Security Insider Lab II

Third Lab - Creating LSTM Model

Group 5: Sayed Alisina Qaderi, Atiqullah Ahmadzai & Dusan Dordevic

---

29. Mai 2024

Contents

List of Figures

List of Tables

1 Introduction

This lab focused on demonstrating a previously trained Long Short-Term Memory (LSTM) model and implementing alternative machine learning models for vulnerability detection in software projects.

1.1 Phase 1: LSTM Model Demonstration

In the first phase, we revisited the LSTM model, observed its performance on test data, and analyzed its strengths and weaknesses.

1.2 Phase 2: Implementation of Alternative Models

In the second phase, we implemented Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), and Gated Recurrent Unit (GRU) models for vulnerability detection.

1.3 Objectives

The primary objectives were to demonstrate the LSTM model's effectiveness, evaluate alternative models, and compare their performance for vulnerability detection.

2 Methods

Before jumping to our main task we want to point out some changes that we brought to the repository. The main purpose of these changes was to prevent more reworking in the future. The below list shows the changes:

- Split the repository codes to different .txt files.
- Adding #endfile to all repository codes, will help us during tokenization. Figure 2.1 shows the changes.
- Code restructure. Figure 2.2 shows the new repository structure.

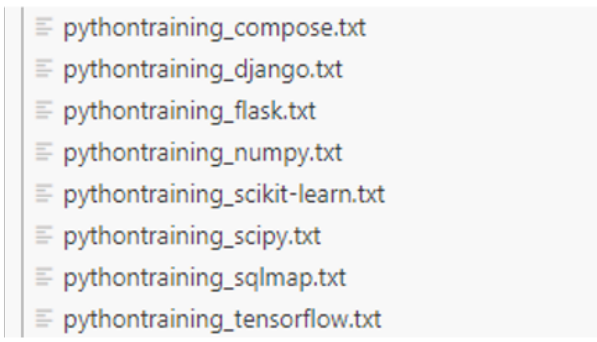


Figure 2.1: FinaltextX Array Element Error

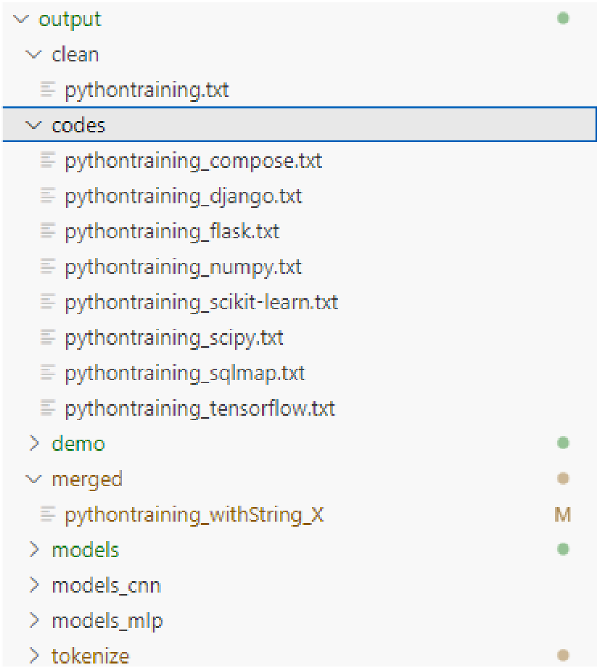


Figure 2.2: FinaltextX Array Element Error

2.1 Demo Presentation

2.1.1 First Step

In the first step, we are required to run the *trymodel.py* script. Running the mentioned script has some challenges due to code deprecation and structural changes brought before. The challenges that we faced are listed below:

- **FinaltextX** conversion to arra, the reason behind this error was due to setting an array element with a sequence. The figure 2.3 shows the error. The solution for this issue was to create a method to convert an array that is compatible with a numpy array. Figure 2.4 shows the solution with the changes.

- **yhat__classes** variable had issue with shaping after adding pad sequence to *X_final-test*, figure 2.5 shows the error. The solutions for the mentioned issue are listed below:

  1. Multipying *max_lenght* by 300 because the vector lenght was changed in *convert_to_array* method which Figure 2.6 shows the solution.

  2. Change the **yhat__classes** variable value to *predict* and take it back with the help of numpy *argmx* method. The changes are shown in figure 2.7.

  3. Adding *zero_devision = 1* and *average=ḿacró* to the precision. Figure 2.8 shows the changes.

  4. Adding *average=ḿacró* to *recall* and *F1Score* varaibles. Figure 2.8 shows the changes.

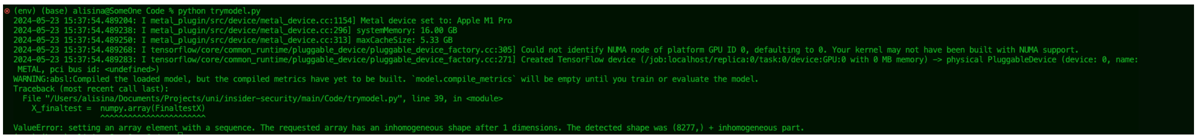


Figure 2.3: FinaltextX Array Element Error

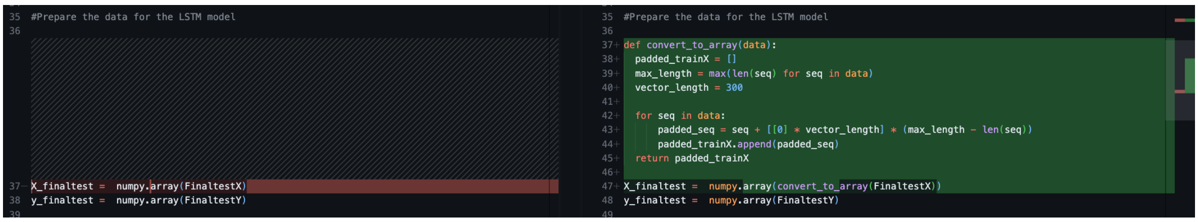


Figure 2.4: Implementation of convert to array method

Figure 2.5: Predict issue

Figure 2.6: Max Lenght multiplication with 300

Figure 2.7: Change on Yhat classes

Figure 2.8: Changes on Precision, Recall, F1Score variables

2.2 Model Demonstration

To demonstrate the model, we have to run the *demonstrat.py* script. While running the mentioned script we have faced with issues in call *myutils.getblocksVisual* method.

- *word_vectors* vocab parameter deprication. Figure 2.9 shows the error.
- *W2v_model* list is required to receive from **wv** parameter. Figure 2.9 shows the error.
- While rendering the demonstrated image the text was layer over layer. In addition, it returned some errors related to text size. Figure 2.10 shows the problem of the demonstrated image.

The solution for this issue was to bring changes in the *getblocksVisual* method. The changes are listed in the following:

- The vocab variable of *word_vectors* changed to *key_to_index*.

- Deu to changes in *W2v_model* library, we need to use **wv** object to get array instead of directly receive it from *W2v_model*.

- Deu to changes in PIL library for python version 3.11 *textsize* has an issue, so instead we used textbbox to handle the situation. Figure 2.11 shows the original code of the repository and figure 2.12 shows the code which was updated by us. The updated code uses *d.textbbox* instead of *d.textsize*. The textbbox method provides a more accurate bounding box for the text, which is particularly useful for correctly calculating the width and ensuring precise placement of subsequent text. The [2] index accesses the width from the bounding box tuple (left, top, right, bottom). The updates ensure that both the x and y positions are incremented consistently, which helps maintain the correct layout and spacing of the text within the image.

After fixing all the above issues we managed to successfully run the *demonstrate.py* script. Figure 2.13 shows a successful log of run *demonstrate.py* script.

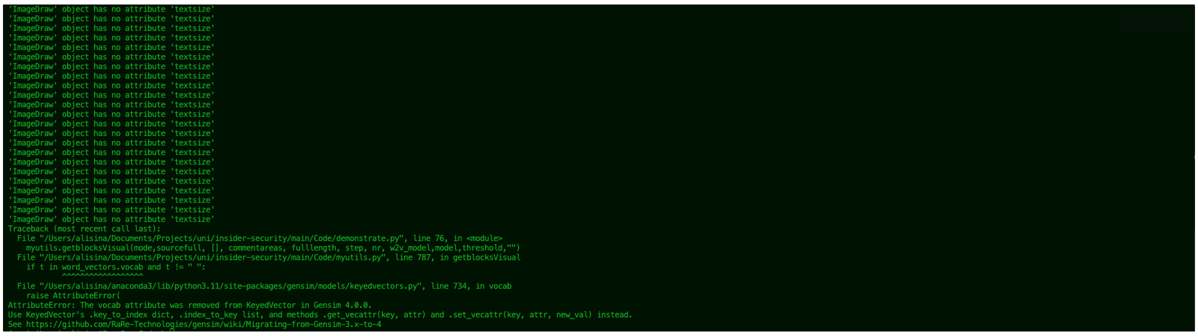


Figure 2.9: W2V code deprecation issues

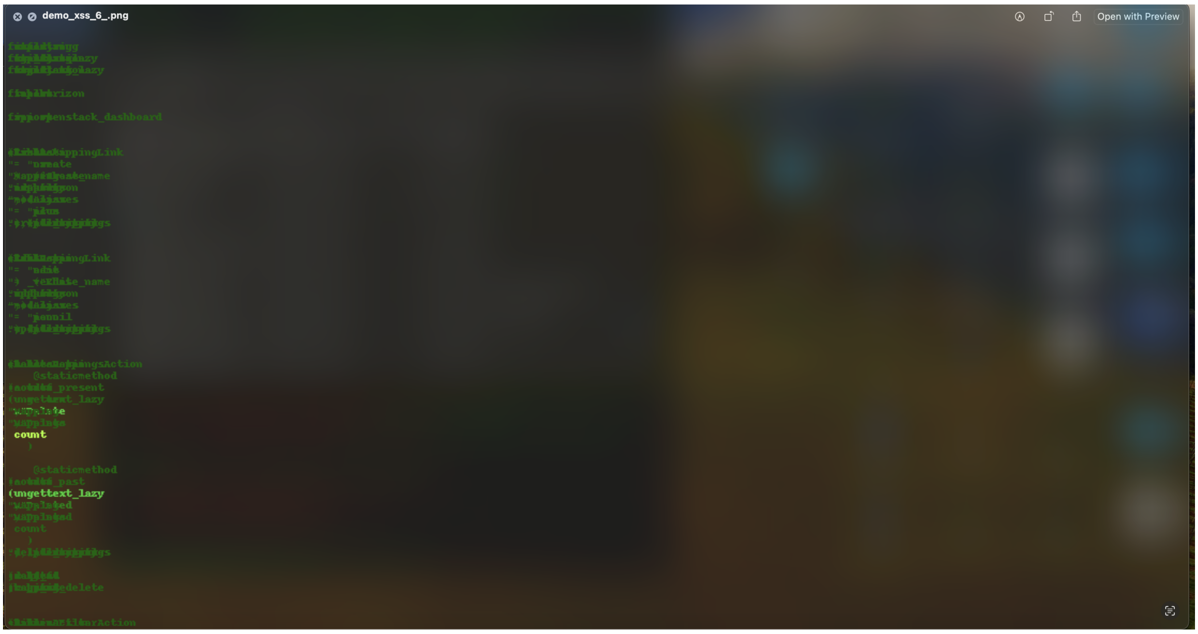


Figure 2.10: Text layers issue in generated image

```
try:
  if len(focusarea) > 0:
    d = ImageDraw.Draw(img)
    if focusarea[0] == "\n":
      ypos = ypos + 11
      xpos = 0
      d.text((xpos, ypos), focusarea[1:], fill=color)
      xpos = xpos + d.textsize(focusarea)[0]
    else:
      d.text((xpos, ypos), focusarea, fill=color)
      xpos = xpos + d.textsize(focusarea)[0]
```

Figure 2.11: Textsize original implemented code

```python
try:
    if len(focusarea) > 0:
        d = ImageDraw.Draw(img)
        if focusarea[0] == "\n":
            ypos += 11
            xpos = 0
            d.text((xpos, ypos), focusarea[1:], fill=color)
            xpos += d.textbbox((0, 0), focusarea, font=None)[2]
        else:
            d.text((xpos, ypos), focusarea, fill=color)
            xpos += d.textbbox((0, 0), focusarea, font=None)[2]
```

Figure 2.12: Text size updated code

```
(env) (base) alisina@SomeOne Code % python demonstrate.py remote_code_execution 3 fine
2024-05-27 16:53:25.639748: I metal_plugin/src/device/metal_device.cc:1154] Metal device set to: Apple M1 Pro
2024-05-27 16:53:25.639785: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 16.00 GB
2024-05-27 16:53:25.639792: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize: 5.33 GB
2024-05-27 16:53:25.639821: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA support.
2024-05-27 16:53:25.639870: I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevice (device: 0, na
me: METAL, pci bus id: <undefined>)
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
finished loading
2024-05-27 16:53:25.791391: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enabled.
saved png.
(env) (base) alisina@SomeOne Code %
```

Figure 2.13: Successfull running demonstrate script

2.3 Other ML Models

2.3.1 CNN

To implement the CNN model we made the below changes:

- We removed (*from keras.preprocessing import sequence*) and add (*from tensorflow.keras.preprocessing.sequence import pad_sequences*).
- We add new lines such as:
  - *from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense*

- *from tensorflow.keras.utils import to_categorical*
- *model.add(Conv1D(64, kernel_size=3, activation='relu'))*
- *model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])*

Besides the above changes, we added new features in the make code as well. The above is a sample of our codes.

2.3.2 MLP

To implement the MLP model we brought the changes added some features in the utils script and made a model file. The sample of the new codes are shown in figure 2.14, 2.15, 2.16.

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
model = MLPClassifier(hidden_layer_sizes=(), activation='logistic', solver='adam', max_iter=1000,
random_state=42)
```

Figure 2.14: Import Libraries for MLP

```
file_path = 'model/mlp/MLP_model_'+mode+'.pkl'
with open(file_path, 'rb') as file:
    model = pickle.load(file)
```

Figure 2.15: Save MLP model

```
file_path = 'model/mlp/MLP_model_'+mode+'.pkl'
with open(file_path, 'rb') as file:
    model = pickle.load(file)
```

Figure 2.16: Load the MLP model

2.3.3 GRU

To implement the GRU model we brought the changes added some features in the utils script and made a model file. The sample of our codes are shown in figure 2.17, 2.18.

```
from tensorflow.keras.layers import GRU, Dense
```

Figure 2.17: Import Libraries for GRU

```
model = Sequential()
model.add(GRU(neurons, dropout=dropout, recurrent_dropout=dropout))
model.add(Dense(1, activation='sigmoid'))
```

Figure 2.18: GRU required code

Note: Due to the limitation in the number of pages we are not able to mention all details. We can share our overall Implementation through a repository

3 Results

The table 3.1 shows which types of vulnerabilities (XSS, Path Disclosure, Remote Code Execution, and Command Injection) are addressed in the LSTM model.

Model	XSS	Path Disclosure	Remote Code Execution	Command Injection
LSTM	✓	✓	✓	✓
CNN	✓		✓	
MLP	✓	✓		
GRU	✓			✓

Table 3.1: Machine Learning Models and Associated Vulnerabilities

3.0.1 LSTM

After successfully executing the demonstrat.py script we received the following result for all four vulnerabilities. In addition, in this report, we are demonstrating only the first test case due to the report's number of pages.

- **XSS:** Figure 3.1 shows the result.
- **Command Injection:** Figure 3.2 shows the result.

Figure 3.1: XSS Demonstration

Figure 3.2: Command Injection Demonstration

- **Remote Code Execution:** Figure 3.3 shows the result.

Figure 3.3: Remote Code Execution Demonstration

- **Path Disclosure:** Figure 3.4 shows the result.

Figure 3.4: Path Disclosure Demonstration

The table 3.2 shows our find out of in LSTM.

Vulnerability Type	Total Samples	% Vulnerable Samples	Absolute Vulnerable Samples	Accuracy	Precision	Recall	F1 Score
General	8277	8.96	742	0.91035	0.91839	0.91035	0.86763
Path Disclosure	19680	11.74	2311	0.88257	0.89636	0.88257	0.82752
Remote Code Execution	14412	9.08	1309	0.90917	0.91742	0.90917	0.86592
Command Injection	18814	12.54	2361	0.87451	0.89026	0.87451	0.81596

Table 3.2: Vulnerability Statistics and Model Performance Metrics

The visualization of the LSTM scores results are shown in chart 3.5.

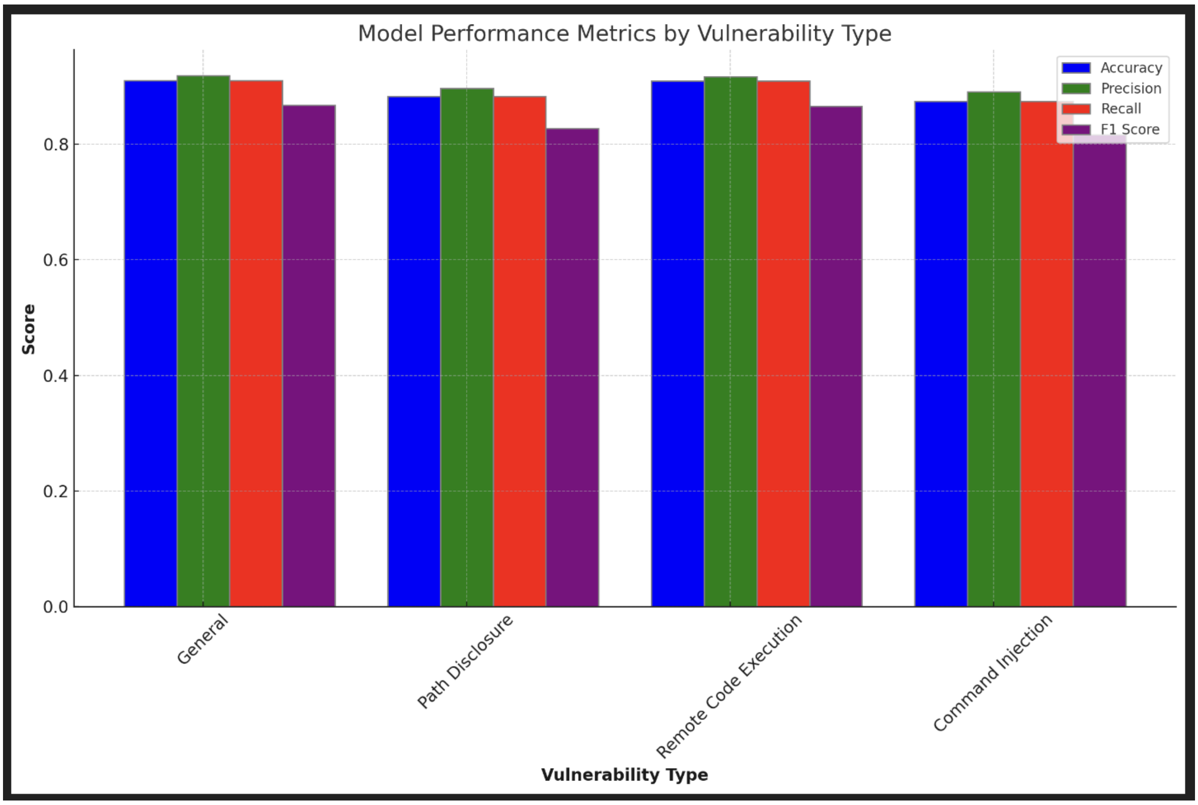


Figure 3.5: Model Performance Metrics by Vulnerability Type

In conclusion, the LSTM model shows good (Accuracy, Precision, Recall, F1Score, Performance consistency), Few false positives, and no false negatives.

3.0.2 CNN

The table 3.3 shows which types of vulnerabilities (XSS and Remote Code Execution) and their scores in CNN.

Vulnerability Type	Total Samples	% Vulnerable Samples	Absolute Vulnerable Samples	Accuracy	Precision	Recall	F1 Score
XSS	8277	8.34	690	0.82651	0.84052	0.82651	0.83341
Remote Code Execution	14412	9.81	1414	0.16847	0.86634	0.16847	0.15487

Table 3.3: Vulnerability Statistics and Model Performance Metrics for XSS and Remote Code Execution

Figure 3.6 shows the demonstrated result of the CNN model.

Figure 3.6: CNN Model Result

In conclusion, the CNN model shows more false positives and fewer false negatives. Table 3.4 demonstrates our conclusion for CNN.

Vulnerability Type	Accuracy	Precision	Recall	F1 Score
XSS	Good	Good	Good	Good
Remote Code Execution	Poor	Good	Poor	Poor

Table 3.4: Model Performance Metrics for Vulnerability Types in CNN

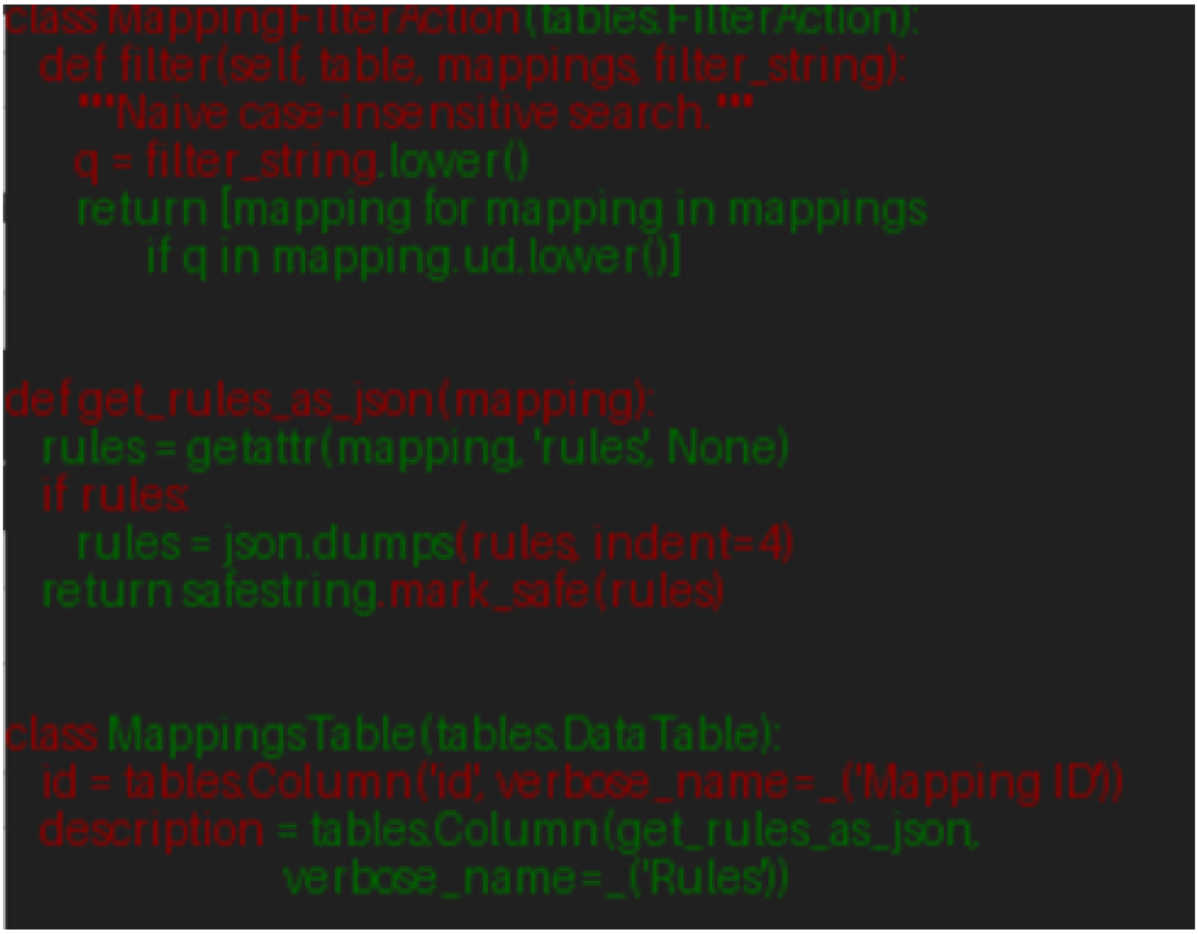
3.0.3 MLP

The table 3.5 shows which types of vulnerabilities (XSS and Path Disclosure) and their scores in MLP.

Vulnerability Type	Total Samples	% Vulnerable Samples	Absolute Vulnerable Samples	Accuracy	Precision	Recall	F1 Score
XSS	8277	8.88%	735	89.33%	86.47%	86.47%	87.69%
Path Disclosure	19680	11.22%	2210	83.56%	71.03%	71.03%	75.75%

Table 3.5: Vulnerability Statistics and Model Performance Metrics for XSS and Path Disclosure in MLP Modal

Figure 3.7 shows the demonstrated result of the MLP model.



```
class MappingFilterAction(tables.FilterAction):
  def filter(self, table, mappings, filter_string):
    """Naive case-insensitive search."""
    q = filter_string.lower()
    return [mapping for mapping in mappings
        if q in mapping.ud.lower()]


def get_rules_as_json(mapping):
  rules = getattr(mapping, 'rules', None)
  if rules:
    rules = json.dumps(rules, indent=4)
  return safestring.mark_safe(rules)


class MappingsTable(tables.DataTable):
  id = tables.Column('id', verbose_name=_('Mapping ID'))
  description = tables.Column(get_rules_as_json,
                verbose_name=_('Rules'))
```

Figure 3.7: MLP Model Demonstration

In conclusion, the MLP model shows few false positives and few false negatives. Table 3.6 demonstrates our conclusion for MLP.

Vulnerability Type	Accuracy	Precision	Recall	F1 Score
XSS	Good	Good	Good	Good
Path Disclosure	Poor	Good	Poor	Poor

Table 3.6: Model Performance Metrics for Vulnerability Types in MLP

3.0.4 GRU

The table 3.5 shows which types of vulnerabilities (XSS and Command Injection) and their scores in GRU.

Vulnerability Type	Total Samples	% Vulnerable Samples	Absolute Vulnerable Samples	Accuracy	Precision	Recall	
XSS	8277	8.61%	713	91.39%	92.13%	91.39%	87.27%
Command Injection	18814	12.73%	2396	87.26%	88.89%	87.26%	81.33%

Table 3.7: Vulnerability Statistics and Model Performance Metrics for XSS and Command Injection in MLP Modal

Figure 3.7 shows the demonstrated result of the MLP model.

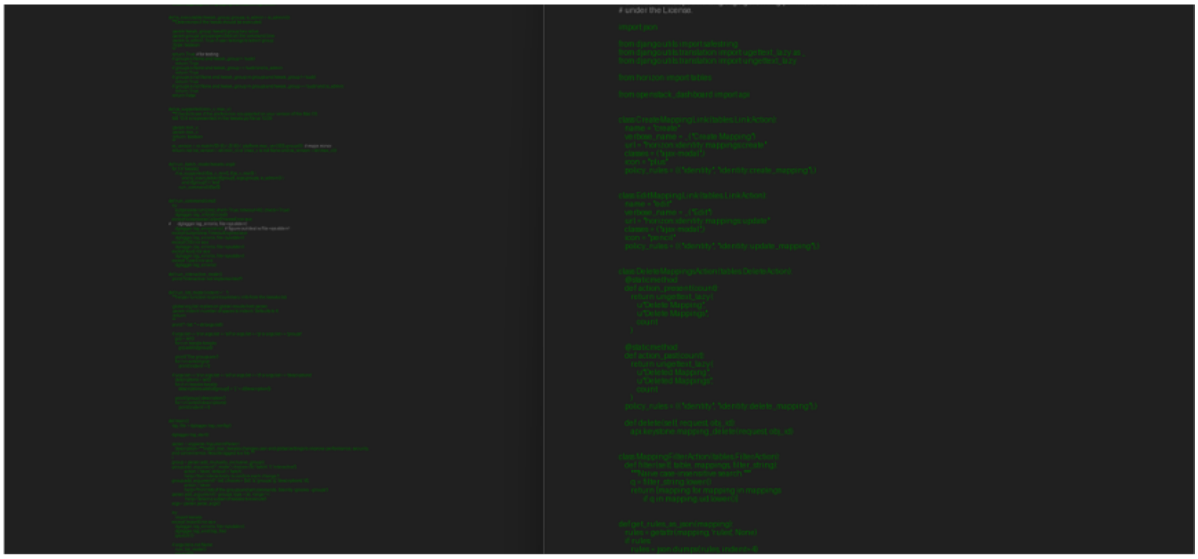


Figure 3.8: GRU Model Demonstration

We did not conclude GRU because the images were not working. Table 3.6 demonstrates our conclusion for GRU.

Vulnerability Type	Accuracy	Precision	Recall	F1 Score
XSS	Good	Good	Good	Good
Path Disclosure	Poor	Good	Good	Poor

Table 3.8: Model Performance Metrics for Vulnerability Types in MLP

3.0.5 Observations

After comparing all results we conclude that the LSTM model performance is better than the others. Table 3.9

Model	Vulnerability	Type	Samples	Vulnerable %	Vulnerable Samples	Accuracy	Precision	Recall	F1 Score
LSTM	XxSS		8277	8.96%	742	0.9104	0.9184	0.9104	0.8676
	Path Disclosure		19680	11.74%	2311	0.8826	0.8964	0.8826	0.8275
	Remote Code Execution		14412	9.08%	1309	0.9092	0.9174	0.9092	0.8659
MLP	Command Injection		18814	12.54%	2361	0.8745	0.8903	0.8745	0.8160
	XSS		8277	8.88%	735	0.8647	0.8933	0.8647	0.8769
	Path Disclosure		19680	11.22%	2210	0.7103	0.8356	0.7103	0.7575
CNN	XSS		8277	8.34%	690	0.8265	0.8405	0.8265	0.8334
	Remote Code Execution		14412	9.81%	1414	0.1685	0.8663	0.1685	0.1549
GRU	XSS		8277	8.61%	713	0.9139	0.9213	0.9139	0.8727
	Command Injection		18814	12.73%	2396	0.8726	0.8889	0.8726	0.8133

Table 3.9: All Models Performance Metrics for Different Vulnerabilities

The table 3.10 shows our observation on average from gathered all results.

Model	Accuracy (Avg)	Precision (Avg)	Recall (Avg)	F1 Score (Avg)	Observations
LSTM	0.8941	0.9056	0.8941	0.8446	Consistently high accuracy, precision, and recall across all types of vulnerabilities.
MLP	0.7875	0.8645	0.7875	0.8172	Strong performance for XSS, but significantly lower for Path Disclosure.
CNN	0.4975	0.8534	0.4975	0.4942	Decent performance for XSS but very poor for Remote Code Execution.
GRU	0.8933	0.9051	0.8933	0.8430	High performance for XSS and Command Injection, comparable to LSTM.

Table 3.10: Performance Metrics and Observations for Different Models