Finetuning CodeBERT for Detecting Web Vulnerabilities

Introduction:

This tool focuses on the development and training of a model designed to classify text for common web vulnerabilities. The goal is to identify specific vulnerabilities, such as SQL Injection and Cross-Site Scripting (XSS), in text-based descriptions. This task is relevant to cybersecurity, where early detection of vulnerabilities can prevent security breaches and data loss.

Description of the Problem Solved:

Web vulnerabilities pose significant security risks to applications and systems. Common vulnerabilities, such as SQL Injection and XSS, can lead to unauthorized access, data manipulation, or other security incidents. Identifying these vulnerabilities in text-based descriptions from code snippets or security reports is crucial for maintaining secure applications.

To address this problem, my model was trained to classify text into predefined vulnerability types. The model takes text as input and predicts whether it contains a specific web vulnerability, allowing for automated detection and analysis.

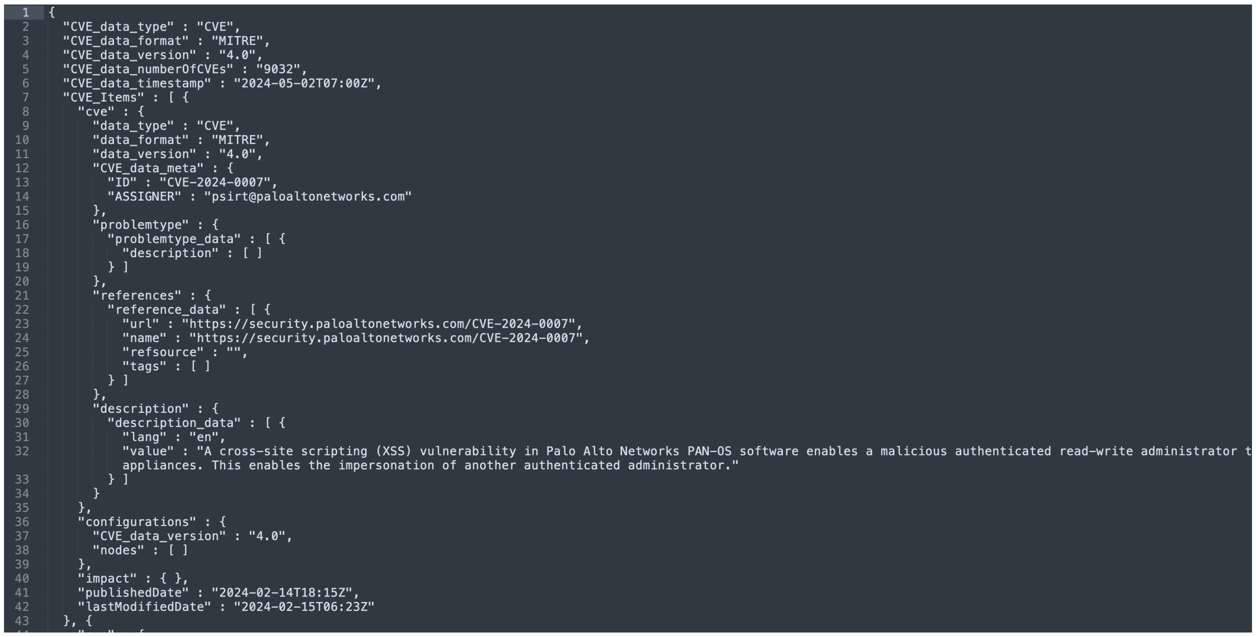
Description of the Algorithm Used:

The algorithm used for this task is based on a pre-trained language model, CodeBERT, which is designed for code-related tasks. This model was fine-tuned for the specific task of classifying web vulnerabilities.

The model was trained on a labeled dataset of text descriptions, using supervised learning techniques. It was trained for three epochs, with evaluation at the end of each epoch to monitor performance. The database that I used is from the National Vulnerability Database, provided by NIST:

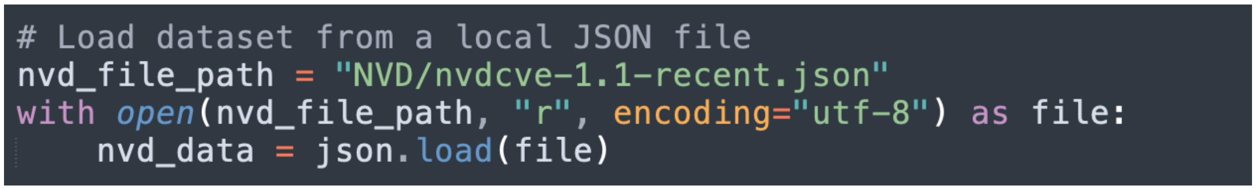
NIST
NVD MENU
Information Technology Laboratory
NATIONAL VULNERABILITY DATABASE
NIST NATIONAL VULNERABILITY DATABASE NVD
VULNERABILITIES

The above dataset provides reliable and structured data. It is an official website of the US government. I specifically selected the dataset that included the recent CTF challenges. The structure of the JSON file is shown in the below image:



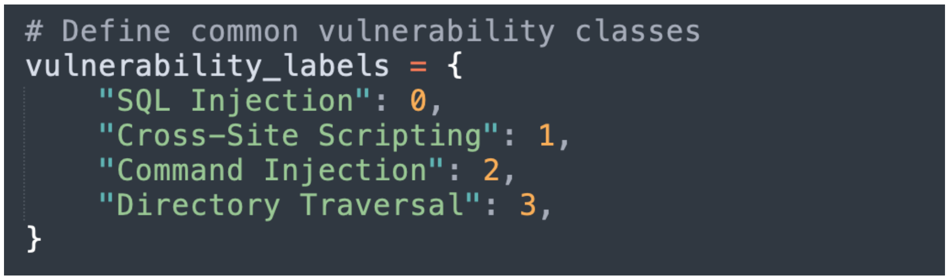
```json
{
  "CVE_data_type" : "CVE",
  "CVE_data_format" : "MITRE",
  "CVE_data_version" : "4.0",
  "CVE_data_numberOfCVEs" : "9032",
  "CVE_data_timestamp" : "2024-05-02T07:00Z",
  "CVE_Items" : [ {
    "cve" : {
      "data_type" : "CVE",
      "data_format" : "MITRE",
      "data_version" : "4.0",
      "CVE_data_meta" : {
        "ID" : "CVE-2024-0007",
        "ASSIGNER" : "psirt@paloaltonetworks.com"
      },
      "problemtype" : {
        "problemtype_data" : [ {
          "description" : [ ]
        } ]
      },
      "references" : {
        "reference_data" : [ {
          "url" : "https://security.paloaltonetworks.com/CVE-2024-0007",
          "name" : "https://security.paloaltonetworks.com/CVE-2024-0007",
          "refsource" : "",
          "tags" : [ ]
        } ]
      },
      "description" : {
        "description_data" : [ {
          "lang" : "en",
          "value" : "A cross-site scripting (XSS) vulnerability in Palo Alto Networks PAN-OS software enables a malicious authenticated read-write administrator t
          appliances. This enables the impersonation of another authenticated administrator."
        } ]
      }
    },
    "configurations" : {
      "CVE_data_version" : "4.0",
      "nodes" : [ ]
    },
    "impact" : { },
    "publishedDate" : "2024-02-14T18:15Z",
    "lastModifiedDate" : "2024-02-15T06:23Z"
  }, {
```

The data was imported into my program as a JSON file:



```python
# Load dataset from a local JSON file
nvd_file_path = "NVD/nvdcve-1.1-recent.json"
with open(nvd_file_path, "r", encoding="utf-8") as file:
    nvd_data = json.load(file)
```

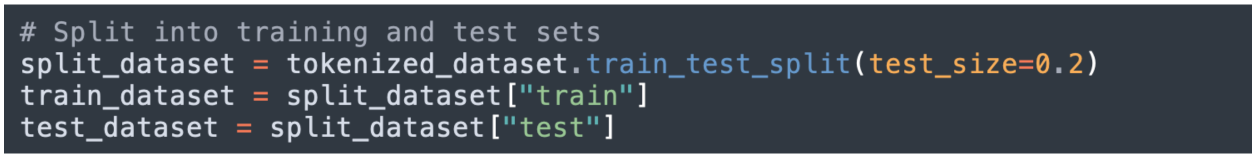
Since my goal is to classify the vulnerabilities, I labelled the following vulnerabilities which are common in web security:



```python
# Define common vulnerability classes
vulnerability_labels = {
    "SQL Injection": 0,
    "Cross-Site Scripting": 1,
    "Command Injection": 2,
    "Directory Traversal": 3,
}
```

We can see that the vulnerabilities that are classified are SQL injection, Cross-Site Scripting and Directory Traversal. I selected just these few popular and common types of web vulnerabilities because training the data and classifying them based on many vulnerability types can be hardware intensive. I tried to train the model locally in my laptop using jupyter notebooks, but due to hardware limitations, the application crashed multiple times. I switched to the FSU's linprog server and trained my model, and it took only 20 minutes and successfully trained my model. This will be further discussed in the results section.

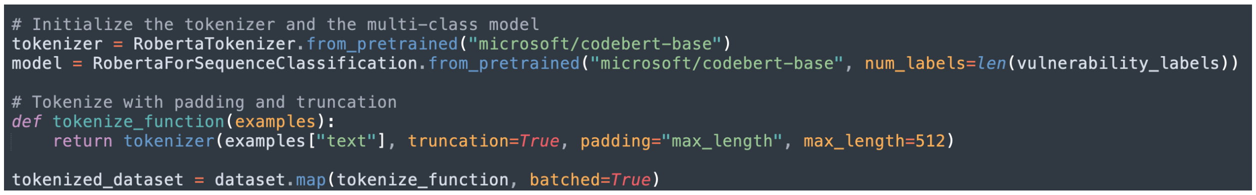
I split the data with 80% for training set and 20% for test set. An 80-20 train-test split is commonly used in machine learning for several reasons, balancing the need for sufficient training data with the importance of evaluating the model's generalization. The below image shows the code used for splitting the data:



```
# Split into training and test sets
split_dataset = tokenized_dataset.train_test_split(test_size=0.2)
train_dataset = split_dataset["train"]
test_dataset = split_dataset["test"]
```

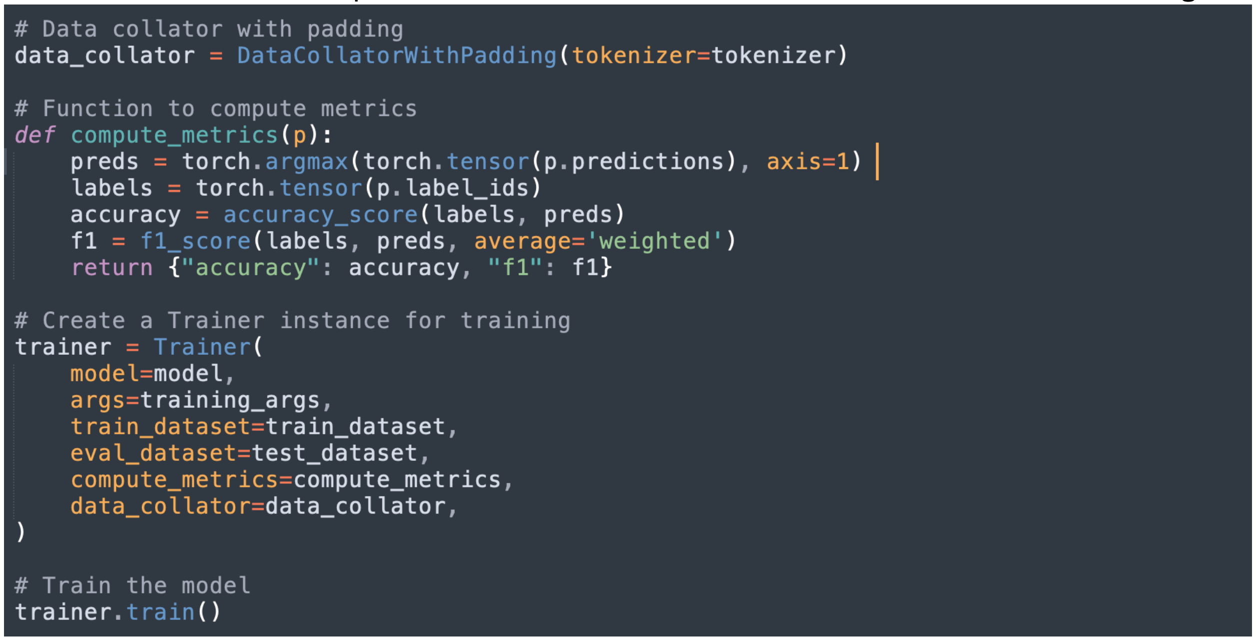
Then I trained the model using a tokenizer. The tokenizer converts text into a format that the model can process. It ensures consistent tokenization, including truncation and padding, to handle variable-length text inputs. The base model is CodeBERT, adapted for classification tasks. It outputs probabilities for each class, indicating the likelihood of a specific vulnerability.

The initialization of the tokenizer and the function used are shown in the below image:



```
# Initialize the tokenizer and the multi-class model
tokenizer = RobertaTokenizer.from_pretrained("microsoft/codebert-base")
model = RobertaForSequenceClassification.from_pretrained("microsoft/codebert-base", num_labels=len(vulnerability_labels))

# Tokenize with padding and truncation
def tokenize_function(examples):
    return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=512)

tokenized_dataset = dataset.map(tokenize_function, batched=True)
```

The function used to compute metrics and train the model are shown in the below image:



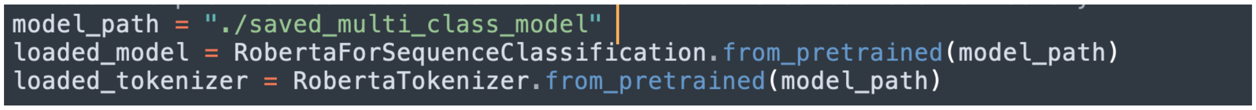
```
# Data collator with padding
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Function to compute metrics
def compute_metrics(p):
    preds = torch.argmax(torch.tensor(p.predictions), axis=1)
    labels = torch.tensor(p.label_ids)
    accuracy = accuracy_score(labels, preds)
    f1 = f1_score(labels, preds, average='weighted')
    return {"accuracy": accuracy, "f1": f1}

# Create a Trainer instance for training
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    compute_metrics=compute_metrics,
    data_collator=data_collator,
)

# Train the model
trainer.train()
```

Accuracy and F1-score are two very important ways to check if a model works right. Accuracy shows how many guesses the model got correct out of all its guesses. It is an easy way to see how well the model works overall. But accuracy has problems when there are not equal amounts of each class in the data. A high accuracy score might still mean the model does not work well for some classes.

The F1-score looks at both precision and recall giving a better view of how the model works. Precision indicates how many of the predicted positives are true positives, while recall measures how many of the actual positives were correctly predicted. The F1-score is especially useful in cases where both false positives and false negatives need to be considered. It provides a more detailed evaluation, especially in scenarios where class imbalance exists, ensuring the model doesn't just achieve high accuracy but also maintains a good balance between precision and recall. Combining accuracy and F1-score gives us a better result, making them excellent metrics for classification tasks.

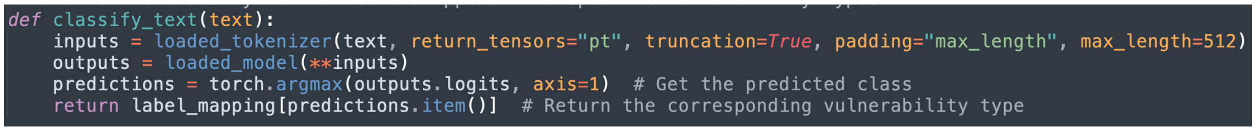
We can see in the last line of the above code; the model is saved to a directory. I wrote a separate program to import the saved model from the directory and solve the CTF problems.

The below image shows how I imported the model and the tokenizer (the tokenizer too was saved using a separate script that analyzed the imported model):



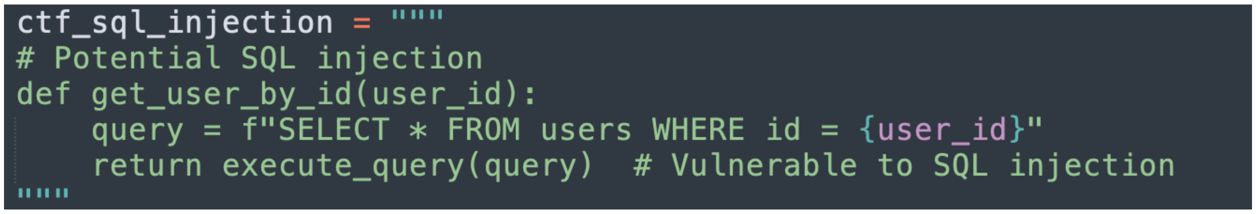
```
model_path = "./saved_multi_class_model"
loaded_model = RobertaForSequenceClassification.from_pretrained(model_path)
loaded_tokenizer = RobertaTokenizer.from_pretrained(model_path)
```

Then I used the following code to classify the text based on the different types of web vulnerabilities detected:



```
def classify_text(text):
    inputs = loaded_tokenizer(text, return_tensors="pt", truncation=True, padding="max_length", max_length=512)
    outputs = loaded_model(**inputs)
    predictions = torch.argmax(outputs.logits, axis=1)  # Get the predicted class
    return label_mapping[predictions.item()]  # Return the corresponding vulnerability type
```

I inserted sample text from code snippets used in CTFs to test the classification of my tool. The sample text was presented in the below format (The below image shows a sample code snippet for a code that is vulnerable to an SQL injection attacks):



```
ctf_sql_injection = """
# Potential SQL injection
def get_user_by_id(user_id):
    query = f"SELECT * FROM users WHERE id = {user_id}"
    return execute_query(query)  # Vulnerable to SQL injection
"""
```

After the model is used to classify the text, it is printed in the following format:

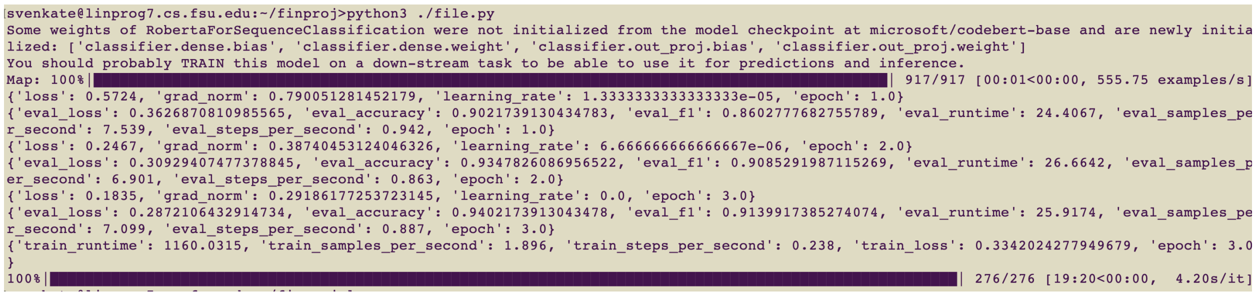
```python
# Classify the CTF code snippets using the loaded model
sql_result = classify_text(ctf_sql_injection)
xss_result = classify_text(ctf_xss)
command_injection_result = classify_text(ctf_command_injection)
directory_traversal_result = classify_text(ctf_directory_traversal)

# Output the results for each classified CTF problem
print("Classification Result for SQL Injection:", sql_result)
print("Classification Result for XSS:", xss_result)
print("Classification Result for Command Injection:", command_injection_result)
print("Classification Result for Directory Traversal:", directory_traversal_result)
```

The above code snippet shows the classification of different code snippets, each potentially having a specific type of web vulnerability. The classify_text function is used to determine which vulnerability is present in each of the code snippets. It returns the type of vulnerability based on the model's predictions. By running this classification for SQL Injection, Cross-Site Scripting (Xss), Command Injection, and Directory Traversal, the code aims to assess whether the pre-trained model correctly identifies the vulnerability types.

The output from these classifications can be used to understand the accuracy and reliability of the model in detecting web vulnerabilities. The results are printed in a way that links each code snippet to its expected vulnerability classification, providing a quick check on the model's performance. This high-level overview can be valuable for security professionals and developers to validate the effectiveness of a text classification model in identifying common security vulnerabilities.

Results:

I tried to train the model locally, but due to hardware limitations, the program just crashed because of the large dataset and complexity. So, I used the linprog server to train the model and took around 20 minutes for the model to get trained:



```
svenkate@linprog7.cs.fsu.edu:~/finproj>python3 ./file.py
Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint at microsoft/codebert-base and are newly initia
lized: ['classifier.dense.bias', 'classifier.dense.weight', 'classifier.out_proj.bias', 'classifier.out_proj.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Map: 100%|████████████████████████████████████████████████████████████████| 917/917 [00:01<00:00, 555.75 examples/s]
{'loss': 0.5724, 'grad_norm': 0.790051281452179, 'learning_rate': 1.3333333333333333e-05, 'epoch': 1.0}
{'eval_loss': 0.3626870810985565, 'eval_accuracy': 0.9021739130434783, 'eval_f1': 0.8602777682755789, 'eval_runtime': 24.4067, 'eval_samples_pe
r_second': 7.539, 'eval_steps_per_second': 0.942, 'epoch': 1.0}
{'loss': 0.2467, 'grad_norm': 0.38740453124046326, 'learning_rate': 6.666666666666667e-06, 'epoch': 2.0}
{'eval_loss': 0.30929407477378845, 'eval_accuracy': 0.9347826086956522, 'eval_f1': 0.9085291987115269, 'eval_runtime': 26.6642, 'eval_samples_p
er_second': 6.901, 'eval_steps_per_second': 0.863, 'epoch': 2.0}
{'loss': 0.1835, 'grad_norm': 0.29186177253723145, 'learning_rate': 0.0, 'epoch': 3.0}
{'eval_loss': 0.2872106432914734, 'eval_accuracy': 0.9402173913043478, 'eval_f1': 0.9139917385274074, 'eval_runtime': 25.9174, 'eval_samples_pe
r_second': 7.099, 'eval_steps_per_second': 0.887, 'epoch': 3.0}
{'train_runtime': 1160.0315, 'train_samples_per_second': 1.896, 'train_steps_per_second': 0.238, 'train_loss': 0.3342024277949679, 'epoch': 3.0
}
100%|██████████████████████████████████████████████████████████████████████| 276/276 [19:20<00:00,  4.20s/it]
```

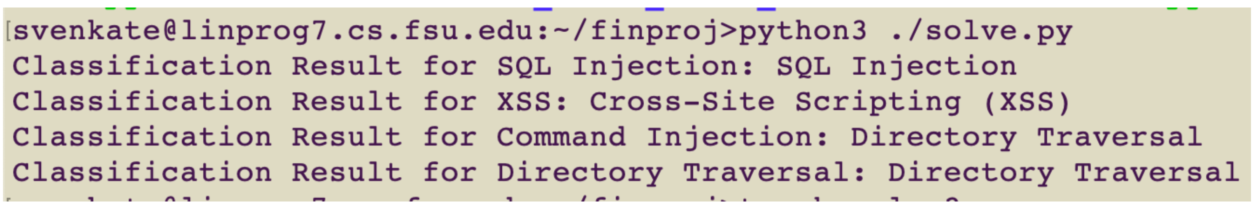
I was able to gather the following information based on the metrics shown in the above image:

The training loss decreased from 0.5724 in the first epoch to 0.1835 in the third epoch, indicating that the model improved as training progressed. The gradient norm, a measure of the gradients' magnitude, decreased from 0.790 in the first epoch to 0.292 in the third epoch, suggesting stable training. The learning rate was adjusted throughout training, starting at 1.333e-05 and ending at zero, indicating a learning rate schedule that decays over time.

Accuracy improved from 90.2% in the first epoch to 94.0% in the third epoch, indicating high accuracy in classification. F1-Score increased from 0.860 to 0.914 over the course of three epochs, suggesting better balance between precision and recall. Precision improved alongside F1-score, showing increased accuracy in positive predictions.

These results suggest that the model performed well, with high accuracy and improving metrics over the training epochs. The decrease in loss and gradient norm, along with increasing accuracy, F1-score, and precision, indicate that the model effectively learned to classify text for common web vulnerabilities.

To test the model against text taken code snippets from CTF, I wrote a script that takes a few examples and classifies them. I also added an example where the code snippet could be vulnerable to two types of attacks:



```
[svenkate@linprog7.cs.fsu.edu:~/finproj>python3 ./solve.py
Classification Result for SQL Injection: SQL Injection
Classification Result for XSS: Cross-Site Scripting (XSS)
Classification Result for Command Injection: Directory Traversal
Classification Result for Directory Traversal: Directory Traversal
```

As shown in the above image, the model was able to classify three out of four types of vulnerabilities successfully. The classification result for the command injection was shown as Directory Traversal because the code had multiple vulnerabilities.