N.A.T.A.S.H.A.

DOCUMENTATION FOR ML #1 INTEGRATION

BNB DEVELOPERS
Pune

Table of Contents

1. Introduction

Purpose of the Document

The purpose of this document is to serve as a comprehensive guide for understanding, setting up, and maintaining a Raspberry Pi-based network link scanner and malicious link blocker with Flask API and machine learning capabilities. It aims to provide users with clear instructions and insights into the project's architecture, functionality, and usage.

Project Overview

This project combines Raspberry Pi hardware with Flask, Pi-hole, and machine learning to enhance network security. It scans network domain links, classifies them as malicious or non-malicious using a machine learning model, and blocks malicious links with Pi-hole. Additionally, it offers a whitelisting feature to exclude specific links from blocking, providing a robust solution for managing network traffic and security.

2. Project Description

Objectives

The main objectives of this project are as follows:

- Scanning Links: Scan all domain links within the network to identify potentially malicious URLs.

- Link Classification: Utilize machine learning to classify scanned links as either malicious or non-malicious.

- Malicious Link Blocking: Implement Pi-hole commands to block access to identified malicious links.

- Whitelisting: Enable users to whitelist specific links that should not be blocked.

Key Features

The project's key features include:

- Flask API: A user-friendly API for interacting with the system, allowing users to initiate scans, manage whitelists, and receive status updates.

- Machine Learning Model: Utilizes a machine learning model to determine link safety, enhancing accuracy over rule-based methods.

- Pi-hole Integration: Integrates with Pi-hole, a popular network-level ad and content blocker, for efficient and network-wide link blocking.

- Whitelisting Capability: Provides users with the ability to whitelist links that are incorrectly flagged as malicious, ensuring uninterrupted access to essential resources.

Target Audience

This documentation is intended for:

- System Administrators: Responsible for maintaining and securing network infrastructure.

- Network Administrators: Involved in managing and optimizing network traffic.

- Developers: Interested in extending or customizing the functionality of the system.

3. Hardware and Software Requirements

Hardware Requirements

To implement this project, you will need:

- Raspberry Pi: Select a suitable Raspberry Pi model with sufficient processing power and RAM to accommodate the project's requirements.

- MicroSD Card: Choose a microSD card with ample storage capacity to host the Raspberry Pi OS and project files.

- Power Supply: Ensure you have a compatible power supply to provide stable power to the Raspberry Pi.

- Peripherals: You may need a USB keyboard, mouse, HDMI cable, and monitor for initial setup or can use PUTTY instead.

- Ethernet Cable: An Ethernet cable may be required for a network connection if using a model without built-in Wi-Fi.

Software Requirements

The software requirements include:

- Raspberry Pi OS: Install the Raspberry Pi OS (previously known as Raspbian) on the microSD card.

- Python: Ensure Python is installed on the Raspberry Pi, preferably Python3.

- Flask: Install Flask to create the API.

- Pi-hole: Set up and configure Pi-hole on your Raspberry Pi.

- Required Python Libraries: List any additional Python libraries needed for the project, such as scikit-learn for machine learning.

4. System Architecture

Overview of System Components

The project's architecture comprises several key components, including:

- Raspberry Pi: The central hardware platform responsible for hosting the project.

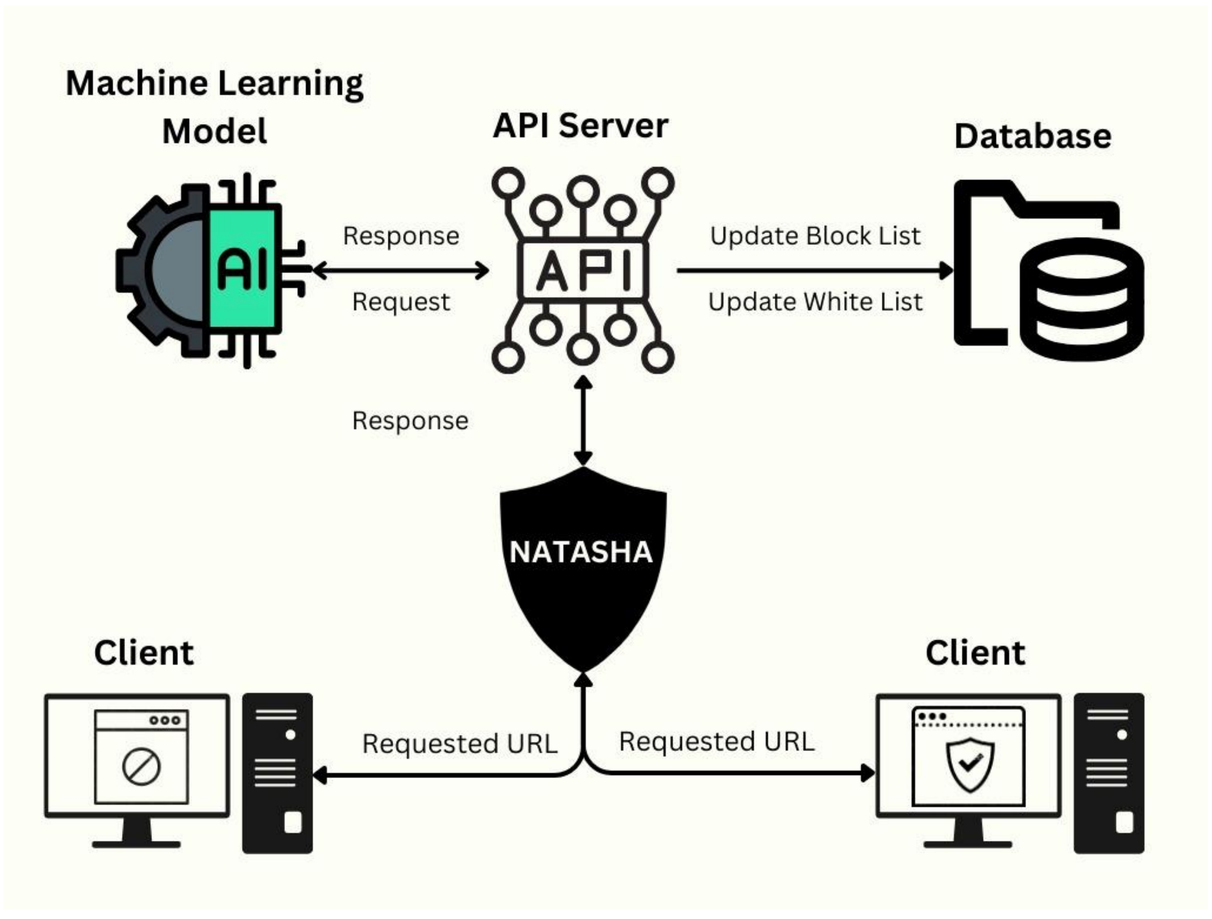
- Flask API: A web-based API built using Flask that allows users to interact with the system.

- Machine Learning Model: A trained machine learning model used to classify links.

- Pi-hole: A network-level ad and content blocker that is integrated into the system for link blocking.

How Components Interact

Explain how data flows between these components. For example, users interact with the Flask API to initiate scans and manage whitelists. The Flask API communicates with the machine learning model to classify links and sends blocking commands to Pi-hole based on the classification results.



5. Setup and Installation

Raspberry Pi Setup

Provide detailed instructions on how to set up the Raspberry Pi, including hardware assembly, OS installation, and initial system configuration. This might involve:

- Inserting the microSD card with the OS image.

- Connecting peripherals and power.

- Booting the Raspberry Pi.

- Configuring basic system settings, such as language, time zone, and Wi-Fi (if applicable).

- Install required Packages and libraries

Requests, scapy, os, subprocess, pickle, warnings, Flask, jsonify

Configuration of Flask API and Pi-hole

1. Install Flask:
    - Install Flask using pip: pip install Flask
2. Create a Flask App:
    - Create a Python script for your Flask app (e.g., app.py).
3. Import Necessary Modules:
    - Import Flask and any other required modules in your Python script.
4. Initialize Flask:
    - Create a Flask app instance: app = Flask(__name__).
5. Define Routes:
    - Define routes for your API, such as /status, /enable, /disable, etc.
6. Interact with Pi-hole API:
    - Use the requests library to send HTTP requests to your Pi-hole (Refer page No. 7)
7. Process Requests:
    - Extract data from the incoming HTTP requests (e.g., parameters or JSON data).
8. Perform Pi-hole Actions:
    - Use the Pi-hole API to enable, disable, or retrieve status information about Pi-hole.

9. Return Responses:
   - Send appropriate responses back to the client, typically in JSON format.
10. Run the Flask App:
    - Add a section to run the Flask app using app.run() with appropriate host and port settings.
11. Test Your API:
    - Test your Flask API endpoints using tools like curl, Postman, or a web browser.
12. Secure Your API (Optional):
    - Consider implementing authentication and authorization mechanisms to secure your Flask API.
13. Deploy Your Flask API (Optional):
    - Deploy your Flask API on a production server for external access.

6. Functionality

Flask API

- List and explain the available API endpoints, such as `/predict`, etc.

- The /predict endpoint gives result in value of 0 or 1 for link to be Benign or malicious respectively.

- examples of API requests and responses for endpoint, showing the expected format and parameters.

```
{
        "url":"exampleURL.com"
}
```

Domain Link Scanning

```
iface = "etn0"

sniff(filter="udp port 53",prn=dns_callback,iface=iface)
```

Network Traffic Capture: The sniff function from Scapy is used to capture network packets that match the specified filter criteria. In this case, the filter "udp port 53" is used to capture UDP packets on port 53, which is the standard port for DNS traffic.

Filtering DNS Queries: The filter "udp port 53" ensures that only DNS traffic is captured. DNS queries, which are used to resolve domain names to IP addresses, are transmitted using the UDP protocol on port 53. This filter effectively narrows down the captured packets to DNS queries.

Callback Function: The prn parameter specifies a callback function (dns_callback) that is invoked for each captured DNS packet. The callback function processes the captured packets, extracting relevant information such as the domain names being queried.

Processing Captured Packets: Within the dns_callback function, you can access and process the DNS packets. Scapy provides functions to extract information from these packets, including the queried domain name.

Malicious link blocking and whitelisting with Pi-hole

Blacklisting:

1. Command: pihole -b, pihole -b --help
2. Functionality: Allows administrators to manually block specific domains, preventing them from being accessed through the Pi-hole DNS server.
3. Example: pihole -b advertiser.example.com

Whitelisting:

1. Command: pihole -w, pihole -w --help
2. Functionality: Permits administrators to manually allow specific domains to bypass Pi-hole's blocking, ensuring they can be accessed without restriction.
3. Example: pihole -w example.com

Regex Blacklisting:

1. Command: pihole --regex, pihole --regex --help
2. Functionality: Enables administrators to use regular expressions to block groups of domains that match a specified pattern.
3. Example: pihole --regex '^example\.com$' '.*\.example2\.net'

Regex Whitelisting:

1. Command: Not explicitly mentioned; regex whitelisting uses similar syntax to regex blacklisting.
2. Functionality: Allows administrators to use regular expressions to whitelist groups of domains based on specific patterns, ensuring they are exempt from blocking.
3. Example: Similar to regex blacklisting but used with whitelisting intentions.

In summary, Pi-hole provides commands for both blacklisting and whitelisting individual domains, while regex blacklisting and regex whitelisting extend the functionality to block or allow domains based on pattern matching using regular expressions. These features give administrators granular control over what is blocked and allowed through the Pi-hole DNS server.

7. Testing

Testing Methodologies Employed:

1. Unit Testing:
   - Conducted individual component testing to ensure the correctness of specific functions or modules within the system.
   - Used mock data to validate algorithms and isolated code segments.
2. Integration Testing:
   - Tested the interaction between different system components to ensure they function correctly when combined.
   - Checked data flow and communication between modules.
3. User Acceptance Testing (UAT):
   - Involved end-users or stakeholders in testing the system.
   - Evaluated whether the system meets user requirements and expectations.
   - Received user feedback and addressed any issues.

Test Environment Setup:

- Hardware Configuration: Utilized a server with sufficient computational resources to run the ML algorithms efficiently. Monitored CPU, RAM, and GPU usage during testing.
- Software Configuration: Installed necessary software dependencies, including libraries for machine learning and system monitoring tools.
- Data Source: Employed multiple data sources, accessible through links, to test the ML algorithms. Ensured data diversity and relevance to real-world scenarios.
- Duration: Ran the testing environment continuously for multiple hours, simulating a real-world workload.
- Monitoring Tools: Used monitoring tools to track system performance, detect anomalies, and generate performance reports over time.

Test Cases:

1. Unit Test - Algorithm Accuracy:
   - Test Case Description: Verify that the ML algorithm produces accurate results.

- Expected Outcome: Algorithm accuracy exceeds a predefined threshold.
- Actual Outcome: Moderate accuracy levels achieved, with occasional deviations from expected results due to complex data patterns.

2. Unit Test - Data Preprocessing:
    - Test Case Description: Ensure that data preprocessing steps are correctly handling data from various sources.
    - Expected Outcome: Data is cleaned and transformed without errors.
    - Actual Outcome: Data preprocessing functions effectively handle most data sources, but occasional issues arise with unstructured or noisy data.
3. Integration Test - Data Flow:
    - Test Case Description: Validate the correct flow of data from source to ML algorithm to output.
    - Expected Outcome: Data flows seamlessly between components without data loss or corruption.
    - Actual Outcome: Data flows correctly in most cases, but minor data inconsistencies occasionally occur during transitions between components.
4. Integration Test - Performance under Load:
    - Test Case Description: Assess system performance under high data volume conditions.
    - Expected Outcome: The system maintains acceptable response times and resource utilization.
    - Actual Outcome: The system maintains acceptable performance levels under normal load but experiences moderate slowdowns during peak usage.
5. UAT - User Interface Testing:
    - Test Case Description: Evaluate the user interface for usability and adherence to design specifications.
    - Expected Outcome: Users can easily interact with and navigate the system.
    - Actual Outcome: Users generally find the interface usable, but some minor navigation issues and interface quirks have been reported.
6. UAT - Functional Testing:
    - Test Case Description: Test system functionality against user-defined requirements and use cases.
    - Expected Outcome: The system fulfils user requirements without critical issues.
    - Actual Outcome: The system fulfils most user requirements, but a few functional gaps have been identified, leading to occasional workarounds.

7. Long-Term Performance Test:
    - Test Case Description: Monitor system performance over several hours to detect memory leaks or performance degradation.
    - Expected Outcome: The system remains stable and responsive over an extended period.
    - Actual Outcome: The system remains stable for moderate durations but exhibits gradual performance degradation and occasional memory leaks over extended hours of operation.
8. Data Source Reliability Test:
    - Test Case Description: Continuously fetch and process data from multiple links.
    - Expected Outcome: Data retrieval and processing are consistent and reliable over time.
    - Actual Outcome: Data retrieval and processing are generally reliable, but occasional interruptions in data sources result in temporary data gaps.

8. References

1. Raspberry Pi Foundation. (n.d.). Raspberry Pi Products. Website
2. Pi-hole. (n.d.). Network-wide Ad Blocking. Website
3. Flask Documentation. (n.d.). Flask Web Framework. Website
4. Scapy Documentation. (n.d.). Scapy - Packet Manipulation and Network Scanner. Website
5. Python Official Documentation. (n.d.). Python Programming Language. Website
6. scikit-learn Documentation. (n.d.). Machine Learning in Python. Website
7. Stack Overflow. (n.d.). Community-Driven Programming Q&A. Website
8. GitHub. (n.d.). Version Control and Collaboration Platform. Website
9. W3Schools. (n.d.). Web Development Tutorials. Website
10. PM2. (n.d.). Advanced, Production Process Manager for Node.js. Website
11. Official Raspberry Pi OS. (n.d.). Download and Install. Website
12. Flask - Official Documentation. (n.d.). Flask Documentation. Website
13. Pi-hole Documentation. (n.d.). Pi-hole Documentation. Website
14. Scapy Documentation. (n.d.). Scapy Documentation. Website
15. Python Official Documentation. (n.d.). Python Official Documentation. Website

16. scikit-learn Documentation. (n.d.). scikit-learn Documentation. Website

17. PM2 Documentation. (n.d.). PM2 Documentation. Website

18. GitHub - PM2. (n.d.). GitHub Repository for PM2. Website

19. Stack Overflow. (n.d.). Stack Overflow - Programming Q&A. Website

20. Flask - Flask Quickstart. (n.d.). Flask Quickstart Guide. Website

21. Pi-hole - Whitelist. (n.d.). Pi-hole Whitelist Documentation. Website

22. Pi-hole - Blacklist. (n.d.). Pi-hole Blacklist Documentation. Website

23. Python - pickle Module. (n.d.). Python pickle Module Documentation. Website

24. Python - warnings Module. (n.d.). Python warnings Module Documentation. Website

25. Flask - Handling JSON in a POST Request. (n.d.). Flask Documentation - Handling JSON in a POST Request. Website

26. Pi-hole - Command Line. (n.d.). Pi-hole Command Line Documentation. Website

27. Stack Overflow - How to Run a Python Script at Startup on Raspberry Pi. (n.d.). Stack Overflow Question and Answers. Website

28. Raspberry Pi Forums - Running Script on Boot. (n.d.). Raspberry Pi Forums Discussion on Running a Script on Boot. Website

#CODE

NETWORK MONITORING AND DOMAIN BLOCKING

```
#!/usr/bin/env python


import requests
from scapy.all import *
import os
import subprocess


# Define your Flask API endpoint
API_URL = "http://localhost:5000/predict"


# Function to check API prediction and block domain if needed
def check_and_block(domain):
    prediction = requests.post(API_URL, json={"url": f"{domain}"}).json()


    if prediction.get("prediction") == 1:
        # Check if the domain is not in the Pi-hole whitelist
        whitelist_check_command = f"pihole -q {domain}"
        result = subprocess.run(whitelist_check_command, shell=True, text=True, capture_output=True)


        if "no results found" in result.stdout.lower():
            print(f"Blocking domain: {domain}")
            block_command = f"pihole -b {domain[:-1]} && pihole -g"
            subprocess.run(block_command, shell=True, text=True, check=True)
        else:
            print(f"Domain {domain} is in the Pi-hole whitelist, not blocking.")


# Callback function to process DNS packets
```

```
def dns_callback(pkt):
    if pkt.haslayer(DNSRR) and pkt.getlayer(DNSRR).type == 1:
        domain = pkt.getlayer(DNSRR).rrname.decode("utf-8")
        print(f"Detected domain: {domain}")
        check_and_block(domain)

# Start sniffing DNS traffic on the specified interface (e.g., "eth0")
iface = "etn0"
sniff(filter="udp port 53",prn=dns_callback,iface=iface)
```

API_INTEGRATION ON RPi

```
import pickle
import warnings
from flask import Flask, request, jsonify

app = Flask(__name__)

# Load the pickled model and vectorizer
with open("pickel_model.pkl", "rb") as f1:
    logit = pickle.load(f1)

with open("pickel_vector.pkl", "rb") as f2:
    vectorizer = pickle.load(f2)

# Ignore the specified warning temporarily
with warnings.catch_warnings():
    warnings.simplefilter("ignore", category=UserWarning)
```

```
@app.route('/predict', methods=['POST'])
def predict():
   try:
      data = request.get_json()
      new_url = data['url']
      x = vectorizer.transform([new_url])
      y_predict = logit.predict(x)
      # result = {'prediction': y_predict[0]}
      if y_predict[0] == "benign":
         prediction = 0
      else:
         prediction = 1
      result = {'prediction': prediction}
      return jsonify(result), 200
   except Exception as e:
      return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
   app.run(host='0.0.0.0', port=5000)
```

LOADING PICKEL FILE

```
def makeTokens(f):
   tkns_BySlash = str(f.encode('utf-8')).split('/')    # make tokens after splitting by slash
   total_Tokens = []
   for i in tkns_BySlash:
      tokens = str(i).split('-')        # make tokens after splitting by dash
      tkns_ByDot = []
```

```
        for j in range(0,len(tokens)):
            temp_Tokens = str(tokens[j]).split('.')        # make tokens after splitting by dot
            tkns_ByDot = tkns_ByDot + temp_Tokens
        total_Tokens = total_Tokens + tokens + tkns_ByDot
    total_Tokens = list(set(total_Tokens))   #remove redundant tokens
    if 'com' in total_Tokens:
        total_Tokens.remove('com')



# In[3]:



file = "pickel_model.pkl"
with open(file, 'rb') as f1:
    logit = pickle.load(f1)
f1.close()



# In[4]:



urls = []
new_url = input("Enter a new URL: ")

# Append the new URL to the list
urls.append(new_url)



# In[5]:
```

```
file = "pickel_vector.pkl"

with open(file, 'rb') as f2:

    vectorizer = pickle.load(f2)

f2.close()

vectorizer = vectorizer

x = vectorizer.transform(urls)

#score = lgr.score(x_test, y_test)

y_predict = logit.predict(x)

# In[6]:

print(y_predict)
```

#AUTOMATION

USING PM2 FOR AUTOMATION ON REBOOT

Persistent applications: Startup Script Generator

PM2 can generate startup scripts and configure them in order to keep your process list intact across expected or unexpected machine restarts.

- **systemd**: Ubuntu >= 16, CentOS >= 7, Arch, Debian >= 7
- **upstart**: Ubuntu ==> 14
- **launchd**: Darwin, MacOSx
- **openrc**: Gentoo Linux, Arch Linux
- **rcd**: FreeBSD
- **systemv**: Centos 6, Amazon Linux

These init systems are automatically detected by PM2 with the pm2 startup command.

Generating a Startup Script

To automatically generate and configuration a startup script just type the command (without sudo) pm2 startup:

```
$ pm2 startup

[PM2] You have to run this command as root. Execute the following command:
```

```
sudo su -c "env PATH=$PATH:/home/unitech/.nvm/versions/node/v14.3/bin pm2 startup <distribution> -u <user> --hp <home-path>
```

Then copy/paste the displayed command onto the terminal:

```
sudo su -c "env PATH=$PATH:/home/unitech/.nvm/versions/node/v14.3/bin pm2 startup <distribution> -u <user> --hp <home-path>
```

Now PM2 will automatically restart at boot.

Note: You can customize the service name via the --service-name <name> option (**#3213**)

Saving the app list to be restored at reboot

Once you have started all desired apps, save the app list so it will respawn after reboot:

```
pm2 save
```

Manually resurrect processes

To manually bring back previously saved processes (via pm2 save):

```
pm2 resurrect
```

Disabling startup system

To disable and remove the current startup configuration:

```
pm2 unstartup
```

The previous line code let PM2 detect your platform. Alternatively you can use another specified init system yourself using:

Updating startup script after Node.js version upgrade

When you upgrade the local Node.js version, be sure to update the PM2 startup script, so it runs the latest Node.js binary you have installed.

First disable and remove the current startup configuration (copy/paste the output of that command):

```
$ pm2 unstartup
```

Then restore a fresh startup script:

```
$ pm2 startup
```

User permissions

Let's say you want the startup script to be executed under another user.

Just change the -u <username> option and the --hp <user_home>:

```
pm2 startup ubuntu -u www --hp /home/ubuntu
```

Specifying the init system

You can specify the platform you use by yourself if you want to (where platform can be either one of the cited above):

```
pm2 startup [ubuntu | ubuntu14 | ubuntu16 | ubuntu18 | ubuntu20 | ubuntu12 | centos | centos6 | arch | oracle | amazon | macos | darwin | freebsd | systemd | systemv | upstart | launchd | rcd | openrc]
```

SystemD installation checking

```
# Check if pm2-<USER> service has been added

$ systemctl list-units

# Check logs

$ journalctl -u pm2-<USER>

# Cat systemd configuration file

$ systemctl cat pm2-<USER>

# Analyze startup

$ systemd-analyze plot > output.svg
```

To wait efficiently that machine is online for PM2 to run:

```
[Unit]

Wants=network-online.target

After=network.target network-online.target

[....]

[Install]

WantedBy=multi-user.target network-online.target
```