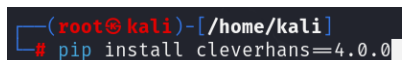


Penetration Testing on Intrusion Detection Systems

We will implement a machine learning model to classify network traffic data into attack types (e.g., DoS, probe, U2R, etc.) using a Multi-Layer Perceptron (MLP). It will preprocess the KDD Cup dataset, train the model, evaluate its performance, and generate adversarial samples to test model (IDS) robustness against attacks.

The program will require the latest version of cleverhans library. Install it using the command:

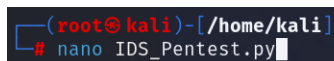
```
$ pip install cleverhans==4.0.0
```



```
(root@kali)-[/home/kali]  
# pip install cleverhans==4.0.0
```

Open required file in nano editor using the command:

```
$ nano IDS_Pentest.py
```



```
(root@kali)-[/home/kali]  
# nano IDS_Pentest.py
```

Download the 2 datasets. They can be found here

<https://github.com/avnishnaithani/pythonforsecurity/blob/main/Domain%20KDDTrain%2B.txt>

<https://github.com/avnishnaithani/pythonforsecurity/blob/main/Domain%20KDDTest%2B.txt>

The code can be copied from

https://github.com/avnishnaithani/pythonforsecurity/blob/main/Domain%20IDS_Pentest.py

Step 1: Import necessary libraries

```

root@kali: /home/kali
File Actions Edit View Help
GNU nano 8.1 IDS_Pentest.py
import tensorflow as tf
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
import pandas as pd
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from cleverhans.tf2.attacks import fast_gradient_method

```

TensorFlow is used for building and training the machine learning model.

Scikit-learn provides metrics for evaluating the model.

Pandas and **NumPy** are used for data manipulation and numerical operations.

CleverHans is a library for adversarial machine learning attacks.

Step 2: Data Loading and Preparation

```

# Data loading and preparation
names = ['duration', 'protocol', 'service', 'flag', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment',
         'num_failed_logins', 'logged_in', 'num_compromised', 'root_shell', 'su_attempted', 'num_root',
         'num_file_creations', 'num_shells', 'num_access_files', 'num_outbound_cmds', 'is_host_login',
         'is_guest_login', 'count', 'srv_count', 'error_rate', 'srv_error_rate', 'rerror_rate',
         'srv_rerror_rate', 'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count',
         'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src',
         'dst_host_srv_diff_host_rate', 'dst_host_error_rate', 'dst_host_srv_error_rate', 'dst_host',
         'dst_host_srv_rerror_rate', 'attack_type', 'other']

df = pd.read_csv('KDDTrain+.txt', names=names, header=None)
dft = pd.read_csv('KDDTest+.txt', names=names, header=None)

```

The KDD Cup 1999 dataset, a common dataset for IDS research, is loaded.

The dataset contains various features related to network traffic.

Step 3: Combining Training and Test Data

```

# Combine the data
full = pd.concat([df, dft])
assert full.shape[0] == df.shape[0] + dft.shape[0]
print("Initial test and training data shapes:", df.shape, dft.shape)

```

The training and testing datasets are combined into a single DataFrame for processing.

This ensures that data integrity is maintained.

Step 4: Label Handling

```
# Check if the 'label' column exists in the full DataFrame
if 'label' not in full.columns:
    # Rename or create the label column if necessary, for example, if 'attack_type' is used
    full['label'] = full['attack_type']
```

If a 'label' column does not exist, it creates one based on 'attack_type'.

Labels are reclassified into broader categories (e.g., 'dos', 'u2r', etc.) for simplicity.

Step 5: Data Preprocessing

```
# Check if the 'label' column exists in the full DataFrame
if 'label' not in full.columns:
    # Rename or create the label column if necessary, for example, if 'attack_type' is used
    full['label'] = full['attack_type']

# Reclassify labels into broader categories
dos_labels = ['neptune', 'back', 'land', 'pod', 'smurf', 'teardrop', 'mailbomb', 'processtable', 'udps']
u2r_labels = ['buffer_overflow', 'loadmodule', 'perl', 'rootkit', 'sqlattack', 'xterm', 'ps']
r2l_labels = ['ftp_write', 'guess_passwd', 'imap', 'multihop', 'phf', 'spy', 'warezclient', 'warezmast',
              'snmpgetattack', 'httptunnel', 'snmpguess', 'sendmail', 'named']
probe_labels = ['satan', 'ipsweep', 'nmap', 'portsweep', 'saint', 'mscan']

# Reclassify labels into broader categories
full.loc[full['attack_type'].isin(dos_labels), 'label'] = 'dos'
full.loc[full['attack_type'].isin(u2r_labels), 'label'] = 'u2r'
full.loc[full['attack_type'].isin(r2l_labels), 'label'] = 'r2l'
full.loc[full['attack_type'].isin(probe_labels), 'label'] = 'probe'

# Drop unnecessary columns
full = full.drop(['other', 'attack_type'], axis=1)
print("Unique labels", full['label'].unique())

# Convert categorical columns to dummy variables
full2 = pd.get_dummies(full, drop_first=False)
features = list(full2.columns[:-5])

y_train = np.array(full2[0:df.shape[0]][['label_normal', 'label_dos', 'label_probe', 'label_r2l', 'label_u2r']])
X_train = full2[0:df.shape[0]][features]
y_test = np.array(full2[df.shape[0]:][['label_normal', 'label_dos', 'label_probe', 'label_r2l', 'label_u2r']])
X_test = full2[df.shape[0]:][features]

# Normalize the data
scaler = MinMaxScaler().fit(X_train)
X_train_scaled = np.array(scaler.transform(X_train))
X_test_scaled = np.array(scaler.transform(X_test))
```

Categorical features are converted into dummy variables to prepare the data for the model.

Data is normalized using MinMaxScaler to scale features to a range between 0 and 1.

Step 6: Model Definition

```
# Define the MLP model
def mlp_model():
    model = Sequential()
    model.add(Dense(256, activation='relu', input_shape=(X_train_scaled.shape[1],)))
    model.add(Dropout(0.4))
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(5, activation='softmax'))
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    model.summary()
    return model
```

A multi-layer perceptron (MLP) model is defined using Keras, which includes layers for input, dropout for regularization, and output.

The model is compiled with an Adam optimizer and categorical cross-entropy loss function, which is appropriate for multi-class classification.

Step 7: Model Training

```
# Train the MLP model
model = mlp_model()
model.fit(X_train_scaled, y_train, batch_size=128, epochs=10, validation_data=(X_test_scaled, y_test))
```

The model is trained using the training data, with validation on the test data to monitor its performance.

Step 8: Model Evaluation

```
# Evaluate the model
def evaluate_model():
    loss, accuracy = model.evaluate(X_test_scaled, y_test, batch_size=128)
    print(f'Test accuracy on legitimate test examples: {accuracy}')

evaluate_model()
```

The model's performance is evaluated on the test set, reporting accuracy.

Step 9: Adversarial Attacks

```
def create_adversarial_samples(model, x_test):
    # Get the logits (raw predictions) from the model
    logits = model(x_test)
    # Use fast_gradient_method to generate adversarial examples
    adv_x = fast_gradient_method(logits, x_test, epsilon=0.3, norm=np.inf)
    return adv_x

# Example of using adversarial attacks
adv_x_test = create_adversarial_samples(model, X_test_scaled)
```

Adversarial attacks involve creating inputs specifically designed to mislead machine learning models. In the context of intrusion detection systems (IDS), this means generating network traffic data that could confuse the model, causing it to misclassify benign traffic as malicious or vice versa.

The function `create_adversarial_samples` is defined as follows:

- **model**: The trained machine learning model.
- **x_test**: The test dataset containing normal and malicious examples.
- The method used here is called **Fast Gradient Method**, a popular technique for generating adversarial examples. The main idea behind FGM is to perturb the input data in the direction of the gradient of the loss with respect to the input, thus maximizing the loss and misclassifying the example.
- **Epsilon (ϵ)**: The epsilon parameter controls the magnitude of the perturbation. In this case, 0.3 indicates how much noise will be added to the input features. A higher epsilon means more aggressive modifications.
- **Norm**: The `norm=np.inf` parameter indicates the use of the infinity norm (L-infinity norm), which means that the perturbation can affect any individual feature by up to 0.3, but the total change will still be constrained within that limit.

Purpose of Adversarial Attacks

1. **Testing Robustness:** The primary goal of generating adversarial samples is to evaluate how robust the model is against adversarial inputs. This is essential for understanding the vulnerabilities of the model.
2. **Improving Security Posture:** By identifying how the model responds to adversarial examples, developers can take corrective actions to strengthen the model against potential real-world attacks.
3. **Understanding Attack Vectors:** Adversarial attacks simulate the techniques attackers might use to compromise the model's integrity. Understanding these vectors helps security professionals devise better defenses.

Step 10: Evaluation on Adversarial Samples

```
# Optionally evaluate on adversarial samples
adv_y_pred = model.predict(adv_x_test)
adv_accuracy = accuracy_score(np.argmax(y_test, axis=1), np.argmax(adv_y_pred, axis=1))
print(f'Accuracy on adversarial samples: {adv_accuracy}')
```

Once adversarial examples are generated, the next step is to evaluate the model's performance on these perturbed inputs. This helps in assessing the impact of adversarial attacks on the model's decision-making capabilities.

- The function `create_adversarial_samples` is called with the trained model and the scaled test dataset (`X_test_scaled`). This returns `adv_x_test`, the set of adversarial examples.
- The model predicts the classes of these adversarial examples using `model.predict(adv_x_test)`. This generates the predicted labels for the adversarial inputs.
- The accuracy of the model on the adversarial examples is calculated using `accuracy_score`. The predicted labels are compared to the true labels (`y_test`), allowing us to see how well the model can classify the perturbed inputs.

- `np.argmax` is used to convert the one-hot encoded true labels and predicted outputs into single class labels for comparison.

Purpose of Evaluating Adversarial Samples

1. **Understanding Model Weaknesses:** The resulting accuracy indicates how well the model can withstand adversarial attacks. A significant drop in accuracy suggests that the model is vulnerable and can be easily fooled by adversarial inputs.
2. **Enhancing Model Robustness:** By analyzing which types of attacks most severely impact accuracy, developers can identify weaknesses in the model and implement strategies to improve robustness, such as adversarial training (training the model on adversarial examples) or feature engineering to make the model less susceptible to such attacks.
3. **Improving Intrusion Detection Systems:** In the context of penetration testing and cybersecurity, understanding how well an IDS can handle adversarial inputs directly correlates to its effectiveness in a real-world scenario where attackers might exploit vulnerabilities. This leads to better-designed security measures and more effective incident response strategies.
4. **Informed Security Policies:** Evaluating performance against adversarial examples can inform broader security policies and practices within an organization, helping to ensure that machine learning systems are not only accurate but also resilient against potential exploitation.

Now Save the file after completing the code. (“CTRL+x “-> “y” -> “Enter”)

Change the file to an executable one using the command:

```
$ chmod +x IDS_Pentest.py
```

```
(root@kali)-[/home/kali]
# chmod +x IDS_Pentest.py
```

Finally run the program using:

```
$ python3 IDS_Pentest.py
```

```
(root@kali)-[/home/kali]
# python3 IDS_Pentest.py
```

OUTPUT

```
Initial test and training data shapes: (125973, 43) (22544, 43)
Unique labels ['normal' 'dos' 'r2l' 'probe' 'u2r']
Training dataset shape (125973, 122) (125973, 5)
Test dataset shape (22544, 122) (22544, 5)
Label encoder y shape (125973,) (22544,)
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an
n using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	31,488
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 256)	65,792
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 5)	1,285

```

Total params: 98,565 (385.02 KB)
Trainable params: 98,565 (385.02 KB)
Non-trainable params: 0 (0.00 B)
Epoch 1/10
985/985 — 3s 3ms/step - accuracy: 0.9492 - loss: 0.1814 - val_accuracy: 0.7546 - val
Epoch 2/10
985/985 — 5s 3ms/step - accuracy: 0.9863 - loss: 0.0397 - val_accuracy: 0.7656 - val
Epoch 3/10
985/985 — 2s 2ms/step - accuracy: 0.9899 - loss: 0.0302 - val_accuracy: 0.7756 - val
Epoch 4/10
985/985 — 3s 3ms/step - accuracy: 0.9916 - loss: 0.0240 - val_accuracy: 0.7742 - val
Epoch 5/10
985/985 — 2s 2ms/step - accuracy: 0.9927 - loss: 0.0206 - val_accuracy: 0.7830 - val
Epoch 6/10
985/985 — 2s 2ms/step - accuracy: 0.9928 - loss: 0.0211 - val_accuracy: 0.7850 - val
Epoch 7/10
985/985 — 2s 2ms/step - accuracy: 0.9936 - loss: 0.0194 - val_accuracy: 0.7734 - val
Epoch 8/10
985/985 — 2s 2ms/step - accuracy: 0.9935 - loss: 0.0178 - val_accuracy: 0.7762 - val
Epoch 9/10
985/985 — 2s 2ms/step - accuracy: 0.9938 - loss: 0.0180 - val_accuracy: 0.7849 - val
Epoch 10/10
985/985 — 2s 2ms/step - accuracy: 0.9946 - loss: 0.0154 - val_accuracy: 0.7664 - val
177/177 — 0s 1ms/step - accuracy: 0.7715 - loss: 2.7211
Test accuracy on legitimate test examples: 0.7663679718971252
```


Initial test and training data shapes: This indicates that the training dataset contains **125,973** instances, while the test dataset has **22,544** instances. Understanding data size helps gauge model training effectiveness and performance evaluation.

Unique labels: The unique labels provide insights into the classes the model is trained to detect, which are crucial for identifying various attack types.

Training and Test dataset shapes: The feature dimensions (122 features) and output dimensions (5 labels) confirm that the data is properly processed and ready for model training.

Model summary: This shows the architecture of the MLP model, including the total number of parameters, indicating the model's complexity. Understanding the architecture helps in optimizing and debugging the model's performance.

This output is typically repeated for each epoch (in our case, for a total of 10 epochs). Each step provides insights into:

- The model's learning progress over time.
- How well the model is fitting the training data (accuracy and loss).
- How well it generalizes to the validation data (validation accuracy and loss).

As training progresses over the epochs, you would ideally want to see:

- Increasing training accuracy and decreasing training loss.

- Validation accuracy increasing (or stabilizing) and validation loss decreasing. If validation accuracy starts to decrease while training accuracy continues to increase, it might indicate overfitting, suggesting that the model is learning noise in the training data rather than generalizable patterns.

The model achieved a test accuracy of about **76.6%** on legitimate test examples, indicating how well the model can identify non-attack traffic.

Monitoring these metrics is crucial for understanding the performance of the model and making decisions about further training, model adjustments, or early stopping.

The above code and its outputs can significantly aid penetration testing by:

Identifying Model Vulnerabilities: By generating adversarial examples and evaluating model performance against them, penetration testers can identify weaknesses in the model that could be exploited in real-world attacks.

Improving Security Measures: Insights gained from adversarial evaluations can help developers fine-tune the model, implement adversarial training techniques, or adjust preprocessing steps to enhance robustness.

Simulating Real-World Attacks: The approach illustrates how attackers might craft inputs designed to mislead the model, allowing security professionals to understand potential attack vectors and develop defensive strategies.

Enhancing Incident Response: The knowledge of model limitations allows teams to devise incident response strategies for when attacks

are successful or when the model fails to accurately classify input data.

In conclusion, this code serves as both a machine learning model for intrusion detection and a framework for evaluating the model's resilience to adversarial attacks, which is a critical aspect of penetration testing in the context of network security.