

# **CYT160 – NIA**

## **Final Project Assignment**

**End-to-End IoT Security Pipeline: Temperature Sensing, MQTT  
Ingestion, Threat Detection, and Analysis**

### **Members:**

Ervan Louie Bermejo (132665233)

Yrnest Justine Radam (136728235)

Kris Sadiasa (114163223)

Radjznairene Jallorina (144342235)

### **Instructor:**

David Chan

## Table of Contents

<b>GITHUB Repository .....</b>	<b>3</b>
<b>Hardware Setup .....</b>	<b>3</b>
<b>Access Raspberry Pi and PuTTY Configuration .....</b>	<b>4</b>
<b>Text File Configuration for DS18B20 on Raspberry Pi.....</b>	<b>5</b>
<b>Create the temp.py Script and Contents of temp.py Script.....</b>	<b>7</b>
<b>Run the Script.....</b>	<b>8</b>
<b>Launch and connect to AWS EC2 .....</b>	<b>9</b>
<b>Install Mosquitto (MQTT Broker) on EC2 .....</b>	<b>9</b>
<b>Set Up Raspberry Pi to Send Temperature via MQTT .....</b>	<b>11</b>
<b>Install and Run Suricata (IDS).....</b>	<b>12</b>
<b>Install and Start the ELK Stack (Elasticsearch, Logstash, Kibana) .....</b>	<b>14</b>
<b>Configuring and Installing Kibana .....</b>	<b>16</b>
<b>Install Filebeat to Ship Logs .....</b>	<b>17</b>
<b>Log Temperature Data from MQTT to a File .....</b>	<b>20</b>
<b>Simulate Attacks .....</b>	<b>20</b>
<b>Data Ingestion and Analysis .....</b>	<b>23</b>
<b>View Logs in Kibana.....</b>	<b>26</b>
<b>Dashboard .....</b>	<b>27</b>
<b>Summary Report .....</b>	<b>28</b>

# GITHUB Repository

<https://github.com/elbermejo/cyt160-FinalProject>

## Hardware Setup

The following components were used in our project:

1. **Raspberry Pi** – A compact, affordable single-board computer developed by the Raspberry Pi Foundation. Despite its size, it functions much like a regular desktop computer and can run full operating systems, including Linux. It's ideal for IoT and embedded projects.
2. **Breadboard** – A tool used for prototyping and testing electronic circuits without the need for soldering. It allows components to be easily added, removed, and rearranged during experimentation.
3. **DS18B20 Temperature Sensor** – A digital temperature sensor that communicates via the 1-Wire protocol, offering precise temperature readings. In this project, we used the waterproof version, encased in a stainless-steel probe, which makes it suitable for measuring temperatures in liquids or outdoor environments.

### DS18B20 Sensor Wire Connections

Sensor Wire Color	Function	Connects To	Raspberry Pi Pin Number
Red	Power (VCC)	3.3V	Pin 1
Black	Ground (GND)	GND	Pin 6
Yellow	Data (DQ)	GPIO4 (1-Wire Data)	Pin 7

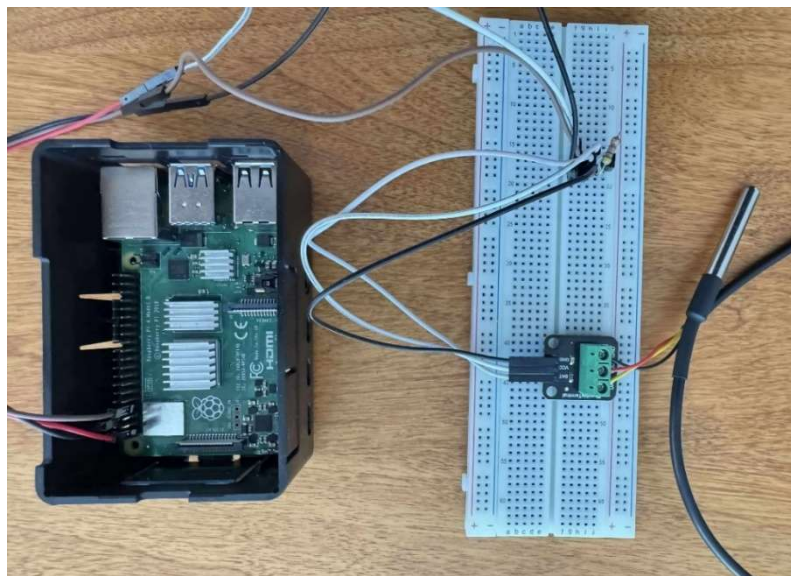


Figure 1: DS18B20 sensor connected to Raspberry Pi via a breadboard, showing proper wiring for power, ground, and data.

## Access Raspberry Pi and PuTTY Configuration

The **PuTTY** application is used to access the Raspberry Pi remotely from a windows computer. Putty us a free and open- source terminal emulator, network file transfer application, and serial console for Windows platforms.

### Steps to Establish an SSH Connection:

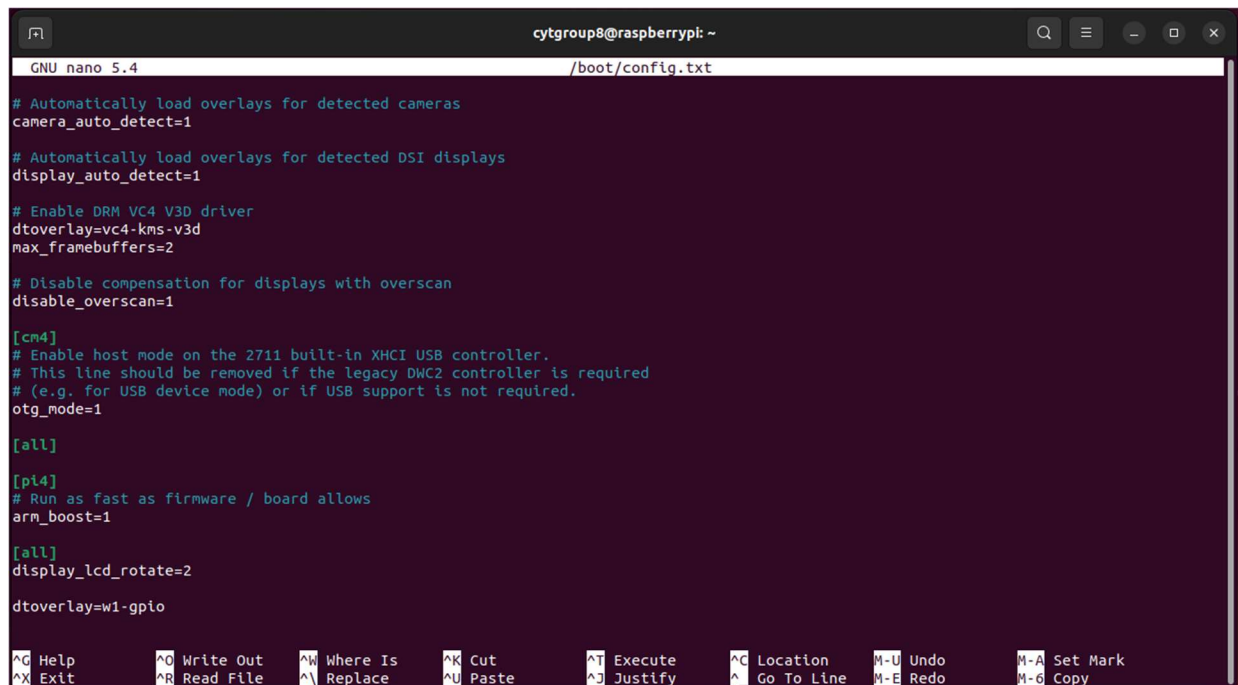
1. **Launch PuTTY** – Open the PuTTY app on your Windows Platform.
2. **Enter the Raspberry Pi's Address** – In the “Host Name (or IP address)” box, type you raspberry Pi's IP address (like 192.168.1.10)
3. **Set the Connection Type** – Ensure that the port is set to 22 and its connection type is SSH.
4. **Save the Session (Optional)** – In the “Saved Sessions” field, enter a file name like RaspberryPi and click Save button. This method allows you to quickly reconnect without re-entering the IP address.
5. **Initiate the Connection** – Click Open to start the SSH Session
6. **Accept Security Warning (For First Time Only)** – If the PuTTY will display a security alert, click YES to trust the connection.
7. **Login via Terminal** – a terminal window will open, prompting you to login. Type your username (usually **pi**) and the default password is **raspberrypi** (unless it has been changed).
8. **Access the Raspberry Pi CLI** – Upon the successful login, you will gain access to the Raspberry Pi's command-line interface that will allow you to execute commands.

# Text File Configuration for DS18B20 on Raspberry Pi

To enable communication with DS18B20 digital temperature sensor via 1-Wire protocol, perform the following configuration on the Raspberry Pi.

**Step 1:** Open the config file by typing: **sudo nano /boot/config.txt**

**Step 2:** Scroll to the bottom and add this line: **dtoverlay=w1-gpio**  
Save the file by pressing **Ctrl + X**, then press **Y**, and hit **Enter**.



```
cytgroup8@raspberrypi: ~
GNU nano 5.4 /boot/config.txt
# Automatically load overlays for detected cameras
camera_auto_detect=1

# Automatically load overlays for detected DSI displays
display_auto_detect=1

# Enable DRM VC4 V3D driver
dtoverlay=vc4-kms-v3d
max_framebuffers=2

# Disable compensation for displays with overscan
disable_overscan=1

[cm4]
# Enable host mode on the 2711 built-in XHCI USB controller.
# This line should be removed if the legacy DWC2 controller is required
# (e.g. for USB device mode) or if USB support is not required.
otg_mode=1

[all]

[pi4]
# Run as fast as firmware / board allows
arm_boost=1

[all]
display_lcd_rotate=2

dtoverlay=w1-gpio

^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location  ^M-U Undo     ^M-A Set Mark
^X Exit      ^R Read File ^A Replace   ^U Paste     ^J Justify   ^_ Go To Line ^M-E Redo     ^M-G Copy
```

**Step 3:** Reboot your Raspberry Pi by typing **sudo reboot**

After the reboot, enter **sudo modprobe w1-gpio** and then **sudo modprobe w1-therm** to load the needed modules.

**Step 4:** Navigate to the sensor directory by typing **cd /sys/bus/w1/devices/**.

Type **ls** to list the folders—you should see one that starts with **28-**



```
cytgroup8@raspberrypi:~ $ cd /sys/bus/w1/devices/
cytgroup8@raspberrypi:/sys/bus/w1/devices $ ls
28-00000083d804 28-000000854937 w1_bus_master1
cytgroup8@raspberrypi:/sys/bus/w1/devices $
```

**Step 5:** Change into that directory `cd 28-00000083d804`

**Step 6:** Type `cat w1_slave` to read the temperature data.

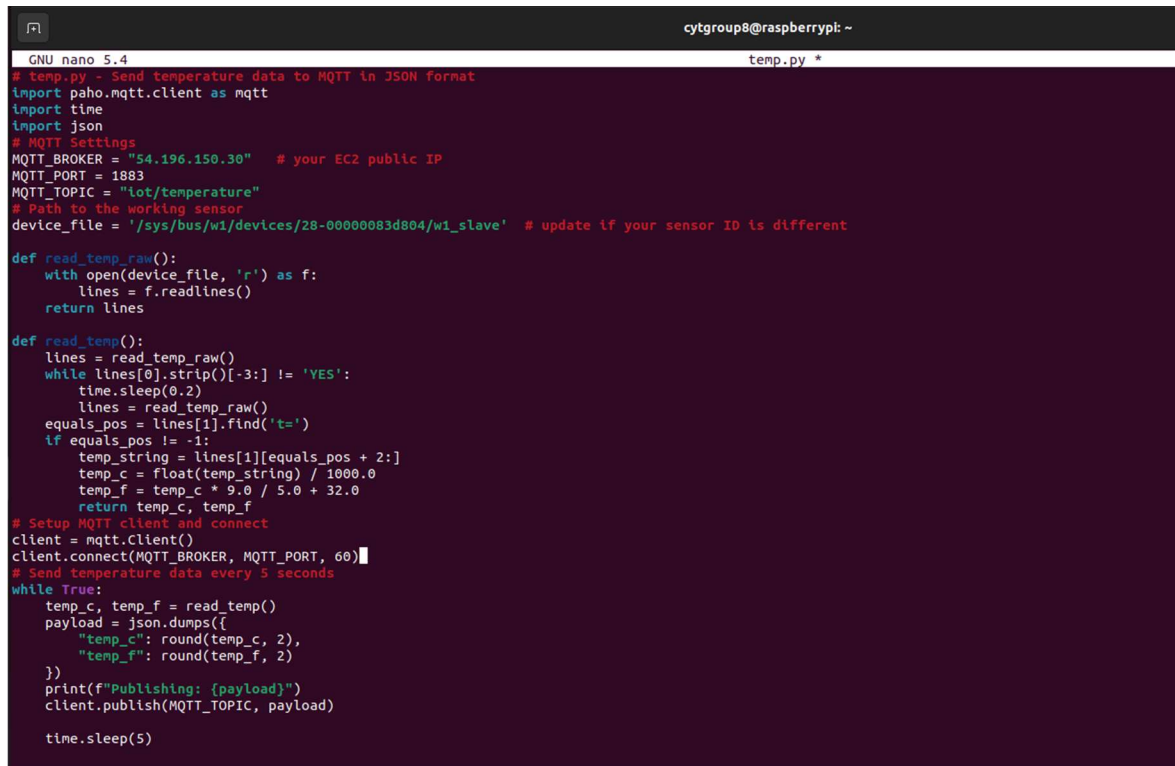
Look for the value after `t=`—that's the temperature in thousandths of a degree

```
cytgroup8@raspberrypi:/sys/bus/w1/devices/28-00000083d804 $ cat w1_slave
63 01 7f 80 7f ff 0d 10 9a : crc=9a YES
63 01 7f 80 7f ff 0d 10 9a t=22187
cytgroup8@raspberrypi:/sys/bus/w1/devices/28-00000083d804 $
```

Celsius (e.g., 22187 means 22.187°C).

# Create the temp.py Script and Contents of temp.py Script

**Step 1:** In the terminal, type `nano temp.py` to create a new Python script.



```
GNU nano 5.4 temp.py *
# temp.py - Send temperature data to MQTT in JSON format
import paho.mqtt.client as mqtt
import time
import json
# MQTT Settings
MQTT_BROKER = "54.196.150.30" # your EC2 public IP
MQTT_PORT = 1883
MQTT_TOPIC = "iot/temperature"
# Path to the working sensor
device_file = '/sys/bus/w1/devices/28-00000083d804/w1_slave' # update if your sensor ID is different

def read_temp_raw():
    with open(device_file, 'r') as f:
        lines = f.readlines()
    return lines

def read_temp():
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = read_temp_raw()
    equals_pos = lines[1].find('t=')
    if equals_pos != -1:
        temp_string = lines[1][equals_pos + 2:]
        temp_c = float(temp_string) / 1000.0
        temp_f = temp_c * 9.0 / 5.0 + 32.0
    return temp_c, temp_f

# Setup MQTT client and connect
client = mqtt.Client()
client.connect(MQTT_BROKER, MQTT_PORT, 60)
# Send temperature data every 5 seconds
while True:
    temp_c, temp_f = read_temp()
    payload = json.dumps({
        "temp_c": round(temp_c, 2),
        "temp_f": round(temp_f, 2)
    })
    print(f"Publishing: {payload}")
    client.publish(MQTT_TOPIC, payload)
    time.sleep(5)
```

**Step 2:** The script imports the `os`, `glob`, and `time` modules.

**Step 3:** Load 1-Wire drivers using `os.system('modprobe w1-gpio')` and `os.system('modprobe w1-therm')`.

**Step 4:** It searches for the DS18B20 sensor folder inside `/sys/bus/w1/devices/`, which usually starts with `28-`.

**Step 5:** Define the function `read_temp_raw()` to read the sensor's raw data.

**Step 6:** Another function `read_temp()` extracts the temperature value from that data.

**Step 7:** Finally, it enters a loop that reads and prints the temperature every second using `print(f"Temperature: {temp:.2f}°C")`.

**Step 8:** Once you've pasted the code, save and exit by pressing **Ctrl + X**, then **Y**, and then **Enter**.

## Run the Script

Type `python3 temp.py` in the terminal and press Enter.

You will now see the temperature being printed every second in Celsius.

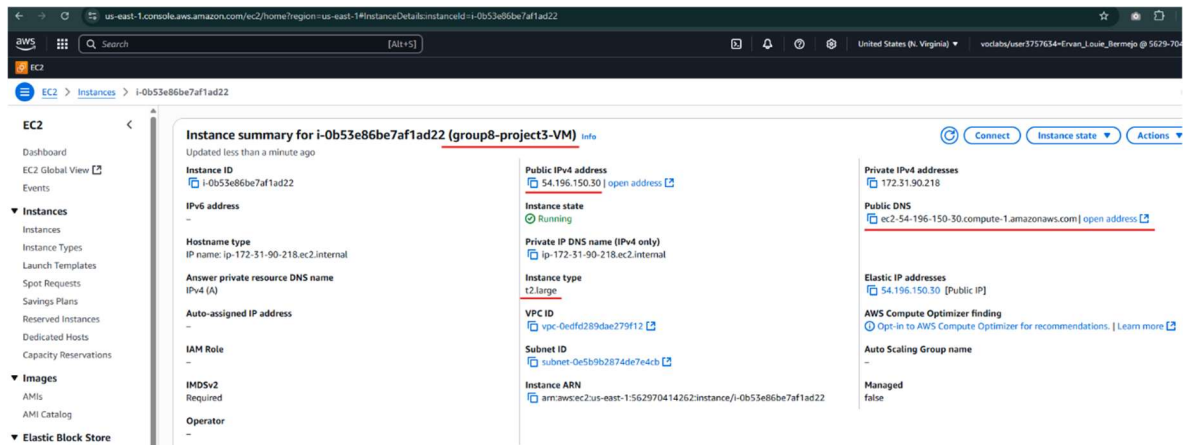
```
cytgroup8@raspberrypi:~ $ sudo python3 temp.py
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.31, "temp_f": 72.16}
Publishing: {"temp_c": 22.31, "temp_f": 72.16}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
Publishing: {"temp_c": 22.25, "temp_f": 72.05}
^CTraceback (most recent call last):
  File "/home/cytgroup8/temp.py", line 45, in <module>
    time.sleep(5)
KeyboardInterrupt

cytgroup8@raspberrypi:~ $
```



## Launch and connect to AWS EC2

We launched an Ubuntu EC2 instance on AWS using t2.Large instance allowing these ports in the security group: **22 (SSH)**, **5601 (Kibana)**, **9200 (Elasticsearch)**, **1883 (MQTT)**, and **5044 (Filebeat)**. Then connected to the instance using SSH from my terminal.



## Install Mosquitto (MQTT Broker) on EC2

**Step 1:** On the EC2 instance, we installed Mosquitto to act as an MQTT broker:

```
sudo apt update
sudo apt install -y mosquitto mosquitto-clients
dpkg -l | grep mosquito-clients
```

```
group8@cyt160:~$ mosquitto -h
mosquitto version 2.0.18

mosquitto is an MQTT v5.0/v3.1.1/v3.1 broker.

Usage: mosquitto [-c config_file] [-d] [-h] [-p port]

-c : specify the broker config file.
-d : put the broker into the background after starting.
-h : display this help.
-p : start the broker listening on the specified port.
    Not recommended in conjunction with the -c option.
-v : verbose mode - enable all logging types. This overrides
    any logging options given in the config file.

See https://mosquitto.org/ for more information.
```

```
group8@cyt160:~$ dpkg -l | grep mosquitto-clients
ii  mosquitto-clients      2.0.18-1build3      amd64      Mosquitto command line MQTT clients
group8@cyt160:~$
```

**Step 2:** Edited the config file to allow remote connections:

```
sudo nano /etc/mosquitto/mosquitto.conf
```

Inside the file, we added:

```
listener 1883
allow_anonymous true
log_dest file /var/log/mosquitto/mosquitto.log
log_type all
```

```
GNU nano 7.2 /etc/mosquitto/mosquitto.conf
listener 1883
allow_anonymous true
log_dest file /var/log/mosquitto/mosquitto.log
log_type all
```

**Step 3:** After saving, we restarted Mosquitto:

```
sudo systemctl restart mosquitto
```

We also ensured that the specified ports were open in the EC2 security group:

- Port 9200 – Elasticsearch (search and analytics engine)
- Port 5044 – Logstash (data ingestion via Filebeat)
- Port 80 – HTTP (web traffic)
- Port 5601 – Kibana (visualization dashboard)
- Port 1883 – MQTT (message queue telemetry transport)
- Port 22 – SSH (remote access to the EC2 instance)

sg-0f51c65c055d3468f - group8-sg Actions

**Details**

Security group name group8-sg	Security group ID sg-0f51c65c055d3468f	Description launch-wizard-1 created 2025-07-23T02:43:17.689Z	VPC ID vpc-0edfd289dae279f12
Owner 562970414262	Inbound rules count 7 Permission entries	Outbound rules count 1 Permission entry	

[Inbound rules](#) [Outbound rules](#) [Sharing - new](#) [VPC associations - new](#) [Tags](#)

**Inbound rules (7)** Manage tags Edit inbound rules

	Name	Security group rule ID	IP version	Type	Protocol	Port range	Source	Description
<input type="checkbox"/>	-	sg-0d29ebafd2bd62d20	IPv4	Custom TCP	TCP	9200	0.0.0.0/0	-
<input type="checkbox"/>	-	sg-0f380c6a66c63fcc5	IPv4	Custom TCP	TCP	5044	0.0.0.0/0	-
<input type="checkbox"/>	-	sg-0e0f4288906c2a4a9	IPv4	HTTP	TCP	80	0.0.0.0/0	-
<input type="checkbox"/>	-	sg-06bb288f3b82bb788	IPv4	All ICMP - IPv4	ICMP	All	0.0.0.0/0	-
<input type="checkbox"/>	-	sg-09be7f21f0c7f29dd	IPv4	Custom TCP	TCP	5601	0.0.0.0/0	-
<input type="checkbox"/>	-	sg-0bc741aaff6376c2	IPv4	SSH	TCP	22	0.0.0.0/0	-
<input type="checkbox"/>	-	sg-02832844648b99ef0	IPv4	Custom TCP	TCP	1883	0.0.0.0/0	-

# Set Up Raspberry Pi to Send Temperature via MQTT

**Step 1:** On the Raspberry Pi, we had a DS18B20 temperature sensor (or a similar 1-Wire sensor) that is properly connected and functioning. We installed the Paho MQTT Python client using the following command:

**pip3 install paho-mqtt**

```
cytgroup8@raspberrypi:~ $ pip3 show paho-mqtt
Name: paho-mqtt
Version: 1.5.1
Summary: MQTT version 5.0/3.1.1 client class
Home-page: http://eclipse.org/paho
Author: Roger Light
Author-email: roger@atchoo.org
License: Eclipse Public License v1.0 / Eclipse Distribution License v1.0
Location: /usr/lib/python3/dist-packages
Requires:
Required-by:
cytgroup8@raspberrypi:~ $
```

**Step 2:** Modifying **temp.py** script to publish temperature readings to MQTT broker that is hosted on our AWS EC2 instance. The image shown below is the revised script.

```
cytgroup8@raspberrypi: ~/Project3

import paho.mqtt.client as mqtt
import time
import json

# MQTT Settings
MQTT_BROKER = "54.196.150.30" # your EC2 public IP
MQTT_PORT = 1883
MQTT_TOPIC = "iot/temperature"

# Path to the working sensor
device_file = '/sys/bus/w1/devices/28-00000083d804/w1_slave' # update if your sensor ID is different

def read_temp_raw():
    with open(device_file, 'r') as f:
        lines = f.readlines()
    return lines

def read_temp():
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = read_temp_raw()
    equals_pos = lines[1].find('t=')
    if equals_pos != -1:
        temp_string = lines[1][equals_pos + 2:]
        temp_c = float(temp_string) / 1000.0
        temp_f = temp_c * 9.0 / 5.0 + 32.0
        return temp_c, temp_f

# Setup MQTT client and connect
client = mqtt.Client()
client.connect(MQTT_BROKER, MQTT_PORT, 60)

# Send temperature data every 5 seconds
while True:
    temp_c, temp_f = read_temp()
    payload = json.dumps({
        "temp_c": round(temp_c, 2),
        "temp_f": round(temp_f, 2)
    })
    print(f"Publishing: {payload}")
    client.publish(MQTT_TOPIC, payload)
    time.sleep(5)
```

**Step 3:** Executing the script on Raspberry Pi, and on the EC2 we could verify it was working by running:

## Python3 temp.py -> Pi

## Python3 mqtt\_logger.py -> EC2 Instance

[illegible]

## Install and Run Suricata (IDS)

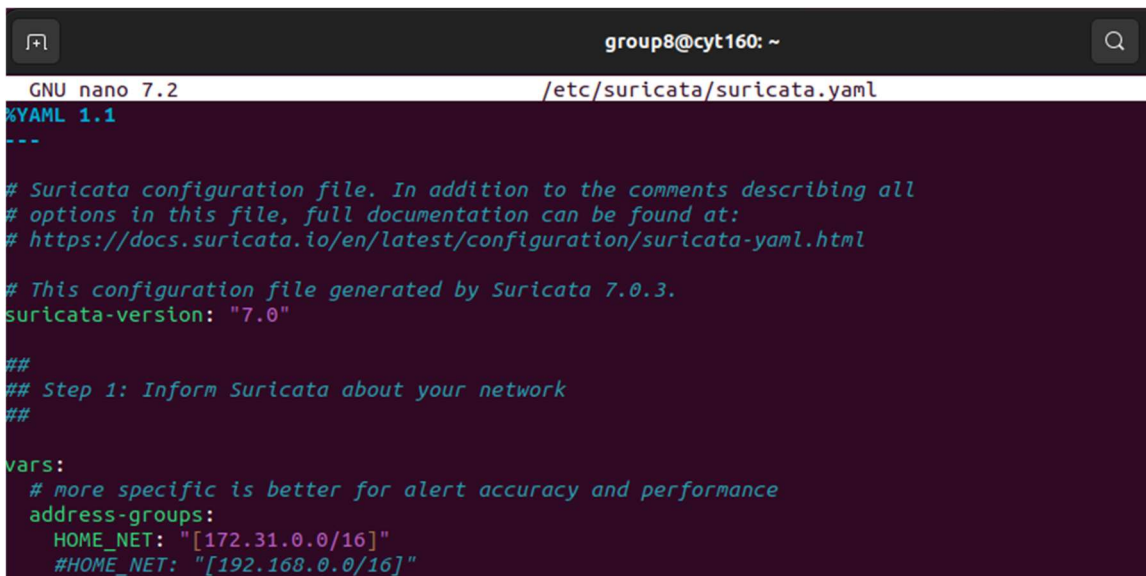
**Step 1:** We installed Suricata on the EC2 with:

```
sudo apt install -y suricata
```

```
group8@cyt160: ~  
  
group8@cyt160:~$ suricata --version  
suricata: unrecognized option '--version'  
Suricata 7.0.3  
USAGE: suricata [OPTIONS] [BPF FILTER]
```

**Step 2:** Modify the configuration file:

```
sudo nano /etc/suricata/suricata.yaml
```



```
GNU nano 7.2 /etc/suricata/suricata.yaml
#YAML 1.1
---
# Suricata configuration file. In addition to the comments describing all
# options in this file, full documentation can be found at:
# https://docs.suricata.io/en/latest/configuration/suricata-yaml.html
# This configuration file generated by Suricata 7.0.3.
suricata-version: "7.0"

##
## Step 1: Inform Suricata about your network
##

vars:
  # more specific is better for alert accuracy and performance
  address-groups:
    HOME_NET: "[172.31.0.0/16]"
    #HOME_NET: "[192.168.0.0/16]"
```

**Step 3:** We set HOME\_NET to the EC2 internal IP range (or 0.0.0.0/0 for simplicity) and made sure the eve.json output was enabled:

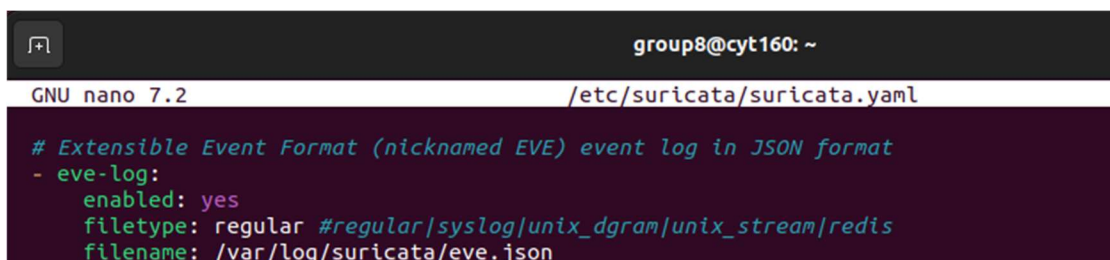
**outputs:**

- eve-log:

enabled: yes

filetype: regular

filename: /var/log/suricata/eve.json



```
GNU nano 7.2 /etc/suricata/suricata.yaml
# Extensible Event Format (nicknamed EVE) event log in JSON format
- eve-log:
  enabled: yes
  filetype: regular #regular/syslog/unix_dgram/unix_stream/redis
  filename: /var/log/suricata/eve.json
```

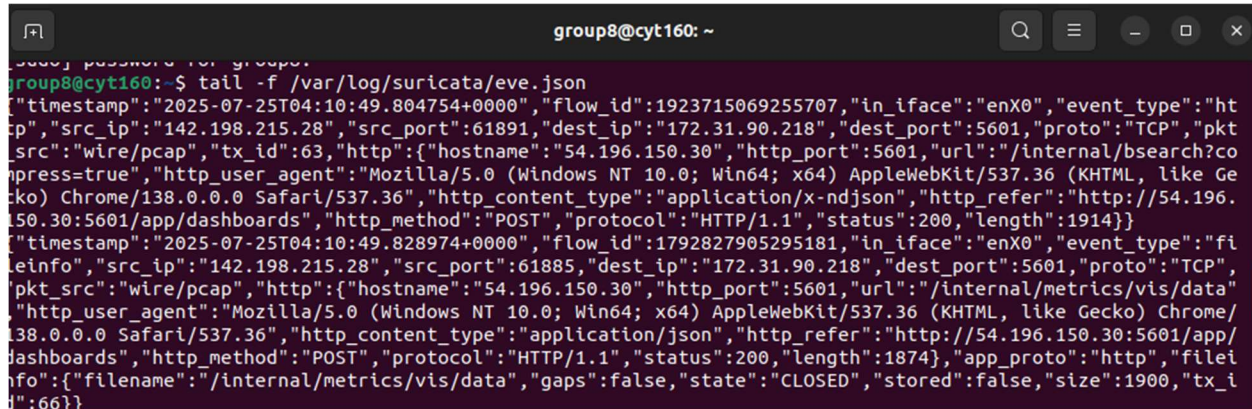
**Step 4:** Running Suricata in IDS mode on the network interface:

```
sudo suricata -c /etc/suricata/suricata.yaml -i eth0
```



**Step 5:** Verify Suricata Alerts to confirm that Suricata was actively monitoring network activity by checking the alerts log in real-time.

```
tail -f /var/log/suricata/eve.json
```

A terminal window titled 'group8@cyt160: ~' showing the command 'tail -f /var/log/suricata/eve.json' being executed. The output displays two JSON alert entries. The first entry has a timestamp of '2025-07-25T04:10:49.804754+0000', flow\_id '1923715069255707', and event\_type 'http'. The second entry has a timestamp of '2025-07-25T04:10:49.828974+0000', flow\_id '1792827905295181', and event\_type 'fileinfo'. Both entries include detailed metadata about the source and destination IP addresses, ports, protocols, and user agents.

```
group8@cyt160:~$ tail -f /var/log/suricata/eve.json
{"timestamp":"2025-07-25T04:10:49.804754+0000","flow_id":1923715069255707,"in_iface":"enX0","event_type":"http","src_ip":"142.198.215.28","src_port":61891,"dest_ip":"172.31.90.218","dest_port":5601,"proto":"TCP","pkt_src":"wire/pcap","tx_id":63,"http":{"hostname":"54.196.150.30","http_port":5601,"url":"/internal/bsearch?compress=true","http_user_agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36","http_content_type":"application/x-ndjson","http_refer":"http://54.196.150.30:5601/app/dashboards","http_method":"POST","protocol":"HTTP/1.1","status":200,"length":1914}}
{"timestamp":"2025-07-25T04:10:49.828974+0000","flow_id":1792827905295181,"in_iface":"enX0","event_type":"fileinfo","src_ip":"142.198.215.28","src_port":61885,"dest_ip":"172.31.90.218","dest_port":5601,"proto":"TCP","pkt_src":"wire/pcap","http":{"hostname":"54.196.150.30","http_port":5601,"url":"/internal/metrics/vis/data","http_user_agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36","http_content_type":"application/json","http_refer":"http://54.196.150.30:5601/app/dashboards","http_method":"POST","protocol":"HTTP/1.1","status":200,"length":1874},"app_proto":"http","fileinfo":{"filename":"/internal/metrics/vis/data","gaps":false,"state":"CLOSED","stored":false,"size":1900,"tx_id":66}}
```

## Install and Start the ELK Stack (Elasticsearch, Logstash, Kibana)

We began by installing Elasticsearch on the AWS EC2 instance. These are the following commands performed to setup the official Elastic package repository and also the latest version of Elasticsearch.

**Step 1:** Install Prerequisites

```
sudo apt install apt-transport-https
```

**Step 2:** Import the GPG Key

```
'wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
```

**Step 3:** Add the Elastic Repository

```
echo "deb https://artifacts.elastic.co/packages/8.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-8.x.list
```

**Step 4:** Update Package Index

```
sudo apt update
```

**Step 5:** Install Elasticsearch

```
sudo apt install elasticsearch
```

## Step 6: Test elasticsearch

```
group8@cyt160: ~  
group8@cyt160:~$ curl -X GET http://localhost:9200  
{  
  "name" : "cyt160",  
  "cluster_name" : "elasticsearch",  
  "cluster_uuid" : "0JrbfDhDSIK8ku_suceVcQ",  
  "version" : {  
    "number" : "8.18.4",  
    "build_flavor" : "default",  
    "build_type" : "deb",  
    "build_hash" : "43b24f613cf25fd3f3369df174e9535a99512aec",  
    "build_date" : "2025-07-16T22:07:56.651816195Z",  
    "build_snapshot" : false,  
    "lucene_version" : "9.12.1",  
    "minimum_wire_compatibility_version" : "7.17.0",  
    "minimum_index_compatibility_version" : "7.0.0"  
  },  
  "tagline" : "You Know, for Search"  
}
```

**Step 7:** To disable security and simplify testing, we edited elasticsearch.yml:

**sudo nano /etc/elasticsearch/elasticsearch.yml**

```
group8@cyt160: ~  
GNU nano 7.2 /etc/elasticsearch/elasticsearch.yml  
#  
#action.destructive_requires_name: false  
#----- BEGIN SECURITY AUTO CONFIGURATION -----  
#  
# The following settings, TLS certificates, and keys have been automatically  
# generated to configure Elasticsearch security features on 23-07-2025 03:08:38  
# -----  
# Enable security features  
xpack.security.enabled: false  
xpack.security.enrollment.enabled: false  
# Enable encryption for HTTP API client connections, such as Kibana, Logstash, and Agents  
xpack.security.http.ssl:  
  enabled: true  
  keystore.path: certs/http.p12
```

# Configuring and Installing Kibana

**Step 1:** After Installing Elasticsearch, we disabled security features by adding the following lines to configuration file.

```
xpack.security.enabled: false
xpack.security.http.ssl.enabled: false
```

**Step 2:** We then restarted the Elasticsearch service to apply the changes.

```
sudo systemctl restart elasticsearch
```

**Step 3:** Afterwards, we installed Kibana using the following syntax.

```
sudo apt install kibana
sudo nano /etc/kibana/kibana.yml
```

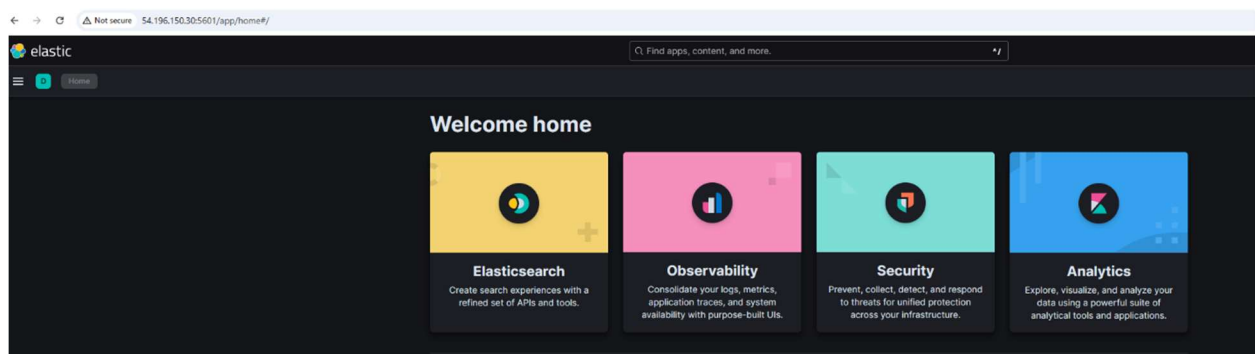
**Step 4:** Ensure that this line was set:

```
server.host: "0.0.0.0"
```

**Step 5:** After saving the changes, we started Kibana:

```
sudo systemctl restart kibana
```

**Step 6:** We checked that port 5601 was open and visited [http://EC2\\_PUBLIC\\_IP:5601](http://EC2_PUBLIC_IP:5601) in the browser.





# Install Filebeat to Ship Logs

Step 1: We installed Filebeat using APT package manager

**sudo apt install filebeat**

```
group8@cyt160: ~  
group8@cyt160:~$ filebeat version  
Filebeat version 8.18.4 (amd64), libbeat 8.18.4 [a9daa983ed2440f7adc8ae3b9a439ab9465c7a4b built 2025-07-16 15:01:22 +0000 UTC]  
group8@cyt160:~$
```

Step 2: Then we edited the config:

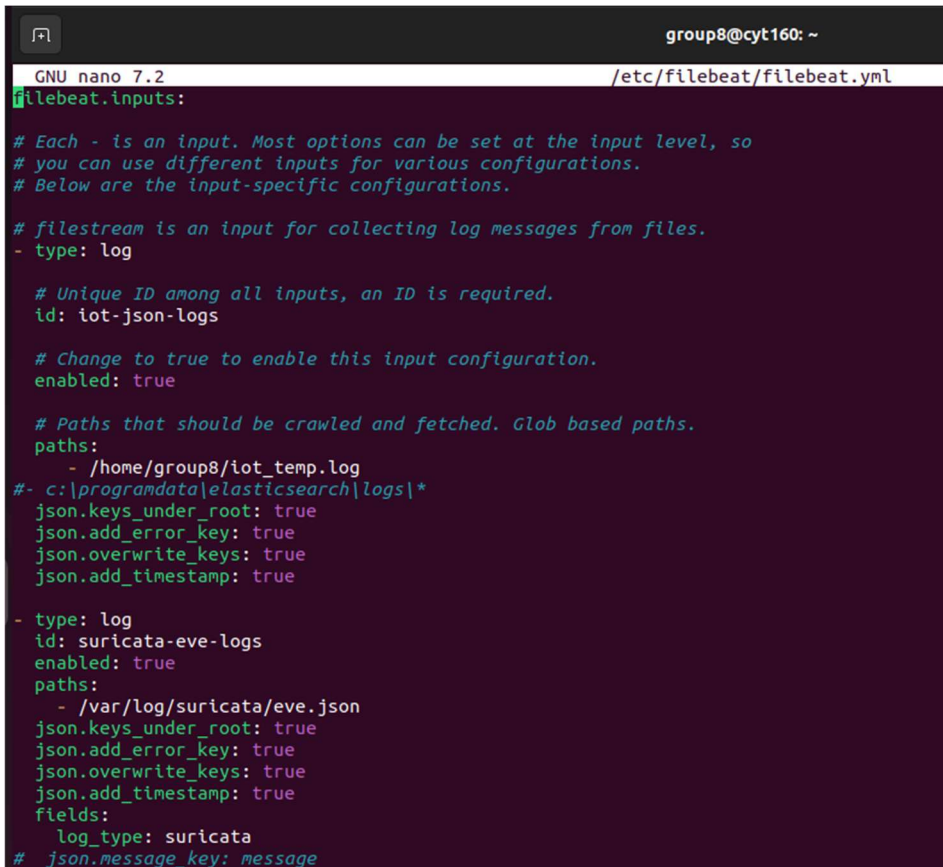
**sudo nano /etc/filebeat/filebeat.yml**

```
GNU nano 7.2 /etc/filebeat/filebeat.yml  
filebeat.inputs:  
# Each - is an input. Most options can be set at the input level, so  
# you can use different inputs for various configurations.  
# Below are the input-specific configurations.  
  
# filestream is an input for collecting log messages from files.  
- type: log  
  
# Unique ID among all inputs, an ID is required.  
id: iot-json-logs  
  
# Change to true to enable this input configuration.  
enabled: true  
  
# Paths that should be crawled and fetched. Glob based paths.  
paths:  
- /home/group8/iot_temp.log  
#- c:\programdata\elasticsearch\logs\*  
  json.keys_under_root: true  
  json.add_error_key: true  
  json.overwrite_keys: true  
  json.add_timestamp: true  
  
- type: log  
  id: suricata-eve-logs  
  enabled: true  
  paths:  
  - /var/log/suricata/eve.json  
  json.keys_under_root: true  
  json.add_error_key: true  
  json.overwrite_keys: true  
  json.add_timestamp: true  
  fields:  
    log_type: suricata  
#  json.message_key: message
```

**Step 3:** We configured Filebeat to read logs from both Suricata and a temperature log inputs.

**filebeat.inputs:**

- type: log  
enabled: true  
paths:
  - /home/group8/iot\_temp.log
- json.keys\_under\_root: true  
json.add\_error\_key: true  
json.overwrite\_keys: true  
json.add\_timestamp: true
- type: log  
id: iot-json-logs  
enabled: true  
paths:
  - /var/log/suricata/eve.json
- json.keys\_under\_root: true  
json.add\_error\_key: true



```
group8@cyt160: ~  
GNU nano 7.2 /etc/filebeat/filebeat.yml  
filebeat.inputs:  
  
# Each - is an input. Most options can be set at the input level, so  
# you can use different inputs for various configurations.  
# Below are the input-specific configurations.  
  
# filestream is an input for collecting log messages from files.  
- type: log  
  
# Unique ID among all inputs, an ID is required.  
id: iot-json-logs  
  
# Change to true to enable this input configuration.  
enabled: true  
  
# Paths that should be crawled and fetched. Glob based paths.  
paths:  
  - /home/group8/iot_temp.log  
#- c:\programdata\elasticsearch\logs\*  
  json.keys_under_root: true  
  json.add_error_key: true  
  json.overwrite_keys: true  
  json.add_timestamp: true  
  
- type: log  
id: suricata-eve-logs  
enabled: true  
paths:  
  - /var/log/suricata/eve.json  
  json.keys_under_root: true  
  json.add_error_key: true  
  json.overwrite_keys: true  
  json.add_timestamp: true  
  fields:  
    log_type: suricata  
#  json.message_key: message
```

**Step 4:** Configure monitoring to elasticsearch

**monitoring.enabled:** true

**monitoring.elasticsearch:**

**hosts:** ["http://localhost:9200"]

```
# Set to true to enable the monitoring reporter.  
monitoring.enabled: true  
monitoring.elasticsearch:  
  hosts: ["http://localhost:9200"]
```

**Step 5:** Configure output to logstash

```
# ----- Logstash Output -----  
output.logstash:  
  # The Logstash hosts  
  hosts: ["localhost:5044"]
```

**Step 6:** To apply the changes, we restarted Filebeat:

**sudo systemctl restart filebeat**

## Log Temperature Data from MQTT to a File

On EC2, we executed a custom Python script named `mqtt_logger.py`. To run the script, we used the following command:

**Python3 `mqtt_logger.py`**

```
group8@cyt160:~$ cat mqtt_logger.py
#!/usr/bin/env python3
import paho.mqtt.client as mqtt

LOG_FILE = "/home/group8/iot_temp.log"
MQTT_BROKER = "localhost"
MQTT_PORT = 1883
MQTT_TOPIC = "iot/temperature"

def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))
    client.subscribe(MQTT_TOPIC)

def on_message(client, userdata, msg):
    message = msg.payload.decode()
    print(f"Received: {message}")
    with open(LOG_FILE, "a") as f:
        f.write(message + "\n")

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(MQTT_BROKER, MQTT_PORT, 60)
client.loop_forever()
```

## Simulate Attacks

**Step 1:** Ensure Suricata is Installed and Running in IDS Mode

**`sudo suricata -c /etc/suricata/suricata.yaml -i enX0`**

```
group8@cyt160:~$ sudo suricata -c /etc/suricata/suricata.yaml -i enX0
Notice: suricata: This is Suricata version 7.0.3 RELEASE running in SYSTEM mode
Info: cpu: CPUs/cores online: 2
Info: suricata: Setting engine mode to IDS mode by default
Info: exception-policy: master exception-policy set to: auto
Info: ioctl: enX0: MTU 1500
Info: logopenfile: fast output device (regular) initialized: fast.log
Info: logopenfile: eve-log output device (regular) initialized: /var/log/suricata/eve.json
Info: logopenfile: stats output device (regular) initialized: stats.log
Info: detect: 1 rule files processed. 5 rules successfully loaded, 0 rules failed, 0
Info: threshold-config: Threshold config parsed: 0 rule(s) found
Info: detect: 5 signatures processed. 0 are IP-only rules, 0 are inspecting packet payload, 1 inspect application layer, 0 are decoder event only
```

## Step 2: Add the Detection Rules : `cat /etc/suricata/rules/suricata.rules`

```
group8@cyt160: ~  
group8@cyt160:~$ cat /etc/suricata/rules/suricata.rules  
alert tcp any any -> any 22 (msg:"SSH Brute Force attempt"; flow:to_server,established; flags:S; detection_filter:track by_src, count 10, seconds 30; sid:1000001; rev:1;)  
alert tcp any any -> any any (msg:"Possible SYN Flood attack"; flags:S; threshold:type both, track by_src, count 100, seconds 10; sid:1000003; rev:1;)  
alert http any any -> any any (msg:"SlowLoris Attack Detected"; content:"User-Agent"; http_header; threshold:type both, track by_src, count 20, seconds 10; sid:1000004; rev:1;)  
alert tcp any any -> any any (msg:"Potential DDOS SYN Flood Detected"; flags:S; threshold:type both, track by_src, count 50, seconds 10; sid:1000010; rev:1;)  
alert tcp any any -> any 1883 (msg:"Malformed JSON Detected"; lua:!malformed_json.lua; sid:1234556; rev:1;)
```

## Step 3: Add malformed\_json.lua :

`sudo nano /etc/suricata/rules/malformed_json.lua`

```
group8@cyt160: ~  
group8@cyt160:~$ cat /etc/suricata/rules/malformed_json.lua  
-- malformed_json.lua  
  
function init(args)  
    local needs = {}  
    needs["payload"] = tostring(true) -- Request packet payload  
    return needs  
end  
  
function match(args)  
    local payload = args["payload"]  
    if not payload then  
        SCLogInfo("Lua: NO RAW PAYLOAD!")  
        return 0  
    end  
  
    SCLogInfo("Lua: PAYLOAD = " .. payload)  
  
    -- Detect malformed JSON: starts with `{` but missing closing `}`  
    if string.match(payload, '{') and not string.find(payload, '}') then  
        return 1  
    end  
  
    -- Detect trailing comma or incomplete quoted string  
    if string.match(payload, '[,,$') or string.match(payload, '"[^"]*$') then  
        return 1  
    end  
  
    return 0  
end
```

## Step 4: We ensured that the Suricata.yaml Includes the default-rule-path:

`/etc/suricata/rules`

```
default-rule-path: /etc/suricata/rules  
rule-files:  
- suricata.rules
```



**Step 5:** To log detected threats and anomalies, we confirmed that alert logging was enabled.

```
# Configure the type of alert (and other) logging you would like.
outputs:
# a line based alerts log similar to Snort's fast.log
- fast:
  enabled: yes
  filename: fast.log
  append: yes
  # filetype: regular # 'regular', 'unix_stream' or 'unix_dgram'

# Extensible Event Format (nicknamed EVE) event log in JSON format
- eve-log:
  enabled: yes
  filetype: regular #regular/syslog/unix_dgram/unix_stream/redis
  filename: /var/log/suricata/eve.json
# Enable for multi threaded eve json outputs output files see README
```

**Step 6:** Simulate Network Threats

Simulate DDOS attack. We used a control traffic generation tool to perform DDoS attack; it allows us to test the Suricata's ability to traffic anomalies.

```
group8@cyt160:~$ tail -f /var/log/suricata/eve.json | jq 'select(.event_type == "alert") | {timestamp: .timestamp, signature: .alert.signature, src: .src.ip}'
{"timestamp": "2025-07-25T22:23:42.235686+0000",
"signature": "Potential DDOS SYN Flood Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:23:42.288959+0000",
"signature": "Possible SYN Flood attack",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:23:58.900842+0000",
"signature": "Potential DDOS SYN Flood Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:23:59.156289+0000",
"signature": "Possible SYN Flood attack",
"src": "142.198.215.28"}

group8@raspberrypi:~/Project3$ sudo hping3 -S -p 80 -l u5000 -c 150 54.196.150.30
HPING 54.196.150.30 (wlan0 54.196.150.30): 5 set, 40 headers + 0 data bytes
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=0 win=0 rtt=26.6 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=1 win=0 rtt=29.4 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=2 win=0 rtt=24.5 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=3 win=0 rtt=27.1 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=4 win=0 rtt=30.8 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=5 win=0 rtt=25.1 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=6 win=0 rtt=27.7 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=7 win=0 rtt=30.7 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=8 win=0 rtt=25.8 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=9 win=0 rtt=28.5 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=10 win=0 rtt=23.6 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=11 win=0 rtt=26.3 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=12 win=0 rtt=29.3 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=13 win=0 rtt=24.4 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=14 win=0 rtt=27.1 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=15 win=0 rtt=30.0 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=16 win=0 rtt=25.2 ms
len=40 ip=54.196.150.30 ttl=55 DF id=0 sport=80 flags=RA seq=17 win=0 rtt=27.9 ms
```

We manually injected malformed JSON payloads in the network to verify that the malformed.lua was successfully detected and recorded the logs events.

```
group8@cyt160:~$ tail -f /var/log/suricata/eve.json | jq 'select(.event_type == "alert") | {timestamp: .timestamp, signature: .alert.signature, src: .src.ip}'
{"timestamp": "2025-07-25T22:25:21.074820+0000",
"signature": "Malformed JSON Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:25:22.100446+0000",
"signature": "Malformed JSON Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:25:22.211139+0000",
"signature": "Malformed JSON Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:25:32.443430+0000",
"signature": "Malformed JSON Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:25:32.467481+0000",
"signature": "Malformed JSON Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:25:33.492086+0000",
"signature": "Malformed JSON Detected",
"src": "142.198.215.28"}
{"timestamp": "2025-07-25T22:25:33.516264+0000",
"signature": "Malformed JSON Detected",
"src": "142.198.215.28"}

cytgroup8@raspberrypi:~/Project3$ python3 mqtt_send_malformed.py
Sending 9 test MQTT payloads...
[1] Publishing: {"sensor": "temp", "value": 100}
[2] Publishing: {"sensor": "temp", "value": 100}
[3] Publishing: {"sensor": "temp", "value": 100}
[4] Publishing: {"sensor": "temp", "value": }
[5] Publishing: {"sensor": "temp", "value": 100}
[6] Publishing: {"sensor": "temp", "value": 100}
[7] Publishing: {"sensor": "temp", "value": 100}
[8] Publishing: {"sensor": "temp", "value": "100"}
[9] Publishing: {"sensor": "temp", "value": 100}
Done.
cytgroup8@raspberrypi:~/Project3$ cat mqtt_send_malformed.py
#!/usr/bin/env python3
import paho.mqtt.publish as publish
import time

BROKER = "54.196.150.30" # <- change to your MQTT broker IP
PORT = 1883
TOPIC = "test/topic"

# List of test payloads: some valid, some intentionally malformed
payloads = [
    {"sensor": "temp", "value": 100}, # valid JSON
    {"sensor": "temp", "value": 100, # missing closing brace
    {"sensor": "temp", "value": 100}, # missing opening brace
    {"sensor": "temp", "value": }, # missing value
    {"sensor": "temp", "value": 100, # trailing comma
    {"sensor": "temp", "value": 100}, # extra comma
    {"sensor": "temp", "value": 100}, # unquoted keys
    {"sensor": "temp", "value": "100"}, # valid with quoted value
    {"sensor": "temp", "value": 100}, # missing comma between keys
]

print(f"Sending {len(payloads)} test MQTT payloads...")
for i, msg in enumerate(payloads):
    print(f"Sending {i+1} Publishing: {msg}")
```

# Data Ingestion and Analysis

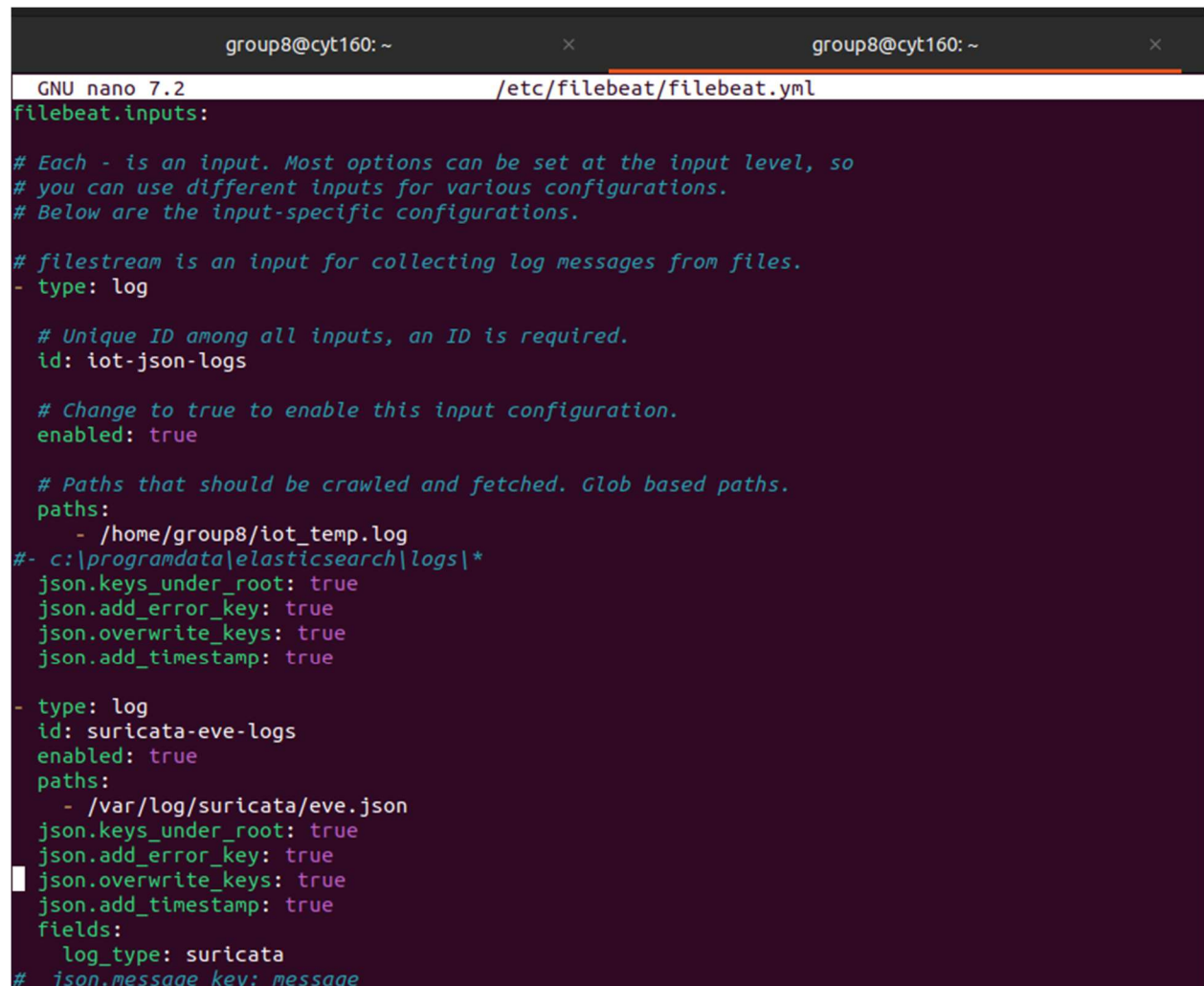
We configured Filebeat on EC2 instance to forward logs. Here are the following steps:

**Step 1:** Configure Filebeat to Send Logs to Logstash:

**/etc/filebeat/filebeat.yml**

Ship Logs to ELK (via Filebeat → Logstash)

Paths: `/var/log/suricata/eve.json` , `/home/group8/iot_temp.log`



```
group8@cyt160: ~  
GNU nano 7.2 /etc/filebeat/filebeat.yml  
filebeat.inputs:  
  
# Each - is an input. Most options can be set at the input level, so  
# you can use different inputs for various configurations.  
# Below are the input-specific configurations.  
  
# filestream is an input for collecting log messages from files.  
- type: log  
  
  # Unique ID among all inputs, an ID is required.  
  id: iot-json-logs  
  
  # Change to true to enable this input configuration.  
  enabled: true  
  
  # Paths that should be crawled and fetched. Glob based paths.  
  paths:  
    - /home/group8/iot_temp.log  
#- c:\programdata\elasticsearch\logs\*  
  json.keys_under_root: true  
  json.add_error_key: true  
  json.overwrite_keys: true  
  json.add_timestamp: true  
  
- type: log  
  id: suricata-eve-logs  
  enabled: true  
  paths:  
    - /var/log/suricata/eve.json  
  json.keys_under_root: true  
  json.add_error_key: true  
  json.overwrite_keys: true  
  json.add_timestamp: true  
  fields:  
    log_type: suricata  
#  json.message_key: message
```

## Step 2: Configure Logstash Pipeline to parse the data from the Filebeat

**/etc/logstash/conf.d/logstash.conf**

```

}
}
}
group8@cyt160:~$ cat /etc/logstash/conf.d/logstash.conf
input {
  beats {
    port => 5044
  }
}

filter {
  # Try to parse JSON from the message field
  json {
    source => "message"
    skip_on_invalid_json => true
  }

  # Parse timestamp if available
  date {
    match => ["@timestamp", "ISO8601"]
    mutate {
      add_tag => ["suricata_alert"]
    }
  }

  # Clean up extra fields for IoT logs (Filebeat)
  if [agent][name] =~ /filebeat/ {
    json {
      source => "message"
      remove_field => ["host", "agent", "ecs", "log", "input"]
    }
  }

  # You can add Metricbeat filters here if needed in the future
}

output {
  # Suricata alerts
  if "suricata_alert" in [tags] {
    elasticsearch {
      hosts => ["http://localhost:9200"]
      index => "suricata-%{+YYYY.MM.dd}"
    }
  }
}

```



### Step 3: Filebeat Log Parsing: `/etc/logstash/conf.d/filebeat.conf`

```
group8@cyt160:~$ cat /etc/logstash/conf.d/filebeat.conf
input {
  beats {
    port => 5044
  }
}

filter {
  json {
    source => "message"
    remove_field => ["host", "agent", "ecs", "log", "input"]
  }
}

output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "iot-temp-%{+YYYY.MM.dd}"
  }

  stdout { codec => rubydebug }
}
```

### Step 4: Metric Beat Parsing : `/etc/logstash/conf.d/metricbeat.conf`

```
group8@cyt160:~$ cat /etc/logstash/conf.d/metricbeat.conf
input {
  beats {
    port => 5044
  }
}

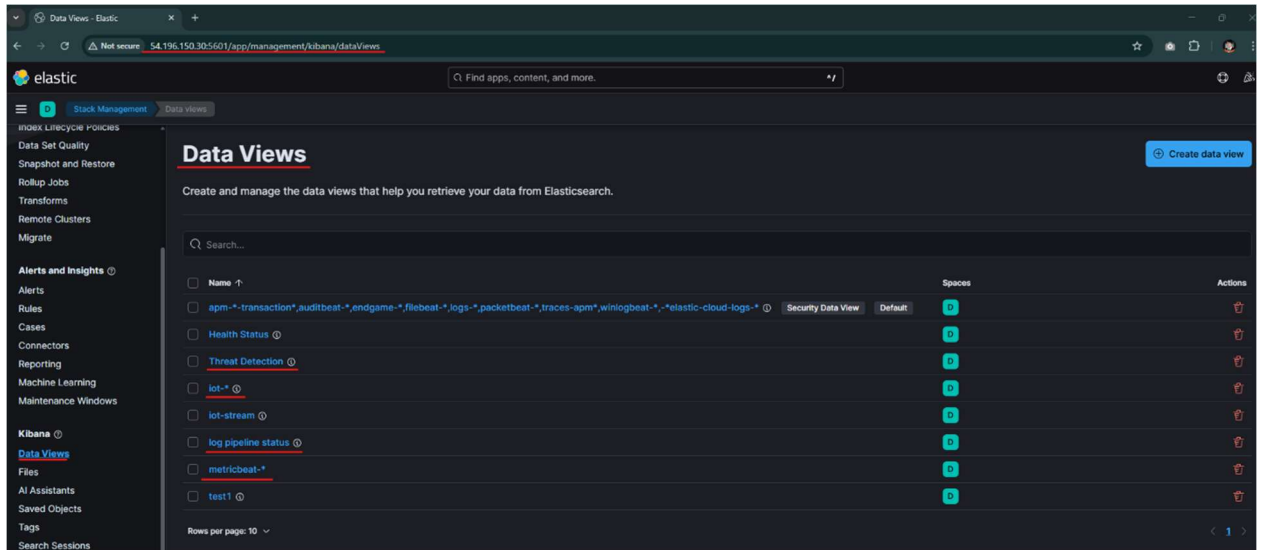
filter {
  # Optional: add filters if needed (e.g., grok, mutate)
}

output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "metricbeat-%{+YYYY.MM.dd}"
  }
}
group8@cyt160:~$
```

# View Logs in Kibana

Steps to View Logs:

1. Open Kibana on a Web Browser.
2. Go to **Kibana > Stack Management > Data Views**

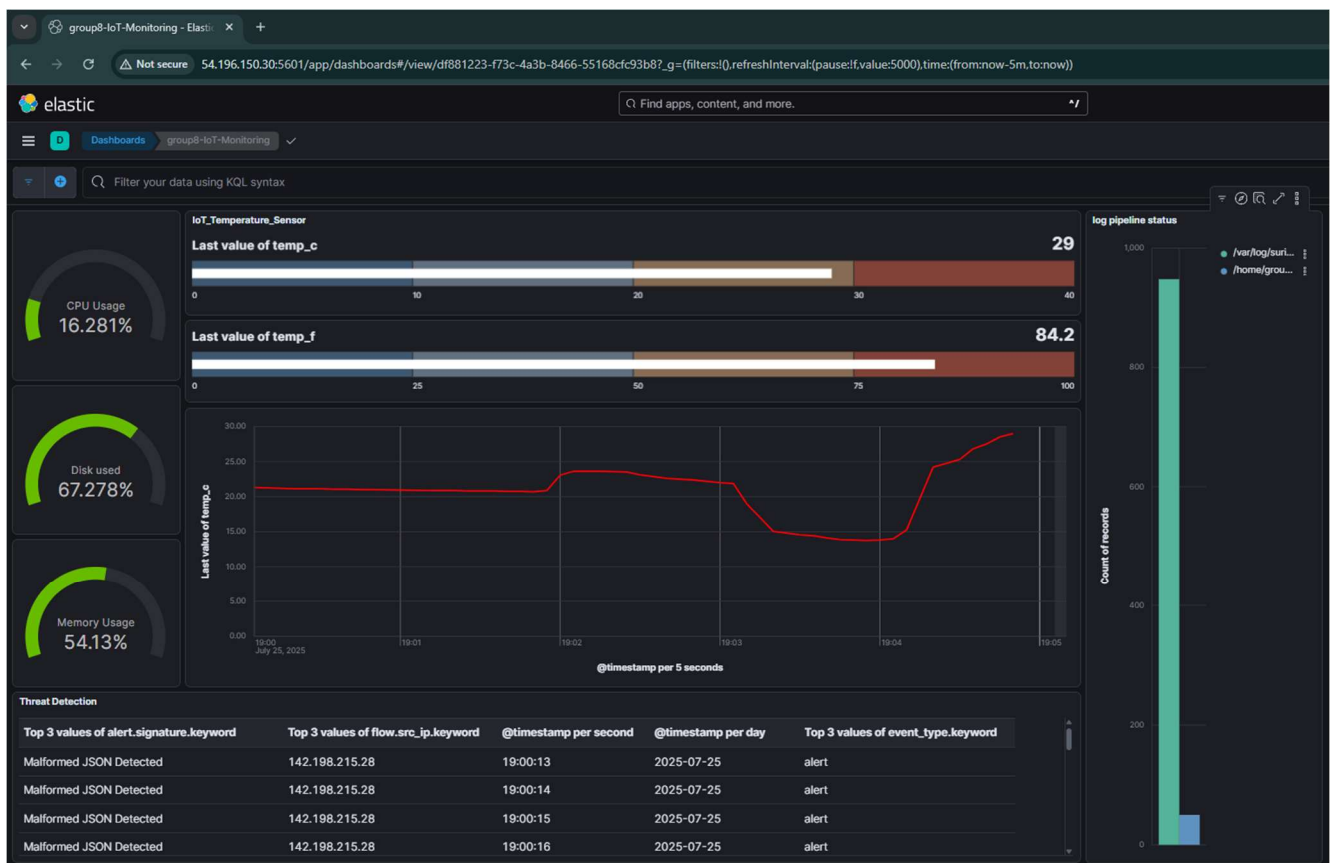


# Dashboard

Create a customized Kibana **dashboards** that visualize and analyze the ingested data.

Listed below are the dashboard components:

1. **IoT temperature data over time** – charts displaying real-time temperature readings over time from DS18B20 sensor.
2. **Detected threats (e.g., alerts, IPs, timestamps, categories)** – visualization of alerts generated by the Suricata.
3. **System health and log pipeline status** – monitoring metrics from system logs such as CPU, memory, and disk usage.



## Summary Report

Our project aimed to implement a complete end-to-end IoT security pipeline that integrates temperature sensing, data transmission using MQTT, intrusion detection using Suricata, and centralized log analysis and visualization using the ELK Stack, which includes Elasticsearch, Logstash, and Kibana. The purpose of this setup was to simulate a real-world IoT environment where data collected from physical sensors can be transmitted remotely, secured through threat detection, and then analyzed through visual dashboards.

The first part of the project focused on the hardware configuration. We used a DS18B20 waterproof digital temperature sensor and connected it to a Raspberry Pi using a breadboard. The sensor was wired to draw power from the Pi's 3.3V pin, grounded to the GND pin, and the data line was connected to GPIO4. We then enabled 1-Wire protocol support by editing the configuration file on the Pi, allowing it to detect and read the sensor's output. After validating that the sensor was properly connected and functional, we created a Python script named `temp.py` that used standard libraries such as `os`, `glob`, and `time` to read the temperature and display it in Celsius. The script was written to continuously output the temperature every second, which simulated the concept of a live stream of environmental data.

To control and monitor the Raspberry Pi remotely, we used an Ubuntu VM inside VMware Workstation to establish an SSH connection. This allowed us to run commands and scripts without the need for a physical monitor or keyboard attached to the Pi. Remote access was a key part of our setup since it reflects how most IoT devices are accessed in practice. After confirming that everything worked locally, we moved on to setting up cloud connectivity.

We launched a virtual machine on AWS using an EC2 t2.Large instance running Ubuntu. We carefully configured its security group settings to allow incoming traffic on the necessary ports. This included port 22 for SSH access, port 1883 for MQTT communication, port 9200 for Elasticsearch, port 5601 for Kibana, and port 5044 for Logstash and Filebeat. After the EC2 instance was running, we installed the Mosquitto broker. This MQTT broker acted as the cloud-based receiver for messages sent by the Raspberry Pi. On the Pi, we installed the Paho MQTT Python client and updated our original temperature script to publish the temperature data to the MQTT topic hosted by Mosquitto on the EC2 instance.

To confirm that the MQTT messages were arriving correctly, we created another script on the EC2 instance called `mqtt_logger.py`. This script subscribed to the same topic and wrote each received message to a file called `iot_temp.log`. This log file later became one of the sources for data analysis and visualization.

Security is an essential aspect of any IoT system, so the next step in our project involved setting up Suricata, a network-based intrusion detection system. We installed Suricata on the EC2 instance and configured it to monitor the primary network interface. The configuration file was edited to define the internal IP range and to enable JSON output through a log file called `eve.json`. Once Suricata was running, we tested it by simulating suspicious behavior. This included a lightweight DDoS simulation using a traffic generation tool and the injection of malformed JSON packets. We also created custom detection rules and included a Lua script to catch invalid traffic. Suricata successfully detected these activities, and alerts were logged in real time. This confirmed that the system was capable of identifying threats, which is a crucial part of any secure IoT solution.

After we had both system logs and sensor data available, we turned our attention to centralized log management and visualization. To handle this, we installed the ELK Stack. This included Elasticsearch for indexing and storing data, Logstash for processing logs, Kibana for dashboard creation, and Filebeat to collect and ship logs from the filesystem. Filebeat was configured to read from both `iot_temp.log` and Suricata's `eve.json` file. The configuration allowed JSON parsing for the Suricata logs and treated the temperature log as plain text. These logs were then forwarded to Elasticsearch, which made them available for searching and visualization inside Kibana.

We also created basic configuration files for Logstash, including pipelines for Filebeat and placeholders for Metricbeat. While Logstash was not heavily used in this phase of the project, preparing it allowed us to understand how future enhancements could be added, such as custom filters or conditional logic for more refined parsing.

Inside Kibana, we created a set of dashboards to make the collected data meaningful. The first set of visualizations showed live temperature data from the Raspberry Pi, plotted over time. Another set of panels displayed alerts generated by Suricata, including source and destination IPs, timestamps, and event categories. We also created visualizations for system-level metrics and log activity. These dashboards gave us a powerful way to observe the performance and security status of our system, all in one place.

During the course of the project, we faced several challenges. MQTT communication between the Raspberry Pi and the EC2 instance initially failed due to port restrictions and firewall configurations, which we resolved by revisiting AWS security group settings. YAML configuration errors in Suricata caused it to crash until they were corrected. In Kibana, we had to manually refresh field mappings and indexes to make our data appear correctly. Despite these issues, the process of debugging and fixing each problem helped us gain hands-on experience in system troubleshooting and gave us a deeper understanding of how these tools work together.

Working on this project taught us how to build a secure, real-time IoT system from the ground up. We learned how to integrate hardware and software components, use cloud infrastructure for data handling, apply intrusion detection systems for security monitoring, and use open-source tools to visualize and analyze the results. Every part of the pipeline depended on the others, so successful integration required attention to detail and proper configuration across the entire stack.

Looking ahead, this project provides a strong base that can be expanded in several ways. We could enhance security by adding encryption and authentication to MQTT communication. We could implement alert notifications using email or SMS. More sensors could be added to collect additional environmental data, such as humidity or light. Logstash could be used more actively for filtering and shaping data before it reaches Elasticsearch. Machine learning models could also be introduced to automatically detect abnormal behavior in either sensor readings or network traffic.

In conclusion, this project helped us bring together multiple areas of knowledge that we've studied throughout the course. It gave us the opportunity to apply those concepts in a real-world scenario that mirrors the type of work done by IT professionals and cybersecurity analysts. We are now more confident in our ability to deploy, secure, and monitor systems that involve both physical devices and cloud infrastructure. Most importantly, we gained a clearer understanding of how the data we collect can be protected and turned into valuable insights.