**LAB13 Cloud Security risk prediction**

**Class: M03 student ID: B2111933 Name: Truong Dang Truc Lam**

Select one model among [modelA] [modelB]

| Items | Resource |
| --- | --- |
| VM type | VirtualBox VM |
| OS | Ubuntu 20.04 |
| IP/URL | http://127.0.0.1:8050 (local dashboard) |
| Language,version | Python 3.6+ |
| framework | Dash (for data dashboard), Scapy (for network attack) |
| Libraries | pandas, plotly, dash, scapy, random, datetime |
| Software tool | Wireshark (monitor packets), VS Code / nano (edit code), browser (dashboard) |
| Protocol; | MQTT, CoAP |
| Message broker | Mosquitto |
| Software tool | Wireshark, Python3, Mosquitto (for optional broker traffic) |
| Container | Docker |
| Code Reference | cam\_flood.py (CAM table overflow), cloud\_predict.py (traffic dashboard) |
|  |  |

**[modelA]**

1. Clone coding model from slide or Ref.site
2. Software setup environment/IDE (multiple choice possible)
3. Design software Architecture
4. Explain your coding process (write used resource)

**[modelB]**

1. Clone coding model from slide or Ref.site

Code for prediction and display on dashboard the cloud server's traffic trend (Slide)

1. Software setup environment/IDE (multiple choice possible)

Operating System: Kali Linux, Ubuntu 20.04 (on VirtualBox)

Python Environment: Python 3.6+ (installed via apt or pyenv)

Code Editor:

* Visual Studio Code (VS Code)
* Nano (lightweight editor in terminal)

Package Manager:

* pip (for Python libraries)
* apt (for system-level tools like Wireshark, Mosquitto)

Browser: Firefox or Chromium (to view Dash app)

Monitoring Tools: Wireshark (for analyzing network traffic)

Command-line Tools: bash, ifconfig, ip, netstat

Optional Containerization: Docker (for isolating services or deploying dashboard)

1. Design software Architecture

The project adopts a modular architecture with two primary components: a network attack module and a cloud traffic monitoring dashboard. The network module, implemented in Python using Scapy, generates and sends spoofed Ethernet packets to simulate a CAM table overflow attack on a local network. This component operates on a virtual machine running Kali Linux and interacts directly with network interfaces. The second component is a data visualization dashboard built using the Dash framework. It runs on a separate Ubuntu-based VM or container, simulates cloud server traffic data, and uses a moving average algorithm for real-time traffic prediction. Traffic data is processed with pandas and plotted using plotly. Communication between modules is optional, with MQTT/Mosquitto reserved for future IoT or distributed traffic sources. The architecture supports scalability by containerizing services with Docker and separating concerns across isolated environments for monitoring and simulation.

1. Explain your coding process (write used resource)

I use the code from slides with some modifications:

import time

import random

from datetime import datetime, timedelta

import pandas as pd

import plotly.graph\_objects as go

import dash

from dash import dcc, html

from dash.dependencies import Input, Output

# Simulate cloud server traffic data (replace with your actual data source)

def generate\_traffic\_data(start\_time, end\_time, interval\_seconds=60):

"""Generates simulated traffic data."""

data = []

current\_time = start\_time

while current\_time <= end\_time:

traffic = random.randint(10, 100) # Simulate traffic (e.g., requests per second)

data.append({'timestamp': current\_time, 'traffic': traffic})

current\_time += timedelta(seconds=interval\_seconds)

return pd.DataFrame(data)

# Predict future traffic (simple moving average example)

def predict\_traffic(df, window=5):

"""Predicts future traffic using a simple moving average."""

df['predicted\_traffic'] = df['traffic'].rolling(window=window).mean()

return df

# Create Dash app

app = dash.Dash(\_\_name\_\_)

# Layout of the dashboard

app.layout = html.Div(children=[

html.H1(children='Cloud Server Traffic Trend'),

dcc.Graph(id='traffic-graph'),

dcc.Interval(

id='interval-component',

interval=1\*1000, # in milliseconds, update every second

n\_intervals=0

)

])

# Callback to update the graph

@app.callback(

Output('traffic-graph', 'figure'),

Input('interval-component', 'n\_intervals')

)

def update\_graph(n):

now = datetime.now()

start = now - timedelta(minutes=60) # show last hour

df = generate\_traffic\_data(start, now)

df = predict\_traffic(df)

fig = go.Figure()

fig.add\_trace(go.Scatter(x=df['timestamp'], y=df['traffic'], mode='lines', name='Actual Traffic'))

fig.add\_trace(go.Scatter(x=df['timestamp'], y=df['predicted\_traffic'], mode='lines', name='Predicted Traffic'))

fig.update\_layout(

title='Cloud Server Traffic',

xaxis\_title='Time',

yaxis\_title='Traffic (Requests/s)',

)

return fig

# Run the app

if \_\_name\_\_ == '\_\_main\_\_':

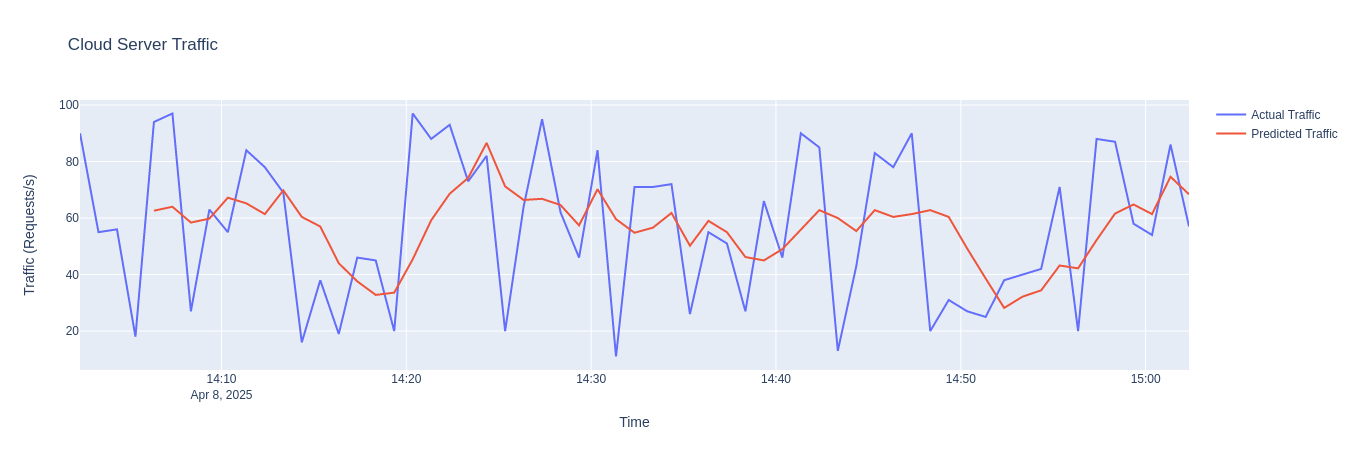
app.run\_server(debug=True)

1. Execute your process and explain

To demonstrate both network security testing and cloud-based traffic monitoring, the project was executed in two distinct environments. First, the CAM table overflow attack was run on a Kali Linux virtual machine. Using Python and the Scapy library, a script (wireless.py) was executed with root permissions to craft and flood the local network interface with random MAC addresses. This simulated an attack that overwhelms the switch’s CAM table, potentially leading to network traffic being broadcasted—ideal for sniffing or denial-of-service testing. The interface used (wlan0 or eth0) was identified using commands like ip a before running the script.

Simultaneously, a second component—cloud server traffic prediction—was launched using a Python script (cloud\_predict.py) on a separate VM or host. This script uses pandas and plotly inside a Dash framework to simulate server traffic data and apply a moving average for prediction. The dashboard was served locally via http://127.0.0.1:8050 and updated every second using Dash’s dcc.Interval component. Users could view live traffic trends and predicted values in a responsive web UI.

Overall, both tools were executed successfully in isolated environments: one to simulate a security vulnerability and the other to visualize real-time analytics. This setup demonstrated a combination of offensive testing and operational monitoring, useful in infrastructure security studies.



Cloud Server Traffic