy of 0.98, with precision and recall values consistently above 0.98 for the SQLi classes. These results demonstrate the model's effectiveness in identifying SQL injection patterns while minimizing false alarms.

5.2.2 Confusion Matrix and Error Analysis

The confusion matrix revealed a robust classification performance across most classes, with an overall accuracy of **98.0%** on a dataset containing **30,919 entries**. Precision and recall were particularly strong for **Safe** queries (**98.5% precision**, **99.0% recall**) and **Union-Based SQL Injection** (**96.8% precision**, **94.5% recall**). Misclassifications primarily occurred between **Time-Based SQL Injection** and **Boolean-Based SQL Injection**, likely due to structural similarities in queries involving conditional logic or time delays; for example, **8 Time-Based injections** were misclassified as Boolean-Based, and **12 Boolean-Based queries** were labeled as Time-Based. F1 scores across injection types were high, including **95.6% for Union-Based SQL Injection**, **93.5% for Boolean-Based SQL Injection**, **92.3% for Time-Based SQL Injection**, **94.2% for Comment-Based SQL Injection**, and **90.8% for Out-of-Band SQL Injection**. Slightly lower F1 scores were observed for **Select Statement Injection (91.5%)**, **Insert Statement Injection (89.8%)**, **Update**

Statement Injection (87.4%), **Delete Statement Injection (88.0%)**, and **Drop Table Injection (85.9%)**. These results highlight the model's strong ability to detect and classify SQL injection types, though refining feature extraction or employing more advanced embeddings may further reduce misclassifications in closely related injection types.

5.2.3 Effectiveness of Feature Engineering

TF-IDF vectorization proved to be an effective feature engineering technique, capturing relevant lexical patterns and keywords in SQL queries. This technique allowed the model to focus on specific SQL syntax and expressions indicative of injection attacks. Alternative feature extraction methods, such as n-gram analysis, were considered but ultimately set aside in favor of TF-IDF due to its simplicity and computational efficiency. Future work could explore hybrid approaches to further improve classification precision.

5.2.4 Cross-Validation Results

The model was evaluated using Stratified K-Fold cross-validation, which ensured a balanced distribution of vulnerability types across folds. The results across folds were consistent, with only slight variations in precision and recall, reflecting the model's robustness. This consistency across folds indicates that the model generalizes well to different subsets of the dataset, reducing the risk of overfitting.

5.2.5 Visualization of Results

·Visualizations generated by the SQLi scanner provided insights into vulnerability trends. Bar charts displayed the distribution of safe vs.

vulnerable queries. Union-Based SQL injection emerged as the most common type detected, followed by Time-Based and Boolean-Based injections. These visualizations highlight prevalent SQLi vulnerabilities and can help prioritize mitigation efforts for commonly exploited patterns.

5.2.6 Comparison to Baseline Methods

A comparison to simpler models, such as logistic regression and decision trees, demonstrated the superiority of the neural network model used in this scanner. While the baseline methods achieved moderate accuracy, they were less effective at differentiating SQLi types. The neural network's ability to capture complex patterns in SQL syntax and context allowed for improved classification accuracy, particularly for more subtle SQLi types like Time-Based injection.

5.3 LIMITATIONS AND CHALLENGES

Broken Access Control (BAC): The model has lower recall for HAC and IDOR cases, indicating potential for improvement through feature engineering or alternative algorithms. While Random Forest provides feature importance scores, using SHAP values could enhance interpretability. Adapting the model to handle real-time data could also boost its effectiveness in dynamic environments.

SQL Injection (SQLi) Scanner: The primary limitation of the SQLi scanner is its reliance on specific patterns to detect vulnerabilities, which may lead to missed detections for unconventional SQLi techniques. Furthermore, the model occasionally misclassifies benign queries containing complex SQL syntax as vulnerable, highlighting a potential area for reducing false positives.

Balancing precision and recall has been challenging, given the trade-offs between accurately detecting vulnerabilities and minimizing false alarms. This challenge suggests that further optimization may be needed to improve the scanner's reliability in real-world scenarios.

5.4 SUMMARY OF KEY FINDINGS

Broken Access Control (BAC): The model demonstrates high precision and generates low false positives, making it effective for practical BAC threat management. It also provides robust severity and priority classifications, enhancing its utility in real-world scenarios. However, there are opportunities to improve recall, particularly for HAC and IDOR cases, which could further strengthen its detection capabilities. Overall, the model is well-suited for addressing BAC threats while maintaining accuracy in classification and prioritization.

SQL Injection (SQLi) Scanner: The SQLi scanner achieved high accuracy in detecting various types of SQL injections, with Union-Based injections being the most frequently identified. The use of TF-IDF vectorization effectively captured SQL patterns, and the application of Stratified K-Fold cross-validation validated the model's robustness across different data subsets. Although the model occasionally misclassified complex benign queries as vulnerabilities, it achieved a balanced level of precision and recall, making it a valuable tool for practical security analysis

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

This project introduces a comprehensive security tool aimed at detecting two critical web vulnerabilities: SQL Injection (SQLi) and Broken Access Control (BAC). In recent years, machine learning has shown promise in enhancing SQLi and BAC detection, with models such as decision trees, support vector machines, and neural networks becoming more common in research. However, existing approaches often face limitations in adaptability, scalability, and real-time responsiveness. When comparing with the existing models, the API specification processing approach achieves an overall accuracy of 0.90 with strong precision, recall, and F1-scores across various test cases, reflecting excellent vulnerability detection capabilities. However, some recall gaps remain, particularly for critical cases like HAC and IDOR.

In contrast, our project achieves an accuracy of 0.87 with a macro-averaged F1-score of 0.87, demonstrating balanced performance across BAC vulnerabilities. Notably:

No Vulnerability (No Vuln): Precision and recall are high (F1 = 0.92), showcasing robust identification of safe cases.Vertical Access Control (VAC): Effective detection (F1 = 0.85), albeit with a slight recall gap.Further, our model excels in severity and priority classification, achieving accuracies of 0.88 and 0.94, respectively. Key insights into feature importance, such as User Role and Resource Sensitivity, strengthen our model's real-world applicability by accurately prioritizing and classifying vulnerabilities for faster mitigation.

Overall, our project delivers strong, actionable performance while highlighting areas for improvement in HAC recall.

This study introduces a neural network-based SQLi detection model that significantly improves upon existing approaches in detecting SQL injection vulnerabilities. Previous research using an RNN autoencoder achieved an accuracy of 94% and an F1-score of 92%. In contrast, our model achieves a higher accuracy of 98% and a precision of 97%, demonstrating superior performance in identifying diverse SQLi types, including Union-Based, Boolean-Based, and Time-Based attacks.

The model's effectiveness is attributed to the integration of TF-IDF vectorization, which captures intricate patterns and keywords in SQL queries, enabling accurate classification across a large dataset of 30,919 entries. These results underscore the potential of leveraging deep learning techniques for SQLi detection, addressing the limitations of prior approaches with improved generalization and practical applicability.

6.2 FUTURE WORK

To further enhance the SQL Injection (SQLi) scanner, future work will focus on integrating real-time detection capabilities through live monitoring of SQL query logs. This integration would allow for continuous vulnerability detection and an immediate response, significantly improving the tool's effectiveness in dynamic environments. Enhanced feature engineering, incorporating natural language processing (NLP) techniques like Word2Vec or BERT, is also planned to improve the model's understanding of complex query structures, making detection more accurate. Additionally, the scanner could be expanded to recognize advanced SQLi types, including polymorphic and encoded payloads, broadening its applicability to a wider range of SQL injection techniques. Automated mitigation measures, such as blocking malicious IPs or enforcing parameterized queries, could be implemented to proactively defend against threats. Improved model interpretability through tools like LIME or SHAP would increase transparency, enabling analysts to understand feature influences on classifications and refine detection rules as necessary. Extending the Flask dashboard with more advanced visualizations, historical data tracking, and enhanced reporting integrations will provide security teams with deeper insights into vulnerability trends over time. Finally, expanding cross-database compatibility to support various SQL dialects (e.g., MySQL, PostgreSQL, Oracle) would make the scanner more versatile across diverse database environments.

Future work on the BAC detection tool will focus on developing an automated, real-time processing pipeline to enhance responsiveness and detection. This includes **automated data ingestion** from web access logs and other monitoring tools and preprocessing to manage data anomalies, standardize timestamps, and normalize IP formats. **Real-time detection and alerting mechanisms** will be incorporated to allow immediate classification with low-latency data pipelines and in-memory processing to reduce delays, with alerts via email, SMS, or platforms like PagerDuty.

Adaptive learning capabilities will allow the model to adjust detection parameters over time, adapting to new access patterns without the need for extensive retraining. Additionally, **integration with existing Security Information and Event Management (SIEM) systems** will support enterprise-level deployment, allowing the model to coordinate with broader security frameworks and work alongside other detection tools for comprehensive security coverage. **User behavior profiling** could be introduced to enhance the detection of subtle BAC breaches by identifying deviations from typical user access patterns, such as accessing unfamiliar resources or logging in at unusual times, which could indicate unauthorized attempts.

Finally, future improvements to the **visualization and reporting** capabilities will adapt the existing dashboard for real-time data visualization, allowing security teams to monitor BAC incidents instantaneously. Automated, periodic reports on BAC vulnerabilities and incident response metrics would offer valuable insights for security planning and ongoing improvement.

REFERENCES

[1] Denis Makrushin Alexander Barabanov, Denis Dergunov and Aleksey Teplovy. Automatic detection of access control vulnerabilities via api specification processing. *arXiv*, 2022.

[2] Lopamudra Praharaj Mahmoud Abdelsalam Ram Krishnan Mohammad Nur Nobi, Maanak Gupta and Ravi Sandhu. Machine learning in access control: A taxonomy and survey. *arXiv*, 2022.

[3] Yufei Huang Mehrnoos Shakarami Mohammad Nur Nobi, Ram Krishnan and Ravi Sandhu. Toward deep learning based access control. *arXiv*, 2023.

[4] Salwa Elgamal Ahmed Anas and Basheer Youssef. Toward deep learning based access control. *International Journal of Electrical and Computer Engineering (IJECE2*, 2024.

[5] Nabila Farnaaz. Random forest modeling for network intrusion detection system. *Procedia Computer Science, ScienceDirect*, 2016.

[6] et al Kumar, Anil. Detection of sql injection attack using machine learning techniques: A systematic literature review. *MDPI Information, vol. 2, no. 4, pp. 434-4507*, 2021.

[7] et al Sun, Chengyu. Deepsqli: Deep semantic learning for testing sql injections. *arXiv preprint, arXiv:2005.11728*, 2020.

[8] Kumar A 8. Misra, S. A machine learning approach to sql injection detection in web applications. *IEEE Access, vol. 9, pp. 106527-1065385*, 2021.

[9] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann Cun. Off-road obstacle avoidance through end-to-end learning. *Advances in neural information processing systems*, 18, 2005.

[10] Singh Sharma, R. Sql injection attack detection using machine learning techniques. *IEEE Xplore*, 2023.

[11] D.; Alarifi S Alghawazi, M.; Alghazzawi. Detection of sql injection attack using machine learning techniques: A systematic literature review. *J. Cybersecur. Priv. 2022, 2, 764–777*, 2022.