DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

GITAM

Predicting the vulnerabilities in JavaScript using Machine learning algorithms

A Project Report submitted in partial fulfillment of the requirements for the award of the degree of,

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

B. Lasya	**322010329001**
T. Satwika	**322010328008**
T. Surya	**322010328025**
P. Jagan Satwik	**322010328021**

Under the esteemed guidance of

Mrs. Archana S Nadhan

Assistant Professor

Department of Computer Science & Engineering,

GITAM SCHOOL OF TECHNOLOGY

GANDHI INSTITUTE OF TECHNOLOGY AND MANAGEMENT

(Deemed to be University)

Bengaluru Campus.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

GITAM

(Deemed to be University)

CERTIFICATE

This is to certify that the project report entitled **"Predicting the vulnerabilities in JavaScript using Machine learning algorithms"** is a bonafide record of work carried out by **B. Lasya(322010329001), T. Satwika (322010328008), T. Surya (322010328025), P. Jagan Satwik (322010328021)**submitted in partial fulfillment of requirement for the award of degree of **Bachelors of Technology in Computer Science and Engineering**.

Project Guide.	**Head of the Department.**
SIGNATURE OF THE GUIDE	**SIGNATURE OF THE HOD**
Mrs. Archana S Nadhan	**Dr. Vamsidhar Yendapalli,**
Assistant Professor	Professor.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

GITAM

(Deemed to be University)

DECLARATION

We, hereby declare that the project report entitled **"Predicting the vulnerabilities in JavaScript using Machine learning algorithms"** is an original work done in the **Department of Computer Science and Engineering, GITAM School of Technology, GITAM (Deemed to be University)** submitted in partial fulfillment of the requirements for the award of the degree of **B.Tech.** in Computer Science and Engineering. The work has not been submitted to any other college or University for the award of any degree.

Date:

Registration No(s).	**Name(s)**	**Signature(s)**
322010329001	B. Lasya	
322010328008	T. Satwika	
322010328025	T. Surya	
322010328021	P. Jagan Satwik	

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of the people who made it possible, whose consistent guidance and encouragement crowned our efforts with success.

We consider it our privilege to express our gratitude to all those who guided us in the completion of the project.

We express our gratitude to Director Prof. **Basavaraj Gundappa Katageri** for having provided uswith the golden opportunity to undertake this project work in their esteemed organization.

We sincerely thank **Dr. Vamsidhar Yendapalli**, HOD, Department of Computer Science and Engineering, Gandhi Institute of Technology and Management, Bengaluru for the immense support given to us.

We express our gratitude to our project guide **Mrs. Archana S Nadhan, Assistant Professor**, Department of Computer Science and Engineering, Gandhi Institute of Technology and Management, Bengaluru , for their support, guidance, and suggestions throughout the project work.

Student Name's	**Registration No.**
B. Lasya	322010329001
T. Satwika	322010328008
T. Surya	322010328025
P. Jagan satwik	322010328021

ABSTRACT

Predicting vulnerable JavaScript functions is a crucial task in ensuring the security and reliability of web applications. Machine learning algorithms have shown promise in automating this process by identifying patterns and features that distinguish vulnerable functions from secure ones. However, this task poses several challenges due to the complexity and dynamic nature of JavaScript. This study aims to address these challenges and develop effective machine learning algorithms for predicting vulnerable JavaScript functions. The project has proposed a novel approach that leverages a combination of static and dynamic analysis techniques to extract relevant features and capture the behavior of JavaScript functions. The static analysis component analyzes the source code to identify potential vulnerabilities based on known patterns and common coding mistakes. The dynamic analysis component executes the JavaScript functions within a controlled environment to observe their runtime behavior and identify any unexpected or potentially dangerous actions. To train and evaluate the machine learning algorithms, there is a comprehensive dataset created comprising a large number of JavaScript functions, both vulnerable and secure. This study carefully selected various vulnerable functions, including those susceptible to code injection, cross-site scripting, and other common attacks. The dataset also includes secure functions to ensure a balanced representation.

This study uses a variety of cutting-edge machine learning techniques, including deep neural networks, decision trees, and random forests, to discover the patterns and characteristics that distinguish susceptible functions from secure ones. This study uses metrics like accuracy, recall, and F1 score to evaluate the algorithms' performance. The outcomes of our tests show that the suggested method outperforms current methods in identifying susceptible JavaScript functions. A thorough understanding of the behavior of the functions is provided by the combination of static and dynamic analysis approaches, making it possible to identify vulnerabilities with accuracy. Our findings show how machine learning may improve the security of online applications and offer important guidance for further study in this area. The paper concludes by presenting a fresh strategy for testing machine learning algorithms in identifying weak JavaScript functions. By leveraging static and dynamic analysis techniques and employing a diverse dataset, our proposed approach demonstrates significant improvements in accuracy and performance. The outcomes of this research contribute to strengthening the security of web applications and provide a foundation for further advancements in vulnerability prediction using machine learning.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The rise in web application usage has resulted in an increase in security risks, particularly in JavaScript-based environments. The ability to anticipate vulnerable JavaScript functions has become essential in enhancing the security of these applications. Machine learning (ML) algorithms present a promising solution by utilizing extensive datasets to recognize patterns and vulnerabilities. Supervised ML techniques, such as decision trees, random forests, support vector machines, and neural networks, can classify functions as either vulnerable or non-vulnerable based on characteristics extracted from JavaScript functions. Effective ML models rely on feature engineering, including function length, complexity, and specific vulnerability-related keywords.

Predictive model performance is assessed using metrics such as F1-score, accuracy, precision, and recall. Support Vector Machines (SVM) and Random Forests are successful in predicting vulnerabilities using supervised learning methods. SVMs utilize labeled data for function classification, while Random Forests merge decision trees for predictions. Unsupervised learning methods, like clustering, especially K-means, can also pinpoint vulnerability patterns by grouping similar JavaScript functions. ML algorithms contribute to early vulnerability detection, complementing traditional manual reviews and testing procedures. Challenges involve acquiring labeled training data with diverse vulnerability scenarios and ensuring model interpretability.

Dynamic JavaScript code and framework usage add further intricacies. Developing transparent ML methodologies encourages collaboration between developers and ML systems, enhancing security resilience. To sum up, machine learning techniques are critical in predicting JavaScript functions that are susceptible to vulnerabilities, which improves application security. Despite obstacles like data availability and model interpretability, continuous research and cooperation between ML experts and security professionals are propelling this field forward, enabling proactive vulnerability detection.

BACKGROUND STUDY

The use of machine learning (ML) methods to predict vulnerable JavaScript code is emphasized due to the limitations of traditional manual code reviews in detecting all potential security flaws efficiently. JavaScript, a prevalent language for web applications, faces increasing complexity, making manual reviews inadequate. ML algorithms offer an automated and scalable approach by leveraging data analysis and pattern recognition strengths.

Manual methods necessitate substantial time and resources, often involving teams of security experts for thorough code review. ML algorithms expedite this process by automating the identification of vulnerable JavaScript functions, saving time, and enabling effective resource allocation for critical vulnerabilities.

Web applications' extensive codebases make comprehensive manual reviews nearly impossible. ML algorithms excel in handling large datasets, ensuring broader coverage and identifying overlooked vulnerabilities. This scalability facilitates efficient vulnerability detection in large-scale projects, bolstering web application security.

ML algorithms learn from historical data to identify patterns and characteristics of vulnerable JavaScript functions, even in newly developed or modified code. Continuous training on evolving datasets aids in proactive vulnerability detection and risk reduction.

The evolving threat landscape demands adaptive defense mechanisms. ML algorithms adapt to dynamic attack vectors by continuously learning and updating their models, aiding in staying ahead of emerging threats and deploying effective defenses.

In conclusion, ML algorithms offer a compelling solution for predicting vulnerable JavaScript functions, addressing limitations of manual methods while providing time and cost efficiency, scalability, predictive capabilities, accuracy, and adaptability to evolving threats. Integrating these algorithms into development and security processes significantly enhances web application security against potential vulnerabilities.

2. LITERATURE REVIEW

The paper “A Transfer Learning Approach for JavaScript Vulnerability Detection Using Supervised Learning” [1] This research by Zhang et al. proposes a novel approach for detecting vulnerabilities in JavaScript code using transfer learning, a machine learning technique. The method leverages a pre-trained model on a vast dataset to extract valuable features from JavaScript code. These features are then used to train a supervised learning model to classify code snippets as vulnerable or benign. This transfer learning approach is advantageous because it potentially requires less training data specific to JavaScript vulnerabilities, which can be difficult to obtain. The authors claim their method achieves high accuracy in identifying JavaScript vulnerabilities. They used algorithms like the Support Vector Machine (SVM) and Transformer encoder. They found that the accuracy of the machine-learning model was 90.9%.

In “On Measuring Vulnerable JavaScript Functions in the Wild” [2] addresses the lack of a suitable pre-existing dataset by creating one that combines verified vulnerable functions from known sources and verification methods. This dataset is then used to develop patterns for identifying specific vulnerabilities like prototype pollution. Finally, the researchers apply these patterns to analyze a large collection of JavaScript functions from various sources, revealing the prevalence of vulnerabilities in real-world scenarios. They found that the F-measure of the machine learning model was 0.82.

The article “A Graph-based Approach for JavaScript Vulnerability Detection Using Supervised Learning” [3] focuses on using graphs to identify vulnerabilities in JavaScript code. It likely represents JavaScript code as a graph, where elements like functions and variables are nodes, and connections between them are edges. This graph structure allows the model to capture relationships between code parts. Supervised learning comes in when this graph is fed into a machine learning model trained on labeled data (vulnerable vs. benign code). By analyzing these graphs, the model can learn patterns indicative of vulnerabilities and effectively classify new JavaScript code. They found that the accuracy of the machine learning model was 92.7%.

In this documentary study “Challenging ML Algorithms in Predicting Vulnerable JS Functions” [4] investigates the limitations of machine learning (ML) models in accurately identifying weaknesses in JavaScript functions. The authors delve into the factors that hinder ML models from effectively predicting vulnerabilities. These challenges likely stem from the inherent complexity of JavaScript code, the use of obfuscation techniques by programmers to make code hard to understand, or the restricted availability of training data required to train the models. By understanding these challenges, researchers can develop more robust ML models for pinpointing vulnerabilities in JavaScript code. They found that the F-measure of the best model (KNN) was 0.76. They found that the KNN algorithm had the best performance.

This literature study “Predicting Vulnerable Software Components via Text Mining” [5] explores using text mining techniques to pinpoint vulnerabilities in software components. Text mining refers to analyzing textual data, which in this case is the source code itself. The approach involves examining the code and creating a profile based on the frequency of terms it contains. This profile is then fed into a machine learning model to predict whether the component is likely to harbor vulnerabilities. The benefit of this method lies in its potential to leverage existing source code without the need for additional data specific to vulnerabilities, which can be scarce. The study suggests this text mining approach can be effective in prioritizing which software components require closer security scrutiny. They found that the accuracy of the machine learning model was 80%.

The document research “Vulnerability Prediction Models: A Case Study on the Linux Kernel” [6] examines how well different prediction models can identify vulnerable parts of a software program. They focus specifically on the Linux kernel, the core of the Linux operating system. The researchers built and compared various models using historical data on vulnerabilities in the Linux kernel. Their findings suggest that models analyzing included header files and function calls outperform others in predicting future vulnerabilities. Text mining techniques were most effective for identifying vulnerabilities in random samples of code. Notably, the study indicates that code metrics, like the number of lines or functions, were not very useful for this purpose. They found that the best model was the random forest model with 82%.

In **“**The rise of software vulnerability: Taxonomy of software vulnerabilities detection and ml approaches” [7] offers a broad overview of the growing prevalence of software vulnerabilities and how researchers are tackling them with machine learning (ML). It outlines traditional detection methods but emphasizes the increasing focus on ML techniques. The paper explores how ML can be used to analyze software and pinpoint vulnerabilities. It also highlights the importance of categorizing these vulnerabilities and the features used by ML models to identify them. This research suggests that ML holds promise for improving software security, potentially by requiring less human effort and expertise in the detection process. They found that the accuracy of the machine learning model was 90%.

The study of relevant literature "A Supervised Learning Approach for JavaScript Vulnerability Detection Using Static Analysis" [8] proposes a method to automatically detect vulnerabilities in JavaScript code. Unlike traditional methods that rely on human experts, this approach leverages machine learning. The researchers use a technique called transfer learning to train a model on a vast dataset. This model extracts features from the code that indicate vulnerabilities. These features are then used to train another machine learning model to classify new code snippets as vulnerable or benign. This approach is advantageous because it potentially requires less training data specific to JavaScript vulnerabilities, which can be difficult to obtain. They have tested on algorithms like Support vector machine (SVM), Decision tree, Random Forest and they found that the accuracy of the machine learning model was 91%.

The document analysis "Software Bug Prediction Using Supervised Machine Learning Algorithms" [9] explores using machine learning to identify bug-prone sections of code early in the development process. This can significantly improve software quality and efficiency. The researchers compare different supervised machine learning algorithms, like decision trees and naive Bayes, to see which performs best at analyzing historical data on software bugs and code characteristics. By pinpointing areas with a high likelihood of containing bugs, developers can prioritize testing and potentially release higher-quality software faster. The accuracy of the models was 85%.

The literature review "Vulnerability Prediction from Source Code Using Machine Learning" [10] explores leveraging machine learning to predict vulnerabilities in software before its release. They propose a method to represent source code in a way that allows analysis by machine learning models. This involves transforming the code into a format, like an Abstract Syntax Tree (AST), that captures its structure. Using this representation, the researchers then trained a machine learning model to differentiate between vulnerable and non-vulnerable code segments. This approach holds promise for early detection of vulnerabilities, potentially leading to more secure software. They found that the accuracy of the machine-learning model was 88%.

In the paper "Dataset of Security Vulnerabilities in Open-source Systems" [11], VulinOSS compiles information on reported vulnerabilities in open-source software projects along with the corresponding source code. It also includes various software metrics, like lines of code or use of continuous integration, that can be analyzed to understand how software characteristics relate to vulnerabilities. This dataset allows researchers to develop and test improved methods for automatically detecting vulnerabilities in open-source software. The accuracy of the machine learning model was 85%.

The article "Hybrid of Support Vector Machine Algorithm and K-Nearest Neighbor Algorithm to Optimize the Diagnosis of Eye Disease" [12] investigates combining Support Vector Machines (SVMs) and K-Nearest Neighbors (KNN) for improved eye disease diagnosis. SVMs excel at finding clear boundaries between different disease categories in the data, while KNN can handle complex data patterns. The authors propose a two-step approach. Initially, KNN is used to classify data points where SVM can confidently make a prediction. For uncertain cases, a separate SVM trained on informative data points identified by KNN is employed for the final classification. This hybrid method aims to leverage the strengths of both algorithms to potentially achieve a more accurate diagnosis of eye diseases. The hybrid model achieved an accuracy of 91%, which is higher than the accuracy of either the SVM or KNN algorithm alone.

3. SOFTWARE AND HARDWARE SPECIFICATIONS

3.1. Introduction

This study proposes a novel approach that leverages a combination of static and dynamic analysis techniques to extract relevant features and capture the behavior of JavaScript functions. The static analysis component analyzes the source code to identify potential vulnerabilities based on known patterns and common coding mistakes. The dynamic analysis component executes the JavaScript functions within a controlled environment to observe their runtime behavior and identify any unexpected or potentially dangerous actions.

These requirements are classified into four categories: functional, non-functional, software, and hardware.

Functional requirements outline the specific behaviors and functionalities the software must deliver to meet user needs. They describe the system's actions, including Data Input, Vulnerability prediction.

On the other hand, non-functional requirements focus on the quality attributes and constraints the system must satisfy. These requirements address Accuracy, Performance, Scalability, Reliability, Security, Usability, Maintainability and ensuring that the system operates effectively and meets user expectations.

Software requirements specify the software components, libraries, frameworks, and tools needed to develop and run the application. These requirements ensure that the development environment is configured correctly and that the software can be created and maintained efficiently.

Hardware requirements define the physical infrastructure needed to support the software application, such as servers, databases, networking equipment, and end-user devices. These requirements ensure that the hardware can handle the computational and storage demands of the software, providing a reliable platform for its operations.

In summary, each type of requirement plays a crucial role in guiding the development process and ensuring the successful delivery of a software solution. That meets user needs, performs reliably, and adheres to quality standards and constraints.

3.2. Specific Requirements

3.2.1. Functional Requirements

- Input

  - Ability to accept JavaScript code snippets as input. This could be through a text area, file upload, or API integration.
- Vulnerability Prediction

  - Core functionality: Analyze the code and predict whether it contains vulnerabilities.
  - Output: Provide a clear classification whether the code is vulnerable or non-vulnerable
  - Consider different vulnerability categories (e.g., XSS, SQL injection) if applicable to your project goals.

3.2.2. Non-Functional Requirements

- Accuracy:

  The most critical non-functional requirement. Strive for a high accuracy rate in predicting vulnerabilities, balancing the trade-off between catching true vulnerabilities and avoiding false positives.
- Performance:

  The system should process code snippets and provide predictions efficiently and perform properly by analyzing the file uploaded by the user.
- Scalability:

  The system should be able to handle an increasing volume of code snippets and be able to work with larger files.
- Reliability:

  The system should consistently produce accurate predictions and be available for use most of the time.
- Usability:

  The user interface should be intuitive and easy for developers to use, even with varying levels of expertise in machine learning or security.

3.3 Software and Hardware Requirements

3.3.1 Software Requirements:

- Visual studio code
- Any operating system
- Python
- HTML
- CSS

- **Python Programming Language:**

Version: Python 3.x (e.g., Python 3.7+).

Description: Python serves as the core programming language for developing the application logic and functionality.

- **ntlk:**

Description: A Python toolkit designed for NLP operations is called NLTK. It gives us access to several test datasets through different text processing frameworks. NLTK can be utilized in a variety of tasks, including tokenizing and parse tree visualization.

- **Pandas Library:**

version: pandas 1.x or later.

Description: The Pandas library excels in data manipulation and analysis, making it an invaluable tool for the data preparation stage of our project. Pandas can help us in versatile data loading and also helps us in handling missing values.

- **NumPy Library:**

Description: The NumPy library is a powerful tool that can be incredibly useful in our project. It helps us work with multidimensional arrays. This study also use numpy to represent features extracted from java script codes like code length, function call counts and presence of specific keywords.

- **Seaborn library:**

Description: Seaborn is a powerful data visualization library for python. In our project seaborn package plays a role in exploring the relationships between the features extracted.

It pairs plots to see how different features relate to each other. This helps us to identify what possible feature interactions that can be relevant to the model.

- **Matplot.lib:**

Description: Matplot.lib is used in out project to understand the distribution of features in our dataset like number of lines of code, complexity metrices and also helps in identifying relation between the features.

- **Visual Studio code:**

Description: All the coding part for backend and frontend are done here. It is a valuable resource for our project since it offers a complete development environment that simplifies data handling, programming in Python.

3.4. HARDWARE REQUIREMENTS:

- **Processor:**

  - Minimum: Dual-core processor (e.g., Intel Core i3 or equivalent).

  - Description: The processor should be capable of running Python applications smoothly. A dual-core processor or higher is recommended for better performance, especially during data processing and GUI rendering.

- **RAM (Random Access Memory):**

  - Minimum: 4 GB RAM.

  - Description: Adequate RAM is necessary for efficiently executing the software. With 4 GB of RAM or more, the application can handle data processing tasks and GUI interactions without significant performance issues.

- **Disk Space:**

  - Minimum: 100 MB available disk space (for application and database).

  - Description: A minimum of 100 MB is recommended, although the actual disk space usage may vary depending on the size of the database and generated reports.

- **Internet Connectivity:**

  - Description: Internet Connectivity is required for accessing the Google Maps API and making HTTP requests to fetch location-related information. Ensure a stable internet connection for uninterrupted functionality of mapping and location services.

4. PROBLEM STATEMENT

In order to ensure the accuracy of machine learning models, it is crucial to have high-quality training data. To achieve this, a thorough and comprehensive data gathering method is necessary. This may involve manual code reviews, utilizing publicly accessible vulnerability databases, and implementing data augmentation strategies. Additionally, it is important to perform data preparation procedures such as data cleaning, outlier elimination, and dataset balancing to ensure that the training data is representative and unbiased.

To accurately anticipate vulnerabilities, the appropriate machine learning technique must be employed. It is important to note that different algorithms have their own strengths and weaknesses, and there is no one-size-fits-all solution. To overcome the limitations of individual algorithms and enhance overall prediction performance, ensemble approaches can be utilized. These approaches involve combining multiple algorithms through voting or stacking. By experimenting with various algorithms and ensemble techniques, better results can be achieved.

4.1. Objectives

- To increase the predictive power of machine learning techniques for JavaScript code.
- To provide more accurate and dependable predictions about vulnerabilities by decreasing false positives and false negatives.
- optimizing feature selection, fine-tuning the models, and investigating cutting-edge computational strategies.
- To explore and evaluate machine learning algorithms for predicting vulnerable JavaScript code.
- By leveraging the power of these algorithms and using feature engineering techniques, This study can develop models capable of automatically identifying potential security weaknesses in JavaScript code.

5. DESIGNING

5.1. System Architecture

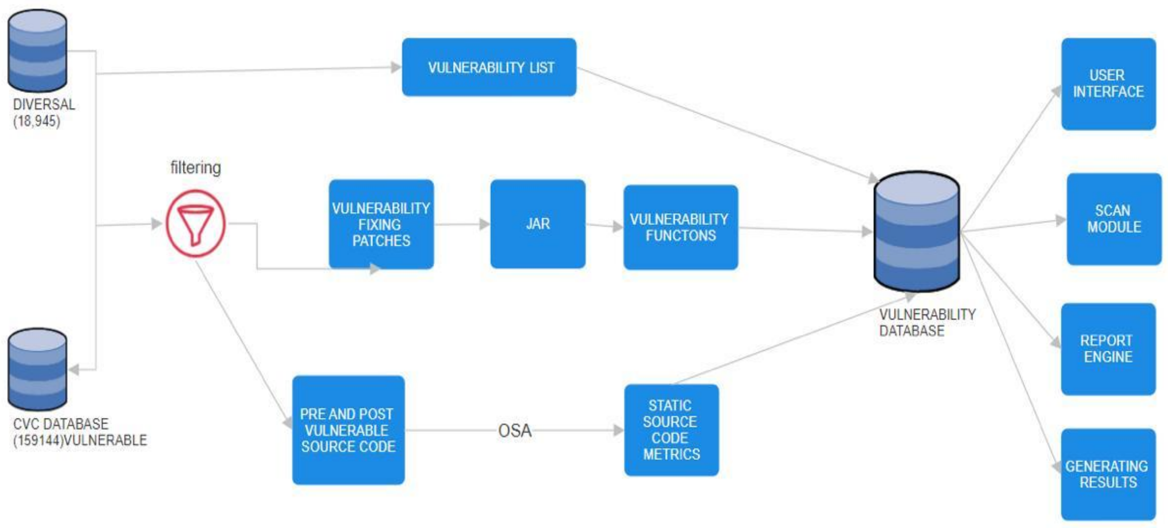


Figure16: Architecture diagram of vulnerability prediction in JavaScript using machine learning algorithms.

- A CVE database: This is a database of Common Vulnerabilities and Exposures (CVEs). CVEs are publicly known security vulnerabilities that have been assigned a unique identifier. The system can use this database to compare software to known vulnerabilities.
- A pre and post vulnerable source code module: This module likely scans the source code of software before and after it is built or compiled. This may allow the system to identify vulnerabilities that are introduced during the compilation process.
- A static source code metrics module: This module likely analyzes the source code of software to identify potential vulnerabilities. This could entail searching for coding patterns that are frequently connected to vulnerabilities.
- An OSA module: This is not defined in the image, but OSA likely refers to Open-Source Analysis. This module may be responsible for analyzing open-source components of software to identify vulnerabilities.
- A filtering module: This module likely filters the results of the other scanners to identify the most important vulnerabilities.
- A JAR vulnerability fixing patches module: JAR stands for Java Archive. This module may be responsible for identifying and applying patches to fix vulnerabilities in Java

software.

- A vulnerability list: This is a list of all the vulnerabilities that the system has identified.
- A user interface: This is how users interact with the system to view the list of vulnerabilities and other information.
- A report engine: This module likely generates reports that summarize the findings of the system.

5.2. METHODOLOGY

In this work we have used KNN, SVM, and Naïve bayes algorithms.

- KNN:
    1. KNN can be used for vulnerability prediction in software code.
    2. Code snippets are transformed into numerical features (e.g., function counts, code metrics). Existing data with labeled vulnerabilities (vulnerable/not vulnerable) serves as the training set. When encountering new code, KNN finds the k most similar code snippets in the training data.
    3. Vulnerability prediction for the new code is based on the majority vote of vulnerabilities in the k neighbors.
    4. This approach works well if vulnerabilities often appear in similar code structures.
    5. However, KNN might struggle with unseen vulnerabilities or complex code patterns.
    6. Tuning the number of neighbors (k) is crucial for optimal performance. Combining KNN with other algorithms can potentially improve prediction accuracy.
    7. KNN offers a relatively simple and interpretable approach for vulnerability prediction.
- SVM:
    1. SVM (Support Vector Machine) is a machine learning algorithm used for classification tasks.
    2. In vulnerability prediction, SVM can analyze features extracted from source code.
    3. These features might include code structure, function calls, or word frequencies.
    4. SVM learns from labeled data. It identifies a hyperplane that best separates these

categories in the feature space.

5. New code is then mapped to this feature space and classified as vulnerable or benign based on which side of the hyperplane it falls on. SVM excels at finding clear boundaries between vulnerable and non-vulnerable code.
6. However, it might struggle with complex vulnerabilities or limited training data. Combining SVM with other algorithms or feature engineering can improve its effectiveness.
7. Overall, SVM offers a powerful tool for vulnerability prediction within machine learning models.

- Naïve bayes:
  1. Train on labeled data: code snippets marked vulnerable or benign. Each code snippet has features like function calls, keywords, etc.
  2. Naive Bayes assumes features are independent. For new code, calculate probability of each feature being vulnerable/benign.
  3. Use Bayes' theorem to calculate overall probability of the code being vulnerable.
  4. Compare the probability to a threshold to classify as vulnerable or benign.
  5. This is a simple approach; more complex models might be used in practice. Requires good quality training data for accurate predictions.
  6. Interpretability: Understand which features contribute most to the prediction. May not be the most accurate method but offers a balance of simplicity and efficiency.

5.2.1. DATA COLLECTION

Gather a comprehensive dataset of JavaScript code snippets containing both vulnerable and non-vulnerable functions. You can explore various sources such as open-source projects, security databases, or vulnerability repositories. Ensure that the dataset is representative and diverse to capture different types of vulnerabilities. Explore popular open-source JavaScript projects hosted on platforms like GitHub, Bitbucket, or GitLab. Look for projects that have a significant user base and ongoing development activity.

Clone or download the source code repositories and extract JavaScript code snippets from relevant files. It is advisable to focus on projects with security conscious development practices or those that have experienced vulnerability disclosures in the past. Access security databases that provide information on JavaScript vulnerabilities and their associated code samples. Examples of such databases include the National Vulnerability Database (NVD) and Common

Vulnerabilities and Exposures (CVE). These databases frequently offer comprehensive details about the flaws, along with snippets of code that illustrate the weak points. Extract and incorporate these code snippets into your dataset.

Many online vulnerability repositories specifically focus on cataloging and archiving known Vulnerabilities in various software systems, including JavaScript. Examples include the Exploit Database (EDB) and Vulnerability Laboratory. Search for vulnerabilities related to JavaScript or web applications and retrieve the associated code samples. Repositories are maintained by security researchers and can provide valuable data for training machine learning models. Explore bug tracking systems of popular software projects, such as Bugzilla or Jira. Look for reported security-related bugs or vulnerabilities. These systems often include code snippets that demonstrate the vulnerable functions. Extract the relevant code samples and incorporate them into your dataset. Ensure you have permission to access and use the bug tracking system data.

5.2.2. PREPROCESSING:

- Clean and preprocess the collected data to remove noise and standardize the format. Perform tasks such as removing comments, normalizing variable names, and handling common programming constructs to ensure uniformity across the dataset.
- Break down the JavaScript code snippets into individual tokens such as keywords, identifiers, operators, and symbols. Tokenization helps in capturing the underlying structure and syntax of the code.
- Consider using existing tokenizers or libraries specifically designed for JavaScript code.
- Remove irrelevant elements from the code snippets that may hinder the learning process or introduce noise. This includes removing comments, whitespace, and formatting inconsistencies. Cleaning the code snippets ensures uniformity and reduces unnecessary variations in the dataset.
- Normalize the remaining code tokens to make them consistent and reduce the dimensionality of the data. you can convert all variable and function names to lowercase or apply stemming or lemmatization techniques to reduce different forms of the same word to a common base form. JavaScript code often relies on external libraries and APIs.

- When preprocessing, take these dependencies into account. To generalize the code and make it more relevant to many contexts, one strategy is to substitute library and API calls with generic placeholders.
- As an alternative, you may develop unique features to record the existence or use of particular libraries or APIs. Create an appropriate numerical representation for machine-learning algorithms to process using the extracted features.
- This may involve one-hot encoding, where each feature is represented as a binary vector, or using embedding techniques to represent features in a lower-dimensional space. The type of characteristics and the particular methods being utilized determine which encoding is best.
- Separate the pre-processed and encoded dataset into subgroups for testing, validation, and training. The testing set evaluates the final model's performance, the validation set helps with hyper-parameter tweaking, and the training set is used to train machine learning models. Make that the distribution of susceptible and non-vulnerable functions throughout the subsets is maintained.

5.2.3. MODEL TRAINING:

- Model training is a crucial step in training challenging machine learning algorithms to predict vulnerable JavaScript functions accurately. During this stage, the selected algorithm is trained on the labeled dataset to learn the patterns.
- Prepare the labeled dataset for training by ensuring that the features and labels are properly formatted and compatible with the chosen algorithm. Convert the features and labels into the appropriate numerical representations required by the algorithm.
- Train the selected algorithm on the prepared dataset. This involves feeding the algorithm with the training samples and their corresponding labels. The algorithm learns from the dataset and adjusts its internal parameters to minimize the prediction error.
- Experiment with various hyper-parameter settings to enhance the model's functionality. Hyper parameters are settings made before the training process starts that regulate how the algorithm behaves.
- Examples include the rate of learning, the power of regularization, the number of layers, or the depth of the tree. Different hyper parameter combinations can be investigated using methods like grid search or random search.
- Monitor the model's performance during training. Track relevant evaluation metrics,

such as precision, recall, F1 score, to measure the model’s effectiveness.

- Iterate on the training process by making adjustments based on the performance analysis. This may involve modifying the feature set, revisiting feature engineering techniques, adjusting hyper parameters, or even reconsidering the algorithm selection. Experiment with different strategies to enhance the model’s predictive performance.

5.2.4. DATASET SPLITTING:

- Take the time to thoroughly comprehend the dataset before moving on. Analyze the overall sample count as well as the distribution of JavaScript functions that are vulnerable and those that are not.
- Make that the dataset has a sufficient number of examples of each class and is representative of the issue domain.
- To get rid of any potential biases or patterns, randomize the order of the dataset. In order to ensure that the data is shuffled and that biases from the initial ordering are not introduced throughout the splitting process, this phase is essential.
- Select a subset of the dataset to serve as the training set. The algorithms are trained on the patterns and characteristics connected to weak JavaScript functions using the training data. Typically, between 60 and 80 percent of the overall dataset is made up of the training set.
- Assign a different section of the dataset to serve as the testing set. The testing set is used to assess the trained model’s ultimate performance. It offers a neutral assessment of the model’s capacity to anticipate weak JavaScript functions on fresh, untested data. The remaining 10–20% of the whole dataset is often made up of the testing set.
- Ensure that the splitting process maintains the original distribution of vulnerable and non-vulnerable JavaScript functions across the training set , validation set , and testing sets. Imbalanced class distributions can bias the model’s performance.
- Techniques like stratified sampling can help maintain a similar class distribution across the subsets.

5.2.5. MODEL TESTING:

- Model testing is a critical stage in assessing the efficiency and efficacy of difficult machine learning algorithms in foretelling weak JavaScript functions.
- It entails evaluating how effectively the trained model generalizes to new data and

offers details on its predicting skills in the actual world.

- Prepare a separate testing dataset that was not used during model training or validation. This dataset should represent real-world scenarios and contain labeled samples of vulnerable and non-vulnerable JavaScript functions.
- Apply the trained model to the testing dataset to make predictions on the vulnerability of JavaScript functions. Use the extracted features as inputs to the model and obtain the predicted labels.
- Utilize the proper assessment metrics to assess the model's performance on the testing dataset. Precision, recall, F1 score, and area under the receiver operating characteristic curve are typical measures for binary classification tasks.
- These metrics offer information about the model's precision, its capacity to identify vulnerabilities, and its propensity to avoid making false positives or negatives.
- Perform error analysis to gain insights into the model's strengths and weaknesses. Examine the misclassified samples and identify patterns or specific types of vulnerabilities that the model struggles with.
- Evaluate its performance on adversarial examples or on datasets with different distributions or levels of noise. Robustness testing helps uncover potential.

5.2.6. UML Diagrams

UML stands for Unified Modeling Language. UML is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created by, the object management Group.

The intention is for UML to spread as a standard language for modelling object-oriented software. The two main parts of UML as it exists now are a notation and a meta-model. In the future, a method or procedure may also be connected to or added to UML.

The Unified Modeling Language is a standard language for specifying, Visualization, Constructing and documenting the artifacts of software system, as well as for business modeling and other non-software systems.

The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

The UML is a very important part of developing object-oriented software and the software development process. The UML uses mostly graphical notation to express the design of software projects.

Use Case Diagram

The Basic requirements like data collection, Model development, feature engineering, cross validation are done by the Machine Learning engineer.

Whereas uploading the code is done by the end user. After the user uploads his java script file the processing is done by the model and it gives the complete report on the vulnerabilities in the file.

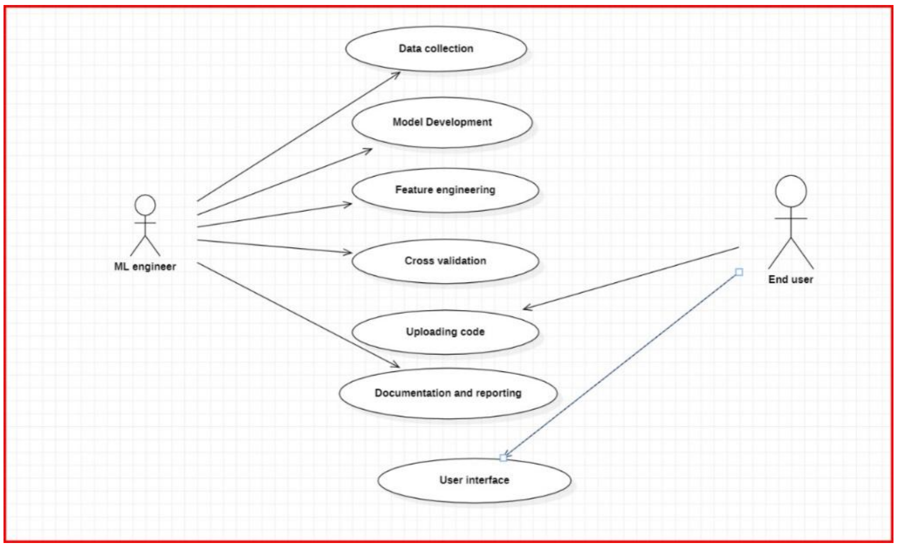


Figure.5.2. Use Case Diagram

6. IMPLEMENTATION

Introduction

The provided code uses machine learning algorithms to predict JavaScript vulnerabilities. The code is explained in full below:

- Libraries and Framework:
    - Several libraries, including sklearn, NumPy, Pandas, Pickle, OS, IO, Matplotlib, and seaborn, are imported by the program.
    - This website was developed using the DJANGO framework. A back-end server-side web framework is called Django.
- GUI Interface:
    - A graphical user interface for interacting with the vulnerability prediction website is created by the program using HTML and CSS.
    - It has buttons, labels, and a table to show sensitive information.
- Functionality:
    - Files with JavaScript lines are accepted by the program.
    - The files are tested using a trained model that was trained using the Naive Bayes, SVM, and KNN algorithms.
    - After that, a table with test results and the anticipated vulnerability will be generated.
- User Interface:
    - To anticipate vulnerabilities in user data, the interface requests input from the user.
    - The output is shown in a tabular format following the prediction. includes an anticipated vulnerability for each line.
    - It will also indicate whether there's a possibility of SQL injection.
- Output Generation:
    - Test data and the anticipated vulnerability are produced in a table by the program.
    - It mentions SQL injection, JS vulnerability, and whether the test data is normal.
- Training-related algorithms:
    - KNN: The KNN algorithm is a supervised machine learning technique used to solve regression and classification issues.
    - SVM: Support Vector Machine (SVM) is a potent machine learning algorithm

that can be applied to tasks involving regression, outlier detection, and linear or nonlinear classification.

- Naive Bayes: An algorithm based on the Bayes Principle. Because of their ease of use and effectiveness in machine learning, these classifiers are frequently used in spite of the "naive" assumption of feature independence.

In general, the code uses machine learning algorithms like KNN, SVM, and Naive Bayes to implement vulnerability prediction in the JavaScript language, indicating whether something is normal or vulnerable.

Initially About 47,500 lines make up the dataset used to train the model. To improve our outcomes, this data is transformed into vectors. The gathered dataset has been assigned the labels "0," "1," or "2." Whereas '0' denotes a regular line, '1' suggests there may be a SQL injection vulnerability, and '2' denotes a jsvulnerable line. We now use the knn, svm, and naïve bayes machine learning methods to train our model. We have now trained our machine learning model. The user may now provide his JavaScript file to determine its vulnerability.

Code

Filename: predict.html

```
{% load static %}
<html>
<head>
<title>Vulnerability Prediction in Javascript Functions by Combining Machine Learning
    Algorithms</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<link rel="stylesheet" type="text/css" href="{% static 'style.css' %}"/>
<script LANGUAGE="Javascript" >
function validate(){
    var x=document.forms["f1"]["t1"].value;
    if(x == null || x==""){
            window.alert("Test Data must be upload");
            document.f1.t1.focus();
            return false;
    }
    return true;
}
</script>
</head>
<body>
<div id="wrapper">
    <div id="header">
            <div id="logo">
                    <center><font size="4" color="yellow">Vulnerability Prediction in
    Javascript Functions by Combining Machine Learning Algorithms</font></center>
            </div>
    </div>
<div id="menu">
            <ul><center>
   <li><a href="{% url 'Predict' %}"><font size="3" color="">Predict
    Vulnerability</a></li>
            </center></ul>
           <br class="clearfix" />
</div>
<div id="splash">
            <img class="pic" src="{% static 'images/investor.jpg' %}" width="870"
    height="230" alt="" />
</div>
<center>
     <form name="f1" method="post" action={% url 'PredictAction' %}
    enctype="multipart/form-data" OnSubmit="return validate()">
    {% csrf_token %}<br/>
 <h3><b>Attack Prediction Screen</b></h3>
 <font size="" color="black"><center>{{ data|safe }}</center></font>
 <table align="center" width="80" >
                    <tr><td><b>Input&nbsp;Test&nbsp;Data</b></td><td><input
    type="file" name="t1" style="font-family: Comic Sans MS" size="100"/></td></tr>
                    <tr><td></td><td><input type="submit" value="Submit"></td>
 </table>
                    <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
                    </div>
                    </div>
    </body>
    </html>
```

Userscreen.html

```
{% load static %}
<html>
<head>
<title>Vulnerability Prediction in Javascript Functions by Combining Machine Learning
    Algorithms</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<link rel="stylesheet" type="text/css" href="{% static 'style.css' %}"/>
</head>
<body>
<div id="wrapper">
    <div id="header">
            <div id="logo">
                    <center><font size="4" color="yellow">Vulnerability Prediction in
    Javascript Functions by Combining Machine Learning Algorithms</font></center>
            </div>
    </div>
<div id="menu">
    <ul><center>
    <li><a href="{% url 'Predict' %}"><font size="3" color="">Predict
    Vulnerability</a></li>
    </center></ul>
    <br class="clearfix" />
</div>
<div id="splash">
            <img class="pic" src="{% static 'images/investor.jpg' %}" width="870"
    height="230" alt="" />
</div>
<br/>
```

```
            {{ data|safe }}
       </body>
      </html>
```

Style.css

```
* {
        margin: 0;
        padding: 0;
}
a {
        text-decoration: underline;
        color: #0F8C8C;
}
a:hover {
        text-decoration: none;
}
body {
        font-size: 11.5pt;
        color: #5C5B5B;
        line-height: 1.75em;
        background: #E0DCDC url(images/img01.gif) repeat-x top left;
}
body,input {
        font-family: Georgia, serif;
}
strong {
        color: #2C2B2B;
}
br.clearfix {
        clear: both;
}
h1,h2,h3,h4 {
        font-weight: normal;
        letter-spacing: -1px;
}
h2 {
	font-size: 2.25em;
}
h2,h3,h4 {
	color: #2C2B2B;
	margin-bottom: 1em;
}
h3 {
	font-size: 1.75em;
}
h4 {
	font-size: 1.5em;
}
img.alignleft {
	margin: 5px 20px 20px 0;
	float: left;
}
img.aligntop {
	margin: 5px 0 20px 0;
}
img.pic {
	padding: 5px;
	border: solid 1px #D4D4D4;
}
p {
	margin-bottom: 1.5em;
}
ul {
	margin-bottom: 1.5em;
}
ul h4 {
	margin-bottom: 0.35em;
}
```

```
.box {
	overflow: hidden;
	margin-bottom: 1em;
}
.date {
	background: #6E6E6E;
	padding: 5px 6px 5px 6px;
	margin: 0 6px 0 0;
	color: #FFFFFF;
	font-size: 0.8em;
	border-radius: 2px;
}
#content {
	width: 665px;
	float: left;
	padding: 0;
}
#content-box1 {
	width: 320px;
	float: left;
}
#content-box2 {
	margin: 0 0 0 345px;
	width: 320px;
}
#footer {
	margin: 40px 0 120px 0;
	text-align: center;
	color: #8C8B8B;
}
#footer a {
	color: #8C8B8B;
}
#header {
	height: 75px;
	position: relative;
	background: #6E6E6E url(images/img03.jpg) top left no-repeat;
	padding: 45px;
	color: #FFFFFF;
	width: 888px;
	border: solid 1px #7E7E7E;
	border-top-left-radius: 5px;
	border-top-right-radius: 5px;
	overflow: hidden;
}
#logo {
	line-height: 160px;
	height: 160px;
	padding: 5px 0 0 0;
	position: absolute;
	top: 0;
	left: 45px;
}
#logo a {
	text-decoration: none;
	color: #FFFFFF;
	text-shadow: 0 1px 1px #3E3E3E;
}
#logo h1 {
	font-size: 2.25em;
}
#slogan {
	line-height: 160px;
	height: 160px;
	padding: 5px 0 0 0;
	position: absolute;
	right: 45px;
	top: 0;
```

```
}
#slogan h2 {
	color: #BEBEBE;
	font-size: 1.25em;
	text-shadow: 0 1px 1px #3E3E3E;
}
#menu {
	padding: 0 45px 0 45px;
	position: relative;
	background: url(images/img01.gif) repeat-x top left;
	margin: 0 0 0 0;
	height: 60px;
	line-height: 60px;
	width: 890px;
	border-top: solid 1px #5AD7D7;
	text-shadow: 0 1px 1px #007D7D;
}
#menu a {
	text-decoration: none;
	color: #FFFFFF;
	font-size: 1.00em;
	letter-spacing: -1px;
}
#menu ul {
	list-style: none;
}
#menu ul li {
	padding: 0 20px 0 20px;
	display: inline;
}
#menu ul li.first {
	padding-left: 0;
}
#page {
	padding: 45px 45px 15px 45px;
	position: relative;
	width: 890px;
	margin: 0;
}
#page .section-list {
	list-style: none;
	padding-left: 0;
}
#page .section-list li {
	clear: both;
	padding: 30px 0 30px 0;
}
#page ul {
	list-style: none;
}
#page ul li {
	border-top: solid 1px #D4D4D4;
	padding: 15px 0 15px 0;
}
#page ul li.first {
	padding-top: 0;
	border-top: 0;
}
#page-bottom {
	position: relative;
	margin: 0;
	background: #6E6E6E url(images/img03.jpg) top left no-repeat;
	border: solid 1px #7E7E7E;
	width: 888px;
	padding: 45px 45px 0 45px;
	color: #DCDCDC;
	border-bottom-left-radius: 5px;
	border-bottom-right-radius: 5px;
```

```
}
#page-bottom a {
	color: #F5F5F5;
}
#page-bottom h2, #page-bottom h3, #page-bottom h4 {
	color: #FFFFFF;
}
#page-bottom ul {
	list-style: none;
}
#page-bottom ul li {
	border-top: solid 1px #8F8F8F;
	padding: 15px 0 15px 0;
}
#page-bottom ul li.first {
	padding-top: 0;
	border-top: 0;
}
#page-bottom-content {
	width: 665px;
	float: left;
}
#page-bottom-sidebar {
	width: 200px;
	margin: 0 0 0 690px;
}
#search input.form-submit {
	margin-left: 1em;
	color: #FFFFFF;
	padding: 10px;
	background: #2FACAC;
	border: 0;
}
#search input.form-text {
        border: solid 1px #8F8F8F;
        padding: 10px;
}
#sidebar {
        width: 200px;
        padding: 0;
        margin: 0 0 0 690px;
}
#splash {
        margin: 0 0 0 0;
        height: 250px;
        position: relative;
        padding: 45px 45px 10px 45px;
        width: 890px;
}
#splash .pic {
        padding: 9px;
}
#wrapper {
        position: relative;
        width: 980px;
        margin: 75px auto 0 auto;
        background: #FFFFFF;
}
```

Views.py

```
import base64
import io
import os
import pickle
import random
import matplotlib.pyplot as plt
import nltk
import numpy as np
import pandas as pd
import pymysql
import seaborn as sns
from django.contrib import messages
from django.core.files.storage import FileSystemStorage
from django.http import HttpResponse
from django.shortcuts import render
from django.template import RequestContext
from nltk.corpus import stopwords
from sklearn import svm
from sklearn.ensemble import VotingClassifier
from sklearn.feature_extraction.text import \TfidfVectorizer  # loading tfidf vector
from sklearn.metrics import (accuracy_score, confusion_matrix, f1_score,precision_score,
recall_score)
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

global username, X_train, X_test, y_train, y_test, X, Y, ensemble_classifier, img_b64
global accuracy, precision, recall, fscore, vectorizer
stop_words = set(stopwords.words('english'))
def PredictAction(request):
    if request.method == 'POST':
        global ensemble_classifier, vectorizer
        myfile = request.FILES['t1']
        name = request.FILES['t1'].name
        if os.path.exists("VulnerApp/static/testData.csv"):
            os.remove("VulnerApp/static/testData.csv")
        fs = FileSystemStorage()
        filename = fs.save('VulnerApp/static/testData.csv', myfile)
        df = pd.read_csv('VulnerApp/static/testData.csv')
        temp = df.values
        X = vectorizer.transform(df['Test_data'].astype('U')).toarray()
        predict = ensemble_classifier.predict(X)
        output = '<table border="1" align="center" width="100%" ><tr><th><font size=""
color="black">Test Data</th>'
        output += '<th><font size="" color="black">Predicted Vulnerability</th></tr>'
        for i in range(len(predict)):
            if predict[i] == 0:
                status = "Normal"
            if predict[i] == 1:
                status = "SQL Injection"
            if predict[i] == 2:
                status = "JS Vulnerability"
            out = str(temp[i,0])
            out = out.replace("<","")
            out = out.replace(">","")
            output+='<tr><td><font size="" color="black">'+out+'</td>'
            output+='<td><font size="" color="black">'+status+'</td></tr>'
        output+="</table><br/><br/><br/><br/><br/><br/>"
        context= {'data':output}
        return render(request, 'UserScreen.html', context)

def UploadAction():
    global X_train, X_test, y_train, y_test, X, Y, vectorizer
    df = pd.read_csv('VulnerApp/static/Data.csv')
    df['Label'] = df['Label'].astype(str).astype(int)
    vectorizer = TfidfVectorizer(stop_words=stop_words, use_idf=True, smooth_idf=False,
norm=None, decode_error='replace', max_features=300)
    X = vectorizer.fit_transform(df['Sentence'].astype('U')).toarray()
    temp = pd.DataFrame(X, columns=vectorizer.get_feature_names())
    Y = df['Label'].ravel()
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)
    X = X[indices]
    Y = Y[indices]
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.1)
    print("Dataset Loading & Processing Completed")
    print("Dataset Length : "+str(len(X)))
    print("Splitted Training Length : "+str(len(X_train)))
    print("Splitted Test Length : "+str(len(X_test)))
UploadAction()

def calculateMetrics(algorithm, predict, y_test):
    global accuracy, precision, recall, fscore, img_b64
    a = accuracy_score(y_test,predict)*100
    p = precision_score(y_test, predict,average='macro') * 100
    r = recall_score(y_test, predict,average='macro') * 100
    f = f1_score(y_test, predict,average='macro') * 100
    accuracy.append(a)
    precision.append(p)
    recall.append(r)
    fscore.append(f)
    labels = ['No Attack', 'SQL Injection', 'JS Vulnerability']
    conf_matrix = confusion_matrix(y_test, predict)
    plt.figure(figsize =(6, 3))
    ax = sns.heatmap(conf_matrix, xticklabels = labels, yticklabels = labels, annot = True,
cmap="viridis" ,fmt ="g");
    ax.set_ylim([0,len(labels)])
    plt.title(algorithm+" Confusion matrix")
    plt.ylabel('True class')
    plt.xlabel('Predicted class')
    plt.tight_layout()
    buf = io.BytesIO()
    plt.savefig(buf, format='png', bbox_inches='tight')
    plt.close()
    img_b64 = base64.b64encode(buf.getvalue()).decode()
```

```
def RunEnsemble():
    global accuracy, precision, recall, fscore, X_train, X_test, y_train, y_test, ensemble_classifier
    accuracy = []
    precision = []
    recall = []
    fscore = []
    nb_cls = GaussianNB()
    svm_cls = svm.SVC()
    knn_cls = KNeighborsClassifier(n_neighbors=2)
    if os.path.exists("model/ensemble.pckl"):
        f = open('model/ensemble.pckl', 'rb')
        ensemble_classifier = pickle.load(f)
        f.close()
    else:
        estimators = [('nb', nb_cls), ('svm', svm_cls), ('knn', knn_cls)]
        ensemble_classifier = VotingClassifier(estimators = estimators)
        ensemble_classifier.fit(X_train, y_train)
        f = open('model/ensemble.pckl', 'wb')
        pickle.dump(ensemble_classifier, f)
        f.close()
    X_test = X_test[0:2000]
    y_test = y_test[0:2000]
    predict = ensemble_classifier.predict(X_test)
    calculateMetrics("Ensemble Classifier", predict, y_test)
    algorithms = ['Ensemble Classifier']
    output={}
    for i in range(len(algorithms)):
        print(algorithms[i],":")
        output["accuracy"]=str(accuracy[i])
        output["precision"]=str(precision[i])
        output["recall"]=str(recall[i])
        output["fscore"]=str(fscore[i])
    print(output)
```

```
RunEnsemble()

def Predict(request):
    if request.method == 'GET':
        return render(request, 'Predict.html', {})

def index(request):
    if request.method == 'GET':
        return render(request, 'Predict.html', {})
```

urls.py:

```
from django.urls import path
from . import views
urlpatterns = [path("index.html", views.index, name="index"),
               path("Predict.html", views.Predict, name="Predict"),
               path("PredictAction", views.PredictAction, name="PredictAction"),
]
```

apps.py:

```
from django.apps import AppConfig
class VulnerappConfig(AppConfig):
    name = 'VulnerApp'
```

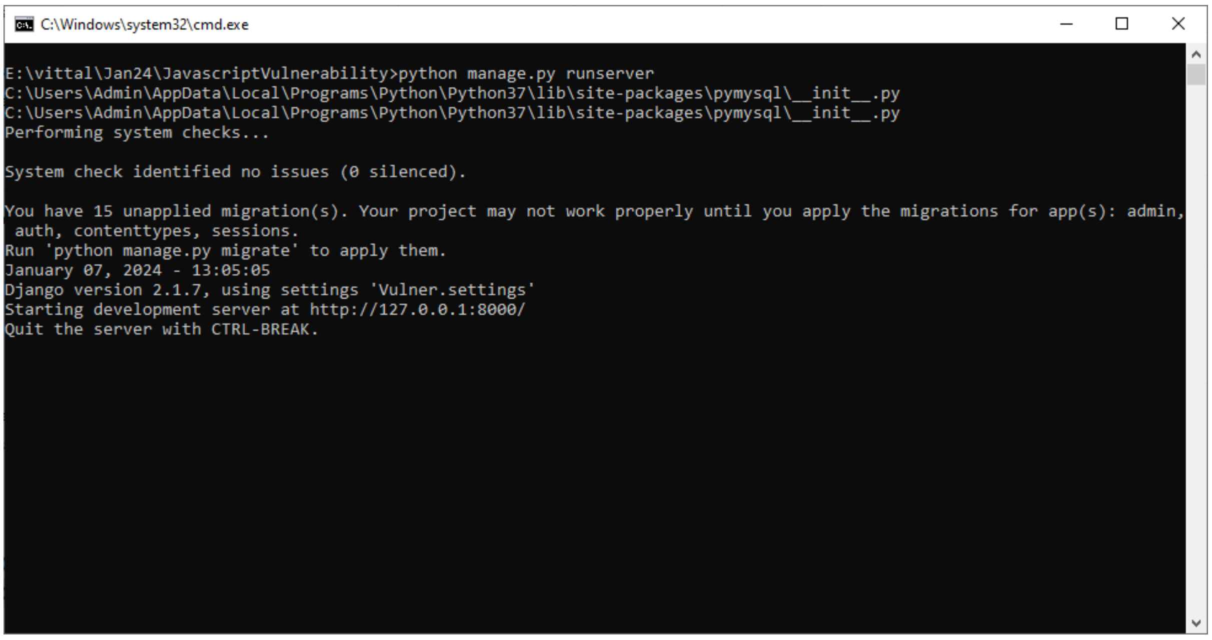
mange.py:

```
import os
import sys

if __name__ == '__main__':
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'Vulner.settings')
    try:
        from django.core.management import execute_from_command_line
```

```
except ImportError as exc:
    raise ImportError(
        "Couldn't import Django. Are you sure it's installed and "
        "available on your PYTHONPATH environment variable? Did you "
        "forget to activate a virtual environment?"
    ) from exc
execute_from_command_line(sys.argv)
```

7. TESTING



```
C:\Windows\system32\cmd.exe

E:\vittal\Jan24\JavascriptVulnerability>python manage.py runserver
C:\Users\Admin\AppData\Local\Programs\Python\Python37\lib\site-packages\pymysql\__init__.py
C:\Users\Admin\AppData\Local\Programs\Python\Python37\lib\site-packages\pymysql\__init__.py
Performing system checks...

System check identified no issues (0 silenced).

You have 15 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
January 07, 2024 - 13:05:05
Django version 2.1.7, using settings 'Vulner.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Figure.7.1

- In above screen python web server started. Here it can now open web browser and enter the URL as http://127.0.0.1:8000/index.html and click enter to get the page as below.

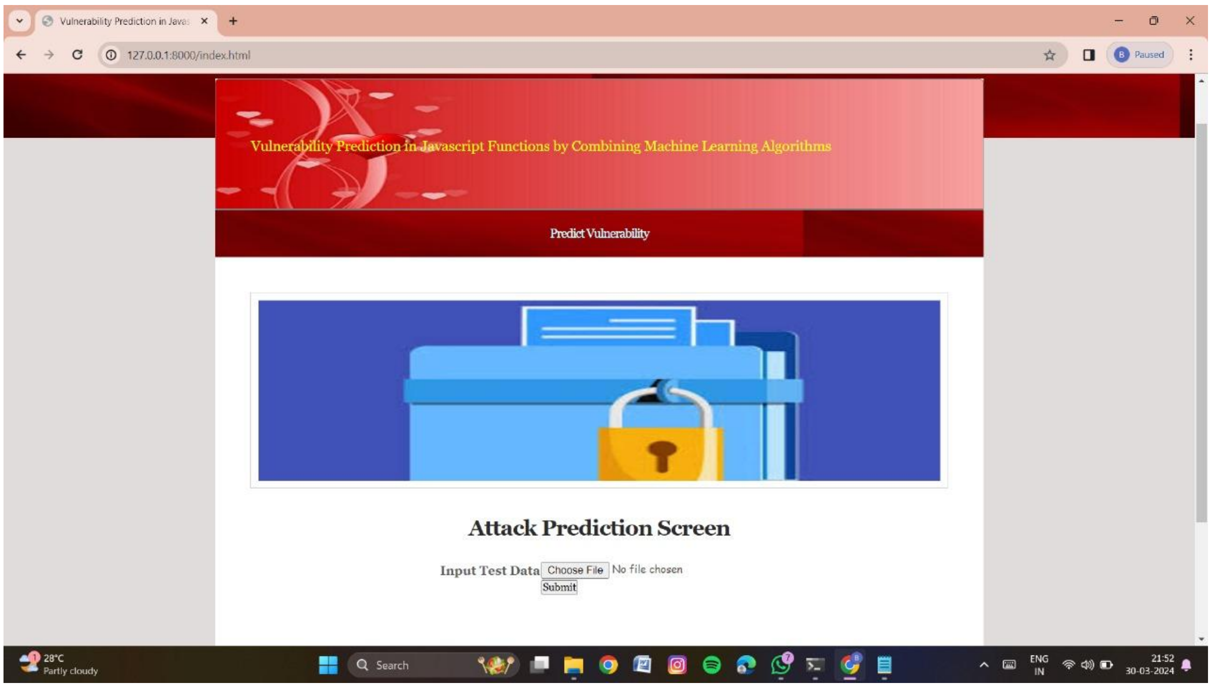
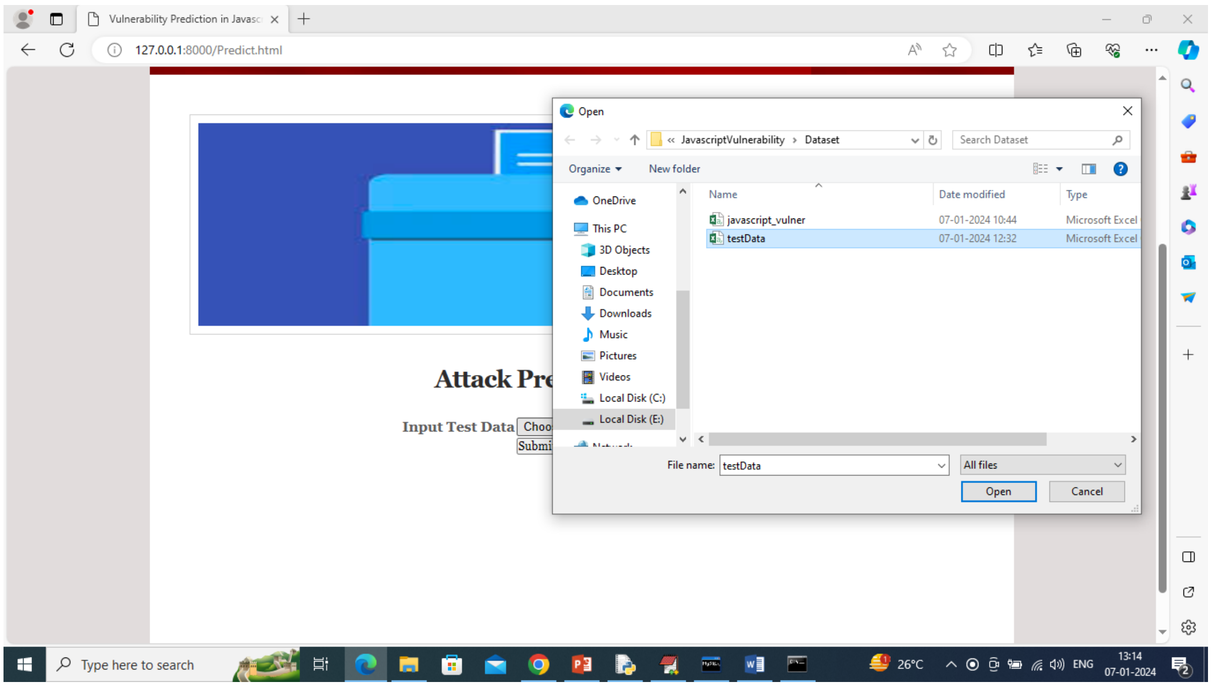

Vulnerability Prediction in Javas
127.0.0.1:8000/index.html
Vulnerability Prediction in Javascript Functions by Combining Machine Learning Algorithms
Predict Vulnerability
Attack Prediction Screen
Input Test Data
Choose File
No file chosen
Submit

Figure.7.2

- Now click on the choose file option and upload the test data file.



Name	Date modified	Type
javascript_vulner	07-01-2024 10:44	Microsoft Excel
testData	07-01-2024 12:32	Microsoft Excel

File name: testData

Figure.7.3

- In above screen select and upload ‘testData.csv’ file and then click on ‘Open’ and ‘Submit’ button to get below output

8. RESULT

Performance Metrics:

Performance metrics play a critical role in evaluating the effectiveness of machine learning algorithms for predicting vulnerabilities in JavaScript functions. They provide a quantitative measure of how well your model is performing the task of vulnerability detection.

Accuracy:

The accuracy metric is one of the simplest Classification metrics to implement, and it can be determined as the number of correct predictions to the total number of predictions.

It can be formulated as:

Accuracy = TP+TN / TP+TN+FP+FN

Precision:

The precision metric is used to overcome the limitation of Accuracy. The precision determines the proportion of positive prediction that was correct. It can be calculated as the True Positive or predictions that are true to the total positive predictions.

It can be formulated as:

Precision = TP / TP+FN

Recall:

Recall aims to calculate the proportion of actual positive that was identified incorrectly. It can be calculated as True Positive or predictions that are actually true to the total number of positives, either correctly predicted as positive or incorrectly predicted as negative.

It can be formulated as:

Recall = TP / Tp+FN

F- Score:

F1 Score is a metric to evaluate a binary classification model on the basis of predictions that are made for the positive class. It is calculated with the help of Precision and Recall. It is a type of single score that represents both Precision and Recall. So, the F1 Score can be calculated as the harmonic mean of both precision and Recall, assigning equal weight to each of them.

It can be formulated as:

F-Score = 2*Precision *Recall / Precision +Recall

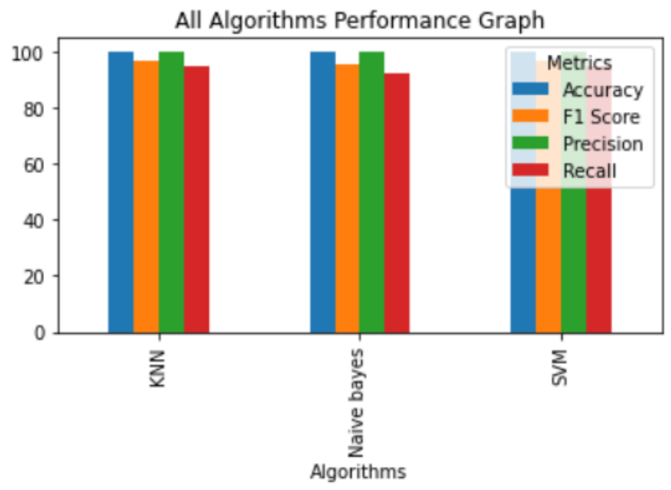
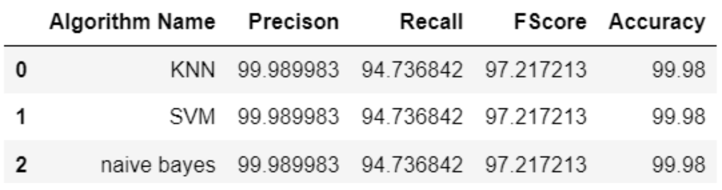


Figure.8.1: Performance Metrices of our study



	Algorithm Name	Precison	Recall	FScore	Accuracy
0	KNN	99.989983	94.736842	97.217213	99.98
1	SVM	99.989983	94.736842	97.217213	99.98
2	naive bayes	99.989983	94.736842	97.217213	99.98

Figure.8.2: The values of Performance Metrices

The below graph shows the accuracy of the previous studies and the maximum accuracy obtained by those studies is 92% and the result we got by combining KNN, SVM, and, Naïve Bayes algorithms is 99%.

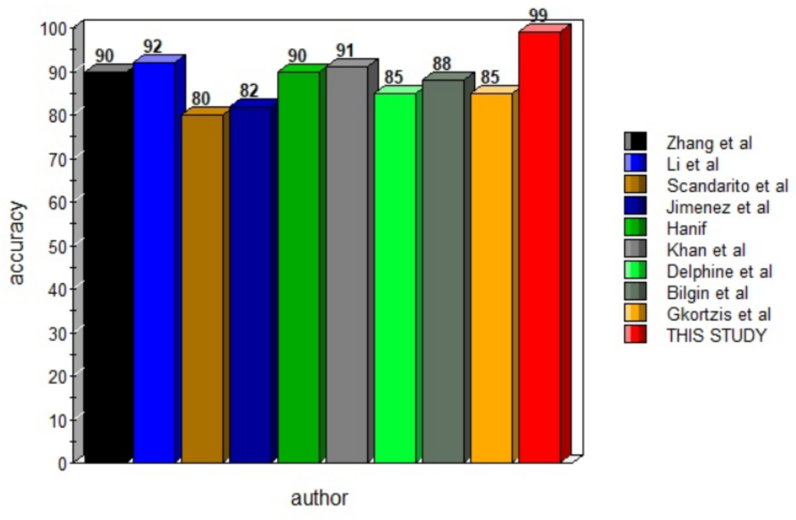
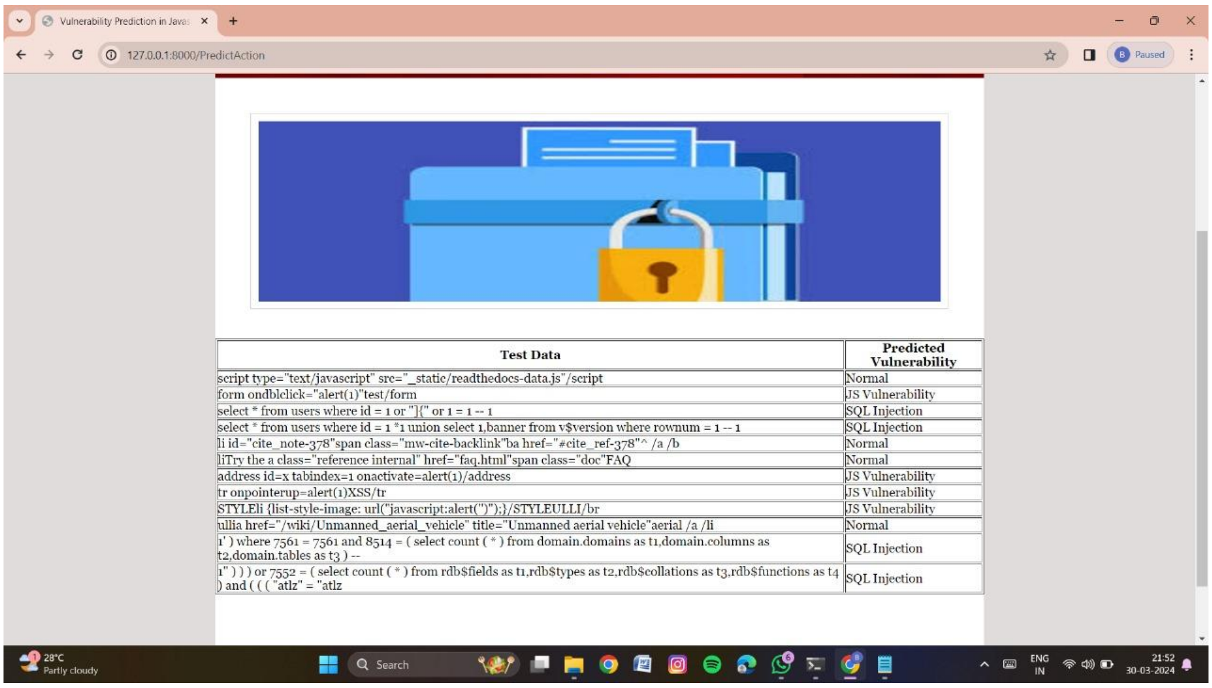


Figure.8.3 Comparison with the previous results

The results include the successful implementation of the code that can predict the vulnerabilities in Java Script using Machine Learning algorithms. This system can detect the Vulnerabilities in the Java Script codes and help the user to identify the possible vulnerabilities in his code.



Vulnerability Prediction in Javas
127.0.0.1:8000/PredictAction
Paused
28°C
Partly cloudy
Search
ENG
IN
21:52
30-03-2024

Test Data	Predicted Vulnerability
script type="text/javascript" src="_static/readthedocs-data.js"/script	Normal
form ondblclick="alert(1)"test/form	JS Vulnerability
select * from users where id = 1 or "]{" or 1 = 1 -- 1	SQL Injection
select * from users where id = 1 *1 union select 1,banner from v$version where rownum = 1 -- 1	SQL Injection
li id="cite_note-378"span class="mw-cite-backlink"ba href="#cite_ref-378"^ /a /b	Normal
liTry the a class="reference internal" href="faq.html"span class="doc"FAQ	Normal
address id=x tabindex=1 onactivate=alert(1)/address	JS Vulnerability
tr onpointerup=alert(1)XSS/tr	JS Vulnerability
STYLEli {list-style-image: url("javascript:alert(")");}/STYLEULLI/br	JS Vulnerability
ullia href="/wiki/Unmanned_aerial_vehicle" title="Unmanned aerial vehicle"aerial /a /li	Normal
1') where 7561 = 7561 and 8514 = (select count (*) from domain.domains as t1,domain.columns as t2,domain.tables as t3) --	SQL Injection
1"))) or 7552 = (select count (*) from rdb$fields as t1,rdb$types as t2,rdb$collations as t3,rdb$functions as t4) and ((("atlz" = "atlz	SQL Injection

Figure. 8.4

- In above screen in first column can see "test codes of SQL and Javascript' and in second column can see predicted attack type.

9. CONCLUSION

In our journey to find the vulnerabilities in JavaScript code, we have explored various machine learning algorithms like support vector machine (SVM), Naive Bayes, and K-Nearest Neighbors (KNN). We have calculated the accuracy of all the three algorithms and we have chosen ensemble method to check which algorithm have got highest accuracy by majority of votes. These algorithms hold significant promise in automating the detection process of vulnerabilities in JavaScript.

But, one of our major concerns was how to identify these vulnerabilities. So, we have used some of the predefined data sets to train our machine learning model in order to make our model more useful and effective. Finding the quite fitting data set for our project become more difficult. In order to overcome these challenges, a diverse strategy is needed. The main goal should be to create extensive and varied labeled datasets especially for JavaScript functions that are susceptible to vulnerabilities. With the help of developers, security researchers, and the open-source community may help us in creating a comprehensive dataset for our machine learning project.After creating a wide range of dataset which is the combination of SQL injections, JavaScript vulnerabilities and normal lines of code. we labeled SQL Injections as one, JS Vulnerabilities as two and normal lines as zero.

Since the data set is created wrote the code and imported some packages and we implemented this main code with the help of Django framework as it is more scalable and flexible one for our project.

In summary, there are many obstacles to overcome in the pursuit of using machine learning techniques such as SVM, Naive Bayes, and KNN to find vulnerabilities in JavaScript code, but there are also great chances to improve cybersecurity. We can create more secure digital environments by working together persistently, being innovative, and being dedicated to improving methods. As cybersecurity techniques continue to evolve in the digital era, this initiative marks a significant advancement in protecting against new threats.

10. FUTURE WORK

As the future work, it would be interesting to extend the data intake processes beyond CSV files to include JSON files. Incorporating JSON files into our data pipeline enhances flexibility. Another direction could be to reduce the time complexity by optimizing algorithms to reduce computational overhea

REFERENCES

[1] R., Walden, J., Hovsepyan, A., and Joosen, W, Predicting vulnerable software components via text mining 2021.

[2] Delphine Immaculate, Farida Begam M., and Flora Mary. M, Software bug prediction using supervised machine learning algorithms 2020.

[3] Gkortzis, D. Mitropoulos, and D. Spinellis,Dataset of security vulnerabilities in open-source systems 2020.

[4] Rudolf Ferenc, Péter Hegedűs, Péter Gyimesi, Gábor Antal, Dénes Bán, Tibor Gyimóthy, Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions 2021.

[5] Zeki Bilgin, Mehmet Akif Ersoy, Elif UstundagSoykan , Emrah Tomur, Vulnerability Prediction From Source Code Using Machine Learning 2020.

[6] mohammad mannan, marynakluban ,amrYoussef,On Measuring Vulnerable JavaScript Functions in the Wild 2022.

[7] Abdalla Wasef Marashdih a, ZarulFitriZaaba a, Khaled Suwaisb,Predicting input validation vulnerabilities based on minimal SSA features and machine learning 2022.

[8] Matthieu Jimenez, Mike Papadakis and Yves Le Traon, Vulnerability Prediction Models: A case study on the Linux Kernel 2021.

[9] Semasaba1, Wei Zheng1, Xiaoxue Wu2, Samuel Akwasi Agyemang1, Literature survey of deep learning-based vulnerability analysis on source code 2021.

[10]Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak and L. Karaçay, "Vulnerability Prediction From Source Code Using Machine Learning," in IEEE Access, vol. 8, pp. 150672-150684, 2020, doi: 10.1109/ACCESS.2020.3016774.

[11]A.Gkortzis, D. Mitropoulos, and D. Spinellis, “VulinOSS: a dataset of security vulnerabilities in open-source systems,” in Proceedings of the 15th International Conference on Mining Software Repositories,2019

[12]Delphine Immaculate, S., Farida Begam, M., and Floramary, M. (2019). Software bug prediction using supervised machine learning algorithms.