

WEBSITE RISK PREDICTION

*Design project report submitted in partial fulfillment of the requirements for
the award of the degree of*

Bachelor of Technology

in

Computer Science & Engineering

Submitted by

M Navaneetha

Krishnapriya A

Naveen B Jacob

Midhul M



**FEDERAL INSTITUTE OF SCIENCE AND
TECHNOLOGY (FISAT)
ANGAMALY-683577**

Affiliated to

**APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY
KERALA**

DECEMBER 2020

**FEDERAL INSTITUTE OF SCIENCE AND
TECHNOLOGY (FISAT)**

Mookkannoor(P.O), Angamaly-683577



FOCUS ON EXCELLENCE

CERTIFICATE

This is to certify that project report entitled “**WEBSITE RISK PREDICTION**” is a bonafide report of the Design Project(CS 341) presented during V^{th} semester by **M Navaneetha,Krishnapriya A,Naveen B Jacob,Midhul M**, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology (B.Tech) in Computer Science & Engineering during the academic year 2020-2021.

Project Guide

Dr. Prasad J C
Head of the Department

ABSTRACT

Web applications are the most widely used technique for providing access to online services. However, these applications poses the threat to manyvulnerable acts. Website security is a prime requirement for any trusted userwho accesses a website that demands their credentials and confidential in-formation. When dealing with everything from hosted websites, securityrisks are always inherent. Web security involves threats (hackers, websitemalware, and so on) that seeks to exploit the vulnerabilities in the web-site ranging from weak passwords to more technical flaws such as Cross-sitescripting.

Cross-Site Scripting (XSS) attacks remain the most widespread form of attack against web applications, which allows the hackers to inject the malicious script code for stealing the user's confidential information. Recent studies show that malicious code detection has become the most common threat. In web browsers, the malicious script codes are executed and used to transfer the sensitive data to the third party (or hackers) domain. Currently, most researchers try to assay the methods to prevent XSS on both the client and server-side. In this project, we present a machine learning technique to classify malicious web pages. The study is centered on some of the possible ways to detect the XSS script on the client-side that is based on the features extracted from the website URL to scan the web pages for checking the malicious scripts.

ACKNOWLEDGMENT

No words are enough to express our gratitude towards our dear **Pankaj Kumar** sir, The Assistant Professor and System Analyst, FISAT who had extended his valuable support and guidance to work towards research-oriented work and reach great success. We also thank all our dear teachers who had offered their helping hands for the success and true fulfillment of this project work by providing all ideas regarding project-related work. Last but not the least, we thank all our classmates and co-teams who worked on par with us and had helped us to explore this new broad area in an organized manner together.

**M Navaneetha
Krishnapriya A
Naveen B Jacob
Midhul M**

Contents

List of Figures	v
List of Tables	vi
1 INTRODUCTION	1
1.1 Overview	1
1.2 Scope of project	2
1.3 Objectives	2
2 Literature Survey	3
2.1 Background Study	3
2.1.1 Classification using Different Classifier Models	3
2.1.2 Block diagram for Classification	4
2.1.3 Methods	5
2.1.4 Algorithms for different classifier models	5
2.1.5 Comparison between different methods	8
3 DESIGN	9
3.1 Problem Statement	9
3.2 Framework Overview	9
3.3 Algorithm	10
3.4 Flowchart	12
3.5 System Requirements	13
3.5.1 Hardware Requirements	13
3.5.2 Software Requirements	13
3.6 Design Methodology	13
3.7 Dataflow Diagram	13
3.7.1 Level 0	13
3.7.2 Level 1	14
3.7.3 Level 2	14
3.8 Data Acquisition and Development	15
3.8.1 Dataset	15
3.8.2 Features	15
3.8.3 Development of the Dataset	16
4 IMPLEMENTATION AND TESTING	17
4.1 Implementation	17
4.2 Testing	18
4.2.1 Unit Testing	18
4.2.2 Integration Testing	18
4.2.3 System Testing	18
4.3 Logging	18
5 RESULTS & ANALYSIS	20
5.1 Comparison With Existing System	20

6	CONCLUSION	21
	Appendices	23
A	Sample Code	24
A.1	decision_tree.py	24
A.2	extract_rules.py	28
A.3	testing.py	30
A.4	prediction.py	32
B	Screenshots	34
C	Datasets	38

List of Figures

1.1	Overview of XSS attack	1
2.1	Block diagram for Classifier Models	4
3.1	Block-Diagram for the proposed system	11
3.2	Flowchart for the proposed system	12
3.3	Dataflow diagram:Level 0	14
3.4	Dataflow diagram:Level 1	14
3.5	Dataflow diagram:Level 2	15
3.6	Feature Extraction System Architecture	16
5.1	Comparison Table	20
B.1	Feature extraction from a given URL	34
B.2	Front End Prediction System	34
B.3	Benign URL Input	35
B.4	Predicted Output with the given Benign URL	35
B.5	Vulnerable URL Input	36
B.6	Predicted Output with the given vulnerable URL	36
B.7	Testing	37
6.1	Dataset with Benign URLs	38
6.2	Dataset with malicious URLs	38
6.3	Dataset Used	39

List of Tables

2.1	Algorithm Comparison Table	8
-----	--------------------------------------	---

Chapter 1

INTRODUCTION

1.1 Overview

The dynamic web pages have an important role in manipulating the resources and providing an interaction between the clients and servers. The features that are currently supported by the web browsers provide a greater interaction in web-based services such as e-commerce, Internet banking, social networking, blogs, forums, etc where the users most sensitive information is brought into the most vulnerable area where one's personal information can be easily acquired and exploited. According to Wasserman and Su [1], most of the web programming languages do not provide safe data transfer by default, and due to which malicious code can be easily injected into any web application, The Cross-Site Scripting (XSS) is one such attack where malicious script codes can be injected on to a vulnerable web page.

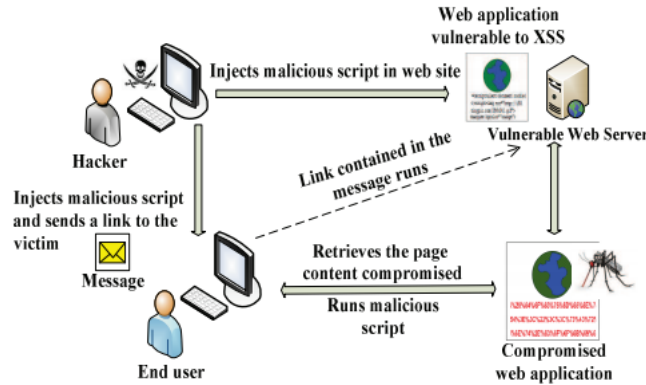


Figure 1.1: Overview of XSS attack

The project identifies a set of features that are present in URLs that anticipate the accurate prediction of an XSS vulnerability on a web page. For the classification of the web page as benign or malicious four distinct algorithms, namely naive Bayes, Decision Tree, Support Vector Machine, and K-Nearest Neighbors were used in the primary phase. A dataset comprised of 20,000 website URLs out of which 10,000 are XSS-attacked (based on studies from previously occurred attacks) was adopted to build the model. For further expansion of the project, a rule-based classification model was built using the existing decision tree model that makes the prediction based on the set of rules constructed from the decision tree algorithm

1.2 Scope of project

Website security is a relevant topic that needs to be addressed in the current scenario as there is a profusion of cyber attacks that is being reported. This requires stringent correction methods to secure a website from cyber attacks. The project intends to contribute in the fields of internet and website security activities on providing a better user-friendly experience and to target a wide audience towards trusted websites. The epicenter of this study is confined to the risk prediction problem of the scandalous Cross-site scripting. The system attacks on a single domain with respect to the features uprooted from the data set.

According to researchers, there is no single source to surmount XSS attacks or to nullify the flaws that existed in the source code of applications. This project focuses on providing a fundamental solution to the XSS attack. The proposed system examines a very finite range of features available from the data set which can be extended for the improvisation of the project.

1.3 Objectives

Website security is the most relevant aspect while using it because no one would prefer a hacked website. Using a hacked website leads to further hassles hence the widespread use of the hacked website should be annihilated at any cost. The basic fact for the success of Cross-site Scripting attacks is that most web users are not aware of the actual code/URLs of hyperlinks and the scripts running behind the web pages. So the web attackers are more successful in executing such attacks using URLs. The proposed system is a URL-based Cross-site Scripting risk prediction system that predicts the vulnerability of a website against Cross-site Scripting attacks. The primary objective of the system is to caution any website user about the possibility of a website being prone to XSS attack before accessing the site. As there is a vast source of websites available to a variety of website users, the possibility of trusted users victimizing such attacks cannot be neglected in the long run. Like any other website attacks, Cross-site scripting is a not easily detectable attack from the user's domain and cannot be blocked, unlike a computer virus that can be blocked by antivirus software. An efficient system that predicts the vulnerability of such attacks is vital in the present scenario where unethical hackers grow in bulk day by day. The proposed system is a research-based solution that intends for at mitigating the chance of confidential data loss from common web users who access the websites hosted by trusted servers.

Chapter 2

Literature Survey

Of late, it has been noticed by researchers that XSS vulnerabilities lists on the top of the greatest website vulnerabilities, Even though there are numerous approaches and techniques proposed for predicting the XSS susceptible web pages. However, machine learning techniques are an effective way to detect the web anomaly. The popular machine learning algorithms namely Naive Bayes, Support Vector Machine, Decision Tree, and K-Nearest Neighbour are the approaches used in the initial phase of the study to determine the vulnerability of XSS in a website.

2.1 Background Study

Studies were conducted about the existing systems and four distinct machine learning algorithms were selected in the primary phase based on the research and discussions.

2.1.1 Classification using Different Classifier Models

Naive Bayes: Naive Bayes is a supervised machine learning algorithm that is based on the Bayes theorem which is used in statistics and probability theory. The fundamental intention of Naive Bayes is that each observed sample in the training set can increase or decrease the probability that a hypothesis arises [2]. Bayes theorem evaluates the probability of a hypothesis (posterior probability) based on the prior probability and class likelihoods.

Support Vector Machine(SVM): Support Vector Machine is a technique for linear classification and regression. In SVM, the primary objective is to find the optimal hyperplane that maximizes the margin. After training, the discriminant is written as a sum of training vectors and are selected as support vectors. The instances which are not supported vectors and which lie inside the margin are omitted since they carry no information [4]. An SVM training algorithm carves a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.

Decision Tree: The Decision Tree algorithm is a supervised machine learning technique that can be used for both Classification and Regression type problems. In Decision Trees, for foretelling a class label for a record, the comparison begins from the root of the tree and the branches are formed till a purified subset is attained. When an unknown input is passed to the tree, it compares the node value with the given inputs attribute. Based on this comparison, it follows the branch corresponding to the attribute value and passes on to the successive node, and the

Predicted class is obtained as an output at the leaf node.

K-Nearest Neighbour(KNN): K-nearest neighbors (KNN) algorithm is a type of supervised ML algorithm that can be used for both classifications as well as regression predictive problems. The following two properties defines KNN:

- **Lazy learning algorithm**:KNN is a lazy learning algorithm because it does not have a specialized training phase and consumes all the data for training the model.
- **Non-parametric learning algorithm**:KNN is also a non-parametric learning algorithm because it does not infer anything about the underlying data.

The KNN algorithm assumes the similarity between the new case/data and available cases and puts the new case into the category that is almost identical to the available categories. KNN algorithm stores all the available data and classifies a fresh data point based on the similarity.

2.1.2 Block diagram for Classification

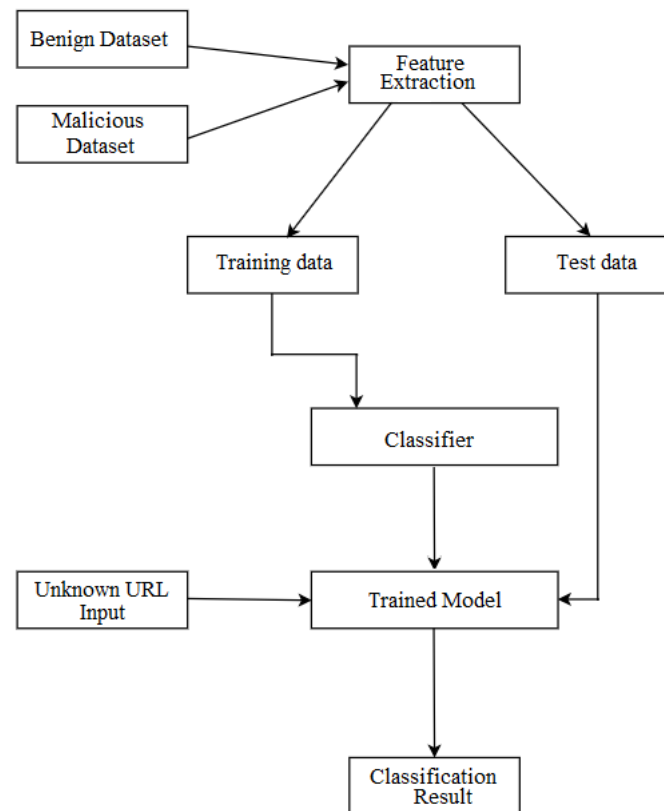


Figure 2.1: Block diagram for Classifier Models

2.1.3 Methods

Features Extraction

The essence of the classifier is the set of features extracted from the obtained data. The dataset used is a list of URLs consisting of both benign and XSS attacked websites. The features were identified and the feature values were computed for every extracted URL. The computed values were used for the preparation of the dataset. For this work, four distinct features were chosen from the URLs based on the research and studies.

1. ***Length of URL***:URL length corresponds to the number of characters present in a single website URL.The URL with considerably large length is considered XSS attacked.
2. ***Number of Keywords***:These are the words which usually appear in the URL as the variable names for assigning user input values.Some of them include iframe,alert,onclick,prompt etc.
3. ***Length of encoded characters***:This feature corresponds to obfuscated strings of characters overshadowed by alternative encodings-Hexadecimal,Decimal, Octal,Unicode,Base64,HTML reference characters.
4. ***Number of special characters***:It corresponds to the character combination which is used as the prefix with script input to make the script code executable. These special characters includes ">", ">", ">" etc.

Training and Testing

The prepared dataset is divided into training and testing data and the process of testing and training the classifier model varies from one another. Training the model involves fitting the feature values and labels from the training data and making the model capable of performing predictions using supervised learning. Testing of the trained model involves evaluating the model using test data based on the training it has done earlier. The accuracy of testing is the ratio of the number of correct predictions made by the model to the total number of predictions made.

Deployment of the trained model

The input retrieved from the user is the URL of the website under consideration. Relevant numerical features are extracted from the URL input to form a single feature vector. Each trained model is deployed to these feature values to make its corresponding prediction. The prediction retrieved is a binary value, the label 1 corresponds to XSS vulnerable websites and 0 corresponds to safe websites.

2.1.4 Algorithms for different classifier models

2.1.4.1 Algorithm:Naive bayes Algorithm

Input : A dataset consisting of list of benign well as XSS attacked URLs

Output : Predicted label(safe or XSS-vulnerable) of a fresh URL input

1. START
2. Extract numerical features and labels separately from the URL data and prepare a dataset.
3. Split the training and testing data.
4. Calculate the mean and variance of each feature for each class in the training data for training.
5. Compute the prior probability of each class based on their frequency in the training dataset.
6. Calculate the conditional probabilities with respect to each feature vector in each sample present in the test data.
7. Perform testing by computing the posterior probabilities of each class using Bayes theorem based on the measured parameters for each test sample.
8. Calculate the accuracy of the trained model and save the model to be used later.
9. For a fresh unlabelled URL input, extract the relevant features and predict whether a website URL is prone to XSS attack or not using the classifier model.
10. STOP

2.1.4.2 Algorithm:Decision Tree Model

Input : A dataset consisting of list of benign well as XSS attacked URLs

Output : Predicted label(safe or XSS-vulnerable) of a fresh URL input

1. START
2. Load the Data
3. Extract Features
4. Based on the features a model is created
5. Best feature is found in the training data-set using entropy and information gain
6. Place the node with the best feature as the root node
7. Divide the root nodes into subsets
8. Recursively make new trees until no further classification possible
9. Then apply test data to the created model
10. Testing is done till the leaf node is encountered
11. The leaf node consists of the purified subset
12. STOP

2.1.4.3 Algorithm:K-Nearest Neighbour Model

Input : A dataset consisting of list of benign well as XSS attacked URLs

Output : Predicted label(safe or XSS-vulnerable) of a fresh URL input

1. START
2. Load the Data
3. Extract Features
4. Input the URL
5. Extract the features of the URL
6. Get the weight
7. Find the closest URL's
8. Note the closest n URL, n=weight
9. Find the closest URL
10. Predict the URL based on the closest URL.
11. STOP

2.1.4.4 Algorithm:Support Vector Machine Model

Input : A dataset consisting of list of benign well as XSS attacked URLs

Output : Predicted label(safe or XSS-vulnerable) of a fresh URL input

1. START
2. Load the Data.
3. Extract Features.
4. Split the dataset into training and testing data.
5. Training Data is trained using SVM linear kernal.
6. Calculate the accuracy of the trained model and save the model.
7. For a fresh unlabelled URL input, extract the relevant features and predict whether a website URL is prone to XSS attack or not using the saved model.
8. STOP

2.1.5 Comparison between different methods

The table below shows the comparison of the algorithms used in the initial phase of the project based on the studies and inferences.

Sl no	Classifier Model	Accuracy(%)	Time taken to build the model(s)
1	Naive Bayes	98.1%	3.79
2	Decision Tree	99.625%	8.71
3	Support Vector Machine	96.3%	79.79
4	K-Nearest Neighbour	99.7%	20.3

Table 2.1: Algorithm Comparison Table

Chapter 3

DESIGN

In the second phase, an efficient URL-based XSS vulnerability prediction system that is a rule-based classifier algorithm is derived from the decision tree algorithm. JavaScript and URL are the frequently used components in a website for implementing and propagating XSS attacks. Hence both these elements can be used for identifying the features for the construction of the dataset that serves the machine learning model. However, the system proposed in this project is a prediction model that feeds on the URL-based XSS features.

3.1 Problem Statement

Web technology has heightened exponentially in daily volume and interactions involving web-based services and has become a vital part of our daily and personal lives. Simultaneously, web applications have unfortunately become the primary targets of cybercriminals to intrude into the users' online privacy. Cybercriminals exploit the poor code experiences of web developers, deficiencies within the code, inappropriate user input sanitization, and non-compliance with security standards by the software package developers[7] to gain personal benefit unethically.

The project intends to utilize efficient machine learning algorithms for the classification of benign and malicious web pages based on the features extracted from URLs to predict the vulnerability of websites against a high-risk security attack. Cross-Site Scripting (XSS) is one of the common high-risk cyber-attack on web application vulnerabilities which has placed web applications, users, and even the industrial field at high risk[5].SAP(Open Web Application Security Project) ranks Cross-site scripting as the most dangerous among the top 10 list of website security attacks[6]. This work focuses on predicting the vulnerability of a website hosted by a trusted server against a Cross-site Scripting attack using classifier machine learning algorithms.

3.2 Framework Overview

The classifier model uses a decision tree classification algorithm from which a rule-based classifier algorithm is derived. Here the rules are extracted from each decision node thus forming a set of rules for each leaf node. For the construction of the decision tree algorithm treelib module is used. Treelib module is a simple module that includes only two sub-modules namely tree and node.

In the Decision Tree algorithm, the tree structure is created which uses the sub-

modules contained in the treelib module for its implementation. It first creates a root node and then creates the branches according to the decision node till a purified subset is obtained. Each node represents a decision or a rule and each leaf node represents the respective class prediction. Initially, the whole Training data is used for the implementation of the decision tree which then finds the adequate rules that can be used to split the dataset into subsets. The Gini index of the given dataset is then computed, which is calculated as

$$GiniIndex = 1 - \sum_{i=1}^n p_i^2 \quad (3.1)$$

The dataset is divided based on the decision in the node, the obtained set of data with the calculated Gini index is used for calculating the information gain. The Information Gain is calculated as

$$Gain(S) = GiniIndex(S) - \sum \frac{|S_v|}{S} . GiniIndex(S_v) \quad (3.2)$$

The information gain helps in finding the node that reduces the uncertainty. The decision that holds the highest gain is then selected as the decision node.

From the constructed decision tree the rule-based classifier algorithm is derived. In the Rule-Based Classifier algorithm, the node rules were extracted from the created decision tree which contains the set of rules for each leaf node. Sub-module **node** is used for the implementation of this algorithm. Here Depth-First Algorithm is implemented to extract the set of rules from the decision tree nodes and hence the rules are created. The remaining test dataset is passed on to the rule-based classifier model which then predicts the predicted class of each test record and compares it with its corresponding label value and the ratio of the number of correct predictions to the number of false predictions computes the accuracy of the model. The model is pickled for further use and when an unknown input is passed onto this model it predicts the class to which it belongs and displays if the URL is safe to use or not.

3.3 Algorithm

Input : A dataset consisting of list of benign as well as XSS attacked URLs

Output : Predicted label(safe or XSS-vulnerable) of a fresh URL input

1. START
2. Extract features and labels from the dataset consisting of a list of URLs.
3. Create a CSV file that consists of feature names and label as columns and their corresponding values as each row
4. Split testing and training data
5. Construct a decision tree algorithm from scratch to fit the features, labels, and their respective values for the training data.



Figure 3.1: Block-Diagram for the proposed system

6. Extract classification rules by the process of tree traversal based on the leaf nodes present in the tree.
7. Test the rule-based classifier model using testing data.
8. Save the model to be used later.
9. For a fresh URL input, extract the feature values and use the model to predict whether or not the website is vulnerable to XSS.
10. STOP

3.4 Flowchart

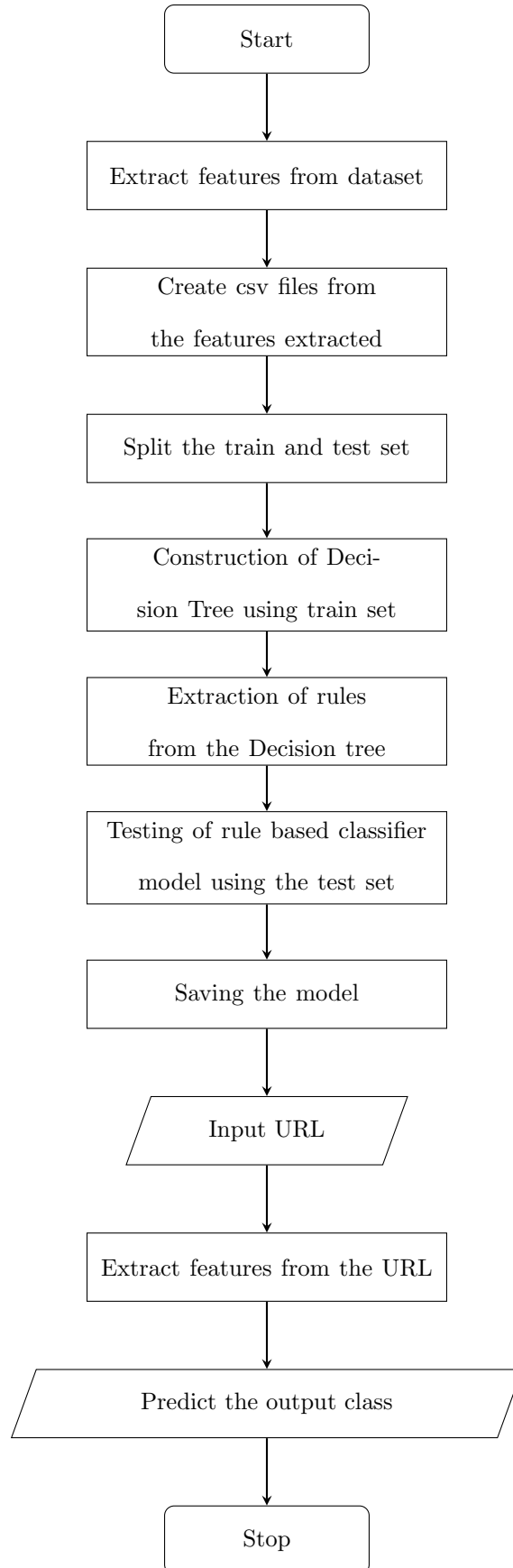


Figure 3.2: Flowchart for the proposed system

3.5 System Requirements

3.5.1 Hardware Requirements

- RAM: 2GB ,Recommended 4GB
- Storage: 10GB , Recommended 15GB

3.5.2 Software Requirements

- Ubuntu(18.04)
- Elementary OS(5.1.6)
- Manjaro(20.2)
- Debian 10

3.6 Design Methodology

- **Treelib**:Treelib module is used to implement a tree data structure. It consists of only two classes **Node** and **Tree**.Both the classes are used in the construction of the decision tree and rule-based classification algorithm
- **Pytest**:pytest is a framework that makes building simple and scalable tests easy. The pytest module is used to perform testing.
- **Pipenv**:It helps in creating and also managing a virtual environment for the created project. It also helps in adding and removing packages from the pip file. It also generates a pipfile.lock, which is used to produce deterministic builds.

3.7 Dataflow Diagram

Data is given as an input which is being classified according to the predefined classes in the training process.The output will be the class in which the input belongs to.

3.7.1 Level 0

Level 0 data flow diagram shows an abstract view of the system where input is a URL provided by the user. The obtained input is then fed to the trained classification system from where output can be retrieved using the class prediction method. The figure below shows the Level 0 dataflow diagram.



Figure 3.3: Dataflow diagram:Level 0

3.7.2 Level 1

Level 1 data flow diagram provides a more detailed understanding of the project, where the input is a URL from the user side and the Output predicts if the entered URL is malicious or not. The input from the user is fed to the feature extractor which extracts the feature according to the frequency of its occurrence in each URL. These features are then imported to the Rule-based classifier model which extracts the rules from the decision tree model hence predicts the output.

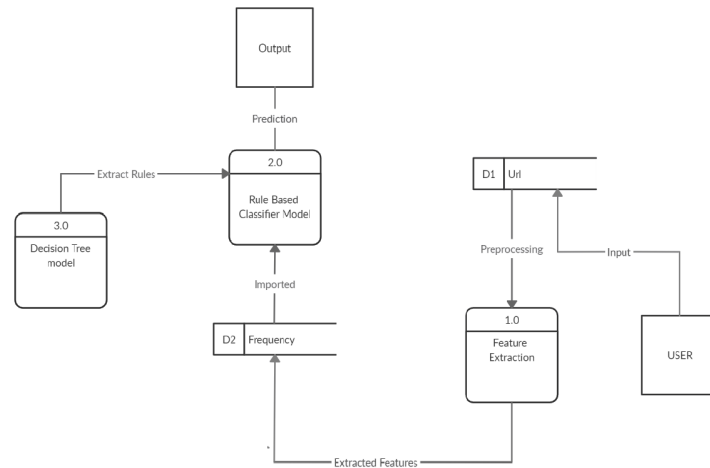


Figure 3.4: Dataflow diagram:Level 1

3.7.3 Level 2

Level 2 data flow diagram provides a holistic view of the project than the level 1 diagram. Here the input is a URL from the user and The Output is the predicted class. The user inputs the URL which undergoes preprocessing and then extracts features which are then used as input data to the rule-based classifier model for the prediction of its class. The training data is used as the input for building the Decision Tree Classifier model, which is then used to extract the node rule or the decision rules to form a set of rules for each leaf. The created rules are then imported to form the rule-based classifier model where the user input is fed for its

class prediction and then from the classifier model the Output is predicted that the given input is malicious or benign.

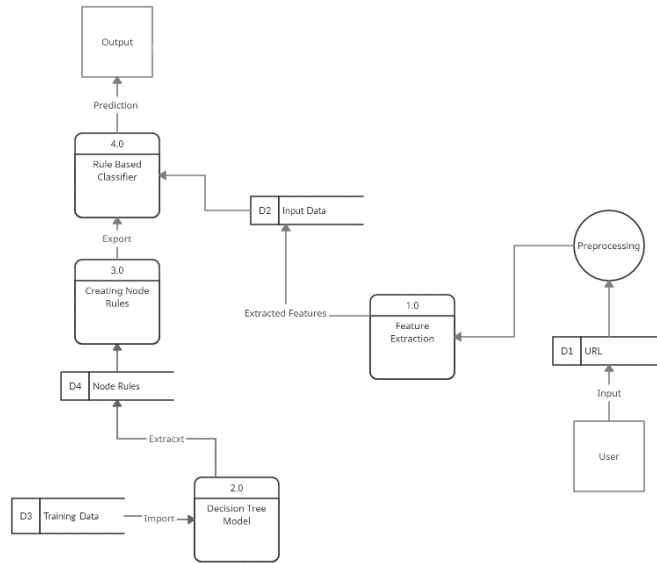


Figure 3.5: Dataflow diagram:Level 2

3.8 Data Acquisition and Development

The initial step includes the collection of URLs of benign web pages and web pages that are affected by the XSS attack separately. Then the extracted features from these website URLs are obtained and are labeled as benign or malicious. These features are extracted and the dataset is created using the collected features. This dataset is then used for training and testing the constructed machine learning algorithm.

3.8.1 Dataset

The dataset consists of two different sets of data with 20,000 data samples out of which 10,000 are XSS attacked(based on previously occurred attacks). The dataset is maintained as a part of an open project in GitHub[3]. From the collected URLs, the features are identified to prepare another main dataset as a CSV file. These feature samples are selected for training and testing.

3.8.2 Features

For the identification of the features, each URL was analyzed from the given dataset. Four distinct features are identified and are programmatically extracted from each URL for the preparation of the final dataset.



Figure 3.6: Feature Extraction System Architecture

- **Length of the URL:** It represents the length of the URL. The malicious URLs contain a JavaScript code so, their length will be longer in length than the benign URL.
- **Number of keywords:** The URL which is used for implementing Cross-site scripting attacks usually contain the direct JavaScript code for redirecting, cookie access, etc. This feature also includes the commonly identified encoded characters that can be used to differentiate malicious and benign URLs.
- **Number of special characters:** It corresponds to the occurrences of script tags and commonly found special characters. These script tags make the script code executable hence this feature is considered very useful
- **Number of HTTP requests:** It checks for the presence of the word HTTP in a URL and is a great feature in differentiating malicious and benign URLs.

3.8.3 Development of the Dataset

The dataset prepared from the extracted URLs consists of 4 attributes and 1 label, which consists of about 10,000 XSS instances and 10,000 non XSS instances. Initially, the dataset is fed to the program which imports the re module to find all the occurrences of the identified features in each URL and returns its count then creates a list for each record in the text file. The list also appends the value 1 as the label if the selected URL is from the malicious dataset and appends the value 0 as the label if the selected URL is from the benign dataset into two different datasets. Then the CSV files are merged and the random module is imported to shuffle the merged CSV file which is used as the dataset.

Chapter 4

IMPLEMENTATION AND TESTING

4.1 Implementation

Installation

The Decision Tree is build Using the module **Treelib**,with two primary classes: **Node** and **Tree**. The tree is a self-contained structure with some nodes and connected by branches. To build the decision tree classifier the treelib module was installed. The treelib module helps in supporting the common tree operations like insertion, node creation, tree traversal, etc.. which was used to build the tree and also to extract the node values . The treelib module was installed using the command as mentioned in Listing 4.1, which shows the command we used to install the module in our respective systems.

Listing 4.1:Installing Treelib
pip install treelib

Implementation

- ***create_node()***:This function is used in our 'decision_tree.py' file. Here this in-built function helps us to create a new node while building the tree. The module has three parameters namely Tag ID and Parent. The tag value is the value displayed on the tree, ID or identifier uniquely identifies the node and the id must be unique. The parent links the created node to the respective parent ID.
- ***subtree()***:The subtree() function is used in the 'extract_rules.py' file.The subtree function is used to identify the left node and the right node of the tree in the DFD function where the rules are built. It takes a single parameter that is the node identifier that identifies the node.
- ***path_to_leaves()***:This function gathers all the identifiers that track leads to a leaf. Every leaf has its path which is identified by this function. The function returns the list of identifiers of the path that leads to the leaf nodes. It takes no argument and returns a list.
- ***all_nodes()***: The all_nodes() function is used to return all the nodes in form of a list. This in-built function is used in 2 of our python files 'extract.py' and 'testing.py'.The all_nodes()function is used while constructing the rule-based classifier and also for testing the model.

4.2 Testing

4.2.1 Unit Testing

A unit test is the micro-level of testing, that checks if every single component operates in the right way. A unit test helps you to isolate what is a broken function in your application and fix the bugs found in them. They form atomic segments of the code, which provides quick results, and only if the unit tests are cleared the code needs to be merged. Unit tests are used to predict the bugs early in the program under development. Unit testing helps in verifying internal design internal logic and internal parts of a program, it also helps in error handling. In our project unit tests have been done on all the user-defined functions. Unit testing was done on all functions with sample data which were extracted from the actual dataset to perform testing on the created functions. The pytest module was imported for this purpose and a function whose name was prefixed with the test was created for each function in the program and the unit testing was concluded successfully.

4.2.2 Integration Testing

Integration testing is done only after the unit testing is passed for every function. Here all individual models are integrated and then tested to know how they work when combined all together. It ensures the successful compilation of the application. In our program for each python file after performing the unit testing integration testing was performed which integrates every individual function and then compiles it as a whole. The pytest module was imported for Integrated testing and a function whose name was prefixed with the test was created for performing the testing and hence the integrated testing was concluded successfully.

4.2.3 System Testing

System testing is testing of a complete integrated software system. System testing takes, all of the integrated components that have passed integration testing as its input. The system testing includes the testing of a fully integrated application to check how the components interact with each other when working as a system and verification of every input to check for the desired output.

4.3 Logging

The logger modules report the progress and problems in the code. It also helps in tracking exceptions or information. The module provides a lot of functionality and flexibility.

The program uses 2 levels of logging:

- **Info():**The level info() reports all the events are working as expected.
 - A logger info is provided at the beginning of every function and class to understand its functioning.

- **Error():**The level error() was used to report that the function is not working as expected.
 - Extraction of rules: While extracting the rules bugs were encountered while changing the symbol.
 - Tree Traversal: For tree traversal, a bug was encountered during the path to leaf extraction

Chapter 5

RESULTS & ANALYSIS

The experiments conducted here uses five different machine learning algorithms in building a predicting system for XSS vulnerability in websites.

In the initial phase of the project, four individual algorithms namely Naive Bayes, Decision Tree, SVM, and KNN were assigned with the problem statement. Four of the algorithms performed well with the provided set of feature values and made correct predictions with considerably fair accuracy. However, among the four, KNN outperformed among other algorithms in terms of accuracy with 99.7%. But concerning the amount of build time, the Naive Bayes classifier has the least value. In the next phase of the project, a more complex rule-based classifier was built from the created decision tree algorithm which holds an accuracy of 98.4% about the data provided and came out considerably efficient in performing correct predictions.

5.1 Comparison With Existing System

Risk Prediction Systems	Decision Tree-Rule based classifier	DOM based XSS checker	PHP-based Sensor
About	URL-based XSS risk prediction system	Cloud-based framework for restraining DOM-based XSS vulnerabilities	A server-side XSS vulnerability identifier
Method opted for making predictions	Generation of classification rules from a programmatically built decision tree	Improve and optimize the context-sensitive sanitization process of HTML5 attack vector	Observe the outgoing HTTP request and scripts present in them
Build complexity	Depends on the extracted features and its values	Depends on the decisive code injected into the nested context of suspected variables	Depends on the scripts existing within the HTTP response
Defense mechanism	Machine learning supported trained model which is user-friendly and less complicated	Traditional mechanism, Complicated	Extremely complicated and conventional method

Figure 5.1: Comparison Table

Chapter 6

CONCLUSION

The project used five distinct machine learning algorithms implemented in two phases as a part of research and studies to predict the vulnerability of a web application against the XSS-based attack. The model performs a prediction of a website security threat, XSS attack, which can reflect in the form of a warning to users who can cancel the subsequent treatment of the pages. The experiments were conducted effectively on the test dataset, and the comparison was performed on existing methods. Based on the results, it can be concluded that the proposed scheme is efficient in utilizing the data provided to make accurate and desired predictions to a great extent.

Pros and Cons

Results of the experiment show that the system has a relatively fair accuracy of **98.4%** for the given set of features. The current system uses only a very small number of features but is capable of making correct predictions with limited relevant features provided. The model that has been trained and tested can be saved as a model(.mdl) file to use anytime later that ensures **reusability** of the system. The system is **highly scalable** as well with the proper exploration of relevant additional features. The design of the project is **user-friendly** as it provides a better user interface to perform prediction. The user can experience the quality of the system from the front end without caring much about its functionality.

Malicious cross-site Scripting code can either be injected into the URL or the HTML source code of the website. The proposed system covers only URL-based XSS attack vulnerabilities. The system ignores the possibilities of JavaScript-based XSS attack and so doesn't concentrate on JavaScript-based features. Also, the system is highly dependent on the features provided and hence is very sensitive to the form of input data. Any form of glitches present in the data fed leads to the failure of the system.

Future Scope of the Project

The scope of the project can be extended further by identifying more features and appending them to the currently created dataset. The research can be extended towards exploring the JavaScript-based features using ethical web scraping to improve the capabilities of the system in predicting JavaScript-based XSS vulnerability as well. The experiment can be repeated by using other machine learning algorithms as well. Since archives with phishing and spam data are available online, a similar study can be done on phishing or spam classification and can be used for extending the capability of the current dataset in representing multiple attack domains.

Bibliography

- [1] Wasserman, G. e Su, Z. “Static Detection of Cross-Site Scripting Vulnerabilities”. In: 30th International Conference on Software Engineering, 2008.
- [2] Ethem Alpaydm. 2010. *Introduction to Machine Learning. Second Edition*. The MIT Press.
- [3] Dataset for research and implementation
<https://github.com/OneRepublicMarchingOn/FI-XSS-SVM>
- [4] Tom M. Mitchell. 1997. *Machine Learning*. McGraw-Hill
- [5] O. Andreeva et al., “Industrial Control Systems Vulnerabilities Statistics”, Kaspersky Lab, Report, 2016. [Online] Available: https://media.kasperskycontenthub.com/wpcontent/uploads/sites/43/2016/07/07190426/KL_REPORT_ICS_Statistic_vulnerabilities.pdf
- [6] OWASP, Foundation. “OWASP Top 10 – 2010. The Ten Most Critical Web Application Security Risks”, release 2010.
<http://owasptop10.googlecode.com/files/OWASP\%20Top\%2010\%20-\%202010.pdf>, Outubro, 2010.
- [7] B.K. Ayeni, J. B. Sahalu, and K. R. Adeyanju, “Detecting Cross-Site Scripting in Web Applications Using Fuzzy Inference System,” J. Comput. Networks Commun., vol. 2018, pp. 1–10, Aug. 2018.
- [8] Rui Wang, Xiaoqi Jia, Qinlei Li and Shengzhi Zhang: Machine Learning based Cross-site Scripting Detection in Online Social Network
- [9] Takeshi Matsudat, Daiki Koizumi and Michio Sonoda: Cross Site Scripting Attacks Detection Algorithm Based on the Appearance Position of Characters
- [10] Ms. Jevitha. K. P and Vishnu. B. A Prediction of Cross-Site Scripting Attack Using Machine Learning Algorithms
- [11] Likarish, P., Jung, E., Jo, I. : Obfuscated malicious javascript detection using classification techniques. 2009 4th International Conference on Malicious and Unwanted Software (MALWARE)
- [12] Sholom M. Weiss and Nitin Indurkha, Rule-based Machine Learning Methods for Functional Prediction

Appendices

Appendix A

Sample Code

A.1 decision_tree.py

```
import treelib
from data_split import *

attributes = ['url_length', 'Keywords', 'Url_count', 'spcl_char', 'Label']

class Data:

    # ..... Returns the Unique value in the dataset .....
    def get_unique_val(self, row, col):
        return set([r[col] for r in row])

    # ..... Counts no of type of data in dataset .....
    def count_Class(self, row):

        count = {}
        check = []
        key = []
        for r in row:

            attribute = r[-1]
            # if attribute ==0 or attribute ==1:
            if attribute not in count:
                count[attribute] = 0

            count[attribute] += 1
        # ..... added .....
        if len(count) == 1:
            return count
        else:
            if len(count) > 1:
                check = list(count.values())
                key = list(count.keys())
                large = check[0]
                pos = 0
                for i in range(len(check)):
                    if large < check[i]:
                        large = check[i]
```



```

        pos = i
        new = {str(key[pos]): large}
    return new

# ..... Class Partitions the Dataset .....
def partition(self, rows, qtn):

    t_row, f_row = [], []
    for r in rows:

        if qtn.comp(r):
            t_row.append(r)
        else:
            f_row.append(r)

    return t_row, f_row

# ..... Class defines the rules in the node .....

class Node_Rules:
    # ..... for numeric data .....
    def __init__(self, col, val):
        self.col = col
        self.val = val

    def comp(self, row):
        values = row[self.col]
        return values >= self.val

    def __repr__(self):
        contn = '>='
        return "Is %s %s %s?" % (attributes[self.col],
                                   contn, str(self.val))

class Calculate:

    # ..... Calculates Gini Impurity .....
    def gini_idx(self, rows):
        d = Data()
        count = d.count_Class(rows)
        impurity = 1
        for target in count:
            prob = count[target] / float(len(rows))
            impurity -= prob ** 2

        return impurity

    # ..... Calculates Information-gain .....
    def information_gain(self, left_set, right_set, uncertainty):

```

```

        total_length = len(left_set) + len(right_set)
        prob = float(len(left_set)) / total_length
        gain = uncertainty - prob * self.gini_idx(left_set) -
(1 - prob) * self.gini_idx(right_set)

    return gain

    # ..... need to push .....

def find_best_split(self, rows):

    d = Data()
    best_gain = 0
    best_question = None # keep track of the feature
    current_uncertainty = self.gini_idx(rows)
    n_features = len(rows[0]) - 1 # number of columns

    for col in range(n_features): # for each feature

        # unique values in the column
        values = set([row[col] for row in rows])

        for val in values: # for each value

            question = Node_Rules(col, val)

            # try splitting the dataset
            true_rows, false_rows = d.partition(rows, question)

            if len(true_rows) == 0 or len(false_rows) == 0:
                continue

            # Calculate the information gain from this split
            gain = self.information_gain(
                true_rows, false_rows, current_uncertainty)

            if gain >= best_gain:
                best_gain, best_question = gain, question

    return best_gain, best_question

class Leaf:

    def __init__(self, rows):
        d = Data()
        self.predictions = d.count_Class(rows)

    # ..... Asks a question .....

```

```

class Decision_Node:
    # This holds a reference to the question

    def __init__(self, question, true_branch, false_branch):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch


class Counter:
    def __init__(self):
        self.count = 0

    def get_count(self):
        self.count = self.count + 1
        return str(self.count + 1)


class DecisionTree:

    def __init__(self):
        self.arr = []
        self.counter = Counter()

    def build_tree(self, dataset, tree, par):

        cal = Calculate()
        d = Data()
        gain, question = cal.find_best_split(dataset)

        # ..... Checking for leaf node.....
        if gain == 0:
            if question in self.arr:
                ID = question + self.counter.get_count()
                tree.create_node(tree.predictions, ID, parent=par)
                par = ID

            else:
                tree.create_node(str(Leaf(dataset).predictions),
self.counter.get_count() + str(question), parent=par)
                par = question
            return tree

        self.arr.append(question)
        # print(self.arr)
        if tree.root == None:

            par = str(question)

            tree.create_node(str(question), par)
            right, left = d.partition(dataset, question)

```

```

        right_branch = self.build_tree(right, tree, par)
        left_branch = self.build_tree(left, tree, par)
    else:

        if question in self.arr:
            ID = self.counter.get_count() + str(question)
            tree.create_node(str(question), ID, parent=par)
            par = ID

        else:
            ID = self.counter.get_count() + str(question)
            tree.create_node(str(question), question, parent=par)
            par = question

        right, left = d.partition(dataset, question)
        right_branch = self.build_tree(right, tree, par)
        left_branch = self.build_tree(left, tree, par)

    # return tree
    return Decision_Node(question, right, left)

```

```

t = DecisionTree()
tree = treelib.Tree()
my_tree = t.build_tree(training_data, tree, '')

```

A.2 extract_rules.py

```

from decision_tree import tree
import pickle as c

class ExtractRules:

    def __init__(self):
        self.rules = []
        self.left = []
        self.l_id = []
        self.r_id = []
        self.right = []

    def dfs(self, visited, tree, node):
        if node not in visited:

            if len(tree.children(node)) != 0:
                self.left.append(node)
                self.right.append(node)
                visited.append(node)
                child = tree.children(node)
                l = child[0]

```

```

        r = child[1]
        # ..... left side traversal .....
        l_sub = tree.subtree(l.identifier)
        self.left.append(l.identifier)
        self.l_id.append(l.identifier)
        self.dfs(visited, l_sub, l.identifier)

        # ..... right side traversal .....
        r_sub = tree.subtree(r.identifier)
        # tag_val=r.tag
        self.r_id.append(r.identifier)
        tag_val = r.tag.replace('>=', '<')
        self.right.append(r.identifier)
        self.dfs(visited, r_sub, r.identifier)
    else:
        if node in self.r_id:
            val = self.right

def create_rules(self, tree, root):
    node = tree.all_nodes()
    paths = tree.paths_to_leaves()
    for path in paths:
        rule = []
        length = len(path) + 1
        p = 0
        for p in range(len(path)):

            if path[p] == root:
                continue
            else:
                par = tree.parent(path[p])
                child = tree.children(par.identifier)

                if path[p] == child[0].identifier and path[p] in
self.left:
                    for n in node:
                        if par.identifier == n.identifier:
                            rule.append(n.tag)

                else:
                    if path[p] == child[1].identifier and path[p]
in self.right:
                        for n in node:
                            if par.identifier == n.identifier:
                                rule.append(n.tag.replace('>=', '<'))

            if p == len(path) - 1:

                par = tree.parent(path[p])
                child = tree.children(par.identifier)

                if path[p] == child[0].identifier and path[p]

```

```

        in self.left:
            for n in node:
                if child[0].identifier == n.identifier:
                    rule.append(n.tag)

            else:

                if path[p] == child[1].identifier and
                path[p] in self.right:
                    for n in node:
                        if child[1].identifier == n.identifier:
                            rule.append(n.tag.replace('>=', '<'))
                            self.rules.append(rule)

    return self.rules

def save(self, clf, name):
    with open(name, 'wb') as fp:
        c.dump(clf, fp)
    print("Rules Saved")

visited = [] # Set to keep track of visited nodes.
r = ExtractRules()
r.dfs(visited, tree, tree.root)
rules = r.create_rules(tree, tree.root)
r.save(rules, "rules.mdl")

```

A.3 testing.py

```

from data_split import testing_data
import pickle as file
# from extract_rules import rules

class Testing:

    def Test(self, rules, data):
        attributes = ['url_length', 'Keywords', 'Url_count',
                     'spcl_char', 'Label']
        qtn = ''
        split = []
        num = ''
        # node = tree.all_nodes()
        for rule in rules:
            c = 0
            for r in range(len(rule) - 1):
                qtn = rule[r]
                split = qtn.split()

```

```

question = split[1]
num = split[-1]
n = num.translate({ord('?'): None})
val = int(n)
for attr in attributes:
    if attr == question:
        pos = attributes.index(attr)
        break

cndtn = split[2]
# print(rule)
# print(data[pos], cndtn, val)
if cndtn == '>=':
    if int(data[pos]) >= val:
        c += 1
        continue
    else:
        break
else:
    if cndtn == '<':
        if int(data[pos]) < val:
            c += 1
            continue
        else:
            break

if c == len(rule) - 1:
    return rule

def accuracy(self, rules):
    c = 0
    for i in testing_data:
        # print(i)
        lab = self.Test(rules, i)
        label = eval(lab[-1])
        d = dict(label)
        l = str(i[-1])
        if l in d:
            c += 1
    return c

with open("rules.mdl", 'rb') as fp:
    rules = file.load(fp)
test = Testing()
c = test.accuracy(rules)
print('len_of_test_data:', len(testing_data))
print('acc=', c / len(testing_data))

```

A.4 prediction.py

```
import re
import _pickle as c
from testing import Testing
#from feature_extraction import Features

with open("rules.mdl", 'rb') as fp:
    rules = c.load(fp)

class Features():
    #..... Returning URL Length .....
    def url_length(self, url):
        return len(url)

    #..... Returning URL keyword count .....
    def Keywords(self, url):
        words = re.findall("(alert)|(script)|(%3c)|(%3e)|(%20)|(onclick)|
        .....(onerror)|(onload)|(eval)|(src)|(prompt)|(iframe)|(style)",
        url, re.IGNORECASE)
        return len(words)

    def url_count(self, url):
        if re.search('(http://)|(https://)', url, re.IGNORECASE) :
            return 1
        else:
            return 0

    #..... Returning count of spcl char .....
    def Spcl_char(self, url):
        spcl = re.findall("<(<)|(>)|(/)|(=)", url)
        return len(spcl)

    #..... Create feature set .....
    def Create_features(self, url):
        features = [self.url_length(url), self.Keywords(url),
        self.url_count(url), self.Spcl_char(url)]
        return features

url = input('Enter the website URL: ')
f = Features()
features = f.Create_features(url)
#print(features)
test = Testing()
r = test.Test(rules, features)
label = eval(r[-1])
d = dict(label)
#print(d.keys())
```



```

for key in d.keys() :

    if(key == '0'):
        print("\nYou are safe! This website is not vulnerable to
XXXXXXXXXXSS attack\n")
    else:
        print("\nAlert! This website is vulnerable to XSS attack\n")

```

Appendix B

Screenshots

```
navi@navi-HP:~/Documents/Main_Project$ /usr/bin/python3 /home/navi/Documents/Main_Project/sample.py
Enter the URL::https://www.researchgate.net/publication/288492297_Prediction_of_Cross-Site_Scripting_Attack_Using_Machine_Learning_Algorithms
[126, 1, 1, 4]
navi@navi-HP:~/Documents/Main_Project$
```

Figure B.1: Feature extraction from a given URL

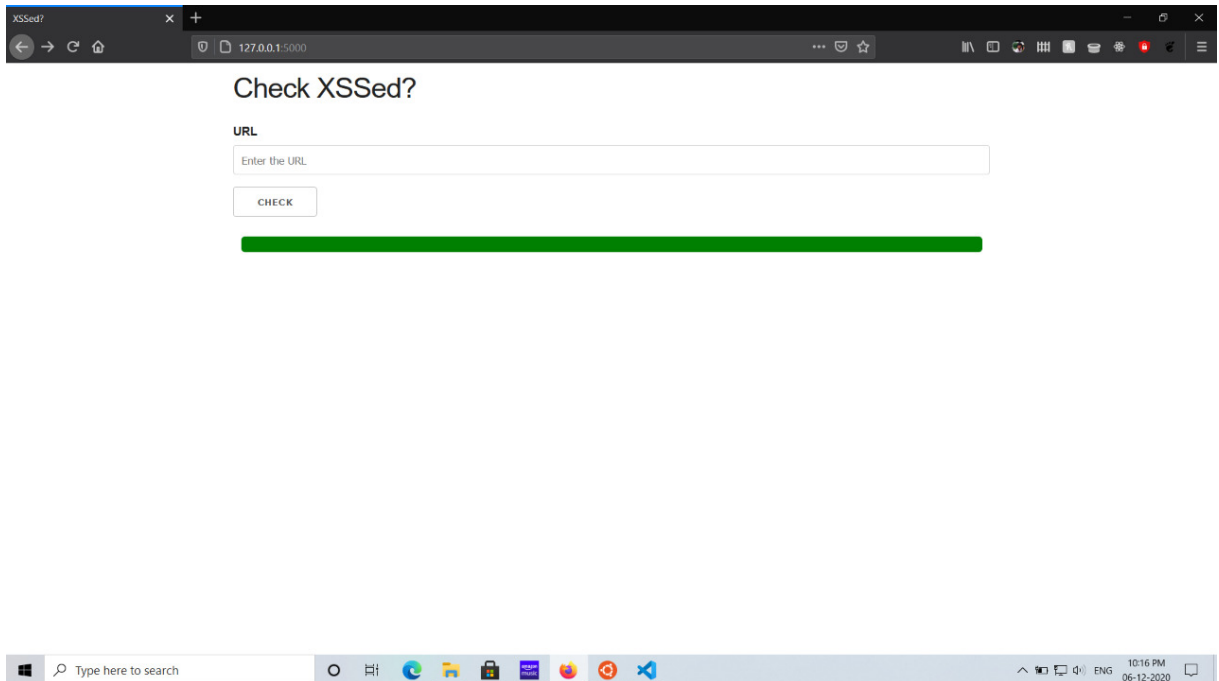


Figure B.2: Front End Prediction System

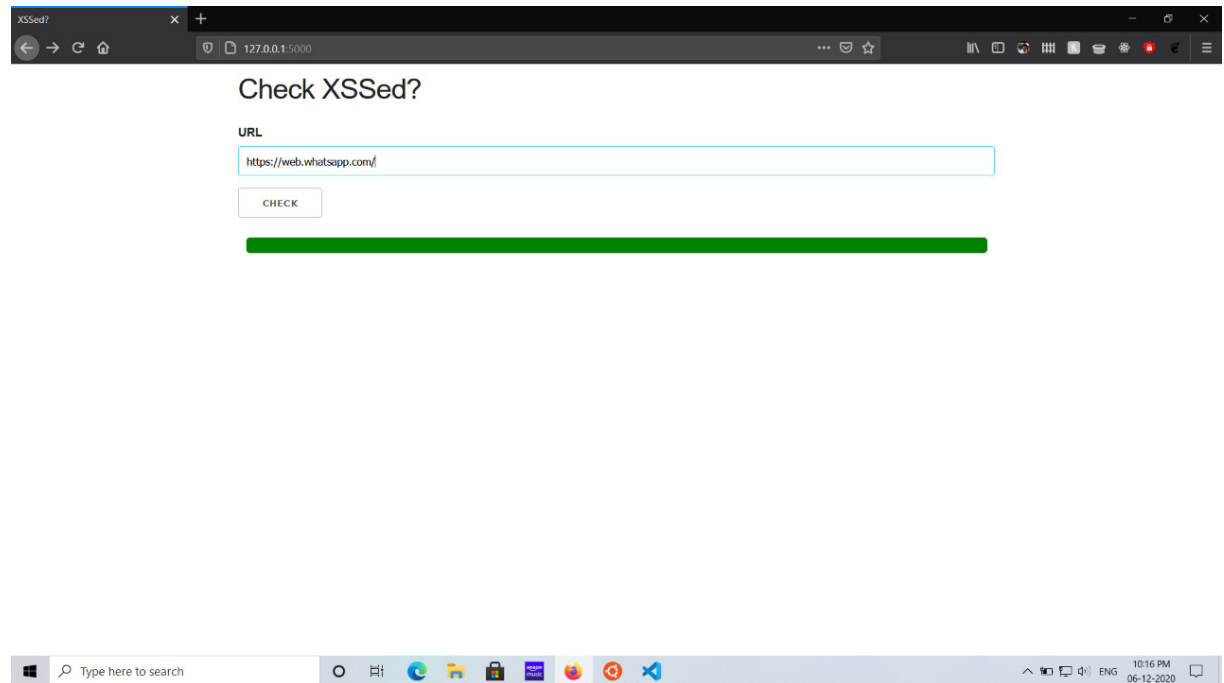


Figure B.3: Benign URL Input

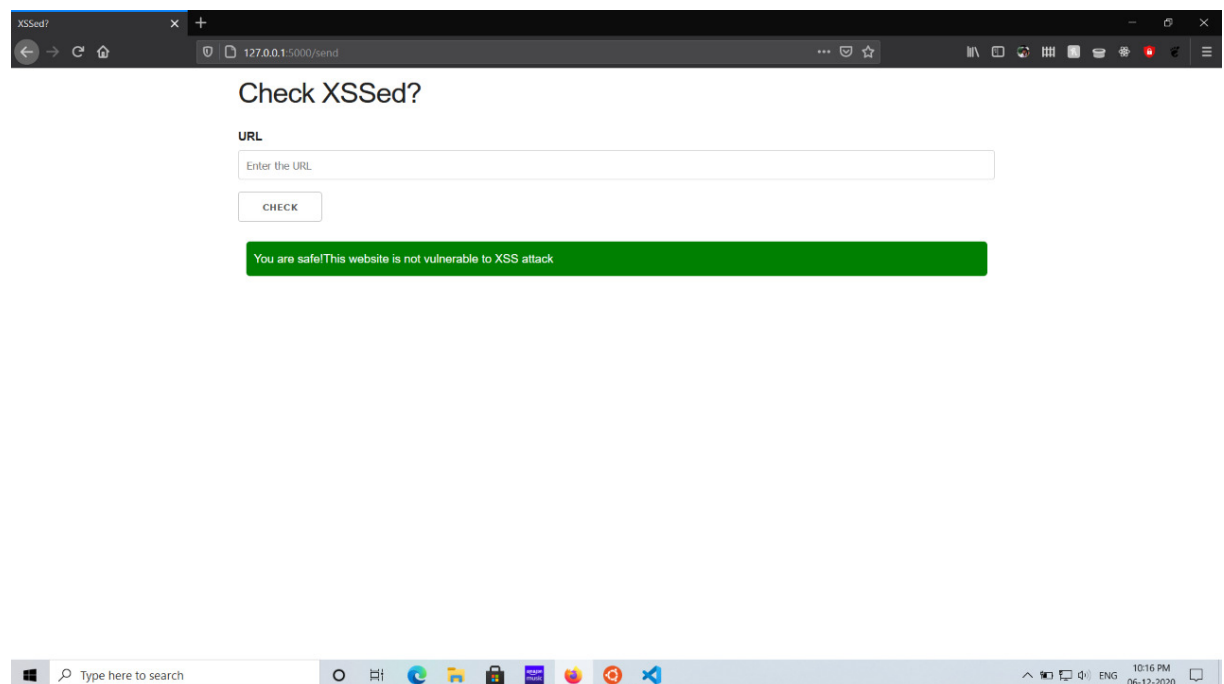


Figure B.4: Predicted Output with the given Benign URL

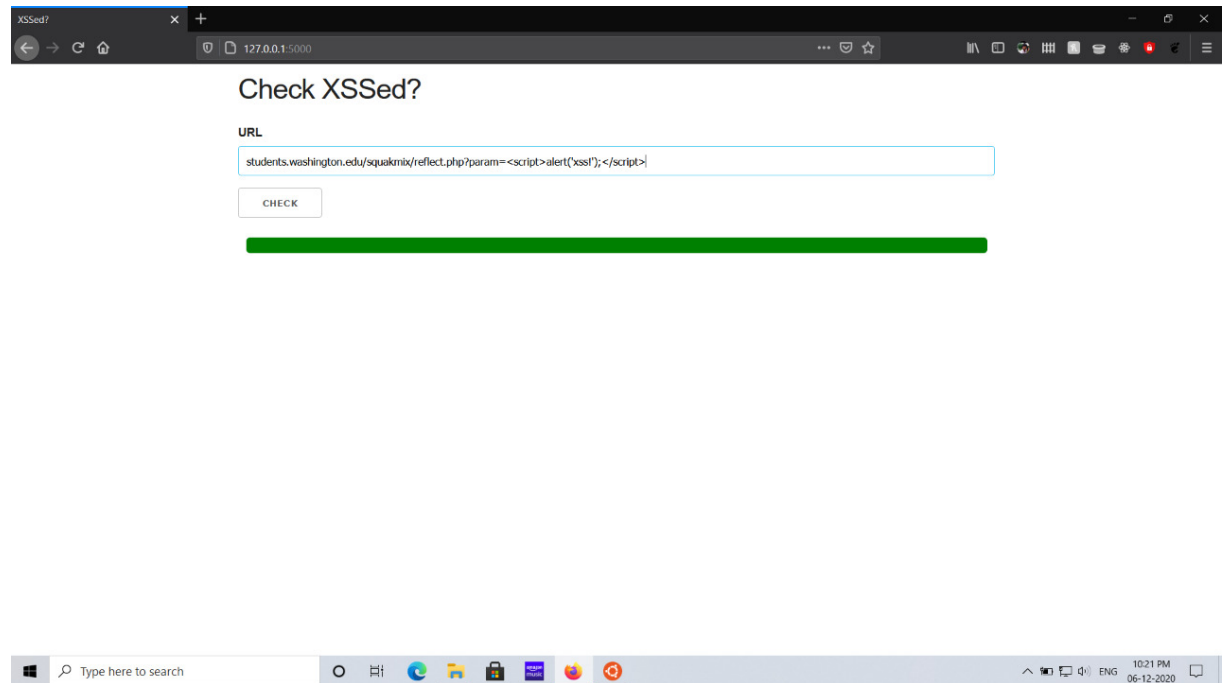


Figure B.5: Vulnerable URL Input

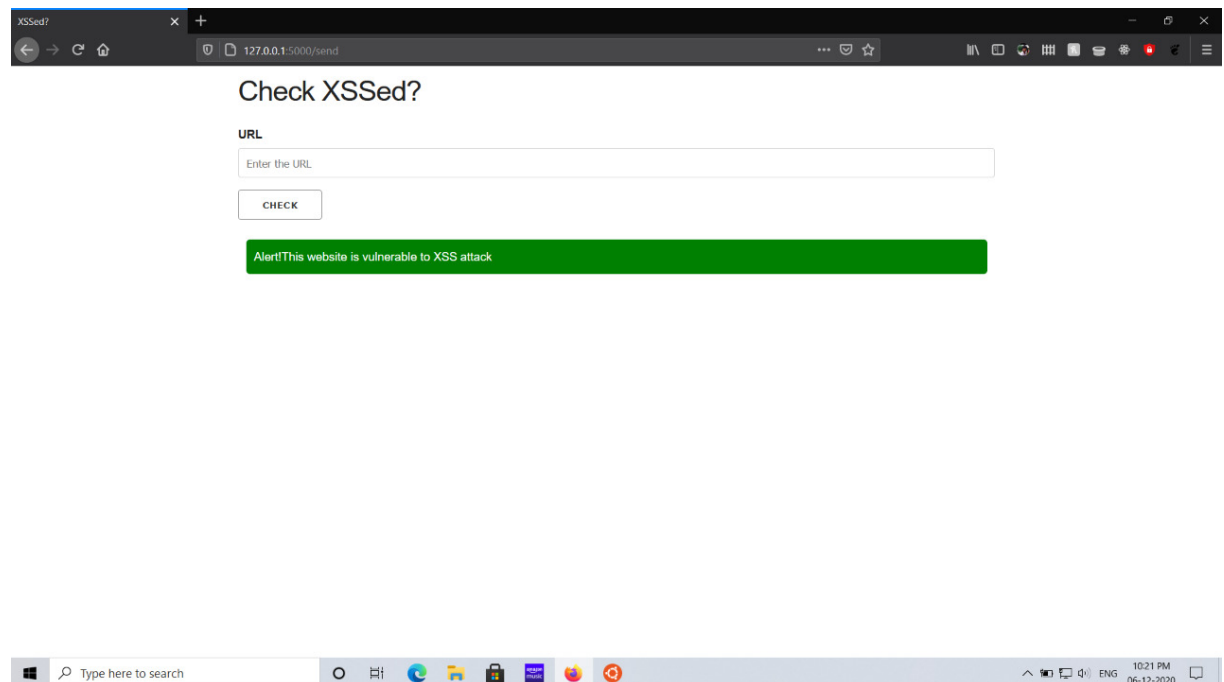


Figure B.6: Predicted Output with the given vulnerable URL

```
navi@navi-HP:/media/navi/New Volume2/project$ python -m pytest
===== test session starts =====
platform linux2 -- Python 2.7.17, pytest-4.6.11, py-1.9.0, pluggy-0.13.1
rootdir: /media/navi/New Volume2/project
collected 8 items

test_decision_tree.py ..... [100%]

===== 8 passed in 7.94 seconds =====
```

Figure B.7: Testing

Appendix C

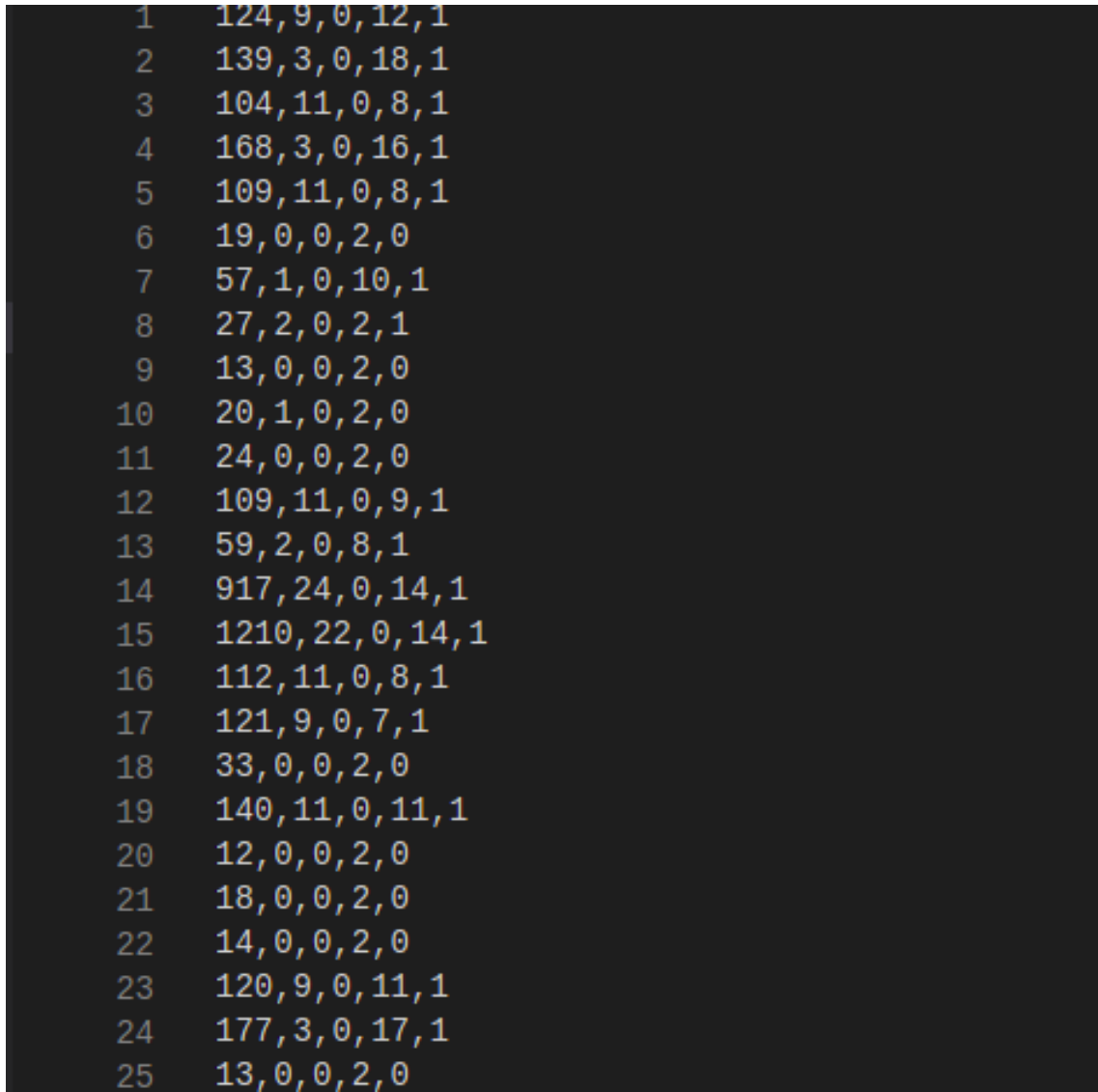
Datasets

```
non_xss.txt
1 /103886/
2 /rcanimal/
3 /458010b88d9ce/
4 /cclogovs/
5 /using-localization/
6 /121006_dakotacwpressconf/
7 /50393994/
8 /169393/
9 /166636/
10 /labview_v2/
11 /javascript/nets.png
12 /p25-03/
13 /javascript/minute.rb
14 /javascript/weblogs.rss
```

Figure 6.1: Dataset with Benign URLs

```
1 /0_1/?%22onmouseover='prompt(42873)'bad=%22%3E
2 /0_1/api.php?op=map&motype=1&city=test%3Cscript%3Ealert%28/42873/%29%3C/script%3E
3 /0_1/api.php?op=map&motype=1&defaultcity=%E5%8C%97%E4%BA%AC&api_key=%22%3E%3C/script%3E%3Cscript%3Ealert%28/42873/%29;%3C/script%3E
4 /0_1/api.php?op=map&motype=1&defaultcity=%E5%8C%97%E4%BA%AC&field=%29%3C/script%3E%3Cscript%3Ealert%2842873%29%3C/script%3E//
5 /0_1/api.php?op=map&motype=1&defaultcity=%E5%8C%97%E4%BA%AC&field=%29%3C/script%3E%3Cscript%3Ealert%2842873%29%3C/script%3E//
6 /0_1/api.php?op=video_api&pc_hash=1&uid=1&snid=%3C/script%3E%3Cscript%3Ealert(/42873/)%3C/script%3E//&do_complete=1%20
7 /0_1/api.php?op=video_api&uid=1&snid=1&pc_hash=%3C/script%3E%3Cscript%3Ealert(/360/)%3C/script%3E//&do_complete=1
8 /0_1/?callback=%3Cscript%3Eprompt(42873)%3C/script%3E
9 /0_1/connect.php?receive=yes&mod=login&op=callback&referer=webscan%5Cu0027.replace(/.%2b/,/javascript:alert(42873)/.source);//
10 /0_1/connect.php?receive=yes&mod=login&op=callback&referer=webscan%bf%5Cu0027.replace(/.%2b/,/javascript:alert(42873)/.source);//
11 /0_1/do/count.php?fid=1'%3E%22)%3C/script%3E%3Cscript%3Ealert(String.fromCharCode(120,%20115,%20115))%3C/script%3E
12 /0_1/do/kindeitor.php?id=%bf%22;alert(%22);//&style=&etype=
13 /0_1/e/data/ecmseditor/infoeditor/epage/TranFile.php?InstanceName=3232%22%3E%3Cscript%3Ealert(/D/)%3C/script%3E%3C%22
14 /0_1/e/data/ecmseditor/infoeditor/epage/TranFlash.php?InstanceName=3232%22%3E%3Cscript%3Ealert(/D/)%3C/script%3E%3C%22
15 /0_1/e/data/ecmseditor/infoeditor/epage/TranImg.php?InstanceName=3232%22%3E%3Cscript%3Ealert(/D/)%3C/script%3E%3C%22
```

Figure 6.2: Dataset with malicious URLs



1	124,9,0,12,1
2	139,3,0,18,1
3	104,11,0,8,1
4	168,3,0,16,1
5	109,11,0,8,1
6	19,0,0,2,0
7	57,1,0,10,1
8	27,2,0,2,1
9	13,0,0,2,0
10	20,1,0,2,0
11	24,0,0,2,0
12	109,11,0,9,1
13	59,2,0,8,1
14	917,24,0,14,1
15	1210,22,0,14,1
16	112,11,0,8,1
17	121,9,0,7,1
18	33,0,0,2,0
19	140,11,0,11,1
20	12,0,0,2,0
21	18,0,0,2,0
22	14,0,0,2,0
23	120,9,0,11,1
24	177,3,0,17,1
25	13,0,0,2,0

Figure 6.3: Dataset Used