nmap-harvester

A supervised machine learning model designed for detecting `NMAP` port scanning, developed as part of a university project.

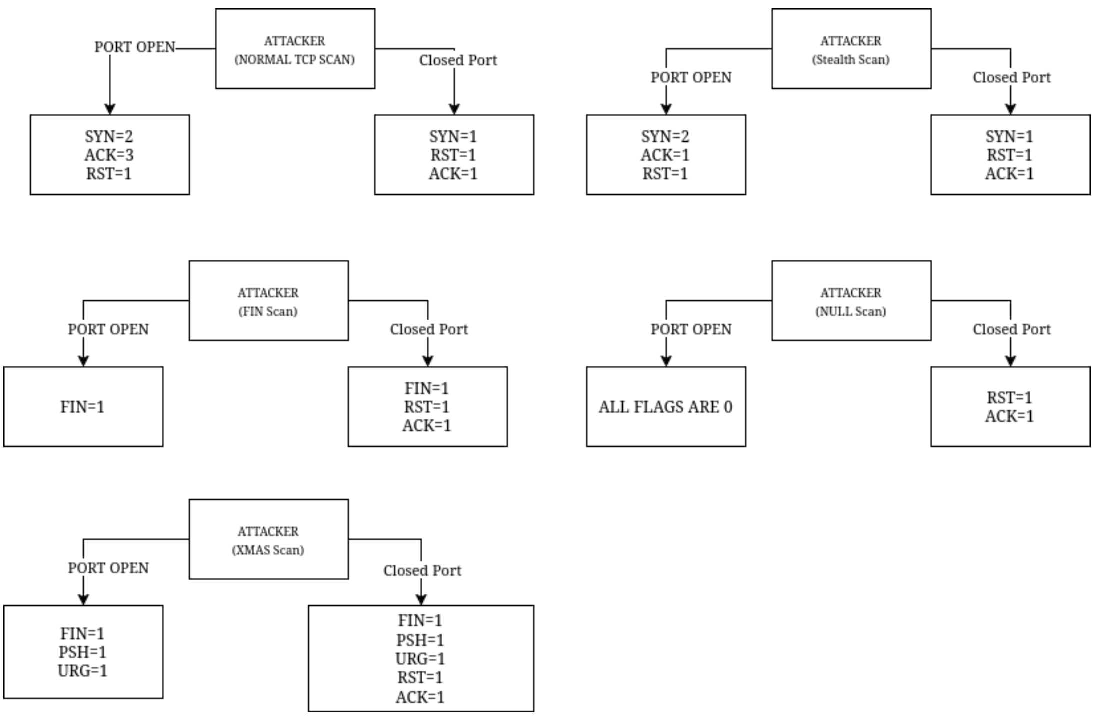
Introduction

This project aims to build a supervised machine learning model to detect real-time `NMAP` port scanning activities.

In many cyber-attacks, the initial step often involves port scanning using tools like `NMAP`. Detecting such scans can be challenging because network packets carry extensive information, and a single packet isn't enough to confirm an `NMAP` scan attempt.

To address this, this project proposes a machine learning-based approach for identifying TCP port scans initiated by `NMAP`.

How the `interceptor.py` collects all TCP flags can be summered in the following scheme.



All this attack data are collected by the `interceptor.py` which listens for coming connection on a specific ip address.

In the project the following files have the described functions:

- `interceptor.py` -> monitor for collecting data
- `algo_chooser.py` -> script for choosing the best machine learning algorithm
- `injector.py` -> injects nmap scans or run normal http requests
- `noiser.py` -> helper script for sending normal http requests
- `detector.py` -> runs a real-time demo of the model using previously mentioned scripts internally
- `model` directory -> contains exported model
- `merger.py` -> merges 2 datasets in one single dataset

- `datasets/delayed/merged.csv` -> contains just another dataset for calculating accuracy and other stats
- `datasets/train/merged.csv` -> contains the datasets used for training the model
- `datasets/runtime` -> contains generated runtime datasets when running `detector.py`

> The entire software is designed for running on a Linux environment and although the only changes needed for running on another operating system are interface names, some checks may fail due to different environments.
>
> For example: `detector.py` (https://github.com/Virgula0/nmap-harvester/blob/main/detector.py#L60) contains a check to run with `root` privileges and this check should be commented if the environment is different from Unix.
>
> This disclaimer has been inserted because I noticed the usage of other operating systems during the course lectures. Running the `detector.py` on a virtual environment should work because not everything is set to run on the `localhost` interface, but even here, the creation of the dataset phases may not work because of interfaces used by pyshark on a virtual machine. A solution can be to create another container and let the 2 isolated container to communicate between them, but this is up to the readers to investigate better eventually.

Train Dataset Creation `datasets/train/merged.csv`

Understanding TCP connections is important for building the dataset. When a TCP packet is sent over the network, it carries specific flags that facilitate the `3-Way Handshake`. `NMAP` can manipulate these flags to evade detection while performing rapid port scans. We cannot create a dataset where each row of the dataset represents a packet since for detecting `NMAP`, multiple packets are needed, so the idea:

Key Dataset Characteristics

- **Session-Based Rows:** Instead of logging each packet individually, each row in the dataset represents a **session** (requests + responses).
- **Flag Summation:** Flags (`SYN`, `ACK`, `FIN`, etc.) are aggregated across the same session. For example:
  - If a `SYN` packet is sent by `NMAP` and another `SYN` is sent by the host in the response (whatever the port is closed or opened), the `SYN` column will record a value of `2` because of their sum.
  - The dataset contains 6 (`SYN`, `ACK`, `RST`, `FIN`, `URG`, `PSH`) out of 9 TCP flags. NMAP uses those flags, but in the case of other new attacks, the dataset can be rebuilt using other TCP flags, too.
- **Duration feature**: `start_response_time`, and `end_response_time` will be set to 0 if only a packet has been found in the entire session. In this case, the duration will be only `end_request_time` - `start_request_time` otherwise the session duration is `end_response_time` - `start_request_time`

Example Dataset Row (normal tcp scan on a closed port 3306)

```
start_request_time,end_request_time,start_response_time,end_respons
2025-01-08 13:52:55.274814,2025-01-08 13:52:55.274814,2025-01-08
13:52:55.274874,2025-01-08 13:52:55.274874,6e-05,"['172.31.0.1',
'172.31.0.2']","['172.31.0.1', '172.31.0.2']","['44031',
'3306']","['44031', '3306']",1,1,0,1,0,0,1
```

- Sessions are grouped using `src_ip`, `dst_ip`, `src_port`, and `dst_port` tuple as keys. However, these grouping keys are excluded and not necessary from the model's training phase.
- The `duration` feature provides valuable information for distinguishing between legitimate traffic and `NMAP` scans, as legitimate HTTP requests may exhibit similar flag behaviour but differ in timing.
- The session window in `interceptor.py` is set to **0.5 seconds** by default, as this is typically enough to capture an `NMAP` scan attempt.

More technical explanations are present via comments in `interceptor.py`. The script takes a while for writing succesfully all the data when a lot of requests are performed.

> During the data collection, some ports were opened intentionally on the host to differentiate some rows in the dataset. For example, an HTTP server on port 1234 has been opened using the following method: `python3 -m http.server 1234` plus, eventually other ports that had already been opened from other services between the range 0-5000.

Common `NMAP` Scans

The following commands were run from the container called `traffic_generator` (the container) having the `sudo python3 interceptor.py` running locally.

```
nmap -sT 172.31.0.1 -p 0-5000 # TCP Scan
nmap -sS 172.31.0.1 -p 0-5000 # Stealth Scan
nmap -sF 172.31.0.1 -p 0-5000 # FIN Scan
nmap -sN 172.31.0.1 -p 0-5000 # NULL Scan
nmap -sX 172.31.0.1 -p 0-5000 # XMAS Scan
```

The result is the creation of `bad.csv`

Then the script `noiser.py` was used for generating `good.csv`

The final dataset consists of a merge (`merged.csv`) used for training the model:

- `bad.csv`: Sessions labelled as `1` (`NMAP` traffic).
- `good.csv`: Sessions labelled as `0` (legitimate traffic).

Machine Learning Model

The `XGBClassifier` was selected as the final model due to its reliable performance in key areas:

1. High accuracy score (`~0.95`)
2. Fast prediction speed (`~3ms` on average for 15,000 rows)
3. High MCC score (`~0.91`)

Why accuracy metric is important?

The dataset generated for training purposes contains a balanced example of normal/anomaly behaviours, half normal and half anomalies, which get shuffled during the dataset splitting phase. Thus, accuracy is an important statistic that can be considered in this case.

Why MCC is not that important?

MCC should normally be preferred when unbalanced datasets are present. This is not our case, but it is taken into account even if it has a minor weight in the final machine-learning model choice.

Why the prediction speed is so important?

The prediction speed played a significant role in choosing this model, as it allows efficient analysis of large volumes of network traffic in real-time. The `RandomForestClassifier` is pretty similar in accuracy (maybe even better sometimes), but it has a slower prediction time in average of `~15ms` compared to `~3ms` of `XGBClassifier`

Model Performance

```
Dataset loaded with 15192 records.
Dataset preprocessed successfully.

+------------+-----+-----+-----+-----+-----+-----+-------+
|  duration  | SYN | ACK | FIN | RST | URG | PSH | label |
+------------+-----+-----+-----+-----+-----+-----+-------+
|  0.000030  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000013  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000012  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000010  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000011  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
+------------+-----+-----+-----+-----+-----+-----+-------+

Dataset split into training and testing sets.

RandomForestClassifier (n_estimators=210): Accuracy: 0.9513, Train
time: 432ms, Prediction time: 16ms, MCC: 0.902033, TP: 661, TN:
785, FN: 33, FP: 41

....

XGBClassifier (n_estimators=210): Accuracy: 0.9507, Train time:
65ms, Prediction time: 3ms, MCC: 0.919739, TP: 660, TN: 785, FN:
34, FP: 41

Best Classifier based on Accuracy
Classifier: RandomForestClassifier
n_estimators: 210
Accuracy Score: 0.9513

Best Classifier based on MCC
Classifier: XGBClassifier
n_estimators: 210
MCC Score: 0.919739
```

How Training Dataset was created (detailed)

The training dataset, `datasets/train/merged.csv`, is generated using the following steps:

1. Create an isolated Docker environment for sending clean packets:

   ```
   docker compose up --build
   ```

2. Access the container:

   ```
   docker attach traffic_generator
   ```

3. Run the interceptor on the host from another terminal:

   ```
   sudo python3 interceptor.py
   ```

   - Before to run, adjust:
     - `interface` : Docker network interface name
     - `scanner_ip` : IP assigned to `traffic_generator`

- `output_file` : Output CSV file path
- `label` : `0` for legitimate traffic, `1` for `NMAP` scans

4. Run `NMAP` scans from the container:

```
nmap -sT 172.31.0.1 -p 0-5000
nmap -sS 172.31.0.1 -p 0-5000
nmap -sF 172.31.0.1 -p 0-5000
nmap -sN 172.31.0.1 -p 0-5000
nmap -sX 172.31.0.1 -p 0-5000
```

5. Run noise traffic for legitimate requests, from the container:

```
python3 noiser.py
```

6. Merge datasets:

```
cd datasets
python3 merger.py
```

7. Train the model:

```
python3 algo_chooser.py
```

Delayed Dataset

A delayed dataset can be created by introducing delays between requests:

```
nmap -p 1-10000 --scan-delay 1s 172.31.0.1
```

You can also adjust the delay in legitimate requests by modifying `SLEEP_SECOND` in `noiser.py`.

Results:

```
Dataset loaded with 11351
records.
Dataset preprocessed
successfully.
+------------+-----+-----+-----+-----+-----+-----+-------+
|  duration  | SYN | ACK | FIN | RST | URG | PSH | label |
+------------+-----+-----+-----+-----+-----+-----+-------+
|  0.000060  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000068  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000062  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000057  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
|  0.000074  |  1  |  1  |  0  |  1  |  0  |  0  |   1   |
+------------+-----+-----+-----+-----+-----+-----+-------+

Dataset split into training and testing sets.

....
XGBClassifier (n_estimators=210): Accuracy: 1.0000, Train time:
28ms, Prediction time: 3ms, MCC: 1.000000, TP: 743, TN: 393, FN:
0, FP: 0

Best Classifier based on MCC
Classifier: XGBClassifier
n_estimators: 210
MCC Score: 1.000000
```

Running the Detector

To run the detector:

```
sudo python3 detector.py
```

- The detector uses `interceptor.py` to monitor session packets.
- `injector.py` simulates normal HTTP traffic with occasional `NMAP` scans (10% probability).
- If at least **30%** of session packets are flagged as anomalies, the system will detect an ongoing `NMAP` attack.

Requirements

Install dependencies with:

```
python3 -m venv venv && source venv/bin/activate && pip install -r requirements.txt
```

Demonstration Video

https://github.com/user-attachments/assets/f10773c6-742e-4394-913e-42beb0cc3683

References

- Medium Article on `NMAP` Detection
- Unix Stack Exchange - Detecting `NMAP` Scans

External Dependencies

- `pyshark`
- `python-nmap`