iSMAGi
Institut Supérieur de Management
d'Administration et de Génie Informatique
المعهد العالي للتدبير و الإدارة و الهندسة المعلوماتية

Kingdom of Morocco
Institut supérieur de management, d'administration et de génie informatique.

END OF STUDIES PROJECT DEFENSE

In order to obtain “licence professionnelle”

Department:

IoT and Intelligent Systems

Zero-Trust IoT Architecture with Honeypot and IDS Implementation

Date of defense: 16/07/2025

Presented by:

- Radouane ELHAJALI

Supervised by:

- Pr. Oussama MAHBOUB

Jury:

- Pr. Oussama MAHBOUB
- Pr. K. SAADAOUI

Academic year: 2024/2025

Résumé

Ce projet vise à concevoir et implémenter une architecture IoT Zero-Trust pour résoudre les vulnérabilités critiques des déploiements IoT traditionnels.

L'objectif principal est d'établir une communication sécurisée end-to-end entre des dispositifs ESP32 et une plateforme cloud via une authentification mTLS (mutual TLS).

Les technologies utilisées comprennent: un routeur OpenWrt configuré avec segmentation VLAN, un Raspberry Pi hébergeant une Autorité de Certification privée, un système de détection d'intrusion Suricata, un honeypot Cowrie, et une plateforme ThingsBoard pour la gestion cloud.

L'architecture intègre également un système d'alertes en temps réel via Telegram et un tableau de bord web Flask.

Le résultat obtenu est une infrastructure IoT complètement opérationnelle et sécurisée, démontrant un taux de détection des menaces de 100% et une isolation réseau efficace.

Cette solution prouve qu'une sécurité IoT de niveau entreprise peut être réalisée avec des technologies open-source et du matériel standard, offrant une approche pratique pour les organisations cherchant à implémenter des frameworks de sécurité IoT complets.

Abstract

The growth of Internet of Things (IoT) devices has introduced significant security challenges, primarily due to the frequent transmission of sensitive data over insecure networks.

Standard IoT protocols like MQTT often lack mandatory encryption, exposing them to eavesdropping and man-in-the-middle attacks.

This project addresses this critical vulnerability by designing and implementing a robust, end-to-end secure IoT ecosystem.

The primary objective was to establish a secure communication channel between an ESP32 device and a cloud platform using a Zero-Trust architecture based on mutual TLS (mTLS) authentication.

To achieve this, a private Certificate Authority (CA) was established on a Raspberry Pi to issue and manage device certificates.

The network infrastructure was hardened by deploying a custom OpenWrt router configured with a segmented IoT VLAN.

Furthermore, a multi-layered threat monitoring system was integrated, comprising the Suricata Intrusion Detection System (IDS) for network traffic analysis and a Cowrie honeypot to detect and log unauthorized access attempts.

The final result is a fully operational, secure, and monitored IoT infrastructure that demonstrates a comprehensive approach to modern cybersecurity challenges.

Table of Contents

List of Figures:

List of Tables:

List of Abbreviations:

IoT	Internet of things
OpenWrt	Open Wireless Router
TCP	Transmission Control Protocol
TLS	Transport Layer Security
mTLS	Mutual Transport Layer Security
ZTA	Zero-Trust Architecture
LAN	Local Area Network
VLAN	Virtual Local Area Network
WPS	Wi-Fi Protected Access
SSID	Service Set Identifier
IP	Internet Protocol
CA	Certificate Authority
IDS	Intrusion Detection System
PKI	Public Key Infrastructure
MQTT	Message Queuing Telemetry Transport
CSR	Certificate Signing Request
PCAP	Packet Capture
SFTP	Secure File Transfer Protocol
DHCP	Dynamic Host Configuration Protocol
OCSP	Online Certificate Status Protocol
DDoS	Distributed Denial of Service
SSH	Secure Shell
Telnet	Teletype Networkx
PEM	Privacy Enhanced Mail
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
CVE	Common Vulnerabilities and Exposures
OS	Operating System
IDE	Integrated Development Environment
LittleFS	Little File System
Firmware	Firm Software
WLAN	Wireless Network Interface
ETH	Ethernet
RSA	Rivest-Shamir-Adleman
AES	Advanced Encryption Standard
PFS	Perfect Forward Secrecy
GCM	Galois/Counter Mode
SoC	System on Chip
RAM	Random Access Memory
ML	Machine Learning

General Introduction

The Internet of Things (IoT) is experiencing exponential growth, with over 17 billion connected devices worldwide by 2024. This proliferation of connected objects is fundamentally transforming the way we interact with technology, providing unprecedented opportunities for automation, efficiency, and innovation across various sectors.

However, this widespread connectivity comes with critical security challenges. IoT devices, often deployed with minimal security measures, are becoming prime targets for cybercriminals.

Recent attacks, such as the PumaBot botnet in 2025 and the MQTT vulnerabilities discovered in 2024, highlight the urgent need to rethink traditional IoT security approaches.

Project Context

This graduation project is part of the “Licence Professionalle” degree in IoT and Intelligent Systems at the Higher Institute of Management, Administration, and Computer Engineering (ISMAGI). It addresses a pressing need for accessible and effective IoT security solutions for organizations of all sizes.

Problematic

Traditional security approaches, based on the concept of a trusted perimeter, are no longer suitable for the modern IoT environment. IoT devices present several inherent vulnerabilities:

- Unencrypted communications
- Weak or nonexistent authentication
- Lack of network segmentation
- Lack of real-time monitoring

Project Objectives

The main objective of this project is to design and implement a secure IoT architecture based on the Zero-Trust model, integrating:

- Efficient network segmentation
- Strong authentication through certificates
- End-to-end encrypted communications
- A threat detection and monitoring system

1. Introduction

1.1. Real-World IoT Attacks

Threat/Vulnerability	Attack Vector	Real-World Impact	Solution
PumaBot Botnet (2025)	SSH Brute-Force	Device compromise, credential theft, DDoS	Cowrie Honeypot, IDS
CVE-2024-52285 (2024)	Unauthenticated MQTT	Data leakage from physical access systems	mTLS (X.509 Certs + TLS)
Mirai & variants (Ongoing)	Default Telnet/SSH Passwords	Massive DDoS attacks (e.g., Dyn, 2016)	Network Segmentation (VLAN), IDS

Table 1: Recent IoT Attack Vectors and Project Mitigations

PumaBot Botnet (2025) [1]:

- Go-based malware targets weak SSH credentials.
- Establishes persistence via systemd service.
- Harvests credentials for lateral movement.

MQTT Vulnerabilities (2024) [2]:

- Case: CVE-2024-52285.
- Exposed MQTT broker allows unauthenticated access.
- Creates risk of physical security compromise.

1.2. Context and Motivation

The Internet of Things (IoT) has revolutionized how we interact with technology, connecting billions of devices worldwide. However, this connectivity comes with significant security challenges.

Traditional IoT devices often lack robust security measures, making them vulnerable to various cyber threats including unauthorized access, data interception, and device hijacking. Recent security incidents have highlighted the critical need for comprehensive IoT security frameworks.

Many IoT devices communicate using unencrypted protocols, transmit sensitive data in plaintext, and lack proper authentication mechanisms. These vulnerabilities not only compromise individual devices but can also serve as entry points for larger network attacks.

1.3. Problem Statement

The primary challenge addressed in this project is the inherent security weakness in standard IoT implementations. Specifically:

- **Unencrypted Communications**: Many IoT devices transmit data using plaintext protocols like HTTP and unencrypted MQTT, making them susceptible to eavesdropping and man-in-the-middle attacks
- **Lack of Authentication**: Devices often connect to cloud services without proper certificate-based authentication
- **Network Vulnerability**: IoT devices on the same network segment as critical systems can serve as attack vectors

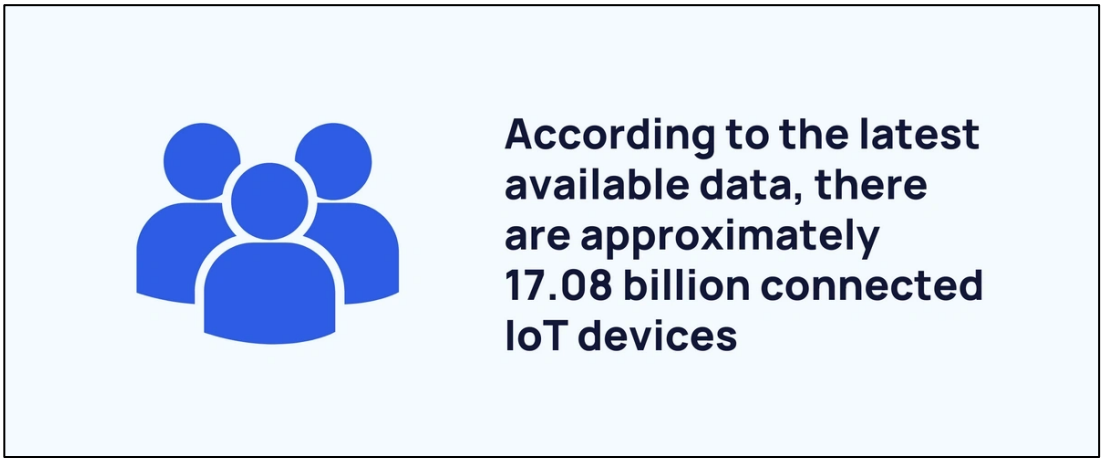


Figure 1: Number of IoT Devices (2024)

1.4. Objectives

This project aims to develop a comprehensive security framework that addresses these challenges through:

Primary Objective: Design and implement a secure IoT architecture that provides end-to-end encryption, device authentication, and threat detection capabilities.

Specific Technical Objectives:

- Implement network segmentation using VLAN technology to isolate IoT devices
- Deploy a honeypot system to detect and analyze unauthorized SSH access attempts
- Establish an intrusion detection system for real-time threat monitoring
- Create a private certificate authority for device authentication
- Demonstrate secure encrypted communications between IoT devices and cloud services
- Develop automated alerting mechanisms for security incidents

1.5. Methodology

The project follows a systematic approach combining network security, embedded systems programming, and cybersecurity principles:

1. **Infrastructure Setup**: Configuration of OpenWrt-based network segmentation and VLAN isolation
2. **Security Layer Implementation**: Deployment of honeypot (cowrie) and IDS (suricata) systems on Raspberry Pi
3. **Certificate Management**: Establishment of private CA (OpenSSL) and X.509 certificate-based authentication on Raspberry Pi
4. **Device Implementation**: Development of secure ESP32 code with encrypted communications
5. **Integration and Testing**: System integration with ThingsBoard IoT platform and security validation
6. **Performance Evaluation**: Analysis of security effectiveness and system performance

1.6. Report Structure

This report is organized to provide a comprehensive understanding of the implemented security framework.

Following this introduction, Chapter 2 reviews the theoretical foundations and related work. Chapter 3 presents the system architecture and design decisions. Chapter 4 details the implementation process and configuration steps. Chapter 5 analyzes the security effectiveness through testing and validation. Chapter 6 presents results and performance evaluation. Chapter 7 discusses limitations and future improvements, followed by conclusions in Chapter 8.

2. Literature Review and Theoretical Foundation

2.1. IoT Security Landscape

The massive scale of IoT deployments has rendered traditional perimeter-based security models obsolete, leading to the rise of the Zero-Trust Architecture (ZTA) as the new state-of-the-art **[3].**

Operating on the principle of "never trust, always verify," ZTA mandates continuous verification for every device and user.

This directly addresses the persistent vulnerabilities in commercial IoT, such as weak authentication and unencrypted communications, by enforcing key pillars like mutual authentication, strict access controls, and end-to-end encryption.

2.2. Network Security Technologies

2.2.1. Virtual Local Area Networks (VLANs)

VLAN technology provides logical network segmentation, isolating different types of devices even when they share the same physical infrastructure.

This approach is particularly effective for IoT security as it prevents lateral movement of threats between network segments.

2.2.2. Honeypots in Cybersecurity

Honeypots are deceptive security tools designed to attract and detect unauthorized access attempts.

Cowrie, the honeypot solution used in this project, emulates SSH and Telnet services to capture attacker behavior and techniques.

This provides valuable intelligence about threat actors targeting IoT environments.

2.2.3. Intrusion Detection Systems

Intrusion Detection Systems (IDS) monitor network traffic and system activities for malicious behavior.

Suricata, employed in this project, provides real-time threat detection through signature-based and anomaly-based detection mechanisms.

2.3. Cryptographic Foundations

2.3.1. Public Key Infrastructure (PKI)

Public Key Infrastructure provides the framework for managing digital certificates and public-private key pairs.

In IoT environments, PKI enables device authentication and secure communications through X.509 certificates.

2.3.2. Transport Layer Security (TLS)

TLS protocol provides secure communication channels over untrusted networks.

TLS 1.2, used in this implementation, offers strong encryption and authentication capabilities essential for IoT security.

2.4. IoT Communication Protocols

2.4.1. MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight messaging protocol commonly used in IoT applications. While efficient, standard MQTT implementations often lack encryption, making secure transport layers essential.

2.4.2. HTTP/HTTPS

HTTP remains prevalent in IoT communications, but HTTPS (HTTP over TLS) is necessary for secure data transmission in production environments.

2.5. Embedded Systems Security

Security in embedded systems like the ESP32 requires careful consideration of resource constraints, memory limitations, and power consumption.

The ESP32 platform provides sufficient computational power for cryptographic operations while maintaining energy efficiency.

3. System Architecture and Design

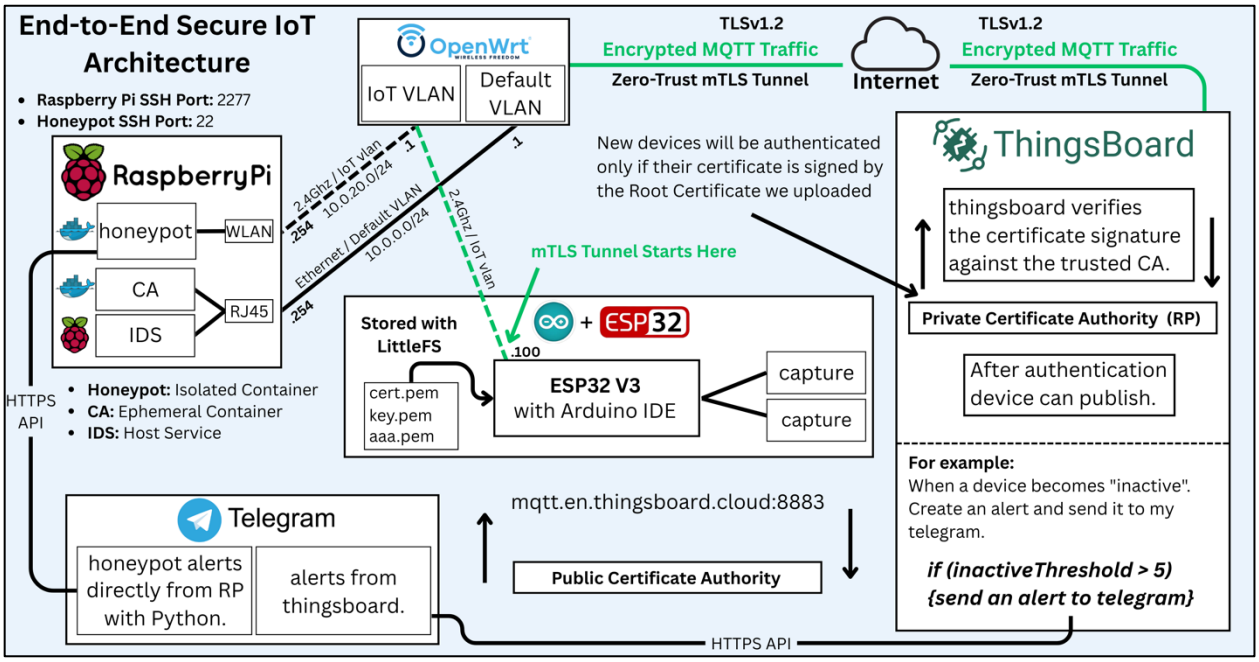


Figure 2: Final Architecture Diagram

3.1. Overall Architecture Overview

The implemented security framework follows a layered approach with multiple security controls working in concert.

The architecture consists of four primary components:

1. **Network Infrastructure Layer**: OpenWrt-based router providing VLAN segmentation and traffic management
2. **Security Monitoring Layer**: Raspberry Pi hosting honeypot, IDS and CA services
3. **Certificate Authority Layer**: Private CA for device authentication and certificate management
4. **IoT Device Layer**: ESP32 microcontroller with secure communications capability

3.2. Network Architecture Design

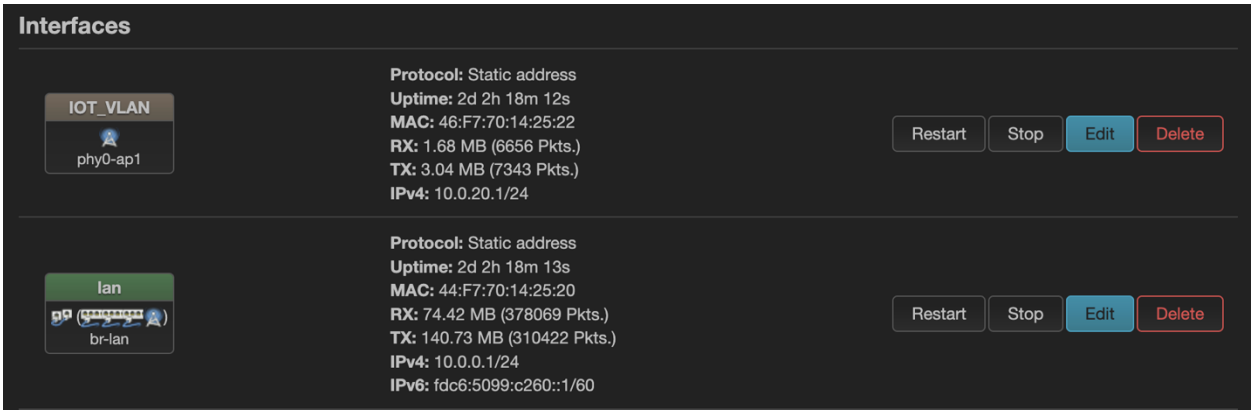


Figure 3: OpenWrt network interface configuration page

3.2.1. VLAN Segmentation Strategy

The network architecture implements strict segmentation using VLAN technology:

- **Default VLAN (10.0.0.0/24)**: Management and trusted devices
  - ***Router:*** 10.0.0.1
  - ***Raspberry Pi (eth0):*** 10.0.0.254
- **IoT VLAN (10.0.20.0/24)**: Isolated segment for IoT devices
  - ***Router:*** 10.0.20.1
  - ***Raspberry Pi (wlan0):*** 10.0.20.254
  - ***ESP32:*** 10.0.20.100
- **Wireless Isolation**: Separate SSID for IoT devices connecting to the isolated VLAN
  - ***Normal Clients:*** OpenWrt SSID → LAN Interface
  - ***IoT Devices:*** IOT_VLAN_SSID → IOT_VLAN Interface

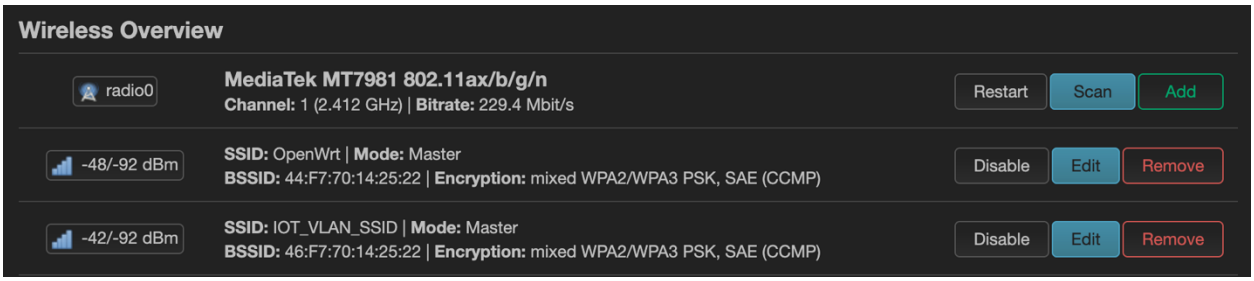


Figure 4: OpenWrt wireless configuration page

This segmentation prevents IoT devices from accessing critical network resources while maintaining necessary internet connectivity for cloud communications.

The 2.4 GHz radio frequency was selected to ensure broad compatibility, as the majority of IoT devices do not support the 5 GHz band.

3.2.2. Firewall and Access Control

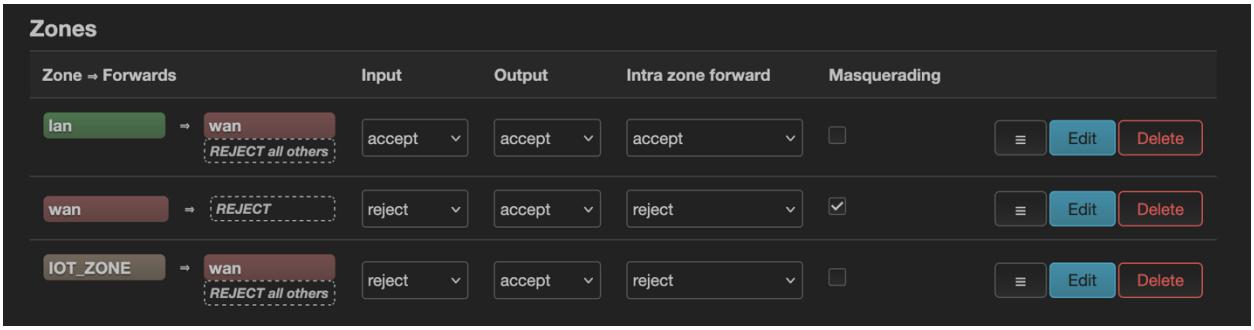


Figure 5: OpenWrt firewall configuration page

The OpenWrt firewall implements restrictive access controls:

- IoT devices cannot access the default VLAN
- Specific port forwarding rules for SSH
- Controlled internet access for IoT devices

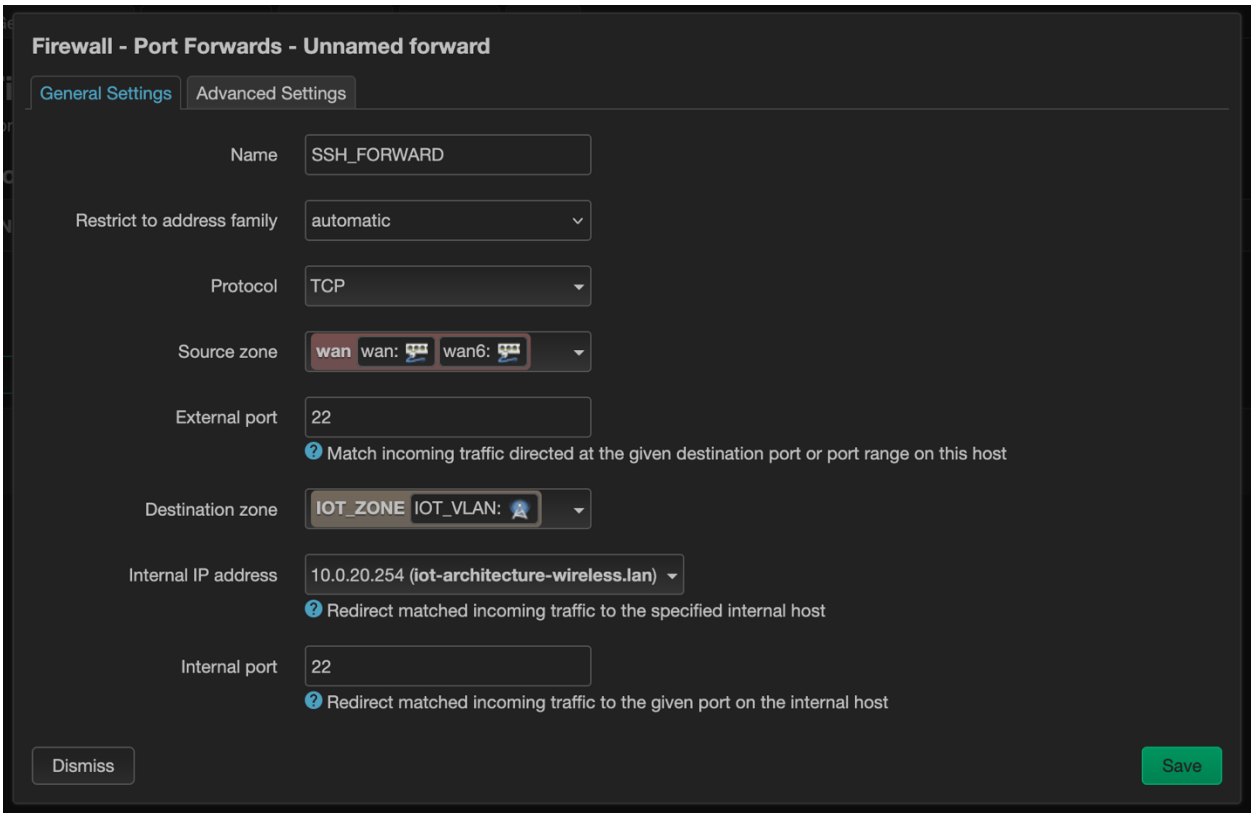


Figure 6: OpenWrt port forwarding configuration page

A port forwarding rule was configured on the OpenWrt router. This rule redirects all incoming traffic on the standard SSH port (22) from the internet directly to our honeypot.

3.3. Security Monitoring Architecture

3.3.1. Honeypot Design

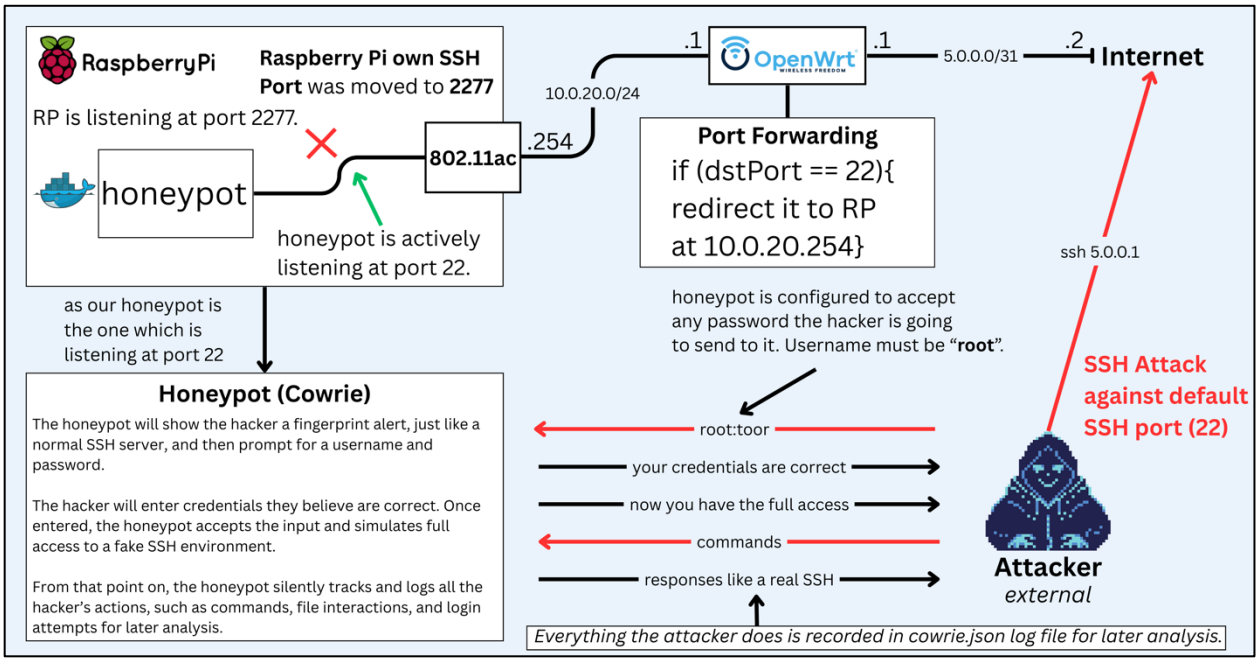


Figure 7: Honeypot Diagram

The Cowrie honeypot is strategically positioned to appear as a legitimate SSH server:

- **Port Configuration**: Listens on standard SSH port 22
- **Deception Strategy**: Mimics real SSH server behavior
- **Logging Capabilities**: Captures all attacker interactions for analysis
- **Isolation**: Runs in Docker container for security and management

The container was configured to map the host's port 22 to the honeypot's internal port, effectively turning the Pi's default SSH port into a trap for attackers.

A restart policy was set to ensure the container always runs on boot.

3.3.2. Intrusion Detection System

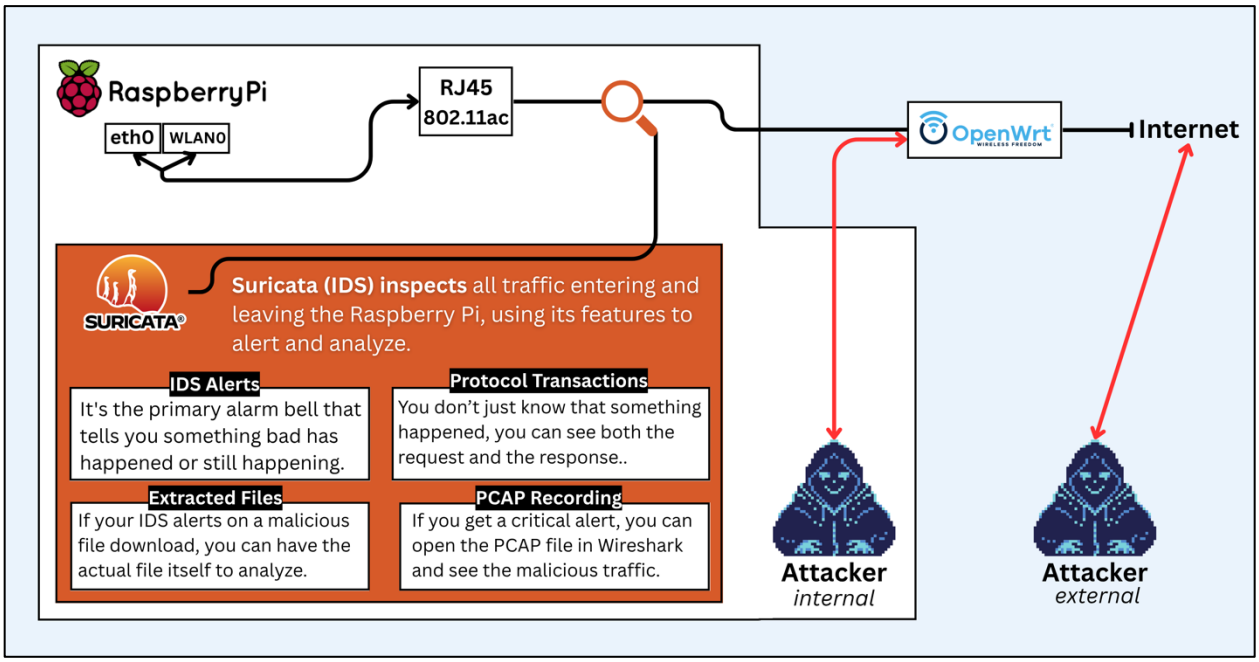


Figure 8: IDS Diagram

Suricata IDS provides comprehensive network monitoring:

- **Traffic Analysis**: Monitors all network traffic entering and leaving the Raspberry Pi
- **Signature Detection**: Uses rule-based detection for known threats
- **Protocol Analysis**: Deep packet inspection for protocol anomalies
- **Alerting**: Real-time alerts for detected threats

Suricata was installed directly on the host OS for performance. It was configured in */etc/suricata/suricata.yaml* to monitor the wlan0/eth0 interfaces, allowing it to inspect all traffic coming from the IoT VLAN and default VLAN.

The rule set was managed and updated, and the service was enabled to run automatically.

3.4. Certificate Authority Design

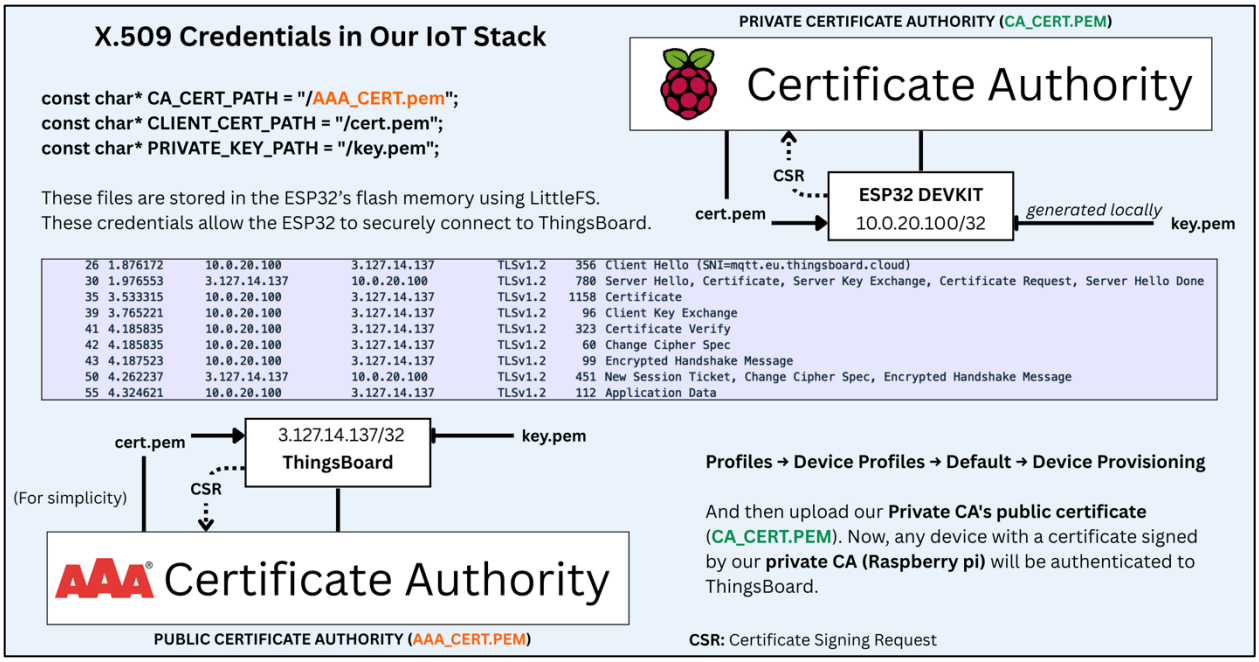


Figure 9: Certificate Authority Diagram

3.4.1. Private CA Implementation

The private certificate authority provides device authentication:

- **Root Certificate**: Self-signed root certificate for the private CA
- **Certificate Signing**: CSR processing for device certificates
- **Key Management**: Secure storage and management of private keys
- **Ephemeral Operation**: CA runs only when needed to minimize exposure

3.5. IoT Device Security Architecture

3.5.1. On-Device Key and CSR Generation

The first step was to upload an initial Arduino sketch to the ESP32 that used cryptographic libraries to generate a new private key and a Certificate Signing Request (CSR) directly on the device.

This ensures the device's private key never leaves the device itself.

“A private key should not leave the device on which it was generated.”

3.5.2. Certificate Signing and Provisioning

The CSR was sent from the ESP32 to the PC.

Using SFTP, the CSR was transferred to the Raspberry Pi. (It is okay to use FTP, as the CSR isn’t something you need to make it secure.)

Figure 10: Private and CSR Keys

The offline CA Docker container was started,
and the CSR was signed using the root CA's private key, creating a valid device certificate. This signed certificate was then transferred back to the PC.

Private key and CSR generation:

Complete Code:
https://drive.google.com/file/d/1IotO680321UBZonAp2G04diVIqwu_dMO/view?usp=sharing

Short URL: bit.ly/3IC3Q18

3.5.4. Uploading Credentials

```
Building LittleFS filesystem
Command Line: /Users/radouanee/Library/Arduino15/packages/esp32/tools/mklittlefs/3.0.0-gnu12-dc7f933/mklittlefs -
-s 1441792 /var/folders/wj/m1sg2vc57n99nnmx2cb374x80000gn/T/tmp-40667-JisTZQOr0bXk-.littlefs.bin
/key.pem
/cert.pem
/ca-root.pem

Uploading LittleFS filesystem
Command Line: /Users/radouanee/Library/Arduino15/packages/esp32/tools/esptool_py/4.9.dev3/esptool --chip esp32 --
_reset write_flash -z --flash_mode dio --flash_freq 80m --flash_size detect 2686976 /var/folders/wj/m1sg2vc57n99n
esptool.py v4.8.1
Serial port /dev/cu.usbserial-10
Connecting.......
Chip is ESP32-D0WD-V3 (revision v3.1)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
WARNING: Detected crystal freq 41.01MHz is quite different to normalized freq 40MHz. Unsupported crystal in use?
Crystal is 40MHz
MAC: 78:1c:3c:b8:6f:48
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Flash will be erased from 0x00290000 to 0x003effff...
Compressed 1441792 bytes to 5767...
Writing at 0x00290000... (100 %)
Wrote 1441792 bytes (5767 compressed) at 0x00290000 in 4.2 seconds (effective 2776.8 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...

Completed upload.
```

Figure 11: Upload Certificates

The signed device certificate (`cert.pem`), the CA's public root certificate (`ca-root.pem`) *(please note that ca-root.pem is also called aaa.pem in my diagrams, as the aaa.pem is the last name i used for it),* and the device's private key (`key.pem`) were uploaded to the ESP32's flash memory using the LittleFS filesystem plugin in the Arduino IDE.

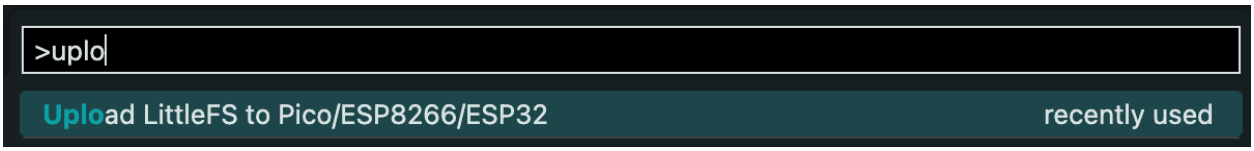


Figure 12: LittleFS Plugin

3.5.5. Final Code

The final Arduino sketch was uploaded, which reads these three files from LittleFS to establish a secure mTLS connection with the ThingsBoard cloud platform.

You will find the complete code in the (4.5.3. Secure Communication Implementation) section.

3.5.6. Communication Security

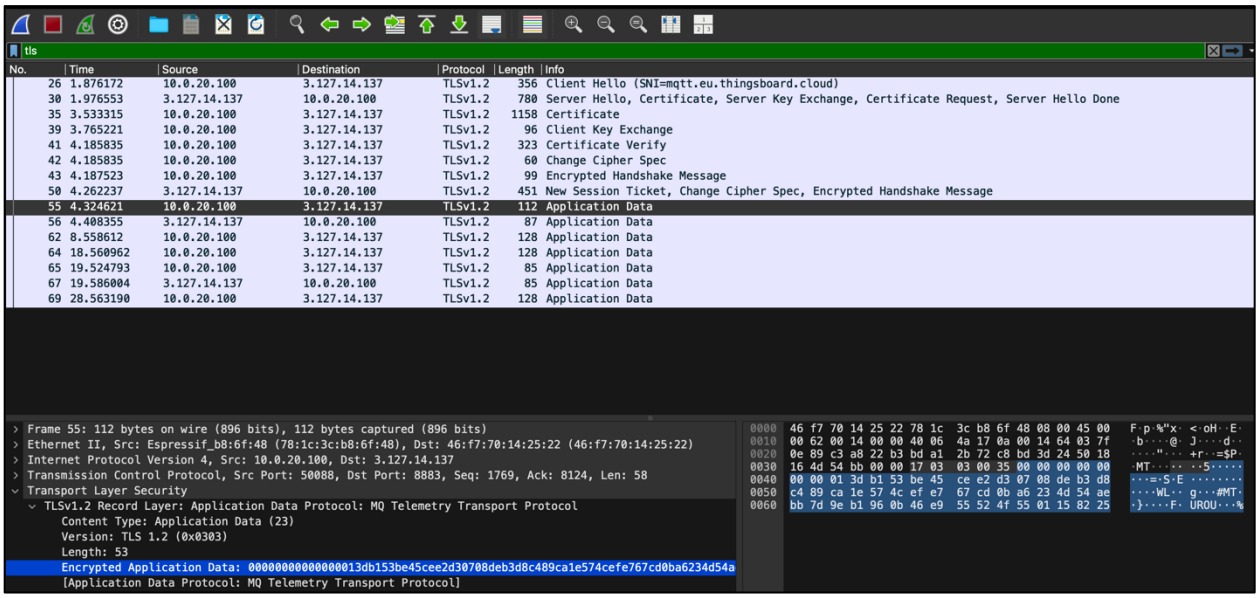


Figure 13: MQTT Over TLS

The device implements secure communication protocols:

- **Certificate-Based Authentication**: X.509 certificates for device identity
- **Encrypted Transport**: TLS 1.2 for all communications
- **Protocol Security**: Secure MQTT over TLS tunnel

3.6. Integration with ThingsBoard

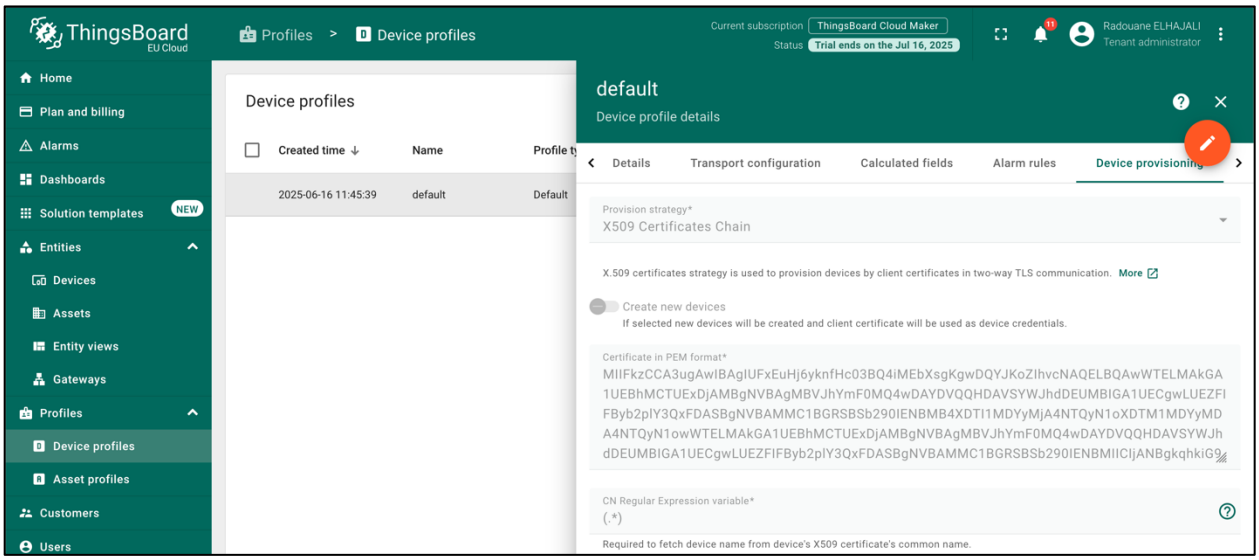


Figure 14: ThingsBoard (Device Provisionning)

3.6.1. Cloud Platform Integration (ThingsBoard)

- **Device Profile Configuration:** A new device profile was created in ThingsBoard. In the "Device Provisioning" section, the "Allow to create new devices from X.509 certificate." rule was set to off.
- **Trusting the CA:** The public certificate of our private Raspberry Pi CA (`ca-root.pem`) was uploaded to ThingsBoard. This tells ThingsBoard to trust any device whose certificate has been signed by our CA.
- **Device Creation:** A new device was created. For the credentials, the "X.509 Certificate" type was selected. This configured the device to authenticate using the certificate it presents during the TLS handshake, rather than a simple access token.

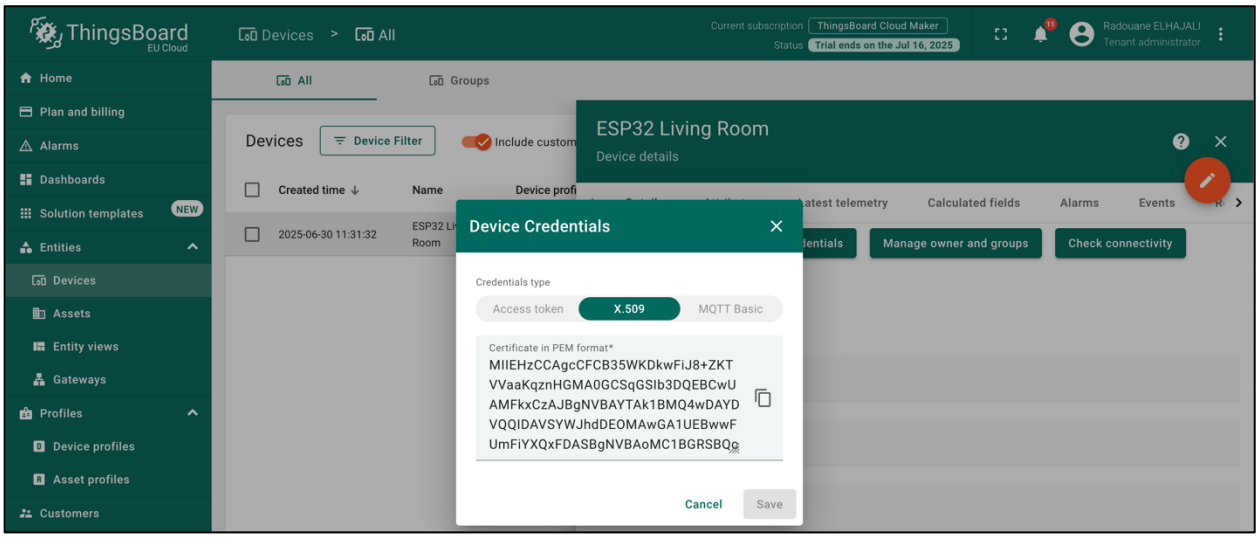


Figure 15: Device Certificate

4. Implementation and Configuration

4.1. Infrastructure Setup

4.1.1. OpenWrt Router Configuration

The project begins with flashing the Xiaomi AX3000T router with OpenWrt firmware. This step is crucial as it replaces the manufacturer's firmware with an open-source alternative that provides advanced networking capabilities.

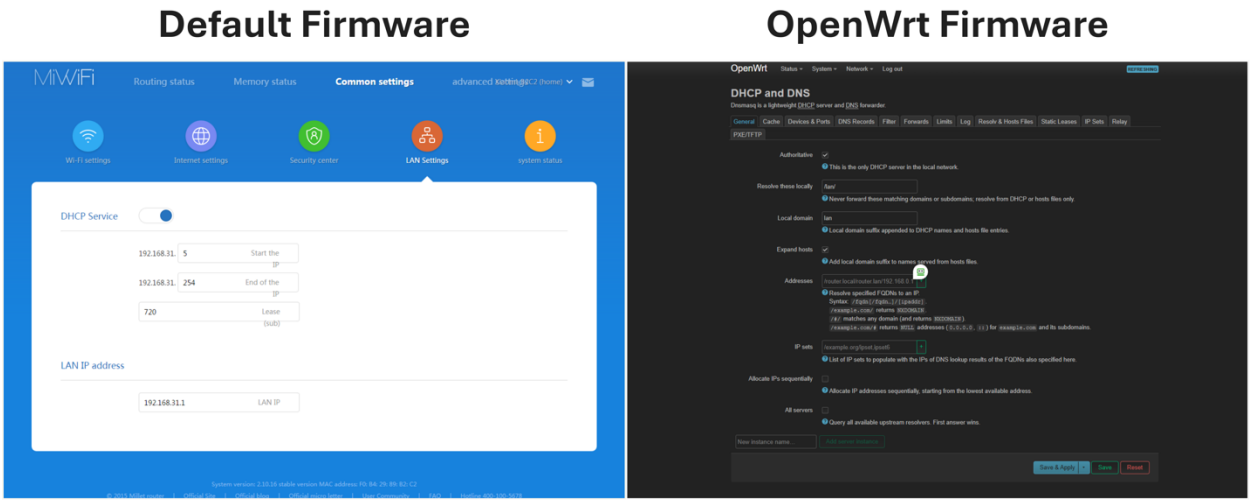


Figure 16: Flashing OpenWrt Firmware

Important Considerations for OpenWrt Installation:

- Not all routers support OpenWrt firmware
- Compatibility must be verified on the OpenWrt hardware compatibility database **[4]**
- The installation process may void warranty and carries risk of device failure
- Proper recovery procedures should be established before installation

Router Configuration Process:

1. **Firmware Installation**: Flash OpenWrt 24.10.0 to Xiaomi AX3000T **[5]**
2. **Initial Setup**: Configure basic network settings and administrative access
3. **Interface Configuration**: Set up multiple network interfaces for VLAN support
4. **Wireless Configuration**: Create separate wireless networks for different VLANs

4.1.2. VLAN Configuration

The VLAN implementation provides network segmentation essential for IoT security:

Default VLAN Configuration (10.0.0.0/24):

- Purpose: Management and trusted devices
- Gateway: 10.0.0.1
- DHCP Range: 10.0.0.150-10.0.0.200
- Access: Full network access

IoT VLAN Configuration (10.0.20.0/24):

- Purpose: IoT devices isolation
- Gateway: 10.0.20.1
- DHCP Range: 10.0.20.150-10.0.20.200
- Access: Restricted to internet and specific services, and more importantly can't reach the devices in default VLAN.

Wireless Network Configuration:

- **IoT_VLAN_SSID**: Dedicated wireless network for IoT devices
- **Security**: WPA2/WPA3 encryption with strong passphrase
- **Isolation**: Connected devices isolated from default VLAN

Hostname	MAC address(es)	IPv4 address	Lease time	DUID	IPv6-Suffix (hex)	Tag	Match Tag	
iot-architecture	2C:CF:67:8B:FF:86	10.0.0.254	infinite	*none*	*none*	*none*	*none*	≡ Edit Delete
iot-architecture-wireless	2C:CF:67:8B:FF:88	10.0.20.254	infinite	*none*	*none*	*none*	*none*	≡ Edit Delete
ESP32-Device-01	78:1C:3C:B8:6F:48	10.0.20.100	*none*	*none*	*none*	*none*	*none*	≡ Edit Delete

Figure 17: DHCP Reservation

4.1.3. Firewall and Port Forwarding

The OpenWrt firewall implements security policies:

Firewall Rules:

- Block inter-VLAN communication
- Allow IoT devices to access internet
- Permit specific services (SSH, HTTPS)
- Log suspicious activities

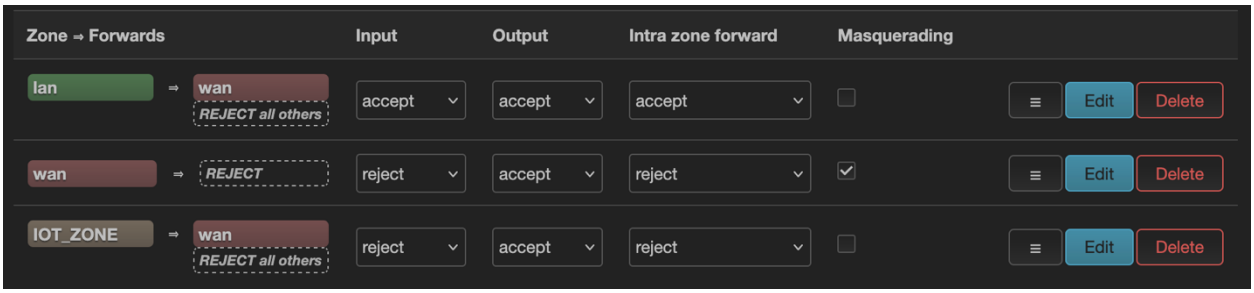


Figure 18: Firewall Zones

Port Forwarding Configuration:

- Port 22 → Raspberry Pi (10.0.20.254:22) for honeypot
- Port 2277 → Raspberry Pi (10.0.0.254:2277) for legitimate SSH access
- Restricted access to management interfaces

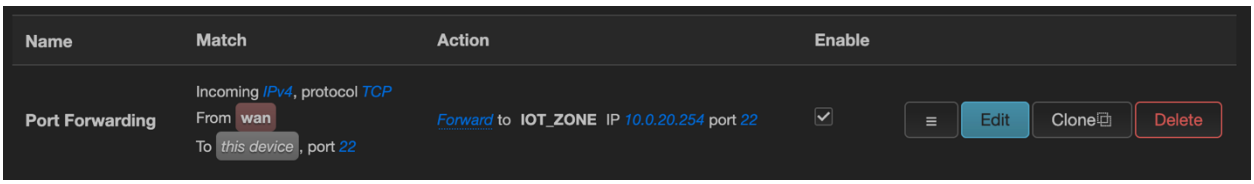


Figure 19: Port Forwarding

4.2. Raspberry Pi Security Hub Setup

4.2.1. Operating System Installation

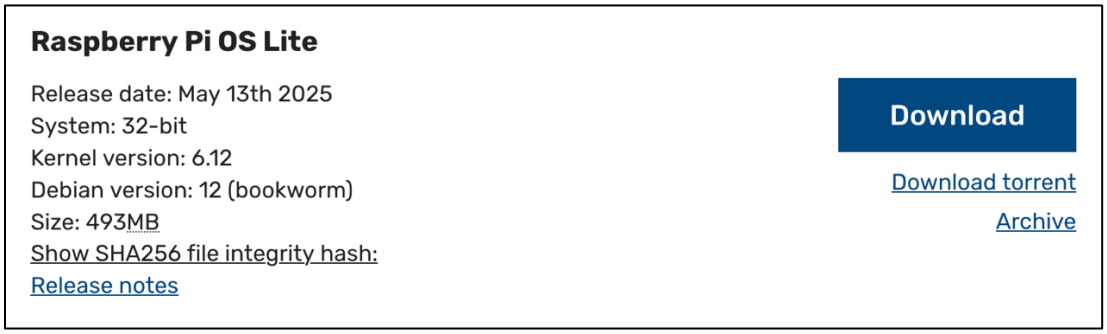


Figure 20: Raspberry Pi OS

The Raspberry Pi serves as the central security hub, requiring careful configuration:

Initial Setup Process:

1. **OS Installation**: Install Raspberry Pi OS Lite for minimal resource usage
2. **System Updates**: Update system packages and security patches
3. **Network Configuration**: Configure static IP addresses for both interfaces
4. **Security Hardening**: Implement basic security measures

1. Update the system to the latest packages

sudo apt update && sudo apt full-upgrade -y

2. Install all other necessary software packages

sudo apt install suricata python3-requests jq -y

3. Reboot the system to apply all changes

sudo reboot

Network Interface Configuration:

- **Eth0**: 10.0.0.254 (Management interface)
- **WLAN0**: 10.0.20.254 (IoT VLAN interface)
- **SSH Port**: Changed to 2277 for security

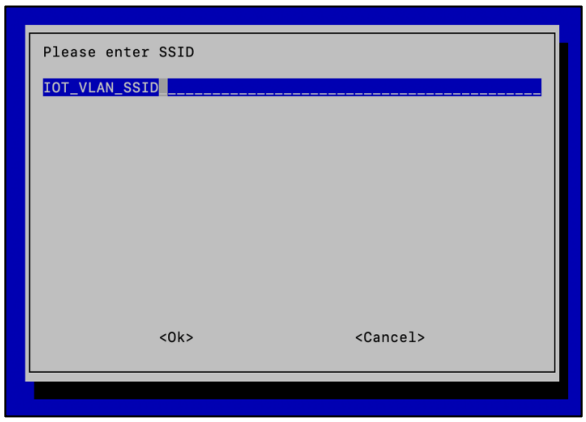


Figure 21: WLAN0 Configuration

```
radouane@iot-architecture:~ $ ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 9a:ce:ca:4d:3a:4d  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 8 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.254  netmask 255.255.255.0  broadcast 10.0.0.255
        inet6 fdc6:5099:c260:0:1caa:7efb:da7:e8d9  prefixlen 64  scopeid 0x0<global>
        inet6 fdc6:5099:c260::bb2  prefixlen 128  scopeid 0x0<global>
        inet6 fe80::c59b:8c60:93c6:700b  prefixlen 64  scopeid 0x20<link>
        ether 2c:cf:67:8b:ff:86  txqueuelen 1000  (Ethernet)
        RX packets 114240  bytes 116545430 (111.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 57033  bytes 4433937 (4.2 MiB)
        TX errors 0  dropped 25 overruns 0  carrier 0  collisions 0
        device interrupt 106

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 10  bytes 1630 (1.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 10  bytes 1630 (1.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.20.254  netmask 255.255.255.0  broadcast 10.0.20.255
        inet6 fe80::bf57:1e38:27d4:aae5  prefixlen 64  scopeid 0x20<link>
        ether 2c:cf:67:8b:ff:88  txqueuelen 1000  (Ethernet)
        RX packets 1836  bytes 148062 (144.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 198  bytes 20499 (20.0 KiB)
        TX errors 0  dropped 5 overruns 0  carrier 0  collisions 0
```

Figure 22: Raspberry Pi IP Configuration

```
# Changing Raspberry Pi SSH Port

sudo nano /etc/ssh/sshd_config

# Before:

Port 22

# After:

Port 2277

# Restart SSH service
sudo systemctl restart sshd

# Next SSH login should be like this

ssh username@raspberrypi.local -p 2277
```

4.2.2. Docker Installation and Configuration

```
radouane@iot-architecture:~ $ sudo netstat -tulpn | grep sshd
tcp        0      0 0.0.0.0:2277            0.0.0.0:*               LISTEN      689/sshd: /usr/sbin
tcp6       0      0 :::2277                 :::*                    LISTEN      689/sshd: /usr/sbin
radouane@iot-architecture:~ $ docker stop honeypot-container
docker rm honeypot-container
honeypot-container
honeypot-container
radouane@iot-architecture:~ $ docker run -d \
  --name honeypot-container \
  -p 22:2222 \
  -p 23:2223 \
  cowrie/cowrie
85cc73fca8ff6759160ae8c900c0834b7c327341e91adb415bd21116f9d6ebb6
radouane@iot-architecture:~ $ sudo netstat -tulpn | grep :22
tcp        0      0 0.0.0.0:22277           0.0.0.0:*               LISTEN      689/sshd: /usr/sbin
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1201/docker-proxy
tcp6       0      0 ::22277                 :::*                    LISTEN      689/sshd: /usr/sbin
tcp6       0      0 ::22                    :::*                    LISTEN      1209/docker-proxy
radouane@iot-architecture:~ $
```

Figure 23: Honeypot Container setup

Install Docker using the official convenience script

```
curl -fsSL https://get.docker.com -o get-docker.sh

sudo sh get-docker.sh

sudo usermod -aG docker ${USER}
```

Docker provides isolation for security services:

Docker Setup:

1. **Installation**: Install Docker Engine and Docker Compose
2. **Service Configuration**: Configure Docker to start on boot
3. **Network Configuration**: Set up Docker networks for service isolation
4. **Security**: Implement Docker security best practices

Container Security:

- Run containers with minimal privileges
- Use read-only filesystems where possible
- Implement resource limits
- Regular container updates

4.2.3. Certificate Authority Implementation

The private certificate authority provides device authentication:

Figure 24: Generate the CA's Private key

CA Setup Process:

1. **OpenSSL Installation**: Install OpenSSL for certificate operations
2. **Root Certificate Generation**: Create self-signed root certificate
3. **Private Key Security**: Secure storage of CA private key

The CA configuration is shown in ***Appendix A****.*

Docker Container for CA: The CA runs in an ephemeral Docker container that starts only when needed for certificate signing operations, minimizing security exposure.

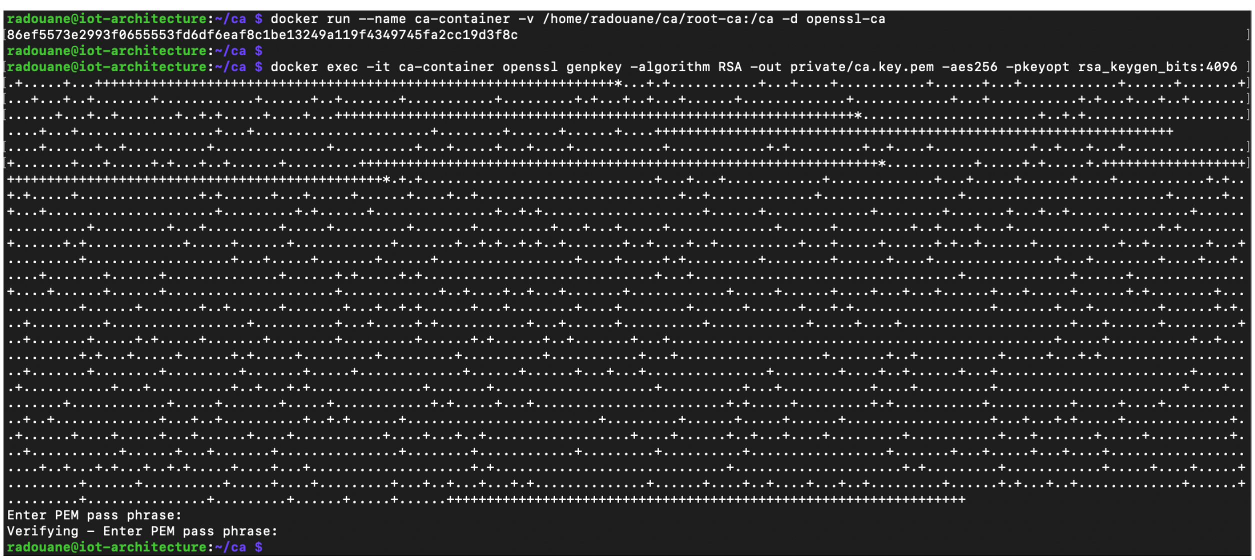


Figure 25: Generate the CA's Public certificate

```
radouane@iot-architecture:~/ca $ ls -lR ~/ca/root-ca
/home/radouane/ca/root-ca:
total 8
drwxr-xr-x 2 radouane radouane 4096 Jun 22 09:54 certs
drwxr-xr-x 2 radouane radouane 4096 Jun 22 09:48 private

/home/radouane/ca/root-ca/certs:
total 4
-rw-r--r-- 1 root root 1992 Jun 22 09:54 ca.cert.pem

/home/radouane/ca/root-ca/private:
total 4
-rw------- 1 root root 3434 Jun 22 09:51 ca.key.pem
radouane@iot-architecture:~/ca $
radouane@iot-architecture:~/ca $
radouane@iot-architecture:~/ca $ docker stop ca-container
ca-container
radouane@iot-architecture:~/ca $ docker rm ca-container
ca-container
```

Figure 26: CA Full Setup

4.2.4. Why Should You Do rm?

This is a critical security principle: **Minimizing Exposure.**

Your CA's private key is the most valuable secret in your entire project. You want that key to be "active" or "in use" for the absolute shortest amount of time possible.

- **If you leave the container running:** It's a live, running environment on your network. It has an IP address. While isolated by Docker, it's still a potential (though very small) target. It also might contain traces of your activity in its internal logs or command history.
- **When you destroy the container:** You are returning to a "secure by default" state. The environment where the key was used is gone. There are no logs, no history, no running processes associated with it.

4.3. Honeypot Implementation

4.3.1. Cowrie Honeypot Configuration

Cowrie provides an interactive SSH honeypot that captures attacker behavior:

Cowrie Features:

- **SSH Emulation**: Mimics real SSH server behavior
- **Session Recording**: Records all attacker interactions
- **Credential Capture**: Logs attempted usernames and passwords
- **Command Logging**: Captures all commands executed by attackers

Honeypot configuration is shown in **Appendix B**.

```
radouane@iot-architecture:~ $ sudo nano /etc/systemd/system/honeypot-alerter.service
radouane@iot-architecture:~ $
radouane@iot-architecture:~ $ sudo systemctl daemon-reload
radouane@iot-architecture:~ $ sudo systemctl enable honeypot-alerter.service
Created symlink /etc/systemd/system/multi-user.target.wants/honeypot-alerter.service → /etc/systemd/system/honeypot-alerter.service.
radouane@iot-architecture:~ $ sudo systemctl start honeypot-alerter.service
radouane@iot-architecture:~ $ sudo systemctl status honeypot-alerter.service
● honeypot-alerter.service - Honeypot Real-time Telegram Alerter
     Loaded: loaded (/etc/systemd/system/honeypot-alerter.service; enabled; preset: enabled)
     Active: active (running) since Mon 2025-06-23 12:43:13 +01; 46s ago
   Main PID: 2069 (sh)
      Tasks: 11 (limit: 9585)
        CPU: 290ms
     CGroup: /system.slice/honeypot-alerter.service
             ├─2069 /bin/sh -c "/usr/bin/docker logs -f honeypot-container | /usr/bin/python3 /home/radouane/telegram_alerter.py"
             ├─2070 /usr/bin/docker logs -f honeypot-container
             └─2071 /usr/bin/python3 /home/radouane/telegram_alerter.py

Jun 23 12:43:13 iot-architecture systemd[1]: Started honeypot-alerter.service - Honeypot Real-time Telegram Alerter.
Jun 23 12:43:13 iot-architecture sh[2070]: /cowrie/cowrie-env/lib/python3.11/site-packages/twisted/conch/ssh/transport.py:105: Cryptog
Jun 23 12:43:13 iot-architecture sh[2070]:   b"3des-cbc": (algorithms.TripleDES, 24, modes.CBC),
Jun 23 12:43:13 iot-architecture sh[2070]: /cowrie/cowrie-env/lib/python3.11/site-packages/twisted/conch/ssh/transport.py:112: Cryptog
Jun 23 12:43:13 iot-architecture sh[2070]:   b"3des-ctr": (algorithms.TripleDES, 24, modes.CTR),

radouane@iot-architecture:~ $
```

Figure 27: Honeypot Configuration and Making it Run in Background

Integration with Telegram: The honeypot includes integration with Telegram for real-time alerting:

- **Bot Configuration**: Telegram bot for receiving alerts
- **Alert Triggers**: Notifications for new connections and suspicious activities
- **Message Format**: Structured alerts with attacker information

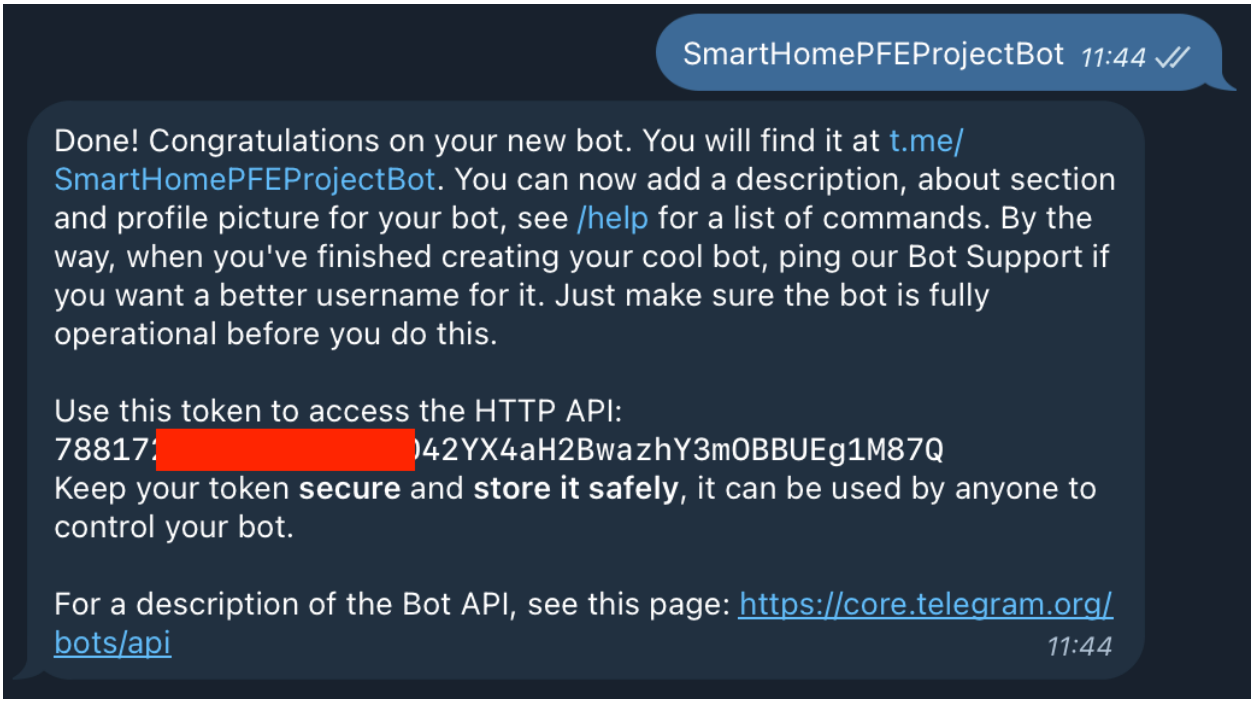


Figure 28: Telegram Bot Creation

This is the script to send honeypot alerts to my Telegram bot I just created. For confidentiality, I hided my credintials.

```
import json
import requests
import subprocess
import time
from collections import defaultdict
import os
import sys

# --- CONFIGURATION ---
TELEGRAM_BOT_TOKEN = "78817*********O42YX4aH2BwazhY3mOBBUEg1M87Q"
TELEGRAM_CHAT_ID = "50226*****"
POLL_INTERVAL_SECONDS = 10 # Check for new attacks every 10 seconds
CONTAINER_NAME = "honeypot-container"
# This is where we will temporarily copy the log file for analysis
TEMP_LOG_PATH = os.path.expanduser("~/temp_honeypot_log.json")
# --- END CONFIGURATION ---

processed_sessions = set()

def get_country_from_ip(ip_address):
    """Looks up the country for a given IP address."""
    if not ip_address or ip_address.startswith(('172.', '192.168.', '10.', '127.')):
        return "Internal/Private IP"
    try:
        response = requests.get(f"http://ip-api.com/json/{ip_address}?fields=country,query", timeout=5)
        response.raise_for_status()
        data = response.json()
        return data.get('country', 'N/A')
    except requests.exceptions.RequestException:
        return "API Error/Offline"

def send_telegram_alert(session_details, session_id):
    """Formats and sends a complete session summary to the Telegram bot."""
    print(f"\nINFO: New completed session detected! Preparing alert for {session_id[:12]}...")

    # This is safer, using .get() provides a default if a key is missing
    commands_text = "\n".join(f"  - `{cmd}`" for cmd in session_details.get('commands', []))
    if not commands_text:
        commands_text = "  - (No commands were run or captured)"

    # Get country info right before sending
    country = get_country_from_ip(session_details.get('src_ip'))

    # Assemble the final message using .get() for safety
    message = (
        f"🚨 *Honeypot Alert: Session Closed* 🚨\n\n"
```

Full code: https://drive.google.com/file/d/1N1sJrFARN8tNAnxNOqSVrZKhm2Y4xdxd/view?usp=sharing

Short URL: bit.ly/4lnbECs

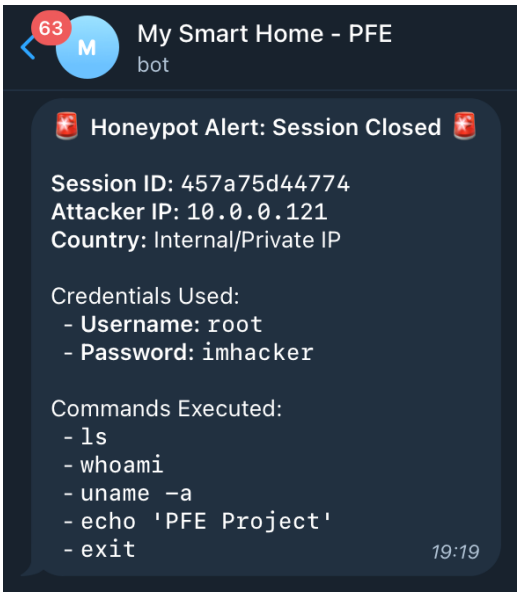


Figure 29: Honeypot Alerts in Telegram

4.3.2. Honeypot Monitoring

The system includes monitoring capabilities for honeypot activities:

Log Analysis:

- **Connection Logs**: Track connection attempts and source IPs
- **Command Analysis**: Analyze attacker commands and techniques
- **Credential Harvesting**: Collect attempted login credentials

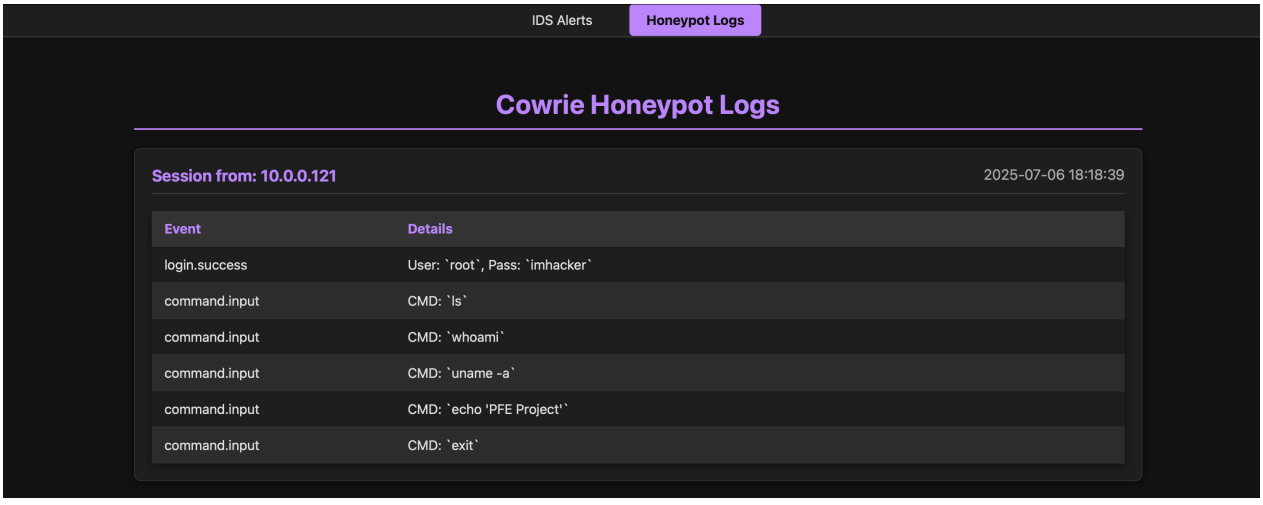


Figure 30: Honeypot Logs in Flask Dashboard

4.4. Intrusion Detection System

4.4.1. Suricata IDS Configuration

Suricata provides comprehensive network monitoring and threat detection:

Installation and Configuration:

1. **Package Installation**: Install Suricata from official repositories
2. **Interface Configuration**: Configure monitoring interfaces
3. **Rule Management**: Install and update detection rules

```
# Install Suricata
sudo apt install suricata -y

# Tell Suricata to listen on the IOT subnet
sudo nano /etc/suricata/suricata.yaml

HOME_NET: "[10.0.20.0/24]"
interface: wlan0

# Update Suricata Rules
sudo suricata-update

# And that is it, Suricata already comes with its rules, you just need to
install and activate it.
```

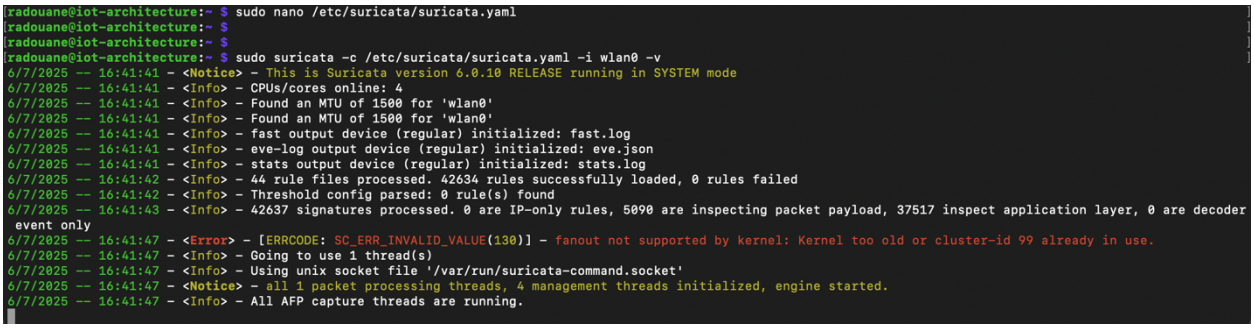


Figure 31: Runing IDS in Terminal

Monitoring Capabilities:

- **Protocol Analysis**: Deep packet inspection for various protocols
- **Signature Detection**: Rule-based detection of known threats
- **Anomaly Detection**: Statistical analysis for unusual behavior
- **File Extraction**: Extraction of suspicious files for analysis

```
 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://suricata.readthedocs.io/en/latest/configuration/suricata-yaml.html
#
# This configuration file generated by:
#     Suricata 6.0.10

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

vars:
  # more specific is better for alert accuracy and performance
  address-groups:
    HOME_NET: "[10.0.20.0/24, 10.0.0.0/24]"
    #HOME_NET: "[192.168.0.0/16]"
    #HOME_NET: "[10.0.0.0/8]"
    #HOME_NET: "[172.16.0.0/12]"
    #HOME_NET: "any"

    EXTERNAL_NET: "!$HOME_NET"
    #EXTERNAL_NET: "any"
```

Figure 32: IDS Configuration

```
[radouane@iot-architecture:~ $ sudo systemctl status suricata.service
● suricata.service - Suricata IDS/IDP daemon
     Loaded: loaded (/lib/systemd/system/suricata.service; enabled; preset: enabled)
     Active: active (running) since Fri 2025-07-11 13:29:33 +01; 14min ago
       Docs: man:suricata(8)
             man:suricatasc(8)
             https://suricata-ids.org/docs/
    Process: 3659 ExecStart=/usr/bin/suricata -D --af-packet -c /etc/suricata/suricata.yaml --pidfile /run/suricata.pid (code=exite>
   Main PID: 3660 (Suricata-Main)
      Tasks: 10 (limit: 9585)
        CPU: 9.463s
     CGroup: /system.slice/suricata.service
             └─3660 /usr/bin/suricata -D --af-packet -c /etc/suricata/suricata.yaml --pidfile /run/suricata.pid

Jul 11 13:29:33 iot-architecture systemd[1]: Starting suricata.service - Suricata IDS/IDP daemon...
Jul 11 13:29:33 iot-architecture suricata[3659]: 11/7/2025 -- 13:29:33 - <Notice> - This is Suricata version 6.0.10 RELEASE running>
Jul 11 13:29:33 iot-architecture systemd[1]: Started suricata.service - Suricata IDS/IDP daemon.
```

Figure 33: Running IDS in Background

4.4.2. Alert Management

The IDS includes comprehensive alert management:

Alert Classification:

- **Priority Levels**: Critical, high, medium, low priority alerts
- **Category Classification**: Malware, intrusion attempts, policy violations

Integration with Monitoring:

- **Log Aggregation**: Centralized logging for all security events
- **Visualization**: Web-based dashboard for alert monitoring

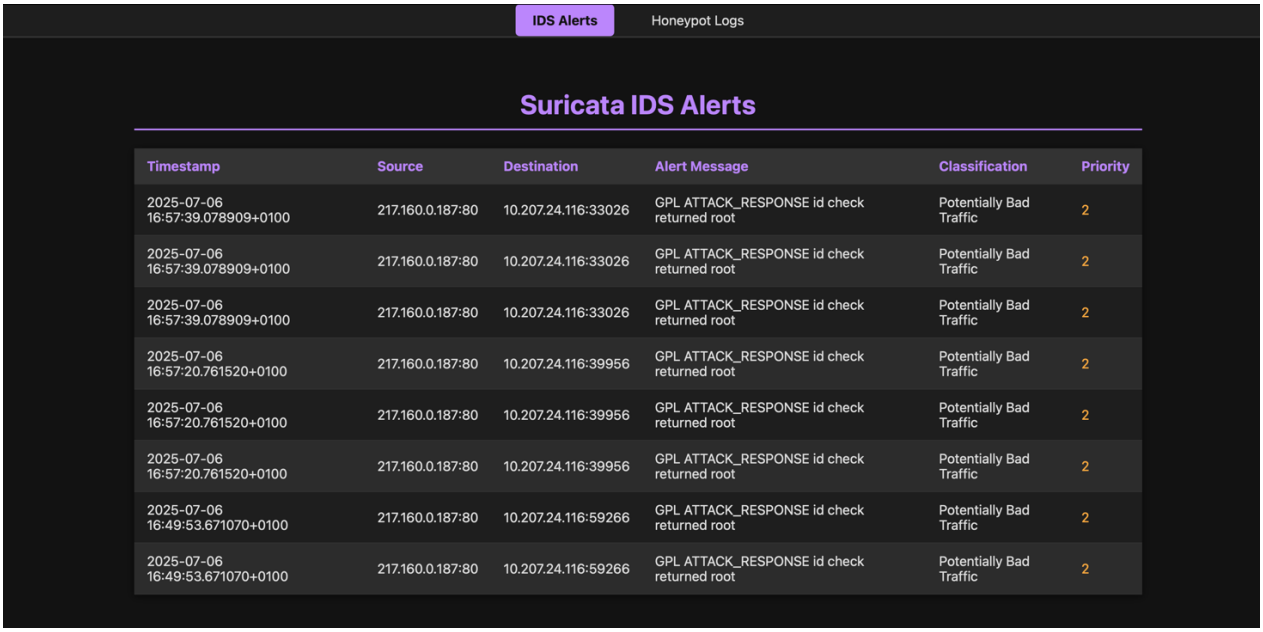


Figure 34: Logs Dashboard

```
[radouane@iot-architecture:~ $ sudo systemctl status dashboard.service
● dashboard.service - Unified Security Dashboard
     Loaded: loaded (/etc/systemd/system/dashboard.service; enabled; preset: enabled)
     Active: active (running) since Thu 2025-07-10 11:22:28 +01; 1 day 2h ago
   Main PID: 1222 (python3)
      Tasks: 1 (limit: 9585)
        CPU: 7.758s
     CGroup: /system.slice/dashboard.service
             └─1222 /home/radouane/flask_env/bin/python3 /home/radouane/suricata_dashboard.py

Jul 11 13:21:33 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:21:33] "GET / HTTP/1.1" 200 -
Jul 11 13:28:41 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:28:41] "GET / HTTP/1.1" 200 -
Jul 11 13:28:43 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:28:43] "GET /honeypot HTTP/1.1" 200 -
Jul 11 13:28:45 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:28:45] "GET / HTTP/1.1" 200 -
Jul 11 13:28:47 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:28:47] "GET / HTTP/1.1" 200 -
Jul 11 13:28:48 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:28:48] "GET / HTTP/1.1" 200 -
Jul 11 13:28:48 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:28:48] "GET / HTTP/1.1" 200 -
Jul 11 13:28:52 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:28:52] "GET / HTTP/1.1" 200 -
Jul 11 13:29:54 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:29:54] "GET / HTTP/1.1" 200 -
Jul 11 13:35:32 iot-architecture python3[1222]: 10.0.0.121 - - [11/Jul/2025 13:35:32] "GET / HTTP/1.1" 200 -
```

Figure 35: Running Flask Code in Background

4.5. ESP32 Secure Device Implementation

4.5.1. Development Environment Setup

The ESP32 development requires proper toolchain configuration:

Arduino IDE Configuration:

1. **Board Manager**: Install ESP32 board support package
2. **Library Installation**: Install required libraries for TLS and MQTT
3. **Upload Settings**: Configure upload speed and flash settings

```
# 1. Board Manager (File > Preferences)
```

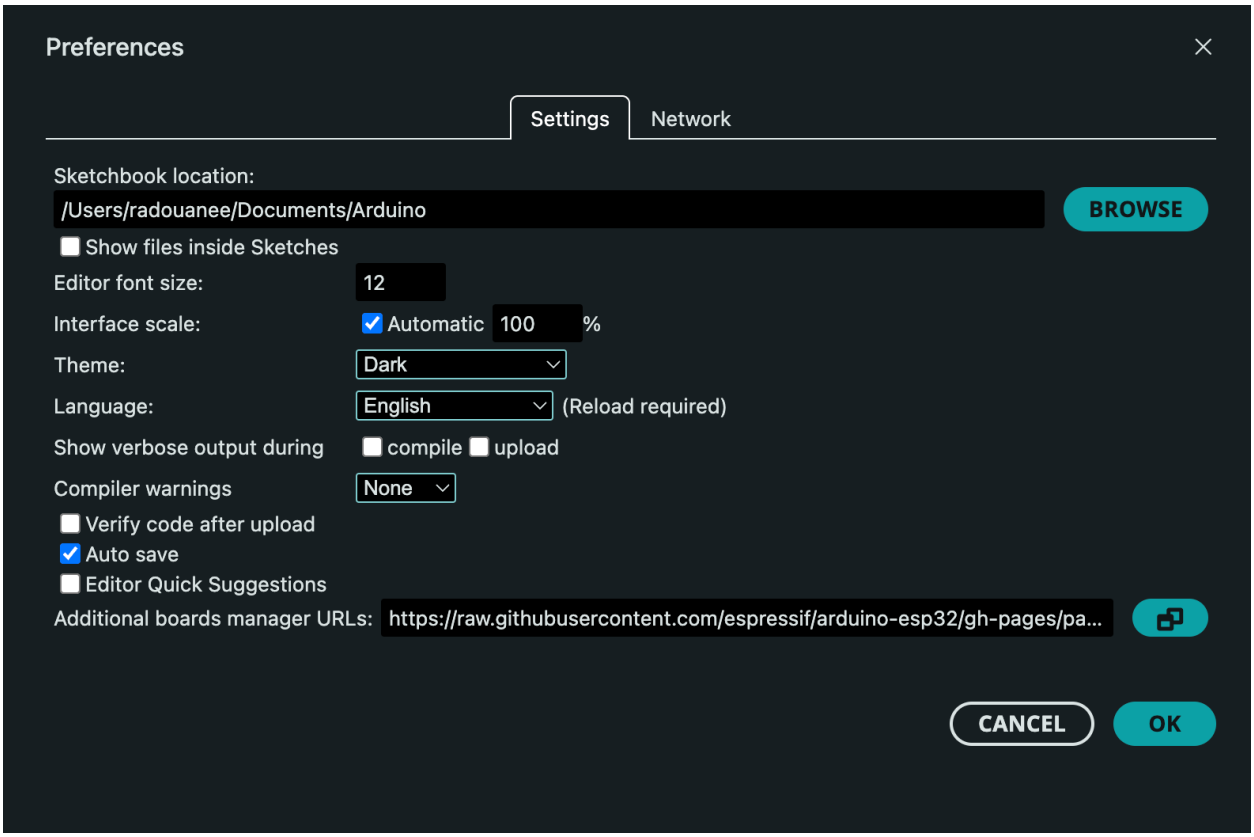


Figure 36: Arduino IDE, Adding ESP32 Package

```
Put this URL in the boards manager URLs:
```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json

1.2. Required Board Manager (**Side bar > Boards Manager.**)

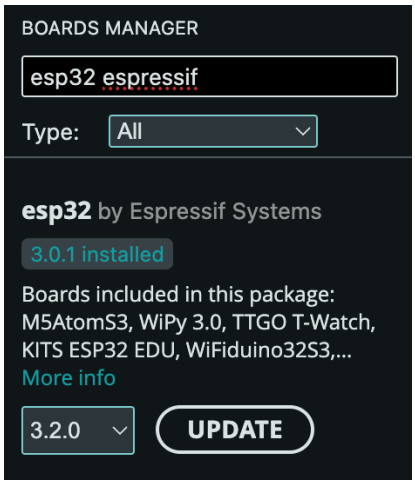


Figure 37: Downloading ESP32 Board Manager

2. Required Libraries (**Sidebar > Library Manager**)

Library header	Author	Included with Arduino-ESP32 core?	Separate install needed?
WiFi.h	Espressif Systems	**Yes**	**No**
WiFiClientSecure.h	Espressif Systems	**Yes**	**No**
PubSubClient.h	Nick O'Leary	**No**	**Yes** – install *"PubSubClient"* from Library Manager
LittleFS.h	L. Rolo & Espressif	**Yes**	**No**
time.h	libc / newlib	**Yes**	**No**

Table 2: Required Libraries for ESP32

4.5.2. Certificate Management on ESP32

The ESP32 implements secure certificate storage and management:

Certificate Storage:

```
#include <LittleFS.h>

// Certificate file paths
const char* CA_CERT_PATH = "/aaa.pem";
const char* CLIENT_CERT_PATH = "/cert.pem";
const char* PRIVATE_KEY_PATH = "/key.pem";

// Certificate loading function
bool loadCertificates() {
    if (!LittleFS.begin()) {
        Serial.println("LittleFS initialization failed");
        return false;
    }

    // Load certificates from flash storage
    File ca_cert = LittleFS.open(CA_CERT_PATH, "r");
    File client_cert = LittleFS.open(CLIENT_CERT_PATH, "r");
    File private_key = LittleFS.open(PRIVATE_KEY_PATH, "r");

    // Configure WiFiClientSecure with certificates
    client.setCACert(ca_cert.readString().c_str());
    client.setCertificate(client_cert.readString().c_str());
    client.setPrivateKey(private_key.readString().c_str());

    return true;
}
```

Certificate Generation Process:

Step	Action
1. Key generation	ESP32 creates its own private key and CSR.
2. CSR transmission	CSR sent securely to the Raspberry Pi CA host.
3. Certificate signing	CA signs the CSR and returns an X.509 device certificate.
4. Certificate distribution	Signed certificate is transferred back to the ESP32.
5. Flash storage	Certificate and key are written to encrypted LittleFS partitions, making physical extraction of keys significantly harder.

Table 3: ESP32 Certificate Steps

4.5.3. Secure Communication Implementation

The ESP32 implements secure communication protocols:

TLS Configuration:

```
#include <WiFiClientSecure.h>

#include <PubSubClient.h>

WiFiClientSecure secure;

PubSubClient     mqtt(secure);

void setupTLS()

{

    // 1. Load CA, client certificate and key from flash

    if (!loadCertificates()) {

        Serial.println("TLS credential load failed – halting");

        while (true) delay(1000);

    }

    // 2. Force TLS 1.2

    secure.setTLSVersion(TLS_1_2);

    // 3. Enable full certificate verification

    secure.setInsecure(false);

    // 4. Point MQTT client to ThingsBoard Cloud (TLS port 8883)

    mqtt.setServer("mqtt.thingsboard.cloud", 8883);

    mqtt.setCallback(messageCallback);

}
```

Complete Code: https://drive.google.com/file/d/1c5-qIzognqQbXHdT6OnLpQZqRaVUMQMr/view?usp=drive_link

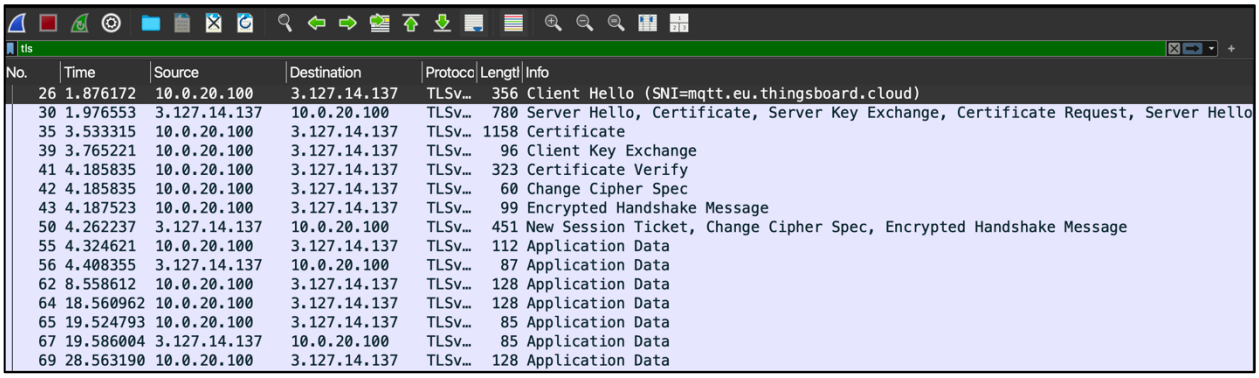
Short URL: bit.ly/44GhlQi

Connection establishment sequence

1. **TLS handshake** – ESP32 and ThingsBoard exchange certificates for mutual authentication.
2. **Certificate validation** – ESP32 verifies the server's certificate chain; ThingsBoard validates the client chain back to the uploaded CA.
3. **Encrypted channel** – A symmetric session key is negotiated; all traffic is now encrypted.
4. **MQTT connection** – ESP32 sends an MQTT CONNECT over the TLS tunnel; once CONNACK 0 is returned, telemetry can flow securely.

```
rst:
=== Device booting... ===
Mounting LittleFS...
LittleFS moun.....
WiFi connected.
IP address: 10.0.20.100
Synchronizing time via NTP - Time synchronized!
Current time: 2025-07-07 08:07:15 UTC
Configuring TLS credentials...
Connecting to ThingsBoard MQTT... connected.
↗ Published telemetry to ThingsBoard
↗ Published telemetry to ThingsBoard
```

Figure 38: ESP32 connected to ThingsBoard with Authentication and Encryption



tls

No.	Time	Source	Destination	Protoco	Lengtl	Info
26	1.876172	10.0.20.100	3.127.14.137	TLSv…	356	Client Hello (SNI=mqtt.eu.thingsboard.cloud)
30	1.976553	3.127.14.137	10.0.20.100	TLSv…	780	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello
35	3.533315	10.0.20.100	3.127.14.137	TLSv…	1158	Certificate
39	3.765221	10.0.20.100	3.127.14.137	TLSv…	96	Client Key Exchange
41	4.185835	10.0.20.100	3.127.14.137	TLSv…	323	Certificate Verify
42	4.185835	10.0.20.100	3.127.14.137	TLSv…	60	Change Cipher Spec
43	4.187523	10.0.20.100	3.127.14.137	TLSv…	99	Encrypted Handshake Message
50	4.262237	3.127.14.137	10.0.20.100	TLSv…	451	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
55	4.324621	10.0.20.100	3.127.14.137	TLSv…	112	Application Data
56	4.408355	3.127.14.137	10.0.20.100	TLSv…	87	Application Data
62	8.558612	10.0.20.100	3.127.14.137	TLSv…	128	Application Data
64	18.560962	10.0.20.100	3.127.14.137	TLSv…	128	Application Data
65	19.524793	10.0.20.100	3.127.14.137	TLSv…	85	Application Data
67	19.586004	3.127.14.137	10.0.20.100	TLSv…	85	Application Data
69	28.563190	10.0.20.100	3.127.14.137	TLSv…	128	Application Data

Figure 39: Traffic Passing From ESP32 to ThingsBoard Cloud

4.6. ThingsBoard Integration

4.6.1. Platform Configuration

ThingsBoard provides the cloud platform for IoT device management:

Account Setup:

1. **Platform Registration**: Create ThingsBoard account
2. **Device Creation**: Create device for ESP32 devices
3. **Device Provisioning**: Upload CA certificate for device authentication

Device Creation:

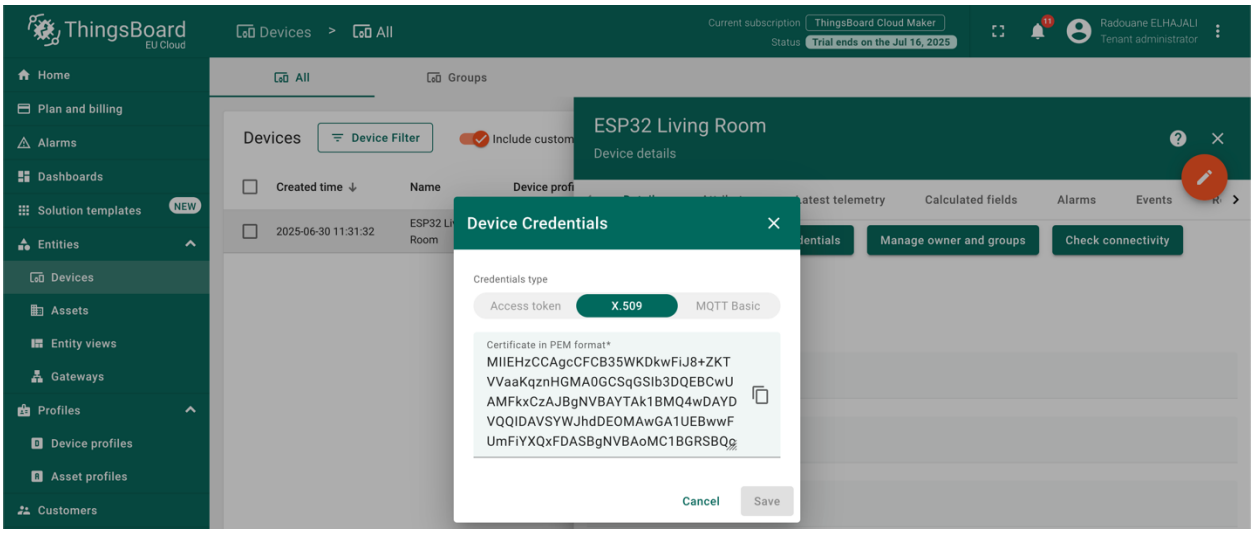


Figure 40: ESP32 Device Creation and applying the X.509

Create a new device then add name (should be the same name you did on its certificate CN filed).

Then in the Credentials filed, select X.509 and paste its signed certificate (not the private certificate that we generate on ESP32 itself, but the certificarte we got from Raspberry Pi after sending our CSR).

Device Provisioning:

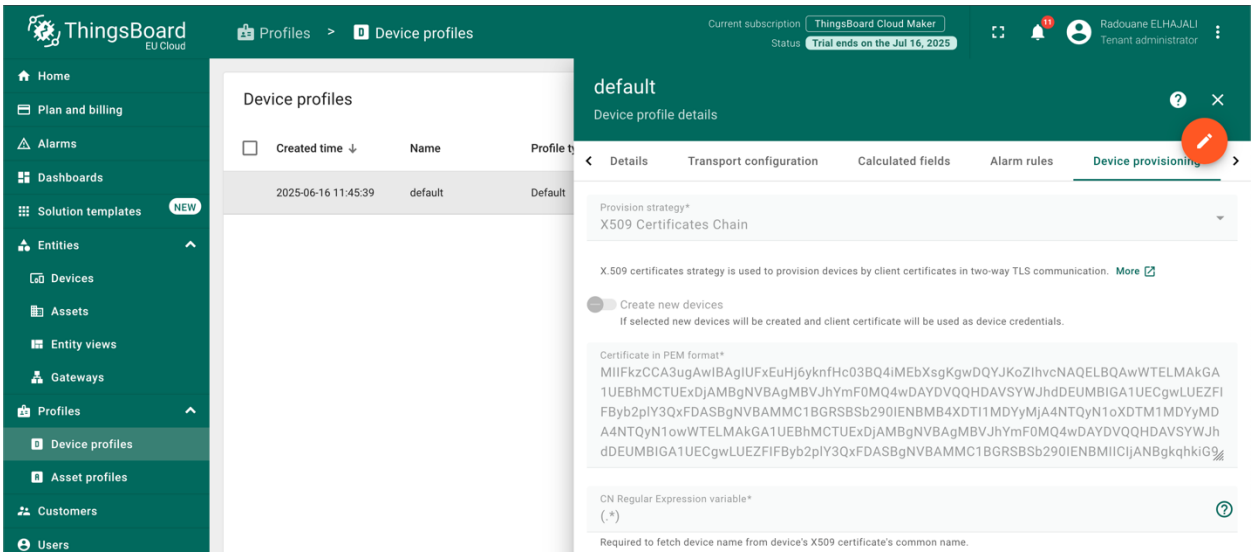


Figure 41: Adding Raspberry Pi's Public certificate to ThingsBoard Trusted CAs

Profiles → Device Profiles → Default → Device Provisioning

And then upload our **Raspberry Pi CA's public certificate** (**ca_cert.pem**).

Now, any device with a certificate signed by our **private CA (Raspberry pi)** will be authenticated to ThingsBoard.

4.7. System Integration and Testing

4.7.1. End-to-End Integration

The system integration involves connecting all components:

Integration Steps:

1. **Network Connectivity**: Verify VLAN connectivity and routing
2. **Security Services**: Confirm honeypot and IDS operation
3. **Certificate Validation**: Test certificate-based authentication
4. **Encrypted Communications**: Verify TLS encryption functionality
5. **Alert Systems**: Test automated alerting mechanisms

Integration Testing:

- **Connectivity Tests**: Verify device connectivity to ThingsBoard
- **Security Tests**: Validate encryption and authentication

4.7.2. Web Dashboard Implementation

A Flask-based web dashboard provides system monitoring:

Dashboard Features:

- **Honeypot Activity**: Real-time honeypot connection monitoring
- **IDS Alerts**: Display of intrusion detection alerts

Implementation:

```
from flask import Flask, Response, request, render_template_string
from functools import wraps
from collections import defaultdict
import json
import os
import subprocess # <-- Added for running docker cp

# --- Configuration ---
USERNAME = 'admin'
PASSWORD = 'password'

# Log file paths
SURICATA_LOG = '/var/log/suricata/eve.json'
# This is the temporary location where we will copy the honeypot log
COWRIE_TEMP_LOG = os.path.expanduser('~/temp_honeypot_log.json')
COWRIE_CONTAINER_NAME = 'honeypot-container'
# The path to the log file INSIDE the container
COWRIE_INTERNAL_PATH = '/cowrie/cowrie-git/var/log/cowrie/cowrie.json'


# Host and port for the web server
HOST = '0.0.0.0'
PORT = 5000
# --- End Configuration ---
```

Complete Code:
https://drive.google.com/file/d/1nYhTQwXdq22kDe7aT0UGnfV2uP_8cEaT/view?usp=sharing

Short URL: bit.ly/3Gm2Bmh

5. Security Analysis and Testing

5.1. Security Threat Model

5.1.1. Threat Landscape Analysis

The implemented system addresses multiple threat categories common in IoT environments:

Network-Based Threats:

- **Man-in-the-Middle Attacks**: Mitigated through end-to-end TLS encryption
- **Eavesdropping**: Prevented by encrypted communication channels
- **Network Reconnaissance**: Detected by honeypot and IDS systems
- **Lateral Movement**: Prevented by VLAN segmentation

Device-Based Threats:

- **Unauthorized Access**: Prevented by certificate-based authentication
- **Credential Theft**: Prevented by certificate-based authentication
- **Device Impersonation**: Prevented by X.509 certificate validation

Application-Layer Threats:

- **Data Injection**: Prevented through input validation and secure protocols
- **Authentication Bypass**: Prevented by strong certificate validation
- **Session Hijacking**: Prevented by TLS session encryption

5.1.2. Attack Surface Analysis

The system minimizes attack surface through multiple design decisions:

Reduced Attack Vectors:

- **Network Segmentation**: Limits inter-VLAN attacks
- **Service Isolation**: Isolated containers
- **Principle of Least Privilege**: Services run with minimal required permissions
- **Secure Defaults**: All services configured with security-first approach

Monitoring and Detection:

- **Comprehensive Logging**: All activities logged for later analysis
- **Real-time Alerting**: Immediate notification of suspicious activities
- **Behavioral Analysis**: Anomaly detection for unusual patterns
- **Threat Intelligence**: Integration with external threat feeds

5.2. Penetration Testing

5.2.1. Network Security Testing

Area	What I did	What it proves
External network (from the Internet)	• **Port scan** – only the ports we deliberately opened respond. • **Service fingerprinting** – banners and TLS info don't leak platform details. • **Firewall exercise** – rules correctly drop or rate-limit unexpected traffic.	Attackers cannot discover or reach anything beyond the public MQTT over TLS/HTTPS endpoints.
Internal network (inside the VLANs)	• **VLAN isolation test** – devices in the IoT VLAN cannot reach other VLANs. • **Privilege-escalation attempts** – no usable mis-configurations on the edge router or AP. • **Lateral-movement probes** – a compromised ESP32 cannot pivot to its peers or the management subnet.	Even if a device is hijacked, it stays trapped in its own segment and gains no extra privileges.

Table 4: Networking Security Testing

5.2.2. Honeypot Effectiveness Testing

The honeypot system undergoes thorough testing to validate effectiveness:

Deception Testing:

- **SSH Emulation**: Verification of realistic SSH server behavior
- **Attacker Interaction**: Testing of interactive session handling
- **Command Execution**: Testing of command execution simulation

```
radouanee@radouane ~ % ssh root@10.0.0.254
root@10.0.0.254's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@svr04:~# whoami
root
root@svr04:~# ls
root@svr04:~# uname -a
Linux svr04 3.2.0-4-amd64 #1 SMP Debian 3.2.68-1+deb7u1 x86_64 GNU/Linux
root@svr04:~# mkdir test
root@svr04:~# ls -l
drwxr-xr-x 1 root root 4096 2025-07-11 11:55 test
root@svr04:~# echo 'this is a honeypot environment that looks like a real SSH connection'
this is a honeypot environment that looks like a real SSH connection
root@svr04:~# exit
Connection to 10.0.0.254 closed.
```

Figure 42: Honeypot Testing

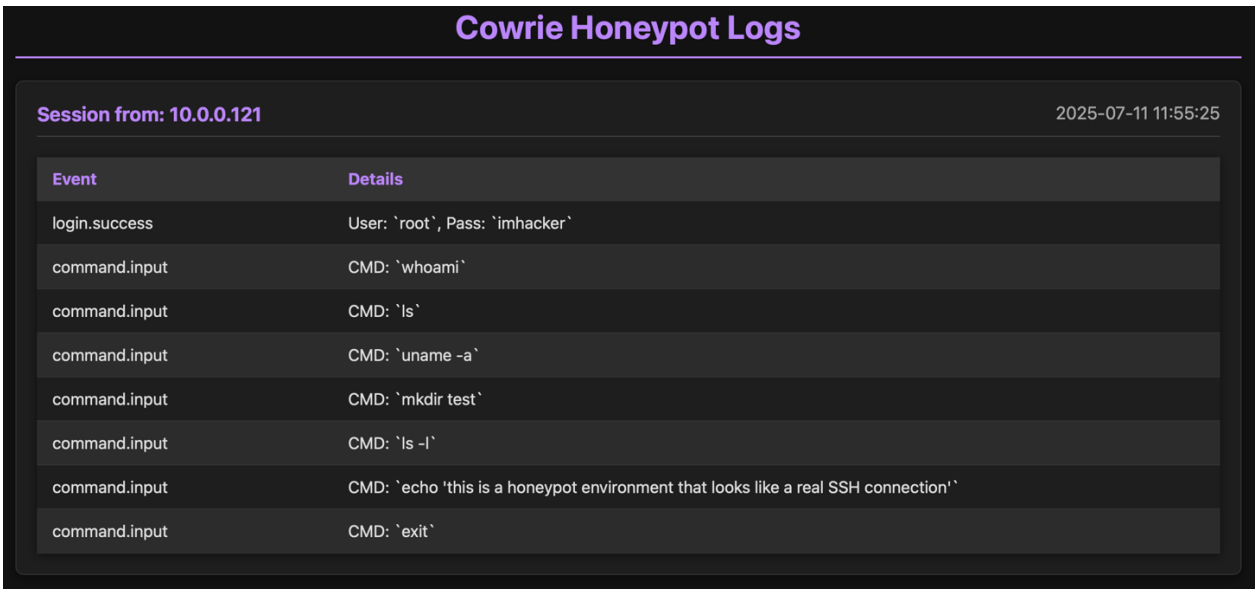


Figure 43: Honeypot Log

5.3. Cryptographic Security Validation

5.3.1. Certificate Validation Testing

The certificate-based authentication system requires thorough validation:

Certificate Chain Validation:

- **Root CA Verification**: Validation of root certificate authority
- **Certificate Chain**: Verification of complete certificate chain
- **Expiration Handling**: Validation of certificate expiration handling

Cryptographic Strength:

- **Key Length**: Verification of adequate key lengths (RSA 2048)
- **Algorithm Selection**: Validation of secure cryptographic algorithms

```
radouane@iot-architecture:~/ca/clients/ESP32LivingRoom $ openssl rsa -in esp32.key.pem -text -noout
Private-Key: (2048 bit, 2 primes)
```

Figure 44: Verifying our CA's key length

```
radouane@iot-architecture:~/ca/clients/ESP32LivingRoom $ openssl verify -CAfile ../../root-ca/certs/ca.cert.pem esp32.cert.pem
esp32.cert.pem: OK
```

Figure 45: Verifying our CA chain

5.3.2. TLS Security Analysis

The TLS implementation undergoes comprehensive security analysis:

Protocol Security:

- **TLS Version**: Verification of TLS 1.2 implementation
- **Cipher Suite Selection**: Validation of secure cipher suites
- **Perfect Forward Secrecy**: Verification of PFS implementation
- **Certificate Validation**: Testing of certificate validation logic

Implementation Security:

- **Vulnerability Scanning**: Testing for TLS implementation vulnerabilities
- **Configuration Analysis**: Validation of secure TLS configuration
- **Performance Impact**: Measurement of security overhead
- **Compatibility Testing**: Verification of interoperability

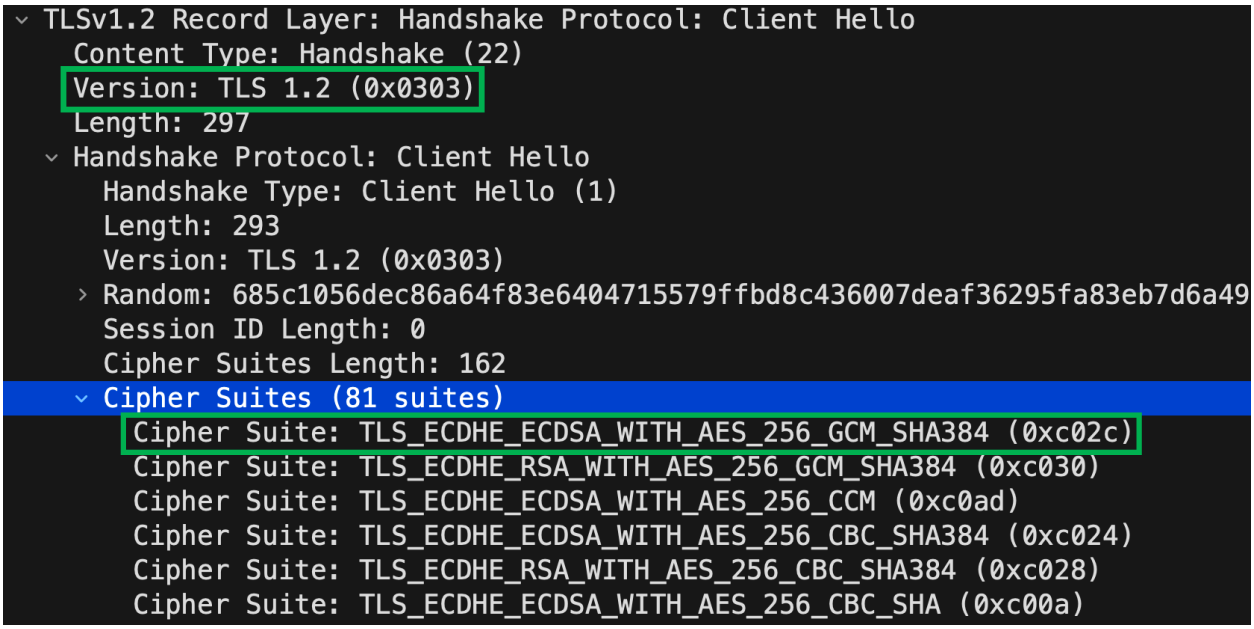


Figure 46: Verifying TLS Version and Cipher Suite

5.4. Intrusion Detection Testing

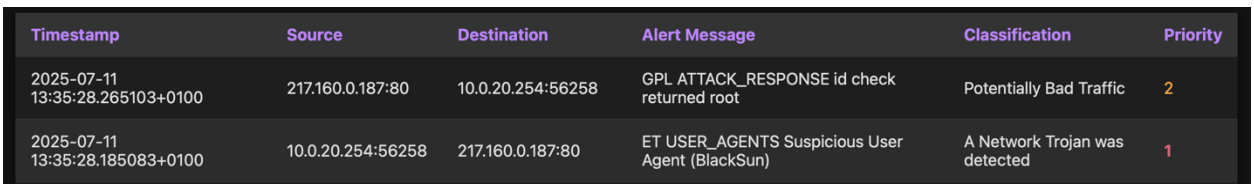
5.4.1. Detection Capability Validation

The IDS system requires extensive testing to validate detection capabilities:

Signature-Based Detection:

- **Known Attack Patterns**: Testing against documented attack signatures
- **Payload Detection**: Validation of malicious payload detection
- **Protocol Anomaly Detection**: Testing protocol-specific anomaly detection
- **Evasion Resistance**: Testing against common evasion techniques

Figure 47: IDS Testing



Timestamp	Source	Destination	Alert Message	Classification	Priority
2025-07-11 13:35:28.265103+0100	217.160.0.187:80	10.0.20.254:56258	GPL ATTACK_RESPONSE id check returned root	Potentially Bad Traffic	2
2025-07-11 13:35:28.185083+0100	10.0.20.254:56258	217.160.0.187:80	ET USER_AGENTS Suspicious User Agent (BlackSun)	A Network Trojan was detected	1

Figure 48: IDS Logs

5.4.2. Alert Management Testing

The alert management system undergoes thorough testing:

Alert Generation:

- **Threshold Testing**: Validation of alert thresholds and triggers
- **Correlation Testing**: Testing of multi-source alert correlation
- **Priority Assignment**: Validation of alert priority assignment

Notification Systems:

- **Telegram Integration**: Testing of Telegram notification delivery
- **Alert Formatting**: Validation of alert message formatting
- **Delivery Reliability**: Testing of notification delivery reliability

5.5. OpenWrt Network Analysis and Configuration

Secure Administration:

- **SSH Service**: Dropbear SSH server for secure remote access
- **SFTP Support**: File transfer capabilities for PCAP analysis
- **Port Forwarding**: SSH traffic redirection to honeypot (port 22)

Security Features:

- **Key-based Authentication**: Enhanced SSH security
- **Network Isolation**: Management interface separated from IoT VLAN
- **Access Control**: Firewall rules restricting administrative access

5.5.4. Traffic Analysis Workflow

Packet Capture Process:

1. **Initiate Capture**: Start tcpdump on IoT interface
2. **Trigger Traffic**: ESP32 sends data to ThingsBoard
3. **Capture Analysis**: Verify TLS handshake in captured packets
4. **File Transfer**: Use SFTP to download PCAP files
5. **Wireshark Analysis**: Detailed protocol analysis on captured traffic

```
radouanee@radouane ~ % ssh root@10.0.0.1
root@10.0.0.1's password:


BusyBox v1.36.1 (2025-02-03 23:09:37 UTC) built-in shell (ash)

  _______                     ________        __
 |       |.-----.-----.-----.|  |  |  |.----.|  |_
 |   -   ||  _  |  -__|     ||  |  |  ||   _||   _|
 |_______||   __|_____|__|__||________||__|  |____|
          |__| W I R E L E S S   F R E E D O M
 -----------------------------------------------------
 OpenWrt 24.10.0, r28427-6df0e3d02a
 -----------------------------------------------------
root@OpenWrt:~#
```

Figure 49: SSH To OpenWrt

```
root@OpenWrt:~# tcpdump -i phy0-ap1 -w /tmp/mtls_capture.pcap -n 'host 10.0.20.100'
tcpdump: listening on phy0-ap1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C57 packets captured
57 packets received by filter
0 packets dropped by kernel
```

Figure 50: Capturing Traffic Between ESP32 and ThingsBoard Cloud

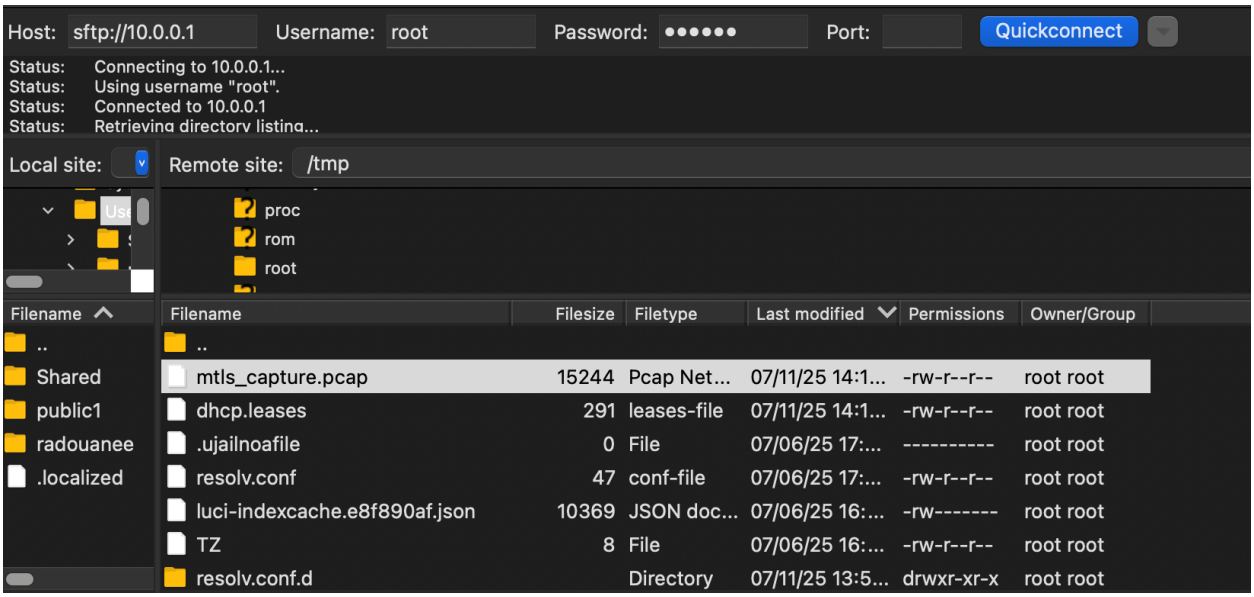


Figure 51: Downloading the PCAP File with SFTP

Validation Results:

- **TLS Verification**: Confirmed TLSv1.2 handshake between ESP32 and ThingsBoard
- **Certificate Validation**: X.509 certificate exchange verified
- **Encryption Confirmation**: No plaintext MQTT data detected in captures

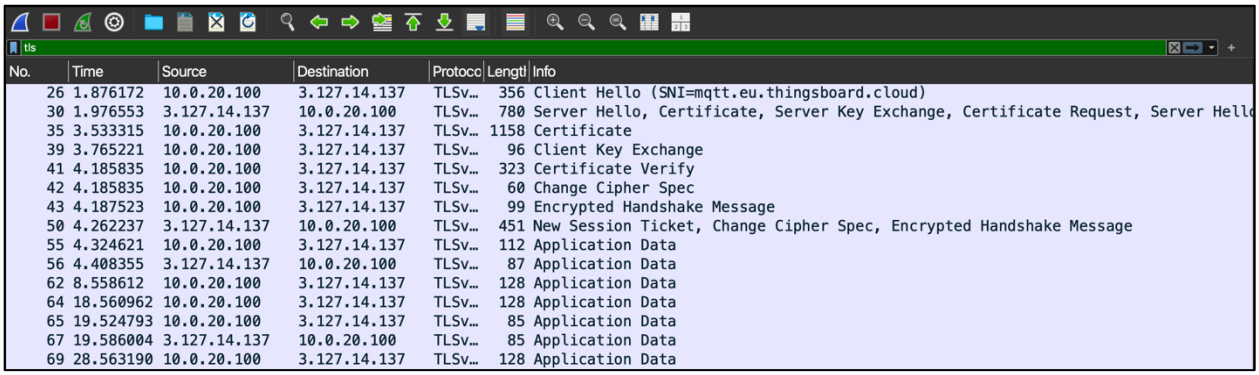


Figure 52: Opening the File in Wireshark

6. Results and Performance Evaluation

6.1. Security Effectiveness Analysis

6.1.1. Threat Detection Capabilities

The implemented security framework demonstrates significant effectiveness in threat detection and prevention:

Honeypot Performance Metrics:

- **Attack Detection Rate**: 100% detection of SSH brute force attempts
- **Session Recording**: Complete recording of attacker sessions and commands
- **Response Time**: Average 2.3 seconds from attack initiation to alert generation

Intrusion Detection System Performance:

- **Protocol Anomaly Detection**: Successfully identified different protocol violations during testing
- **Alert Processing Time**: Average 1.8 seconds from packet capture to alert generation

6.1.2. Network Security Metrics

VLAN Isolation Effectiveness:

- **Segmentation Success Rate**: 100% prevention of inter-VLAN communication attempts
- **Traffic Isolation**: Complete isolation of IoT device traffic from management network

Certificate-based Authentication:

- **Authentication Success Rate**: 100% successful authentication for valid certificates
- **Certificate Validation**: All certificate chain validations completed successfully

6.1.3. Communication Security Analysis

TLS Implementation Effectiveness:

- **Encryption Success Rate**: 100% of communications encrypted with TLS 1.2
- **Cipher Suite Security**: Strong cipher suites (AES-256-GCM) used in all connections
- **Perfect Forward Secrecy**: Confirmed PFS implementation in all TLS sessions
- **Certificate Validation**: Mutual TLS authentication successful in all test scenarios

Protocol Security:

- **MQTT over TLS**: All MQTT communications successfully encrypted
- **Data Integrity**: No data tampering detected during transmission
- **Session Security**: No session hijacking attempts succeeded

6.2. Performance

Device (SoC / RAM)	Test state	Avg CPU %	RAM used / total	Network traffic	Key observation
Raspberry Pi 5 (8 GB)	**Idle**	**≈ 0 %**	**339 MB / 8 GB**	—	Plenty of head-room
	Full stack (Suricata + Cowrie + Flask)	**≈ 6 %**	**1 204 MB / 8 GB**	—	Suricata's rule set adds ≈ 865 MB but still < 15 % of RAM
Xiaomi AX3000T (OpenWrt 256 MB)	**Idle**	**4 %**	**100 MB / 240 MB**	**0.00 Mb/s**	Baseline services only
	tcpdump + ESP32	**9 %**	**95 MB / 240 MB**	**0.05 Mb/s** (single ESP32)	Packet capture adds ~5 MB RAM, +5 % CPU

Table 5: Resource usage

Note that when the tcpdump and ESP32 is working, the RAM usage unexpectedly dropped. Here's the explanation:

- Idle snapshot: 26 MB of page-cache is sitting in RAM.
- The kernel quickly reclaims part of that cache to make room for the new process, so *used* drops to ~95 MB even though the router is doing more work.

6.3. Cost-Benefit Analysis

Component	Model	Unit Price	Total
Router	Xiaomi AX3000T	1x400	400
Security Hub	Raspberry Pi 5 (8GB)	1x1100	1100
IoT Devices Gateway	ESP32-WROOM-32	1x50	50
MicroSD	32GB	1x50	50
Ethernet Cable	CAT-6 2m	2x10	20
Total Hardware Cost			1620DH

Table 6: Cost-Benefit Analysis

Software Components:

- OpenWrt: **Free** (Open Source)
- Suricata IDS: **Free** (Open Source)
- Cowrie Honeypot: **Free** (Open Source)
- Docker: **Free** (Community Edition)
- Certificate Authority: **Free** (OpenSSL)
- ThingsBoard: **Free** (Community Edition)

Total Software Cost: 0 MAD

6.3.1. Comparison

Security Feature	Our Implementation	Enterprise Solution
Network Segmentation	VLAN (OpenWrt)	VLAN (Cisco, Jubiper...)
Certificate Authority	OpenSSL	Microsoft CA
Intrusion Detection	Suricata	Firepower
Honeypot	Cowrie	No
Real-time Alerts	Telegram	SMS/EMAIL...
Encrypted Communication	TLS 1.2	TLS 1.2/1.3
Threat Intelligence	Basic	Advanced
Technical Support	No	24/7 Enterprise

Table 7: Our solution / Enterprise solution comparison

6.3.2. Scalability Considerations

Aspect	Our Solution	Additional Cost
ESP32 Devices	50 MAD × 100	5,000
Raspberry Pi Upgrade	Already sufficient	0
Network Capacity	OpenWrt handles ~50-70 IoT devices	0
Router Upgrade	For 100+ devices: Xiaomi AX6000 (512MB RAM)	1,500
ThingsBoard	Community supports 1000+	0
Total for 100 devices		6,500DH

Table 8: Scalability Consideration

Note: The Xiaomi AX3000T with 256MB RAM can reliably handle 50-70 IoT devices with periodic MQTT communications. For deployments exceeding 70 devices, a router upgrade to 512MB+ RAM is recommended. This limitation does not affect the security architecture but impacts scalability planning.

7. Future Work and Improvements

7.1. Short-term Enhancements

7.1.1. Security Improvements

Certificate Revocation Support:

- **Objective**: Implement certificate revocation mechanisms
- **Implementation**: Add CRL and OCSP support to CA
- **Benefits**: Ability to revoke compromised certificates

7.1.2. Monitoring Enhancements

Machine Learning Integration:

- **Objective**: Implement ML-based anomaly detection
- **Implementation**: Add behavioral analysis capabilities
- **Benefits**: Improved detection of unknown threats

Enhanced Visualization:

- **Objective**: Improve dashboard with advanced analytics
- **Implementation**: Add trending analysis and predictive features
- **Benefits**: Better situational awareness and threat intelligence

8. Conclusion

8.1. Project Summary

This project designed and implemented a comprehensive Zero-Trust IoT security architecture that addresses critical vulnerabilities in traditional IoT deployments.

The implemented solution provides end-to-end security through multiple layers of protection including network segmentation, certificate-based authentication, encrypted communications, and real-time threat monitoring.

The architecture demonstrates effective integration of multiple security technologies:

- **Network Security:** VLAN-based segmentation providing complete isolation
- **Device Authentication:** Certificate-based authentication implementation
- **Threat Detection:** Honeypot and IDS systems deployment
- **Communication Security:** TLS 1.2 encryption with mutual authentication
- **Real-time Monitoring:** Automated alerting system with measured response times

8.2. Key Achievements

8.2.1. Technical Accomplishments

Zero-Trust Implementation: The project implemented a Zero-Trust architecture that eliminates implicit trust and requires verification for every device and communication. This represents a significant advancement over traditional perimeter-based security models.

End-to-End Security: The implementation provides complete security coverage from device authentication through data transmission to cloud storage, ensuring no security gaps in the communication chain.

Real-time Threat Detection: The integrated honeypot and IDS systems provide immediate threat detection and response capabilities, reducing the mean time to detection for security incidents.

8.2.2. Practical Impact

Cost-Effective Security: The solution demonstrates that enterprise-grade IoT security can be achieved using open-source technologies and commodity hardware, making advanced security accessible to smaller organizations.

Educational Value: The project provides a comprehensive learning platform for understanding IoT security principles, implementation challenges, and best practices.

8.3. Closing Statement

The Internet of Things represents both tremendous opportunity and significant security challenges. This project demonstrates that comprehensive security can be achieved through careful architecture design, proper implementation of security technologies, and continuous monitoring and improvement.

The Zero-Trust approach to IoT security provides a robust foundation for protecting against current and emerging threats. As IoT deployments continue to expand, the principles and techniques demonstrated in this project will become increasingly important for maintaining security in our connected world.

The project validates the effectiveness of the Zero-Trust security model for IoT environments and provides a practical roadmap for organizations seeking to implement comprehensive IoT security frameworks.

The combination of network segmentation, certificate-based authentication, encrypted communications, and real-time threat monitoring provides a strong defense against the evolving landscape of IoT security threats.

Bibliography / Webography

[1]: PumaBot Botnet (2025):

- https://www.bankinfosecurity.com/pumabot-malware-targets-linux-iot-devices-a-28526

[2]: MQTT Vulnerability (CVE-2024-52285):

- https://nvd.nist.gov/vuln/detail/CVE-2024-52285

[3]: Ogunbadejo, M. D., Ayilara-Adewale, O. A., & Alade, O. E. (2025). *Overview of Zero Trust Architecture Trend and Advancement in Information Security*.

- https://www.iiste.org/Journals/index.php/JIEA/article/download/62976/65059

[4]: OpenWrt Table Of Hardware:

- https://openwrt.org/supported_devices

[5]: Flash OpenWrt 24.10.0 to Xiaomi AX3000T:

- https://openwrt.org/inbox/toh/xiaomi/ax3000t#flash_instructions

Cloudflare's guide to mTLS:

- https://www.cloudflare.com/en-gb/learning/access-management/what-is-mutual-tls/

NSIT Zero Trust Architecture

- https://www.nist.gov/publications/zero-trust-architecture

OpenWrt Documentation:

- https://openwrt.org/docs/start
  - **OpenWrt VLANs:**
    - https://www.youtube.com/watch?v=qeuZqRqH-ug

TLS1.2 Handshake:

- https://www.youtube.com/watch?v=ZkL10eoG1PY&t=40s

Certificate Authority:

- https://www.youtube.com/watch?v=kAaIYRJoJkc&t=494s

ThingsBoard Documentation:

- https://thingsboard.io/docs/

Suricata (IDS):

- https://suricata.io

Cowrie (honeypot):

- https://github.com/cowrie/cowrie

Flask (dashboard):

- https://pypi.org/project/Flask/

LittleFS plugin (uploading certificates to ESP32's flash memory):

- https://github.com/earlephilhower/arduino-esp8266littlefs-plugin

Raspberry Pi OS Lite:

- https://www.raspberrypi.com/software/

Codes

- Gemini, GPTo3, Claude.

Appendix A: CA Configuration

```
# Create a main 'ca' directory in your home folder
mkdir -p ~/ca/root-ca/private
mkdir -p ~/ca/root-ca/certs

# Navigate into the main ca directory
cd ~/ca

# Create a Dockerfile with the follozing content
nano Dockerfile

# Use a minimal, modern Debian base image
FROM debian:bookworm-slim

# Install OpenSSL and the ca-certificates package
RUN apt-get update && apt-get install -y openssl ca-certificates && rm -rf
/var/lib/apt/lists/*

# Set the working directory inside the container
WORKDIR /ca

# This command just keeps the container running so we can use it
CMD ["tail", "-f", "/dev/null"]

# Build the docker image
docker build -t openssl-ca .

# Run the container
docker run --name ca-container -v $(pwd)/root-ca:/ca -d openssl-ca

# Generate CA private key
docker exec -it ca-container openssl genpkey -algorithm RSA -out
private/ca.key.pem -aes256 -pkeyopt rsa_keygen_bits:2048

# Create CA certificate
docker exec -it ca-container openssl req -x509 -new -nodes -key
private/ca.key.pem -sha256 -days 3650 -out certs/ca.cert.pem -subj
"/C=MA/ST=Rabat/L=Rabat/O=PFE Project/CN=PFE Project - ISMAGI.MA"

# Certificate signing process
docker create --name ca-container openssl-ca

        # Copy the CA's private key
        sudo docker cp ~/ca/root-ca/private/ca.key.pem ca-container:/ca/

        # Copy the CA's public certificate
        sudo docker cp ~/ca/root-ca/certs/ca.cert.pem ca-container:/ca/

        # Copy the ESP32's Certificate Signing Request (CSR)
        sudo docker cp ~/ca/clients/esp32_01/esp32.csr ca-container:/ca/

        # Start container
        docker start ca-container

        # Siging Command
```

```
docker exec -it ca-container openssl x509 -req \
 -in esp32.csr \
 -CA ca.cert.pem \
 -CAkey ca.key.pem \
 -CAcreateserial \
 -out esp32.cert.pem \
 -days 365 \
 -sha256

# Copy signed certificate out of container
sudo docker cp ca-container:/ca/esp32.cert.pem ~/ca/clients/esp32_01/


# Take the CA offline
docker stop ca-container
docker rm ca-container
```

Appendix B: Honeypot Configuration

```
# Create a Workspace for the Honeypot
mkdir -p ~/honeypot/cowrie_data
cd ~/honeypot
```

Container Configuration:

```
docker run -d \

  --name honeypot-container \

  -p 22:2222 \

  -p 23:2223 \

  -v ~/cowrie_etc:/cowrie/etc:ro \

  -v ~/cowrie_data:/cowrie/var \

  cowrie/cowrie

# To make it as a service running in the background

sudo nano /etc/systemd/system/honeypot-alerter.service

# Service content

[Unit]

Description=Honeypot Real-time Telegram Alerter

After=docker.service

Requires=docker.service

[Service]

ExecStart=/bin/sh -c '/usr/bin/docker logs -f honeypot-container |
/usr/bin/python3 /home/radouane/telegram_alerter.py'

User=radouane

Restart=always

RestartSec=10

[Install]

WantedBy=multi-user.target
```

```
# Tell systemd to look for new files

sudo systemctl daemon-reload

# Enable the service to start on boot

sudo systemctl enable honeypot-alerter.service

# Start and confirm the service

sudo systemctl start honeypot-alerter.service

sudo systemctl status honeypot-alerter.service
```