Configuration Manual for CSIC web application attacks classifier

Instructions for setting up and implementing a classifier for web application assaults are provided in this manual. The classifier assists in identifying possible security risks by utilizing a machine learning model to distinguish between legitimate and suspect web traffic.

1. Requirements

Software requirements	Data Files	Hardware requirements
Python Pandas Streamlit Anaconda	Normal_class.csv file Suspicious_class.csv	Ram: 16 GB Processor: Intel i5 OS: Windows 11

2 Code execution

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import re
import math
import eli5

#dataset pre-processing realated imports
import sklearn
from eli5.sklearn import PermutationImportance
from urllib.parse import urlparse
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
```

```
#imports related to classifiers
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import precision_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.linear_model import SGDClassifier
from sklearn.svm import LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.neural_network import MLPClassifier

#DNN imports
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import ReduceLROnPlateau

#Explainability imports
from sklearn.inspection import PartialDependenceDisplay
```

Figure1. This code imports libraries and modules for various tasks:

- Data Handling: `numpy`, `pandas` for numerical operations and data manipulation.
- Visualization: `seaborn`, `matplotlib.pyplot` for plotting.
- Text Processing: `re` for regular expressions.
- Math Operations: `math` for mathematical functions.

- Explainability: `eli5`, `PartialDependenceDisplay` for model interpretability.

- Machine Learning: `sklearn` and `xgboost` for preprocessing, classifiers, metrics, and model evaluation.

- Deep Learning: `tensorflow` for building neural networks.

The code imports libraries for data processing, visualization, and machine learning, including various classifiers, deep learning with TensorFlow, and tools for model explainability such as ELI5 and Partial Dependence Displays.

```
csic_filepath='/content/drive/MyDrive/csic_database.csv'
csic_data=pd.read_csv(csic_filepath)
print('Done!')
```

Figure2. This code loads a CSV file located at `'/content/drive/MyDrive/csic_database.csv'` into a pandas DataFrame called `csic_data` and prints 'Done!' once the operation is complete.

```
n_features=csic_data.shape[1]
n_samples =csic_data.shape[0]


print("Number of samples:", n_samples)
print("Number of features:", n_features)
```

Figure3. The code calculates and prints the number of samples and features in the `csic_data` dataset using its shape attribute. `n_samples` represents the total number of rows, and `n_features` represents the total number of columns.

```
# get the number of missing data points per feature
print(f'number of features: {n_features}')
missing_values_count = csic_data.isnull().sum()
missing_values_count[0:n_features]
```

Figure4. This code computes and prints the count of missing values for each feature in the `csic_data` DataFrame. It shows the number of missing values for all features.

```
total_cells = np.product(csic_data.shape)
total_missing = missing_values_count.sum()
percent_missing = (total_missing/total_cells) * 100
print('percentage missing:',(f'{percent_missing:.2f}') ,'%')
```

Figure5. This code calculates the percentage of missing data in the `csic_data` DataFrame. It finds the total number of cells, sums up the missing values, computes the percentage, and prints it.

Feature Engineering:

```
#compute the number of unique values in each feature
for feature in csic_data.columns:
    if feature in csic_data.columns:
        unique_count = csic_data[feature].nunique()
        print(f"Number of unique values for {feature}: {unique_count}")
    else:
        print(f"Column '{feature}' does not exist in the DataFrame.")
```

Figure6. This code iterates through each feature in `csic_data` and prints the number of unique values for each feature. If a feature doesn't exist, it prints a corresponding message.

```
X = X.rename(columns={'Unnamed: 0': 'Class'})
X = X.rename(columns={'lenght': 'content_length'})

feature_names=[ 'Class','Method','host','cookie','Accept', 'content_length', 'content','classification','URL']

# Print the remaining data
X = X[feature_names]
print(X)
```

Figure7. This code renames specific columns in the DataFrame `X` for clarity, adjusts the feature names, and then selects only the specified columns from `X`. It finally prints the resulting DataFrame.

```
size=X.shape[1]
# Get list of categorical variables
s = (X.dtypes == 'object')
object_cols = list(s[s].index)

print("Categorical variables:")
print(object_cols)
```

Figure8. This code identifies and prints the names of categorical variables (features with data type 'object') in the DataFrame `X`. It checks each feature's data type and lists those that are categorical.

```
X['content_length'] = X['content_length'].astype(str)
X['content_length'] = X['content_length'].str.extract(r'(\d+)')
X['content_length'] = pd.to_numeric(X['content_length'], errors='coerce').fillna(0)
print(X.content_length)
```

Figure9. This code handles missing values in the `content_length` column by:

1. Converting `content_length` to strings.

2. Extracting numeric values from the strings using regex.

3. Converting the extracted values back to numeric format, replacing any non-numeric entries with `NaN`.

4. Filling these `NaN` values with `0`.

It then prints the cleaned `content_length` column.

```
filtered_length = X.loc[X['Method'] == 'GET', 'content_length']
print(filtered_length)
```

Figure10. This code filters the `X` DataFrame to show the `content_length` values only for rows where the `Method` column is 'GET'. It then prints these filtered values.

URL Preprocessing:

```
url_counts = X['URL'].value_counts()
most_common_urls = url_counts.head(10)  # Extract the top 10 most common strings

print("Most common URLs:")
for i, (url, count) in enumerate(most_common_urls.items(), 1):
    print(f"{i}. URL: {url} - Count: {count}")
```

Figure11. This code counts the occurrences of each unique URL in the `URL` column, extracts the top 10 most frequent URLs, and prints them along with their counts.

```
def count_dot(url):
    count_dot = url.count('.')
    return count_dot


def no_of_dir(url):
    urldir = urlparse(url).path
    return urldir.count('/')

def no_of_embed(url):
    urldir = urlparse(url).path
    return urldir.count('//')

def shortening_service(url):
    match = re.search('bit\.ly|goo\.gl|shorte\.st|go2l\.ink|x\.co|ow\.ly|t\.co|tinyurl|tr\.im|is\.gd|cli\.gs|'
                      'yfrog\.com|migre\.me|ff\.im|tiny\.cc|url4\.eu|twit\.ac|su\.pr|twurl\.nl|snipurl\.com|'
                      'short\.to|BudURL\.com|ping\.fm|post\.ly|Just\.as|bkite\.com|snipr\.com|fic\.kr|loopt\.us|'
                      'doiop\.com|short\.ie|kl\.am|wp\.me|rubyurl\.com|om\.ly|to\.ly|bit\.do|t\.co|lnkd\.in|'
                      'db\.tt|qr\.ae|adf\.ly|goo\.gl|bitly\.com|cur\.lv|tinyurl\.com|ow\.ly|bit\.ly|ity\.im|'
                      'q\.gs|is\.gd|po\.st|bc\.vc|twitthis\.com|u\.to|j\.mp|buzurl\.com|cutt\.us|u\.bb|yourls\.org|'
                      'x\.co|prettylinkpro\.com|scrnch\.me|filoops\.info|vzturl\.com|qr\.net|1url\.com|tweez\.me|v\.gd|'
                      'tr\.im|link\.zip\.net',
                      url)
    if match:
        return 1
    else:
        return 0
```

```python
def count_http(url):
bug (Ctrl+Shift+D)l.count('http')

def count_per(url):
    return url.count('%')

def count_ques(url):
    return url.count('?')

def count_hyphen(url):
    return url.count('-')


def count_equal(url):
    return url.count('=')


def url_length(url):
    return len(str(url))

#Hostname Length

def hostname_length(url):
    return len(urlparse(url).netloc)
```

```python
import re

def suspicious_words(url):
    score_map = {···

    matches = re.findall(r'(?i)' + '|'.join(score_map.keys()), url)

    total_score = sum(score_map.get(match.lower(), 0) for match in matches)
    return total_score


def digit_count(url):
    digits = 0
    for i in url:
        if i.isnumeric():
            digits = digits + 1
    return digits

def letter_count(url):
    letters = 0
    for i in url:
        if i.isalpha():
            letters += 1
    return letters

def count_special_characters(url):
    special_characters = re.sub(r'[a-zA-Z0-9\s]', '', url)
    count = len(special_characters)
    return count
```

```python
# Number of Parameters in URL
def number_of_parameters(url):
    params = urlparse(url).query
    return 0 if params == '' else len(params.split('&'))

# Number of Fragments in URL
def number_of_fragments(url):
    frags = urlparse(url).fragment
    return len(frags.split('#')) - 1 if frags == '' else 0

# URL is Encoded
def is_encoded(url):
    return int('%' in url.lower())


def unusual_character_ratio(url):
    total_characters = len(url)
    unusual_characters = re.sub(r'[a-zA-Z0-9\s\-._]', '', url)
    unusual_count = len(unusual_characters)
    ratio = unusual_count / total_characters if total_characters > 0 else 0
    return ratio
```

Figure13. These functions extract and compute various features from URLs:

- `count_dot(url)`: Counts dots (`.`) in the URL.

- `no_of_dir(url)`: Counts directory separators (`/`) in the URL path.

- `no_of_embed(url)`: Counts embedded URL separators (`//`) in the URL path.

- `shortening_service(url)`: Checks if the URL uses a known shortening service.

- `count_http(url)`: Counts occurrences of 'http' in the URL.

- `count_per(url)`: Counts percentage signs (`%`) in the URL.

- `count_ques(url)`: Counts question marks (`?`) in the URL.

- `count_hyphen(url)`: Counts hyphens (`-`) in the URL.

- `count_equal(url)`: Counts equal signs (`=`) in the URL.

- `url_length(url)`: Returns the length of the URL.

- `hostname_length(url)`: Returns the length of the hostname part of the URL.

- `suspicious_words(url)`: Scores the URL based on the presence of suspicious words and patterns.

- `digit_count(url)`: Counts digits in the URL.

- `letter_count(url)`: Counts letters in the URL.

- `count_special_characters(url)`: Counts special characters in the URL.

- `number_of_parameters(url)`: Counts the number of parameters in the URL query string.

- `number_of_fragments(url)`: Counts the number of fragments in the URL.
- `is_encoded(url)`: Checks if the URL contains URL encoding.
- `unusual_character_ratio(url)`: Calculates the ratio of unusual characters in the URL

```
X['URL'] = X['URL'].astype(str)


X['count_dot_url'] = X['URL'].apply(count_dot)
X['count_dir_url'] = X['URL'].apply(no_of_dir)
X['count_embed_domain_url'] = X['URL'].apply(no_of_embed)
X['short_url'] = X['URL'].apply(shortening_service)
X['count-http'] = X['URL'].apply(count_http)
X['count%_url'] = X['URL'].apply(count_per)
X['count?_url'] = X['URL'].apply(count_ques)
X['count-_url'] = X['URL'].apply(count_hyphen)
X['count=_url'] = X['URL'].apply(count_equal)
X['hostname_length_url'] = X['URL'].apply(hostname_length)
X['sus_url'] = X['URL'].apply(suspicious_words)
X['count-digits_url'] = X['URL'].apply(digit_count)
X['count-letters_url'] = X['URL'].apply(letter_count)
X['url_length'] = X['URL'].apply(url_length)
X['number_of_parameters_url'] = X['URL'].apply(number_of_parameters)
X['number_of_fragments_url'] = X['URL'].apply(number_of_fragments)
X['is_encoded_url'] = X['URL'].apply(is_encoded)
X['special_count_url'] = X['URL'].apply(count_special_characters)
X['unusual_character_ratio_url'] = X['URL'].apply(unusual_character_ratio)
```

Figure14. This code applies various URL feature extraction functions to the `URL` column in the DataFrame `X` and creates new columns for each extracted feature:

- Dot count, directory slashes, embedded slashes: Adds columns for counts of dots, slashes, and embedded slashes.

- Shortening service, HTTP count, percent signs, etc.: Adds columns for URL shortening service presence, HTTP occurrences, percent signs, and other URL-specific metrics.

- Hostname length, suspicious words score, digit/letter count: Adds columns for hostname length, suspicious words score, and counts of digits and letters.

- URL length, query parameters, fragments, encoding, special characters, unusual character ratio: Adds columns for URL length, number of parameters and fragments, encoding check, special characters count, and unusual character ratio.

```python
unique_count = X['cookie'].nunique()
print(f"Count of unique values in 'cookie': {unique_count}")
```

Figure15. This code counts and prints the number of unique values in the `cookie` column of the DataFrame `X`.

```python
X['Accept'] = X['Accept'].astype(str)
X['Accept'] = X['Accept'].str.extract(r'(\d+)')
X['Accept'] = pd.to_numeric(X['Accept'], errors='coerce').fillna(1)
```

```python
lb_make = LabelEncoder()
X["Method_enc"] = lb_make.fit_transform(X["Method"])
X["host_enc"] =lb_make.fit_transform(X["host"])
X["Accept_enc"] =lb_make.fit_transform(X["Accept"])
```

```python
unique_count_met = X["Method_enc"].nunique()
unique_count_host = X["host_enc"].nunique()
unique_count_acc = X["Accept_enc"].nunique()


print(f"Number of unique values for 'Method_enc': {unique_count_met}")
print(f"Number of unique values for 'host_enc': {unique_count_host}")
print(f"Number of unique values for 'Accept_enc': {unique_count_acc}")
```

Figure16. This code performs the following:

1. Processes `Accept`: Converts values to strings, extracts numeric values, and fills non-numeric entries with `1`.

2. Encodes categorical features: Uses `LabelEncoder` to convert the `Method`, `host`, and `Accept` columns into numeric representations.

3. Counts unique encoded values: Prints the number of unique values for the encoded columns (`Method_enc`, `host_enc`, and `Accept_enc`).

```python
def apply_to_content(content,function):
    if pd.isna(content):
        return 0
    elif isinstance(content, str):
        return function(content)

#"""
#               'count_dot_content','count_dir_content','count_embed_domain_content','count%_content','count?_content',
 #              'count-_content','count=_content','hostname_length_content','sus_content','count_digits_content',
  #             'count_letters_content','content_length','number_of_parameters_content','number_of_fragments_content',
   #            'is_encoded_content','special_count_content','unusual_character_ratio_content'
    #           ]"""

X['count_dot_content'] = X['content'].apply(apply_to_content, function=count_dot)
X['count_dir_content'] = X['content'].apply(apply_to_content, function=no_of_dir)
X['count_embed_domain_content'] = X['content'].apply(apply_to_content, function=no_of_embed)
X['count%_content'] = X['content'].apply(apply_to_content, function=count_per)
X['count?_content'] = X['content'].apply(apply_to_content, function=count_ques)
X['count-_content'] = X['content'].apply(apply_to_content, function=count_hyphen)
X['count=_content'] = X['content'].apply(apply_to_content, function=count_equal)
X['content_length'] = X['content'].apply(apply_to_content, function=url_length)
X['sus_content'] = X['content'].apply(apply_to_content, function=suspicious_words)
X['count_digits_content'] = X['content'].apply(apply_to_content, function=digit_count)
X['count_letters_content'] = X['content'].apply(apply_to_content, function=letter_count)
X['special_count_content'] = X['content'].apply(apply_to_content, function=count_special_characters)
X['is_encoded_content'] = X['content'].apply(apply_to_content, function=is_encoded)
#X['unusual_character_ratio_content'] = X['content'].apply(apply_to_content, function=unusual_character_ratio)
```

Figure17. This code applies a set of functions to the `content` column in the DataFrame `X` using the `apply_to_content` function:

1. Handles missing values: Returns `0` if the content is `NaN`.

2. Applies functions: For non-missing content, it applies various functions to extract features, such as dot count, directory slashes, and encoded content.

It calculates and adds these features to new columns in `X`. Note that the `unusual_character_ratio_content` line is commented out.

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Select the features and class variable for plotting
new_content_features = ['count_dot_content', 'count_dir_content', 'count_embed_domain_content', 'count%_content', 'count?_content',
                        'count-_content', 'count=_content', 'sus_content', 'count_digits_content',
                        'count_letters_content', 'content_length', 'is_encoded_content', 'special_count_content']

# Create a DataFrame with the selected features
selected_features_df = X[new_content_features]

for feature_name in selected_features_df.columns:
    if feature_name in X.columns:
        unique_count = selected_features_df[feature_name].nunique()
        print(f"Number of unique values for {feature_name}: {unique_count}")
    else:
        print(f"Column '{feature_name}' does not exist in the DataFrame.")
```

Figure18. This code does the following:

1. Defines features: Lists the features to be plotted.

2. Creates a DataFrame: Extracts these features from `X` into `selected_features_df`.

3. Counts unique values: Prints the number of unique values for each feature in `selected_features_df`. If a feature does not exist, it prints a corresponding message.

```
labels=['count_dot_url', 'count_dir_url', 'count_embed_domain_url', 'count-http',
        'count%_url', 'count?_url', 'count-_url', 'count=_url', 'url_length', 'hostname_length_url',
        'sus_url', 'count-digits_url', 'count-letters_url', 'number_of_parameters_url',
        'is_encoded_url','special_count_url','unusual_character_ratio_url',
         #method
        'Method_enc',
        #content
        'count_dot_content','count%_content',
         'count-_content','count=_content','sus_content','count_digits_content',
          'count_letters_content','content_length',
       'is_encoded_content','special_count_content']
print(X[labels])
```

Figure19. This code selects and prints specific columns from the DataFrame `X`, defined in the `labels` list. It shows the values of these columns for each row in `X`.

Final labels of the dataframe before classification.

```
random_forest_model = RandomForestClassifier(random_state=1000)
print('Computing....')
# Fit the model
random_forest_model.fit(x_tr,y_tr)
print('Done!')
```

Figure20. This code initializes a `RandomForestClassifier` with a fixed random state for reproducibility, fits the model to training data (`x_tr` and `y_tr`), and prints status messages before and after the fitting process.

```
final_model = KNeighborsClassifier(n_neighbors=9)
final_model.fit(x_tr, y_tr)
knn_predictions = final_model.predict(x_ts)
```

Figure21. This code initializes a `KNeighborsClassifier` with 9 neighbors, fits the model to the training data (`x_tr` and `y_tr`), and then makes predictions on the test data (`x_ts`).

```
DT_model = DecisionTreeClassifier(random_state=2)
print('Computing....')
DT_model.fit(x_tr,y_tr)
print('Done!')
```

Figure22. This code initializes a `DecisionTreeClassifier` with a fixed random state, fits the model to the training data (`x_tr` and `y_tr`), and prints status messages before and after the fitting process.

```
MLP_model = MLPClassifier(hidden_layer_sizes=(50,50),activation='relu',alpha=0.0001,learning_rate='adaptive',max_iter=500)
MLP_model.fit(x_tr,y_tr)
print('Done!')
```

Figure23. This code initializes an `MLPClassifier` with a neural network architecture of two hidden layers (each with 50 neurons), ReLU activation, a small regularization parameter

(`alpha`), adaptive learning rate, and a maximum of 500 iterations. It then fits the model to the training data (`x_tr` and `y_tr`) and prints 'Done!' upon completion.

```
SVC_model = LinearSVC(max_iter=300,C=0.1, penalty='l2',random_state=10)
print('Computing....')
# Fit the model
SVC_model.fit(x_tr,y_tr)
print('Done!')
```

Figure24. This code initializes a `LinearSVC` model with a maximum of 300 iterations, a regularization parameter `C` of 0.1, an L2 penalty, and a fixed random state. It then fits the model to the training data (`x_tr` and `y_tr`) and prints status messages before and after the fitting process.

```
SGD_model=SGDClassifier(alpha=0.001,max_iter=4000,penalty='l1')
SGD_model.fit(x_tr,y_tr)
SGD_predictions= SGD_model.predict(x_ts)
print("MAE",mean_absolute_error(y_ts,SGD_predictions))
print("Accuracy", accuracy_score(y_ts, SGD_predictions))
print("Precision", precision_score(y_ts, SGD_predictions, average='weighted', labels=np.unique(SGD_predictions)))
print("Recall", recall_score(y_ts, SGD_predictions, average='weighted', labels=np.unique(SGD_predictions)))
print("F1", f1_score(y_ts, SGD_predictions, average='weighted', labels=np.unique(SGD_predictions)))
print("ROC AUC", roc_auc_score(y_ts, SGD_predictions, average='weighted', labels=np.unique(SGD_predictions)))
error_sgd = (SGD_predictions != y_ts).mean()
print("Test error: {:.1%}".format(error_sgd))
```

Figure25. This code trains an `SGDClassifier` with specified hyperparameters and evaluates its performance using the test data (`x_ts` and `y_ts`):

1. Fits the model: Trains the Best Model with the training data.

2. Makes predictions: Predicts labels for the test data.

3. Evaluates performance:

- MAE: Mean Absolute Error.
- Accuracy: Overall accuracy.
- Precision: Weighted precision score.
- Recall: Weighted recall score.
- F1: Weighted F1 score.
- ROC AUC: ROC Area Under the Curve score.
- Test Error: Proportion of misclassified instances.

The results are printed for each metric.

3 Steps to Run and Execute the Codes

Setting Up the Environment for Streamlit Platform

- Install required Python libraries

Run the code to install essential libraries.

Pip install streamlit

- Prepare the Data files

Prepared the normal_class.csv and suspicious_class.csv are placed in the directory.

- Load the Trained model

The trained model is saved as ‘trained_model.pk1’ is accessible.

Update and run the code

```
base) C:\Users\angel\OneDrive\Documents\Main Project>python -m streamlit run app.py

 You can now view your Streamlit app in your browser.

 Local URL: http://localhost:8501
 Network URL: http://192.168.0.122:8501
```

References

OWASP (2020). *OWASP Web Security Testing Guide*. [online] owasp.org. Available at: https://owasp.org/www-project-web-security-testing-guide/.

Cloudflare (n.d.). What Is Web Application Security? | Web Security | Cloudflare UK. *Cloudflare*. [online] Available at: https://www.cloudflare.com/en-gb/learning/security/what-is-web-application-security/.

GitHub. (2020). *zaproxy/zaproxy*. [online] Available at: https://github.com/zaproxy/zaproxy.

Hasan, M.M. (2024). *A Guide to Common Web Application Security Vulnerabilities and Mitigation*. [online] WebDevStory. Available at: https://www.webdevstory.com/web-application-security-vulnerabilities/.