CENTRALIZED WEB-BASED IOT SECURITY ASSESSMENT AND PENETRATION TESTING DASHBOARD

BY

TAN HOI JIAN

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) (COMMUNICATIONS AND NETWORKING)

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2026

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Technology (Honours) (Communications and Networking) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, **Dr. Rohani binti Bakar**, for her valuable time, guidance, constructive feedback, and continuous encouragement throughout this project. Her expertise and patience have been instrumental in shaping the direction of my work and in helping me grow both academically and professionally.

I am also grateful to the **Faculty of Information and Communication Technology (FICT), Universiti Tunku Abdul Rahman (UTAR)**, for providing the resources, facilities, and a conducive environment that made this project possible.

Last but not least, I wish to extend my heartfelt thanks to my parents and family for their love, unwavering support, and encouragement throughout my studies. Their belief in me has been the foundation of my perseverance and determination in completing this final year project.

ABSTRACT

This project developed a centralized web-based IoT Security Assessment and Penetration Testing Dashboard for small-scale and laboratory IoT environments. The project addressed the problem of fragmented IoT security assessment, where network scanning, web vulnerability testing, device monitoring, result interpretation, and reporting were commonly performed using separate tools and manual workflows. To overcome this limitation, the system was designed and implemented using Python Flask, MySQL, HTML, CSS, and JavaScript, with Flask-SocketIO for real-time communication, while integrating Nmap for network scanning, OWASP ZAP for web vulnerability assessment, MQTT and REST for telemetry ingestion, ping sweep host discovery, and SSL/TLS certificate analysis. The development process involved system design, backend and frontend implementation, third-party tool integration, and functional testing of the completed modules. The final system was able to perform Nmap and ZAP scans, monitor device telemetry in real time, detect reachable hosts, analyse SSL/TLS configurations, visualise findings through charts and tables, and generate structured reports in printable and spreadsheet formats. The results showed that the dashboard successfully consolidated multiple IoT security assessment functions into a single platform and improved usability, monitoring visibility, and workflow efficiency. The project contributed a practical and educational prototype for centralized IoT security assessment by combining scanning, telemetry, reporting, and visualisation features in one web-based system.

Area of Study: IoT Security, Web-based Dashboard

Keywords: IoT Security Assessment, Penetration Testing Dashboard, Nmap Scanning, OWASP ZAP, MQTT Telemetry, Real-Time Monitoring, SSL/TLS Scanning, Vulnerability Assessment

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CIDR	Classless Inter-Domain Routing
CISA	Cybersecurity and Infrastructure Security Agency
CLI	Command-Line Interface
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DAST	Dynamic Application Security Testing
DNS	Domain Name System
DPI	Deep Packet Inspection
ELK	Elasticsearch, Logstash, and Kibana
ERD	Entity Relationship Diagram
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization
IT	Information Technology
JSON	JavaScript Object Notation

MQTT	Message Queuing Telemetry Transport
NAC	Network Access Control
NSE	Nmap Scripting Engine
NVD	National Vulnerability Database
OASIS	Organization for the Advancement of Structured Information Standards
OSSIM	Open Source Security Information Management
OSTE	OWASP Security Testing Extension
OT	Operational Technology
OWASP	Open Web Application Security Project
PC	Personal Computer
PDF	Portable Document Format
PETIoT	Penetration Testing Framework for Internet of Things
QoS	Quality of Service
REST	Representational State Transfer
RTSP	Real-Time Streaming Protocol
SIEM	Security Information and Event Management
SMB	Server Message Block
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
SQLi	SQL Injection
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time to Live
UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator

VAPT	Vulnerability Assessment and Penetration Testing
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
XSS	Cross-Site Scripting
ZAP	Zed Attack Proxy

CHAPTER 1

Introduction

This chapter introduces the background and motivation of the project, which focused on addressing security challenges in Internet of Things (IoT) environments through the development of a centralized web-based security assessment and penetration testing dashboard. It presents the problem statement, project objectives, scope, contributions, and background information that supported the direction of the completed system.

1.1 Problem Statement and Motivation

The IoT has experienced explosive growth in recent years, connecting billions of devices across domains like smart homes, transportation, agriculture, and healthcare. Cisco predicts over 75 billion IoT devices will be active by 2025 [1]. However, many of these devices are inherently insecure. Studies indicate roughly 70% of consumer IoT devices have at least one security flaw, averaging over 25 vulnerabilities per device [2]. These flaws contributed to a 300% surge in IoT-targeted attacks in early 2019 [3]. Furthermore, IoT gadgets often have minimal resources (low CPU, memory, and power), making them unable to run traditional security software [1]. The combination of vast scale and limited defences means vulnerabilities in one device can be leveraged to compromise entire networks. For example, smart cameras or medical monitors can serve as entry points for malware or privacy breaches. These trends underscore a critical need for systematic IoT security assessment. In particular, a unified platform that continuously scans devices, monitors for threats, and visualizes security metrics would help organizations manage the growing risk in IoT environments [1]-[3].

Despite the severity of IoT security issues, current tools and practices are fragmented. Device manufacturers may provide minimal configuration options, and existing security products (often vendor-specific) focus on narrow aspects like network traffic. Verma and Sheetlani note that “Due to the lack of security features in IoT devices across the environment, a security dashboard is needed to find what type of security controls are required to stop threats” [4]. In practice, security analysts rely on a mix of open-source tools such as Nmap for network probing and OWASP ZAP for web interface testing. However, these tools are usually operated separately in ad hoc ways, producing unstructured outputs that require technical expertise to

interpret. There is no easy-to-use, centralized platform for routine vulnerability assessment and monitoring of diverse IoT devices. As IoT deployments become more critical, this gap can lead to undetected breaches and delayed response. The problem, therefore, is to create a single dashboard that orchestrates essential IoT security functions – scanning, real-time alerting, and reporting – in a coherent manner accessible to security analysts and administrators.

The motivation for this project comes from both the escalating IoT threat landscape and the opportunity to leverage modern web technologies for security tooling. Automated security testing is increasingly regarded as necessary; Jaafar et al. observe that "automated penetration testing can provide consistent and comprehensive assessments, enabling end users and organizations to monitor and fortify their IoT devices continuously against emerging threats" [5]. This project responds to that need by developing a centralized web-based IoT security dashboard that integrates Nmap and OWASP ZAP within a single interface, supported by a Flask backend, an HTML/CSS/JavaScript frontend, and a MySQL database for structured storage. By testing the prototype on real devices such as PCs, laptops and mobile phones, the project demonstrates both practical feasibility and educational value. The dashboard aimed to make IoT penetration testing more practical, beginner-friendly, and cost-effective, particularly for small labs, student projects, or organizations with limited resources. In summary, the project is motivated by the urgent need for continuous IoT security assessment [3], the benefits of automation [5], and the availability of accessible development tools that can make such a dashboard a reality.

Building on these motivations, the project was extended beyond basic scan integration into a more complete dashboard that included live telemetry monitoring, role-based access control, device discovery, SSL/TLS assessment, ping sweep host discovery, activity logging, report export, and user guidance features. These enhancements strengthened the practicality of the system for small-scale IoT security assessment and academic demonstration.

1.2 Project Objectives

This project was carried out based on three main objectives to improve IoT security assessment through a centralized web-based dashboard:

Objective 1: To implement and demonstrate IoT-focused security assessment and penetration testing capabilities

- Conduct automated vulnerability assessments using open-source tools such as Nmap for network scanning and OWASP ZAP for web interface testing.
- Integrate automated web and network security assessment capabilities through open-source tools and supporting modules for practical IoT security evaluation.
- Store scan results in a structured database for tracking historical vulnerabilities.
- Generate alerts based on scan outcomes to support real-time decision making.

Objective 2: To develop a centralized web-based dashboard for managing and visualising IoT security data

- Design and implement a responsive frontend interface for initiating scans, displaying device status, and viewing analytics.
- Enable real-time monitoring of IoT devices through protocols such as MQTT or HTTP polling.
- Visualize security metrics such as device risk scores using charts and tables.

Objective 3: To enhance and validate the system through real IoT device testing and advanced features

- Test the dashboard in practical scanning and monitoring scenarios to evaluate functionality and usability.
- Implement live monitoring and telemetry integration through MQTT, REST, and WebSocket communication.
- Extend the system with additional supporting features such as device discovery, SSL/TLS checking, host discovery, reporting, and activity logging.

1.3 Project Scope

This project was scoped to the design, development, and demonstration of a prototype Centralized IoT Security Assessment and Penetration Testing Dashboard. The dashboard focused on consolidating essential penetration testing and monitoring functions into a single, web-based platform. The system targets small-scale IoT environments on a local network, such as laboratory test, academic demonstrations, or small organizations with limited resources, rather than enterprise-level deployments.

Specifically, the project covered the following components:

- **Integration of open-source scanning tools:** Nmap is used for network discovery and service probing, while OWASP ZAP is employed for assessing vulnerabilities in IoT device web interfaces. These tools are orchestrated programmatically through the backend to enable automated assessments.

- **System implementation using accessible technologies:** The backend is implemented in Python Flask, the frontend uses HTML, CSS, and JavaScript, and a MySQL database is employed to store device metadata, scan results, and alert history. This stack was selected for its simplicity, accessibility, and suitability for small projects.
- **Dashboard functionalities:** The system provides device discovery, vulnerability scanning, real-time alerting, data visualization through charts and tables, and historical reporting of previous scans. These functions enable users to monitor IoT device security continuously and identify vulnerabilities efficiently.
- **Testing and validation with real IoT devices:** The prototype is evaluated using a small laboratory setup consisting of PC and laptop units including mobile phones, which serve as representative IoT devices with real services and web interfaces. This ensures that the system is validated on realistic targets.

Not only that, the project was including but not limited to:

- **Real-time telemetry monitoring** through MQTT, REST API, and WebSocket updates.
- **Role-based access control** for admin and user views.
- **Ping sweep host discovery** for identifying reachable devices in a subnet.
- **SSL/TLS certificate analysis** for checking protocol and certificate-related issues.
- **Report generation and export** in printable and spreadsheet formats.
- **Activity logging and user guidance support** for improved usability.

1.4 Contributions

The system achieved the following contributions:

Integration of Multiple Open-Source Security Tools: This project demonstrates the feasibility of orchestrating Nmap, OWASP ZAP, SSL/TLS inspection, and ping sweep host discovery within a single Flask-based backend. The system unifies tools that are conventionally operated in isolation into a cohesive, automated assessment pipeline, eliminating the need for manual tool switching and result correlation. This integration approach serves as a replicable reference architecture for developers seeking to consolidate fragmented security tooling.

Real-Time IoT Device Monitoring via MQTT and WebSocket: The system implements a live telemetry pipeline in which IoT devices publish status updates via MQTT or REST, and the dashboard reflects these updates in real time through Flask-SocketIO WebSocket communication. The auto-scan feature further extends this by automatically triggering an

Nmap assessment when a previously unseen device is detected, enabling continuous security coverage without manual intervention. This demonstrates a practical approach to bridging IoT device monitoring and security assessment in a unified platform.

Usability Improvement through Centralised Web Interface: The dashboard consolidates all security assessment functions which are scanning, monitoring, alerting, logging, and reporting, into a single browser-based interface that requires no client-side installation. Role-based access control provides differentiated views for regular users and administrators. An in-browser user guide, custom modal dialogs, real-time progress indicators, and toast notifications collectively reduce the technical barrier to conducting IoT security assessments, making the platform accessible to users without prior command-line security tool experience.

CVE Intelligence Integration via NVD API: Each completed Nmap scan is automatically enriched with known vulnerability data retrieved from the National Vulnerability Database (NVD) API v2. The system maps detected service names and version strings to CVE records and stores the results in a local cache with a seven-day TTL to minimise redundant external API calls. This transforms raw port scan output into actionable vulnerability intelligence without requiring the user to perform manual CVE lookups.

Educational Value for IoT Security Assessment Training: The system is designed and validated at a scale appropriate for academic laboratories, student projects, and small organisations with limited resources. Its open-source technology stack which are Python Flask, MySQL, HTML/CSS/JavaScript, Nmap, and OWASP ZAP, is widely taught in computing curricula, making the codebase readily understandable and extensible by students and researchers. The structured report export, activity logging, and in-browser documentation collectively provide a complete learning artefact that covers the full IoT security assessment workflow from device discovery through to vulnerability reporting.

Comparative Advancement over Existing Fragmented Tooling: As documented in the literature review, existing approaches to IoT security assessment rely either on individually operated open-source tools that produce unstructured outputs, or on costly enterprise platforms that are inaccessible to small organisations. This project delivers a middle-ground solution: a zero-licensing-cost, open-source platform that provides the automation and centralisation of enterprise tools while remaining deployable on a standard personal computer. The system

directly addresses the gap identified by Verma and Sheetlani [4], who called for dashboards that organise and standardise security assessment across IoT environments.

1.5 Report Organisation

This report is organised into seven chapters. Chapter 1 is the introduction of the project, which covers the problem statement and motivation, project objectives, project scope and direction, project contributions, and report organisation. It establishes the context and rationale for developing a centralised web-based IoT security assessment and penetration testing dashboard as an accessible and integrated alternative to fragmented open-source tooling.

Chapter 2 is the literature review, which examines previous works and existing tools in the domain of IoT security assessment. It reviews open-source tools including Nmap and OWASP ZAP, evaluates industry approaches and proprietary platforms such as ThingsBoard and Cisco Cyber Vision, identifies the gaps that remain in current solutions, and justifies the technology stack selected for this project based on those findings.

Chapter 3 is the proposed method and approach, which describes the overall system architecture, use case diagram, system requirements, estimated cost, and the development timeline for both FYP1 and FYP2. It presents the three-tier architecture of the system, the modular design of the Flask backend, and the workflow from device telemetry ingestion through to scan result storage and visualisation.

Chapter 4 is the system design, which provides a detailed component-level view of the dashboard. It describes the system block diagram, the specifications of all software components and libraries, the internal design of each module including the Flask Blueprint structure and MySQL database schema, and the runtime interaction sequences between components for each major system operation.

Chapter 5 is the system implementation, which documents the complete hardware and software setup, environment configuration, and startup procedure required to bring the system to a running state. It also presents the operation of each feature through annotated screenshots, covering user registration and login, live telemetry monitoring, manual and automatic Nmap scanning, OWASP ZAP web vulnerability assessment, SSL/TLS certificate inspection, ping

sweep, activity logging, and security report export. Implementation issues encountered during development and their resolutions are also discussed in this chapter.

Chapter 6 is the system evaluation and discussion, which evaluates the completed system against the three project objectives and their associated sub-goals. It presents the functional test results across twenty-two test cases, performance measurements for each scanning module, and usability observations. The chapter also revisits the key project challenges and discusses the limitations of the current implementation.

Chapter 7 is the conclusion and recommendation, which summarises the outcomes of the project, reflects on its contributions relative to the identified problem, and proposes a set of concrete recommendations for future development. These recommendations cover areas including deployment automation, security hardening of the MQTT broker, scheduled scanning, extended CVE intelligence, and formal usability testing.

CHAPTER 2

Literature Review

2.1 Previous Works

The IoT has revolutionized how modern systems interact, but it also presents unique and urgent security concerns. Numerous researchers have addressed these issues from various angles. Jaafar et al. introduced **PETIoT**, a penetration testing framework for IoT, which formalizes an IoT "kill chain" to assist testers in discovering vulnerabilities systematically within devices such as smart cameras, wearables, or home sensors. Their tests identified multiple **zero-day vulnerabilities** in consumer IP cameras, reinforcing the importance of structured VAPT (Vulnerability Assessment and Penetration Testing) processes in IoT environments [7].

Meanwhile, Gyamfi and Jurcut presented a comprehensive review of **intrusion detection systems (IDS)** in IoT environments, highlighting the limitations of conventional detection methods in resource-constrained devices. Their work suggested leveraging **edge computing and machine learning** to distribute security workloads closer to the data source, allowing real-time anomaly detection without overburdening IoT devices [1].

Additionally, tools like **RouterSploit**, **Binwalk**, and **Firmadyne** have emerged to support security analysts in identifying vulnerabilities within embedded firmware or network stacks. RouterSploit focuses on exploiting router firmware [8], while Binwalk performs deep analysis of binary images—a critical need in IoT where manufacturers often ignore security hygiene during firmware development. However, despite their power, these tools are primarily CLI-based and require technical proficiency, making them inaccessible for many users.

2.2 Open-Source Tools and Security Community Contributions

2.2.1 Nmap: Network Exploration and Security Auditing Tool

Nmap is one of the most widely used open-source tools for network scanning and host discovery. Originally developed for identifying live hosts and open ports, it has since evolved into a comprehensive network auditing platform. Nmap supports a wide range of scanning techniques, including TCP SYN scans, UDP scans, service version detection, and operating

system fingerprinting [9]. It also includes the Nmap Scripting Engine (NSE), which allows security researchers to write scripts for advanced tasks such as vulnerability detection, malware scanning, and brute force authentication checks. In IoT environments, Nmap has been particularly valuable for identifying exposed services such as Telnet, SSH, HTTP, or RTSP on embedded devices like routers, IP cameras, and sensors. Since many IoT devices ship with weak configurations or open services, Nmap provides an efficient first step in assessing their security posture.

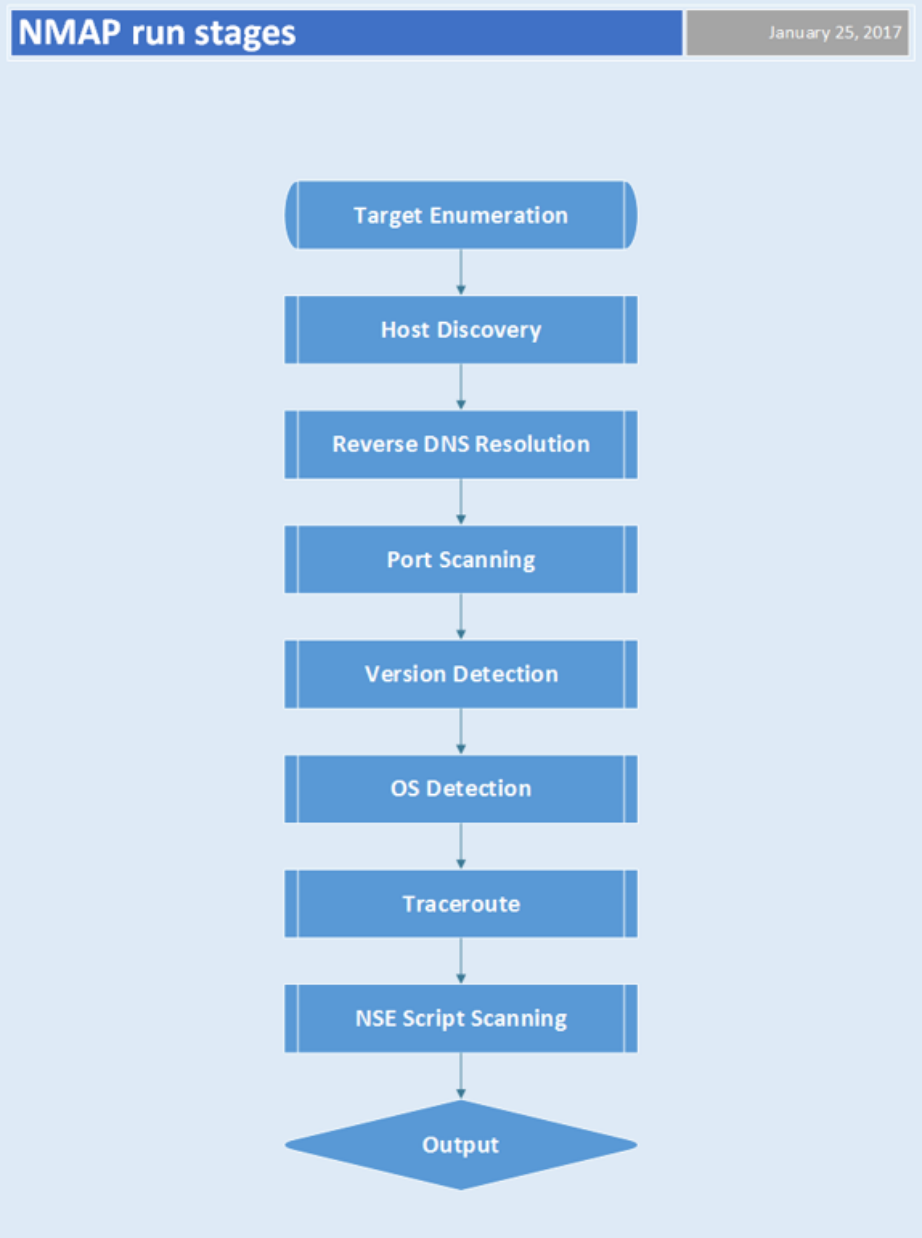

Figure 2.1 Nmap scanning workflow

According to Figure 2.1, the general workflow of an Nmap scan typically follows these several distinct phases. It begins with **target enumeration**, where the user specifies the IP addresses,

ranges, or hostnames to be scanned. Next, Nmap performs **host discovery**, often using techniques such as ICMP echo requests, ARP requests, or TCP probes, to determine which systems are alive on the network. If active hosts are found, Nmap proceeds with **reverse DNS resolution**, mapping IP addresses back to domain names where possible. The next step is **port scanning**, in which Nmap probes selected ports to identify which services are open or closed. Once open ports are detected, **version detection** is applied to determine the specific software and version number running on each service. This may be followed by **operating system detection**, where Nmap analyzes network responses to fingerprint the host's OS. To provide network path visibility, Nmap can also perform a **traceroute**, revealing the intermediate hops between the scanner and the target. Beyond these standard checks, the **NSE (Nmap Scripting Engine)** can be invoked to run specialized scripts that test for vulnerabilities, weak configurations, or even malware indicators. Finally, Nmap generates an **output report**, which can be displayed in formats such as plain text, XML, or logs for further processing.

Nmap's primary strengths lie in its flexibility and extensibility. It can be used for simple discovery scans as well as complex vulnerability assessments through its scripting engine. Nmap is also a lightweight tool, making it well-suited for assessing IoT networks, where devices often run on minimal resources. Furthermore, it is open source and has a large community support system, ensuring continuous updates and contributed scripts for newly discovered vulnerabilities.

Despite its strengths, Nmap is not without limitations. Its command-line interface (CLI) can be intimidating for beginners, particularly those without prior networking or security experience. While graphical front-ends exist, such as Zenmap, they still require familiarity with the tool's many options and scan profiles. Moreover, Nmap focuses primarily on network- and transport-layer assessments, leaving application-layer vulnerabilities such as SQL injection or Cross-Site Scripting (XSS) outside its scope. Finally, while efficient for small- to medium-scale environments, deep scans with NSE can be resource-intensive and may overwhelm low-powered IoT devices, potentially causing service disruption during testing. These challenges highlight the need for integration with complementary tools and for dashboards that can simplify interpretation of Nmap's raw output.

In this project, Nmap was not only reviewed as a general-purpose network exploration and security auditing tool, but was also adopted as one of the core components of the implemented dashboard. It was used to identify live targets, enumerate open ports and services, support

service and version detection, and provide the main input for risk assessment and result visualisation. The Nmap results were further integrated into the centralized dashboard, stored in the database for historical review, and linked to other modules such as telemetry monitoring and reporting. This demonstrated the practical suitability of Nmap as a lightweight and effective scanning tool for small-scale IoT security assessment.

2.2.2 OWASP ZAP: Zed Attack Proxy

The OWASP Zed Attack Proxy (ZAP) is an open-source Dynamic Application Security Testing (DAST) tool designed to identify vulnerabilities in web applications. Developed by the OWASP community, ZAP offers automated scanning, crawling, and fuzzing capabilities, making it particularly suitable for identifying flaws in web interfaces commonly found in IoT devices [10]. Many IoT products, such as smart cameras, routers, and home automation hubs, expose web dashboards or administrative portals. These interfaces are often weakly protected and vulnerable to attacks such as SQL injection, Cross-Site Scripting (XSS), insecure cookies, and missing security headers. ZAP can automatically explore these interfaces, simulate attacks, and generate structured reports that highlight vulnerabilities. Furthermore, ZAP provides a REST API that allows programmatic control, making it possible to integrate its functionality into larger platforms such as security dashboards.

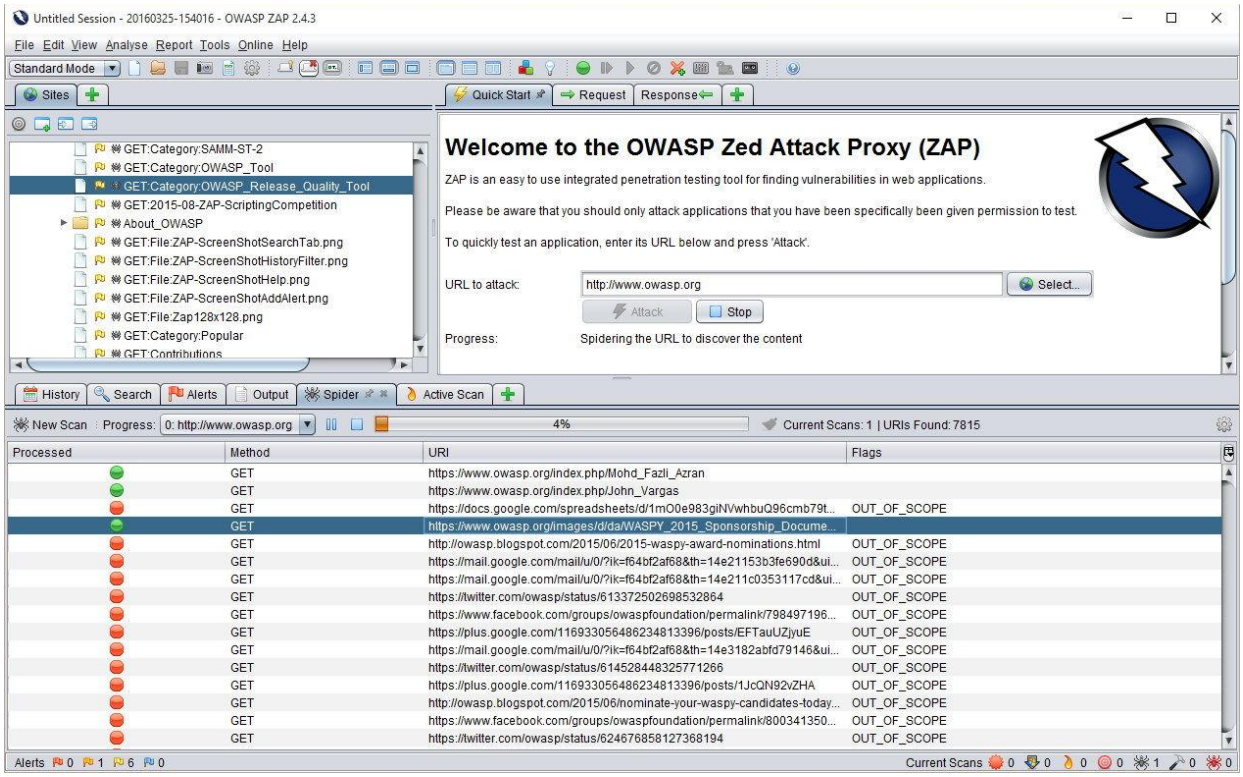

Figure 2.2 OWASP ZAP overview

One major strength of OWASP ZAP is its comprehensive vulnerability detection capabilities for web applications. It supports automated and semi-automated testing, with features ranging from simple passive scanning to advanced active attacks. ZAP is also free and supported by an active global community, ensuring frequent updates and new plugins. The availability of a REST API makes it suitable for automation and integration with other tools, which is highly relevant to projects aiming to create centralized dashboards. Despite these advantages, ZAP has several weaknesses that limit its accessibility for non-experts. Automated scans can generate false positives that require manual verification, which may be challenging for beginners. Its scans are also resource-intensive and may overwhelm low-powered IoT devices with weak processors or limited memory. Another limitation is its narrow focus: ZAP specializes in web vulnerabilities but does not provide visibility into lower-level network services or firmware-related issues. Therefore, while ZAP is a valuable component of IoT penetration testing, it cannot serve as a complete solution without being combined with other tools.

In the implemented system, OWASP ZAP was integrated into the dashboard as the main web vulnerability assessment component for HTTP and HTTPS-based interfaces exposed by IoT or local network devices. Its role in the project extended beyond standalone vulnerability scanning, as the scan results were captured, processed, stored, and presented within the centralized dashboard together with timestamps, risk levels, descriptions, and recommended solutions. The final implementation also supported practical workflow improvements by allowing scan progress feedback and structured result review, making OWASP ZAP suitable for a web-based academic prototype focused on IoT security assessment.

2.3 Industry Approaches and Proprietary Dashboards

On the commercial front, several **enterprise-grade IoT security solutions** have emerged, typically integrated into **Network Access Control (NAC)** systems or extended from **SIEM** platforms. Palo Alto Networks, for instance, offers an IoT security dashboard that enables **device fingerprinting, policy enforcement, and risk scoring**. These systems often use deep packet inspection (DPI) and artificial intelligent (AI) models to classify and monitor IoT device behaviour [11].

However, commercial tools are typically **vendor-locked, costly**, and not well-suited for academic experimentation or small-scale deployment [12]. Their dashboards are often designed

for enterprise infrastructure and are ill-equipped for **active penetration testing**, focusing instead on passive observation and policy management. Furthermore, such systems are usually proprietary, with limited extensibility and customization options, hindering integration with new scanning tools or telemetry protocols.

Open-source IoT dashboards such as **ThingsBoard** or **Node-RED** provide alternatives with visualization capabilities. ThingsBoard, in particular, has been praised for simplifying the display of telemetry from multiple devices via MQTT, or HTTP. Nonetheless, these platforms focus primarily on data monitoring and lack features for active security diagnostics, leaving a gap in their suitability for vulnerability assessment and threat management.

2.3.1 Importance of Dashboards in IoT Security

Dashboards play a key role in modern cybersecurity operations. While Security Information and Event Management (SIEM) platforms like **Splunk**, **ELK Stack**, and **AlienVault OSSIM** are widely used in enterprise environments, they are often **too complex, high skill and knowledge requirements or over-engineered** for lightweight IoT scenarios [13]. These platforms are typically designed for server logs, endpoint behaviour, and enterprise-grade traffic—not constrained IoT ecosystems.

Several academic studies, such as Bella et al. [6], emphasize the benefits of **visual dashboards** for improving analyst productivity and response time. Dashboards that display vulnerability severity, affected assets, and real-time alerts can reduce the time needed to understand and mitigate security risks [14]. Such visual clarity becomes essential when managing **heterogeneous and distributed systems**, such as those found in IoT environments.

However, most existing IoT dashboards focus on **telemetry visualization** rather than **security diagnostics**. Platforms like **ThingsBoard** and **Node-RED** allow users to visualize temperature, humidity, or usage logs, but offer little to no support for real-time scanning or alerting. According to Verma and Sheetlani [4], “a security dashboard is needed to find what type of security controls are required to stop threats,” which underlines the necessity of a **more proactive security monitoring interface**.

Furthermore, many of these platforms suffer from the following limitations:

- They are **passive**: limited to visualizing incoming telemetry without actively probing for threats.

- They are **proprietary or cloud-dependent**, limiting flexibility and control. They **lack modular integration** with security scanning tools like Nmap or ZAP, making them unsuitable for vulnerability assessments.

2.3.2 ThingsBoard: IoT Dashboard Platform

ThingsBoard is an open-source IoT platform designed primarily for device management, telemetry collection, and data visualization [11]. It allows administrators to connect IoT devices, collect sensor readings, and display results through customizable dashboards. With features such as rule-based event processing and data analytics, ThingsBoard is widely used in smart city, industrial, and agricultural projects. From a usability perspective, its drag-and-drop interface makes it easy to configure dashboards without deep programming expertise.

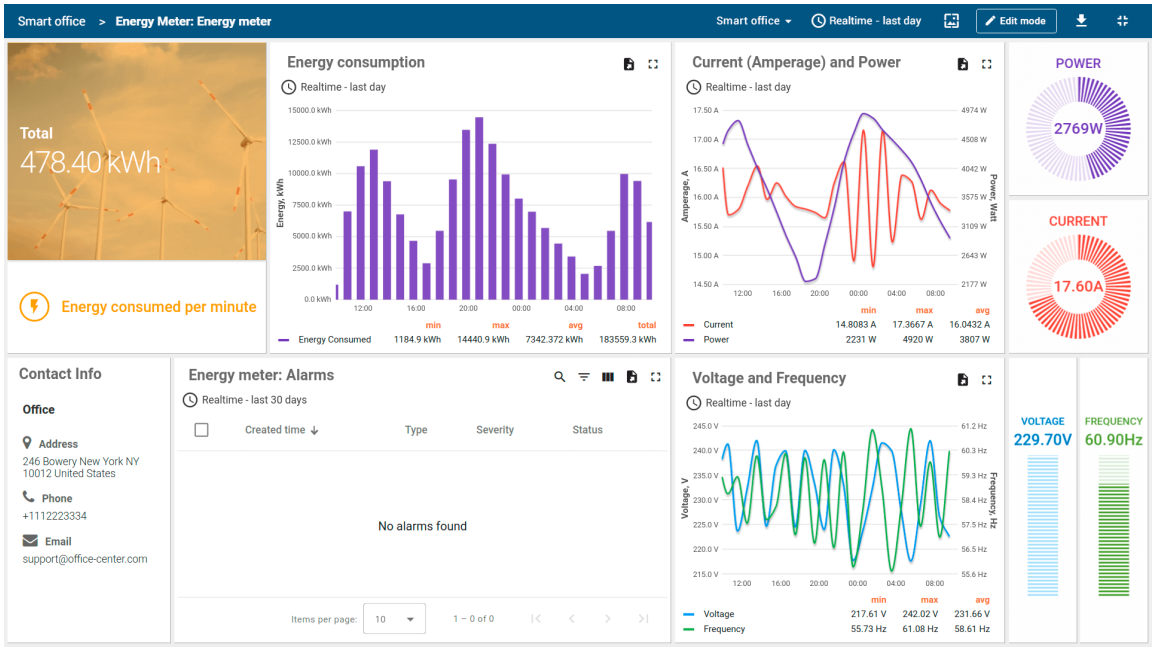

Figure 2.3 ThingsBoards dashboard

The main strength of ThingsBoard is its ability to handle large-scale telemetry data and present it in a visually appealing, real-time dashboard. Its scalability and open-source nature make it attractive for academic and commercial projects. It also supports a wide range of device communication protocols such as MQTT, CoAP, and HTTP, which are common in IoT environment. However, ThingsBoard is not designed for security purposes. It does not provide penetration testing capabilities, nor does it integrate with scanning tools such as Nmap or OWASP ZAP. Its primary focus is monitoring device performance and environmental metrics rather than identifying vulnerabilities. Consequently, while it excels as a data visualization and telemetry management tool, it lacks the security-oriented features necessary to function as an IoT security dashboard.

2.3.3 Cisco Cyber Vision: Industrial IoT Security Platform

Cisco Cyber Vision is a commercial platform designed for monitoring and securing industrial IoT and operational technology (OT) environments [15]. It provides visibility into connected devices by passively monitoring network traffic and performing device fingerprinting. It integrates with Security Information and Event Management (SIEM) systems to provide centralized threat visibility and can enforce network access control policies in collaboration with other Cisco products.

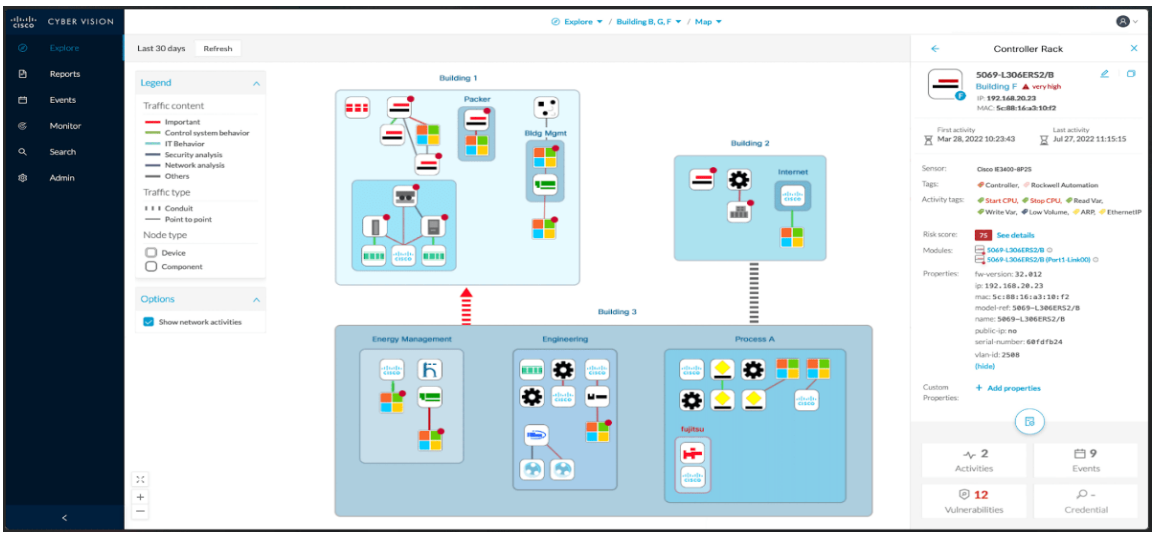

Figure 2.4 Cisco Cyber Vision GUI dashboard

Cisco Cyber Vision's strengths lie in its enterprise-grade reliability and integration with broader security infrastructure. It is capable of handling large-scale industrial deployments and provides detailed visibility into both IT and OT networks. By integrating with Cisco's ecosystem of security products, it offers a holistic solution for organizations that can afford its deployment.

On the other hand, its weaknesses are significant in the context of smaller-scale or educational projects. Its proprietary nature and high licensing costs make it inaccessible for students, researchers, or small organizations. Additionally, Cisco Cyber Vision emphasizes passive monitoring and device fingerprinting rather than active penetration testing. It does not perform vulnerability scans or orchestrate tools like Nmap and ZAP, which limits its relevance to hands-on IoT security testing.

Although these industry platforms demonstrated strong monitoring, device visibility, and centralized management capabilities, they were generally oriented towards enterprise or

industrial use cases and often depended on proprietary ecosystems, complex deployment requirements, or commercial licensing models. In contrast, this project focused on a smaller-scale and more accessible academic implementation that combined open-source security tools, lightweight web deployment, live telemetry support, and integrated reporting within a single dashboard. This distinction helped define the project direction as a practical and educational alternative rather than a full enterprise replacement.

2.4 Gaps Identified in Existing Solutions

The literature review above demonstrates that despite significant progress in IoT security, several key gaps remain, hindering its practical application in small-scale or educational settings. Open-source tools such as Nmap and OWASP ZAP remain mainstays of vulnerability assessment and penetration testing, but they operate independently. Running these tools independently requires advanced technical knowledge, manual correlation of results, and careful interpretation of complex output. For example, Nmap can reveal open ports and exposed services, while ZAP can identify unsecured web endpoints. However, due to the lack of a centralized platform, the results are often fragmented. This fragmentation leads to inefficiencies and increases the risk of overlooking critical vulnerabilities.

Visibility platforms such as ThingsBoard fill another gap by providing real-time dashboards, but their focus is on telemetry and device performance, not security. They lack integration with scanning tools and cannot highlight whether IoT devices are vulnerable to cyberattacks. Similarly, commercial solutions like Cisco Cyber Vision and similar enterprise-grade tools, while providing comprehensive visibility, are prohibitive for small organizations, universities, or individual researchers due to their high cost, proprietary nature, and reliance on vendor ecosystems. These solutions are designed for industrial deployment, so there is still a gap in the market for lightweight and affordable alternatives. Thus, these gaps justify the need for a project such as the centralized IoT security assessment and penetration testing dashboard, which combines automation, visualization, and accessibility into a single platform.

2.5 Review of the Technologies

This section reviews the main technologies selected for the development of the centralized web-based IoT Security Assessment and Penetration Testing Dashboard. The review focuses on the suitability of the chosen hardware platform, firmware or operating system environment, database system, programming language, and algorithmic approaches used in the

implementation. The purpose of this review was to justify the selected technologies based on practicality, compatibility, performance, ease of integration, and suitability for small-scale IoT security assessment environments.

2.5.1 Hardware Platform

The hardware platform used in this project consisted primarily of a personal computer or laptop that acted as the central server for the dashboard, together with IoT-related target devices available within the local network environment. The server machine was responsible for running the Flask backend, MySQL database, frontend dashboard, MQTT communication, and integrated security tools such as Nmap and OWASP ZAP. This type of hardware platform was suitable because it provided sufficient processing capability, memory, and storage for development, scanning tasks, telemetry handling, and report generation in a small-scale environment.

In addition to the server machine, the project environment also involved network-connected devices that acted as scanning targets or telemetry sources. These included common local devices and simulated or real IoT-style endpoints. The use of a standard computer-based deployment platform was practical because it reduced development cost, simplified testing, and supported easy installation of open-source tools and libraries. For an academic prototype, this hardware arrangement was appropriate as it enabled the complete system to be developed and demonstrated without requiring specialised enterprise infrastructure.

2.5.2 Firmware/OS

The software environment of the project depended on the operating system of the host machine and the runtime environment required by the integrated tools. The operating system provided the execution environment for Flask, Python libraries, MySQL connectivity, MQTT communication, Nmap, and OWASP ZAP. A desktop operating system was suitable for this project because it supported both development and deployment tasks in a convenient manner, especially for testing web interfaces, scanning tools, and local network communication.

From the IoT integration perspective, the project also interacted with devices or simulated devices that communicated through MQTT or REST-based telemetry submission. This made the operating system environment important, since it needed to support networking utilities, background scanning processes, Socket.IO communication, and external tool execution. The

chosen environment was appropriate because it allowed smooth integration between the web application, database, telemetry pipeline, and external security assessment tools.

2.5.3 Database

MySQL was selected as the database management system for this project. It was used to store user information, scan results, telemetry records, activity logs, SSL/TLS scan results, ping sweep results, and report-related data. MySQL was a suitable choice because it is widely used, reliable, easy to integrate with Python Flask applications, and appropriate for structured relational data.

The project required persistent storage for several modules rather than only one scanning component. Nmap results needed to be stored for later review and risk visualisation, ZAP results needed session-based and historical retrieval, telemetry data needed device-based tracking, and logs needed user-based recording. MySQL supported these requirements effectively through relational tables and indexed queries. For a centralized dashboard that handled multiple types of structured assessment data, MySQL provided a stable and practical storage solution.

2.5.4 Programming Language

Python was the primary programming language used for backend development in this project. It was used together with the Flask framework to implement routing, API development, scan orchestration, telemetry processing, report generation, and integration with external tools. Python was selected because of its readability, strong library ecosystem, ease of prototyping, and suitability for security automation tasks.

For frontend development, HTML, CSS, and JavaScript were used to implement the web dashboard interface, interactive components, result tables, progress indicators, real-time updates, and user interactions. JavaScript was especially important for handling asynchronous requests, Socket.IO communication, live dashboard updates, and dynamic rendering of scan results and telemetry information. The combination of Python on the backend and JavaScript on the frontend was suitable because it enabled rapid development of a modular, responsive, and interactive web-based dashboard.

2.5.5 Algorithm

The project did not rely on a single complex mathematical algorithm, but it employed a set of practical algorithmic and rule-based approaches to support scanning, monitoring, and result processing. Nmap scanning logic was used to enumerate open ports and services, while OWASP ZAP scanning logic was used to detect web vulnerabilities. In addition, the system applied rule-based processing to derive risk levels from scan findings and to enrich device status information.

For live monitoring, the system used event-driven processing through MQTT, REST input, and WebSocket updates to maintain a real-time telemetry view. Ping sweep operations used subnet-based host probing to identify reachable devices, while SSL/TLS assessment used certificate and protocol inspection to determine issues such as expiry, weak configurations, or hostname mismatch. The dashboard also used queue-based scan handling for certain workflows, especially where sequential or managed scan execution was required. These approaches were appropriate for the project because they prioritised practical integration, usability, and real-time system behaviour rather than theoretical algorithmic complexity.

2.5.6 Summary of the Technologies Review

Based on the technology review, the selected hardware and software stack was suitable for the implementation of the proposed IoT security dashboard. The hardware platform was sufficient for small-scale scanning and monitoring tasks, while the operating system environment supported the required development tools and network utilities. MySQL was appropriate for storing structured security and telemetry data, Python and Flask were effective for backend orchestration, and HTML, CSS, and JavaScript provided a practical frontend environment for dashboard interaction. The scanning, telemetry, and rule-based processing approaches used in the project were also suitable for building a centralized platform that combined security assessment, real-time monitoring, and reporting within one system.

2.6 Chapter Summary

The literature review presented in this chapter highlights the strengths and weaknesses of current tools and platforms in the domain of IoT security assessment. Tools like Nmap and OWASP ZAP are indispensable for network and web-based vulnerability scanning, respectively, but they remain fragmented and challenging for non-experts to use effectively.

Visualization platforms such as ThingsBoard demonstrate the value of centralized dashboards but focus primarily on telemetry rather than vulnerability management. Enterprise-grade solutions such as Cisco Cyber Vision deliver robust monitoring and integration features but are inaccessible to students and small organizations due to cost and complexity.

Taken together, these findings highlight the fact that no solution currently exists that offers a centralized, beginner-friendly, open-source IoT security dashboard that integrates multiple penetration testing tools into a unified platform. Existing environments are either too fragmented, too narrow in scope, or too resource-intensive for small-scale applications. This gap has provided strong impetus for the current project, which aimed to integrate Nmap and OWASP ZAP into a web dashboard built on Flask, HTML/CSS/JavaScript, and MySQL. By automating scan tasks, correlating results, and presenting them in a structured and user-friendly manner, the system aimed to bridge the gap between legacy security tools and high-cost enterprise platforms.

The dashboard will contribute to both research and practice by providing a practical, cost-effective, and educational solution. It demonstrates that open-source tools, when orchestrated within an integrated dashboard, can significantly improve the accessibility and usability of IoT penetration testing, thereby promoting better security practices in small labs, academic environments, and organizations with limited technical or financial resources.

CHAPTER 3

System Methodology/Approach

A development-oriented methodology guided the project. Requirements were elicited by identifying common IoT assessment workflows and the needs of novice users. The system was designed using a modular architecture to separate concerns (scanning, persistence, presentation). Iterative development cycles were used: core backend orchestration and database design were implemented first, followed by frontend visualisation and progressive integration of scanning workflows. The prototype was validated through a series of test cases executed against common devices.

3.1 System Design

3.1.1 System Architecture Diagram

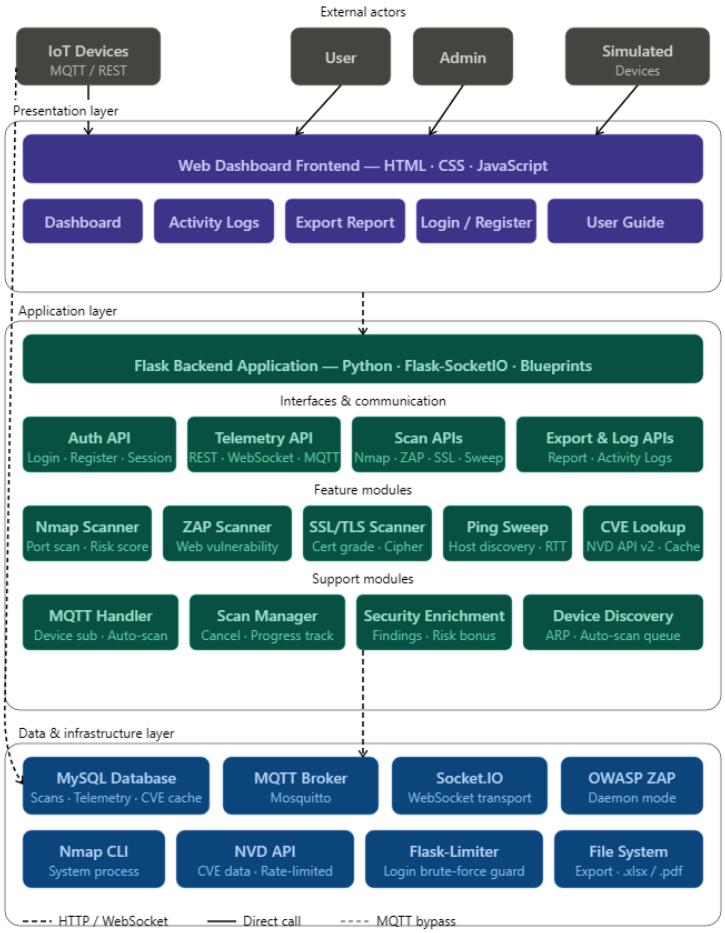

Figure 3.1 System Architecture Diagram for Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard

The system follows a three-tier architecture that separates the presentation layer (frontend), the application layer (backend), and the data and infrastructure layer (external tools and services). This modular design ensures that each component is independent yet integrated, making the system easier to develop, maintain, and extend. Compared to FYP1, the architecture has grown significantly — incorporating real-time device telemetry, automated scanning pipelines, SSL/TLS inspection, network host discovery, CVE intelligence lookup, and a role-based access control system across both user and admin accounts.

1. Presentation Layer (Frontend)

The user interface remains the entry point of the system and is accessible through a web browser. It is developed using HTML, CSS, and JavaScript, with additional visualization support from Chart.js for risk distribution charts and real-time telemetry summaries.

The frontend is now structured into five dedicated pages, each serving a distinct function:

- **Dashboard** — The primary interface where users initiate scans, monitor live telemetry, view network scan results, ZAP findings, SSL/TLS grades, ping sweep results, and receive real-time notifications. All dashboard panels update dynamically via WebSocket without requiring a page reload.
- **Activity Logs** — A persistent log viewer showing a chronological record of all user actions stored in the database. Admins see all users' logs with role badges; regular users see only their own entries.
- **Export Report** — A dedicated reporting page that generates a comprehensive security report from all system data. Users can apply a date range filter and download the report as a formatted Excel workbook (.xlsx) or print it as a PDF.
- **Login / Register** — Handles user authentication and new account creation, now protected by a rate limiter to prevent brute-force attacks.
- **User Guide** — An in-browser documentation page covering all features, role differences, MQTT integration, risk level explanations, and FAQs.

A **custom modal system** replaces all native browser dialogs (alert, confirm, prompt) for a consistent user experience. A WebSocket connection badge in the header provides a live connectivity indicator, and a real-time device counter shows how many devices are online.

2. Application Layer (Flask Backend)

The backend remains built on Python Flask, now extended with Flask-SocketIO for bidirectional real-time communication and Flask-Limiter for rate-controlled endpoints. It is

organised into three functional sub-layers using Flask Blueprints, making each module independently maintainable.

2a. APIs and Communication Interfaces

Flask exposes a set of REST API endpoints grouped by responsibility:

- **Auth API** — Handles login, logout, registration, and session management. Passwords are hashed and user sessions are tracked with a 30-minute inactivity timeout. The login endpoint is rate-limited to mitigate brute-force attacks.
- **Telemetry API** — Accepts device status updates via HTTP POST and relays them to connected browsers over WebSocket in real time. Devices that have not sent data for more than 10 seconds are automatically marked offline.
- **Scan APIs** — Four distinct endpoints handle Nmap scanning, OWASP ZAP scanning, SSL/TLS certificate scanning, and ping sweep. All scan-triggering routes are protected with authentication decorators. Nmap and ZAP scans return a scan ID immediately and execute in background threads, allowing the frontend to poll for progress without blocking.
- **Export and Log APIs** — Serve the full security report data (with optional date range filtering) and activity log entries, with output scoped by role.

2b. Feature Modules

Each major security capability is implemented as a standalone module:

- **Nmap Scanner** — Invokes Nmap via Python's *subprocess* module with service version detection (*-sV*). The scan runs in a background thread managed by the Scan Manager. Results are parsed into structured records and enriched with a risk score and security findings before being stored in MySQL.
- **ZAP Scanner** — Connects to a locally running OWASP ZAP daemon via its REST API. A preflight health check confirms ZAP is available before the scan begins, surfacing a clear error if it is not running. ZAP performs active web vulnerability scanning against HTTP and HTTPS targets discovered on the local network.
- **SSL/TLS Scanner** — Establishes a direct TLS connection to any specified host and port, extracts the certificate using the *cryptography* library, and evaluates the cipher suite, TLS version, expiry, self-signed status, and hostname match. Each host is assigned a letter grade from A to F and results are upserted (one record per host:port) to avoid duplicates on re-scan.

- **Ping Sweep** — Concurrently pings all addresses in a user-supplied CIDR range using *ThreadPoolExecutor*, recording each host's reachability status and round-trip time. Results are upserted by IP address and included in exported reports.
- **CVE Lookup** — After each Nmap scan, the module automatically queries the NVD API v2 for known CVEs matching each detected service and version. Lookup strategies are applied in priority order: product + version fingerprint first (high confidence), then product name alone (medium), then a service keyword mapping (low). Up to five CVEs per service are fetched and stored in a MySQL cache table with a seven-day TTL, eliminating redundant API calls on repeated scans.

2c. Support Modules

- **MQTT Handler** — Subscribes to the MQTT broker and processes incoming telemetry messages from IoT devices publishing to *iot/devices/<device_id>/telemetry*. It also manages the auto-scan queue, triggering a background Nmap scan via the Scan Manager when a previously unseen device appears on the network.
- **Scan Manager** — A shared in-process registry that holds references to all running scan subprocesses. It enables the Stop Scan feature to terminate the OS-level Nmap or ZAP process immediately, and exposes the active scan list for progress tracking and cancellation across the auto-scan queue.
- **Security Enrichment** — A rule-based module that analyses each open port and service for protocol-level risks independent of CVEs — flagging issues such as Telnet exposure, cleartext web services, SMB exposure, SNMP reachability, MQTT running without encryption, and exposed camera management interfaces. Each finding carries a severity label (HIGH, MEDIUM, LOW) and contributes a weighted bonus to the device's overall risk score, capped at a maximum of 35 additional points.
- **Device Discovery** — Performs ARP table inspection and periodic host sweeps to maintain a live list of devices present on the local network. New device arrivals and departures trigger toast notifications and log entries on the dashboard, and new devices can be automatically queued for scanning when the auto-scan toggle is enabled.

3. Data and Infrastructure Layer

Persistent storage and external service dependencies are consolidated in the data and infrastructure layer.

- **MySQL Database** — Stores all persistent application data across multiple tables: scan results (Nmap, ZAP, SSL/TLS, ping sweep), device telemetry, user and admin

accounts, activity logs, and the CVE cache. Role-based queries ensure that regular users only retrieve their own records while admins have full visibility.

- **MQTT Broker (Mosquitto)** — Receives telemetry published by IoT devices and simulated devices. The Flask MQTT handler subscribes to the relevant topic pattern and forwards incoming data into the telemetry pipeline.
- **Socket.IO / WebSocket** — Provides the real-time communication channel between the Flask backend and all connected browser clients. Scan progress events, device status changes, and telemetry updates are all pushed over this connection without polling.
- **OWASP ZAP (Daemon Mode)** — Runs as a background process on the server, exposing a local REST API that the ZAP Scanner module calls to initiate and retrieve active scan results.
- **Nmap CLI** — Invoked as a system subprocess for each port scan. Input IP addresses are validated against a strict regex pattern before being passed to the process to prevent command injection.
- **NVD API** — The external National Vulnerability Database API v2, queried by the CVE Lookup module. Requests are rate-limited and results are cached locally in MySQL to avoid repeated external calls.
- **Flask-Limiter** — Enforces request rate limits on the login endpoint to block brute-force credential attacks.
- **File System** — Used by the Export module to generate and serve .xlsx workbooks (via openpyxl) and print-formatted HTML pages for PDF export via the browser's print dialog.

4. System Workflow Summary

The interaction between all components follows an expanded logical flow:

1. **Actor → Frontend** — A user or admin accesses the dashboard through a browser. IoT devices and simulated devices connect directly to the MQTT Broker, bypassing the frontend entirely.
2. **Frontend → Flask** — User interactions (initiating scans, loading results, exporting reports) are sent to Flask via HTTP requests. The WebSocket connection established at login keeps the browser updated in real time.
3. **Flask → Feature Modules** — Flask routes incoming requests to the appropriate module: Nmap Scanner, ZAP Scanner, SSL/TLS Scanner, Ping Sweep, or CVE Lookup. Scans are dispatched to background threads and tracked by the Scan Manager.

4. **Feature Modules → External Tools** — Nmap is invoked as a subprocess, ZAP is called via its REST API, NVD is queried for CVE data, and TLS connections are made directly for SSL inspection.
5. **Feature Modules → MySQL** — Parsed and enriched results are written to the database. CVE results are cached. All writes are scoped with the authenticated user's ID for role-based retrieval.
6. **MySQL → Flask → Frontend** — Flask retrieves structured results from MySQL and delivers them to the frontend via REST responses or WebSocket events, where they are rendered in tables, risk charts, telemetry cards, and notification feeds.
7. **MQTT Broker → MQTT Handler → Scan Manager** — Device telemetry arriving via MQTT is processed by the handler, which updates the telemetry table and, if the auto-scan toggle is enabled, submits the device's IP to the Scan Manager queue for automatic Nmap scanning.

3.1.2 Use Case Diagram and Description

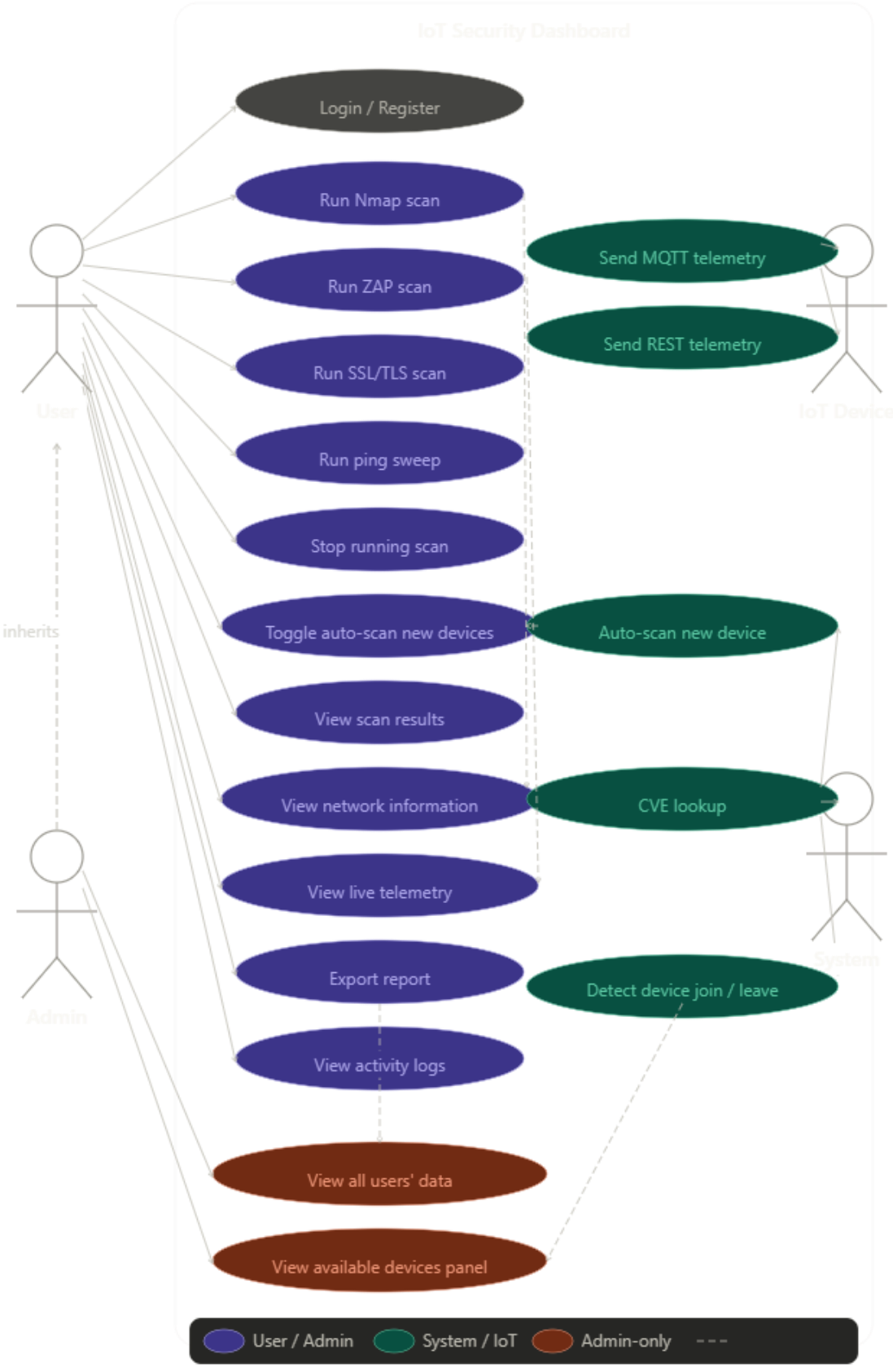

Figure 3.2 Use Case Diagram for Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard

The use case diagram above illustrates the complete set of interactions between all actors and the IoT Security Dashboard system. The system boundary encompasses nineteen use cases distributed across four actors: User, Admin, IoT Device, and System. Each actor represents a distinct role with its own set of responsibilities and access privileges within the platform.

Actors

User is the primary human actor of the system. A regular user accesses the dashboard through a web browser after authenticating via the Login / Register use case. Once authenticated, the user can initiate all four types of security scans — Nmap, ZAP, SSL/TLS, and Ping Sweep — as well as stop any scan that is currently running and toggle the auto-scan new devices setting. Users can also view scan results, network information, live device telemetry, exported reports, and their own activity logs. All data visible to a regular user is scoped exclusively to their own account — they cannot access other users' scan results or logs.

Admin is an elevated human actor who inherits the complete set of use cases available to the User. This is represented in the diagram by the dashed inheritance arrow from Admin to User, indicating that everything a User can do, an Admin can also do without restriction. In addition to inherited capabilities, the Admin has exclusive access to two use cases not available to regular users: viewing all users' data across the entire system, and accessing the Available Devices panel which displays a live list of all devices currently detected on the local network. Admin accounts are created directly in the database by the system administrator rather than through the public registration page.

IoT Device represents any physical or simulated device that communicates with the dashboard autonomously, without going through the web frontend. IoT devices interact with the system through two use cases: sending telemetry via the MQTT broker by publishing to a structured topic, or sending telemetry directly via the REST API using an HTTP POST request. Both pathways feed into the View Live Telemetry use case through an «include» relationship, meaning that any telemetry received — regardless of transport — is always reflected in the live telemetry panel visible to authenticated users.

System represents the automated backend processes that execute independently of direct human interaction. The System actor is responsible for three use cases: automatically scanning

newly detected devices when the auto-scan toggle is enabled, performing CVE lookups against the NVD API after each Nmap scan completes, and detecting when devices join or leave the local network. These automated behaviours are triggered by internal events rather than user requests, which is why they are attributed to a separate System actor rather than to the User or Admin.

Use Case Relationship

Association lines connect each actor to the use cases they directly initiate. Solid lines are drawn from User and Admin to all shared use cases on the left side of the system boundary, and from IoT Device and System to their respective use cases on the right side.

Summary of Access Control

The use case diagram also serves as a visual representation of the system's role-based access control model. Purple use cases are accessible to both User and Admin roles. Coral use cases are restricted exclusively to Admin accounts. Teal use cases on the right side represent automated system or device-initiated actions that operate independently of the user interface. This colour-coded separation makes it immediately clear which parts of the system are open to all authenticated users, which require elevated privileges, and which are driven by the system itself rather than by human interaction.

3.1.3 Activity Diagram

Activity diagrams use the Unified Modelling Language (UML) notation to describe the step-by-step flow of control through each major workflow in the system. Each diagram is presented using horizontal swim lanes that assign every action to the responsible party — User, Flask Backend, Database, external tools, or the frontend — making it immediately clear who performs what and when. Decision nodes (diamond shapes) represent branching logic, while filled-circle start nodes and double-circle end nodes mark the boundaries of each workflow.

Six activity diagrams are presented in this section, covering the most significant workflows in the system: user authentication and registration, report export, ZAP web application scanning, SSL/TLS certificate scanning, ping sweep host discovery, and IoT device telemetry ingestion. The Nmap port scan activity diagram is presented as AD1 earlier in this section.

AD1 Run Nmap Scan

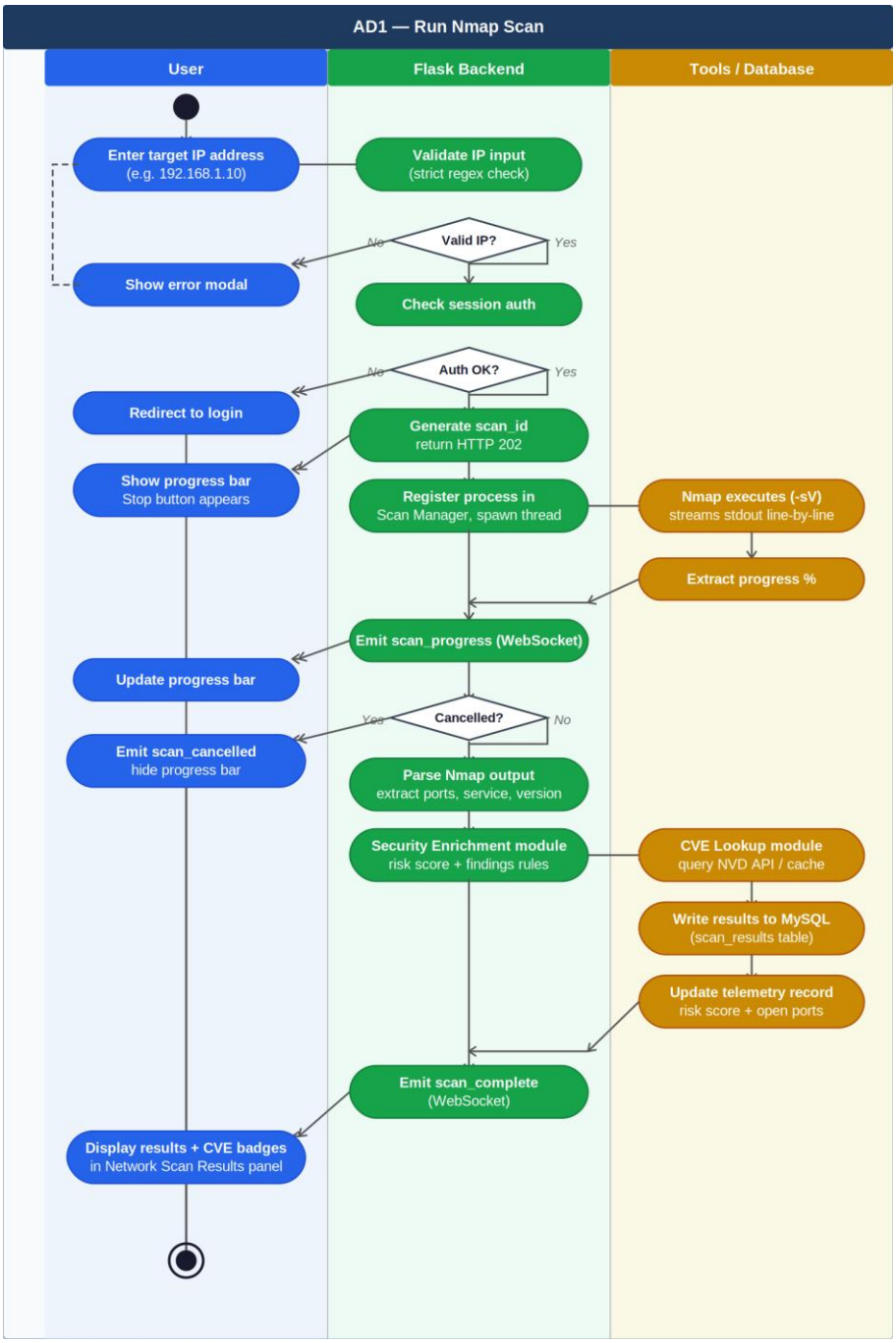

Figure 3.3 Activity Diagram 1 — Run Nmap Scan

This diagram illustrates the complete Nmap port scan lifecycle across three swim lanes: User, Flask Backend, and Tools / Database. The User enters a target IP address which Flask immediately validates against a strict regular expression with invalid input loops back to an error modal. Flask then verifies the active session, redirecting unauthenticated requests to the login page. On success, Flask generates a scan_id, returns HTTP 202, and spawns a background thread registered with the Scan Manager. The User sees a progress bar and a Stop button. Nmap

executes as a system subprocess, streaming progress percentage back to the frontend via WebSocket. A cancellation decision node allows the User to terminate the scan at any point. On completion, Flask parses the output, runs the Security Enrichment module to calculate a risk score and findings, queries the CVE Lookup module against the NVD API, writes all results to MySQL, updates the telemetry record, and emits a scan_complete WebSocket event — causing the results table with CVE badges to appear on the dashboard.

AD2 User Login / Register

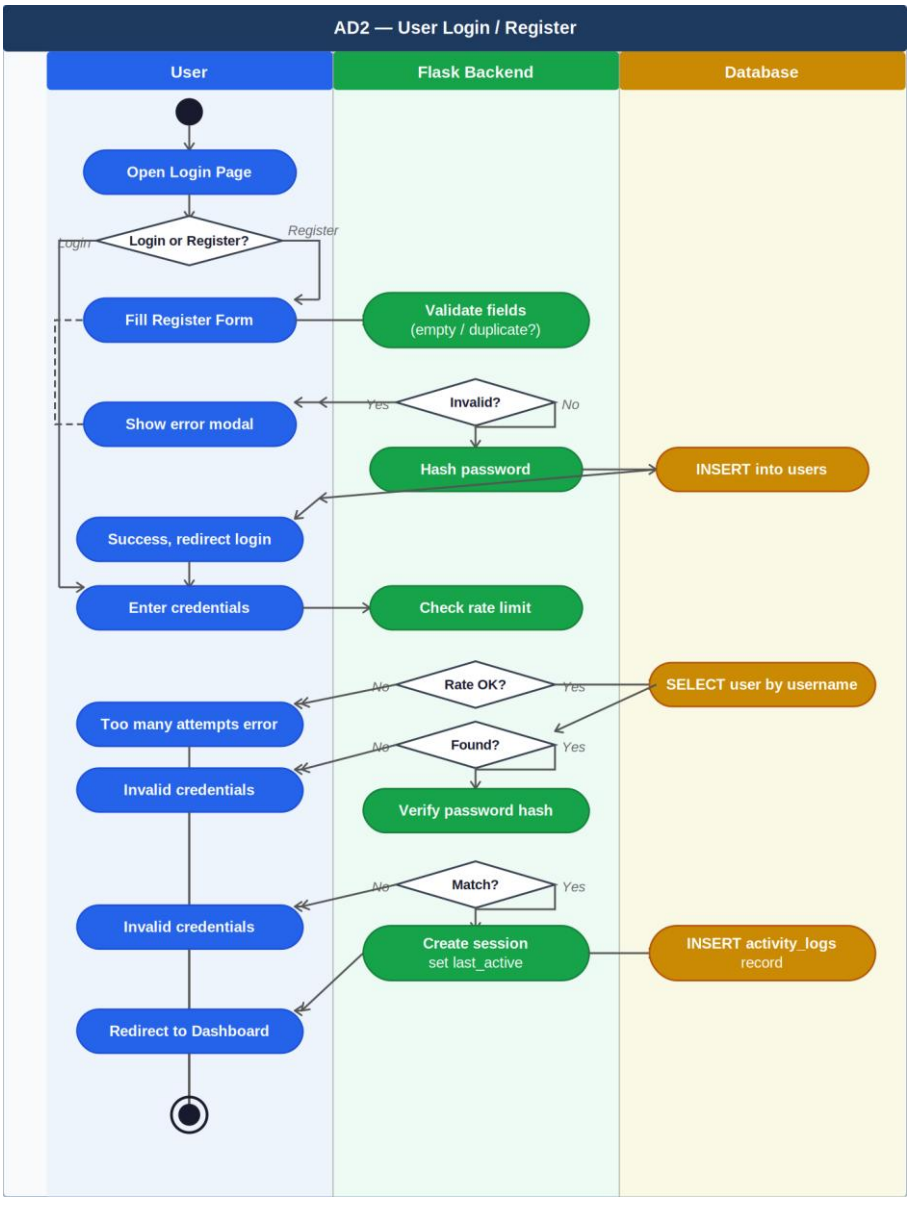

Figure 3.4 Activity Diagram 2 — User Login / Register

This diagram illustrates the complete login and registration flow across three swim lanes: User, Flask Backend, and Database. The User opens the login page and chooses between login and

register. On the register path, the User fills the form and Flask validates the fields — a decision node checks for empty fields or duplicate usernames, looping back to the form with an error modal if invalid. If valid, Flask hashes the password and the Database inserts the new user record. On the login path, Flask enforces the rate limiter before querying the Database for the supplied username. A decision node checks whether the user was found, and if so, whether the password hash matches. Both failure cases return an invalid credentials modal to the User. On success, Flask creates a server-side session, the Database logs the activity, and the User is redirected to the Dashboard.

AD3 Export Report

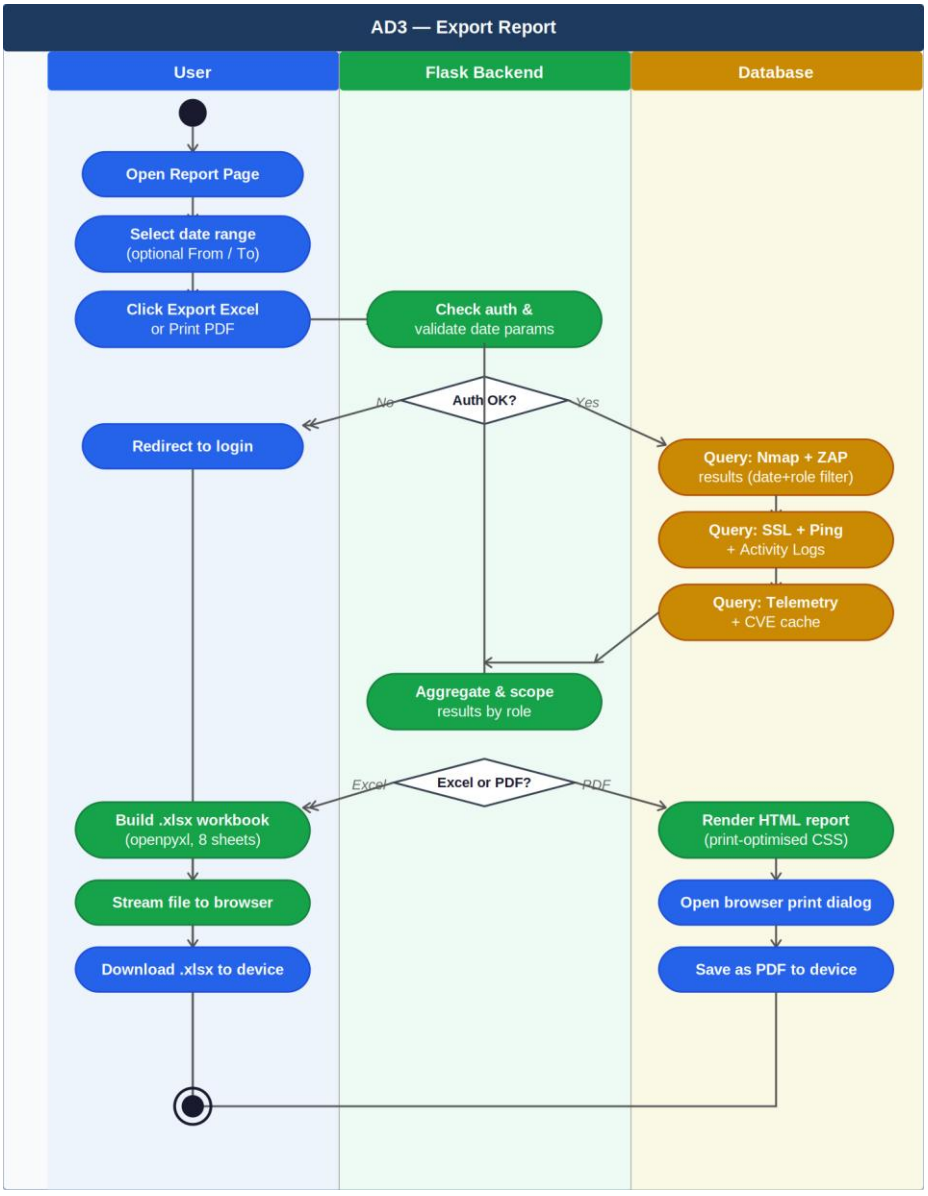

Figure 3.5 Activity Diagram 3 — Export Report

This diagram shows the report export workflow across User, Flask Backend, and Database lanes. The User selects an optional date range (From and To) and clicks either Export Excel or Print PDF. Flask validates authentication and the date parameters before querying the Database across multiple tables like Nmap results, ZAP results, SSL/TLS results, ping sweep data, activity logs, and telemetry which all filtered by role and date range. The aggregated data is then scoped by role. A decision node branches the flow: the Excel path builds a multi-sheet workbook using openpyxl and streams the file to the browser as a download, while the PDF path renders a print-optimised HTML page and opens the browser print dialog for the User to save as PDF.

AD4 Run ZAP Scan

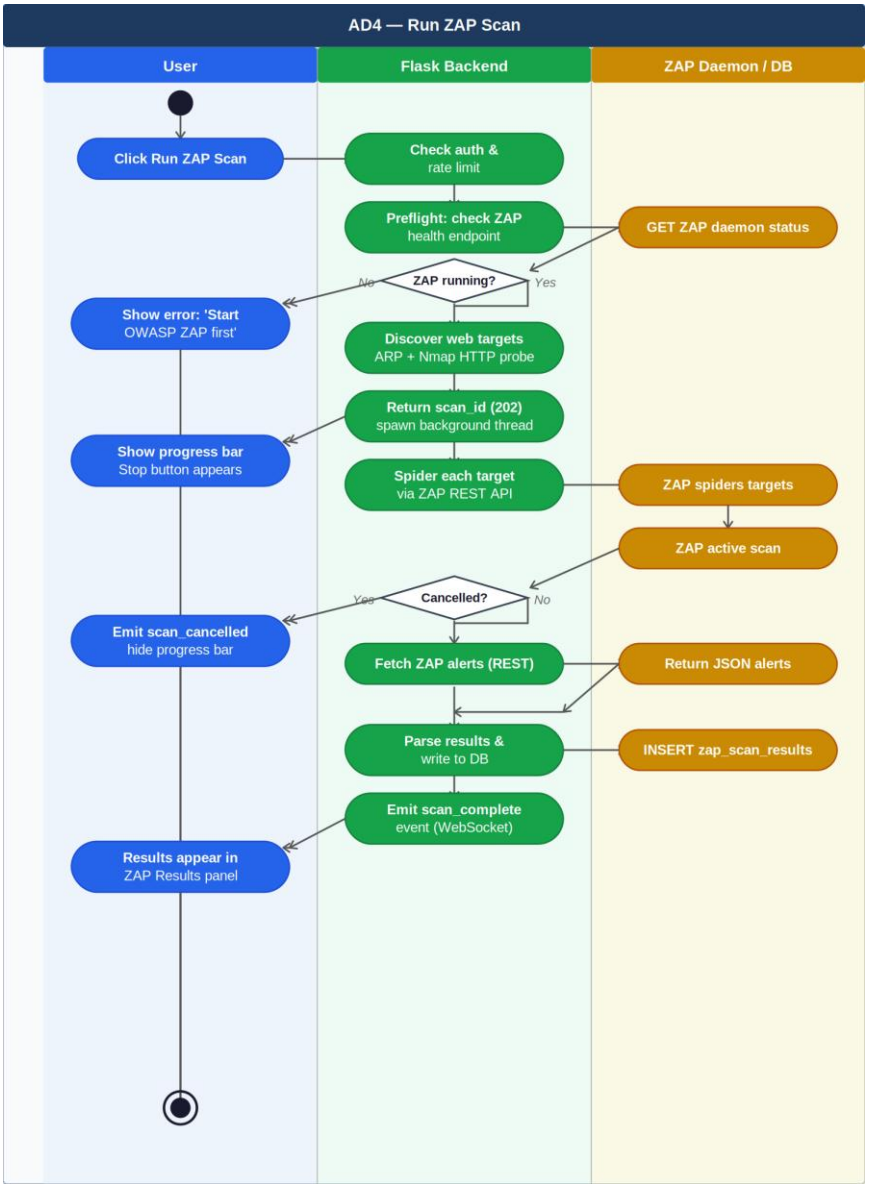

Figure 3.6 Activity Diagram 4 — Run ZAP Scan

This diagram covers the full ZAP scan lifecycle across User, Flask Backend, and ZAP Daemon/DB lanes. After the User initiates the scan, Flask performs a preflight health check against the ZAP daemon, if ZAP is not running a clear error is immediately shown. On success, Flask discovers web targets via ARP and Nmap, then returns a scan_id with HTTP 202 and spawns a background thread. The User sees a progress bar and a Stop button. The background thread sends spider and active scan requests to the ZAP daemon. A cancellation decision node checks whether Stop was pressed, if so, a scan_cancelled event is emitted. Otherwise, Flask fetches JSON alerts from ZAP, parses them, writes results to the database, emits a scan_complete WebSocket event, and the results appear in the ZAP Results panel.

AD5 Run SSL/TLS Scan

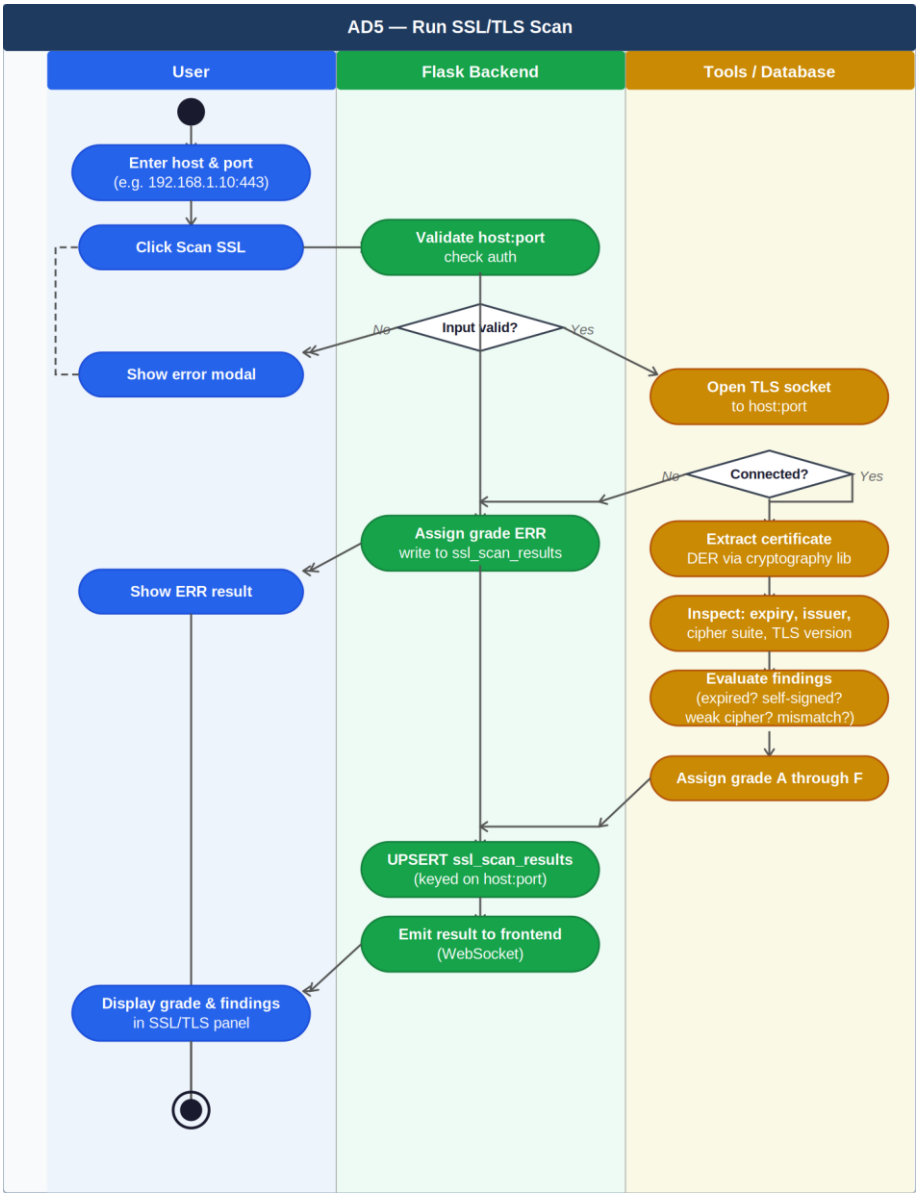

Figure 3.7 Activity Diagram 5 — Run SSL/TLS Scan

This diagram describes the SSL/TLS scan flow across User, Flask Backend, and Tools/Database lanes. The User enters a hostname and port. Flask validates the input and authentication. A TLS socket connection is opened directly to the target, if it fails, grade ERR is assigned and written to the database immediately. If connected, the raw certificate is extracted and parsed using the cryptography library. The scanner inspects the subject, issuer, expiry date, cipher suite, and TLS protocol version, then evaluates a set of findings rules covering expired certificates, self-signed issuers, weak cipher suites, outdated TLS versions, and hostname mismatches. A letter grade of A through F is assigned. Results are written using an UPSERT keyed on host:port to avoid duplicates on re-scan, and the frontend is updated via WebSocket.

AD6 Run Ping Sweep

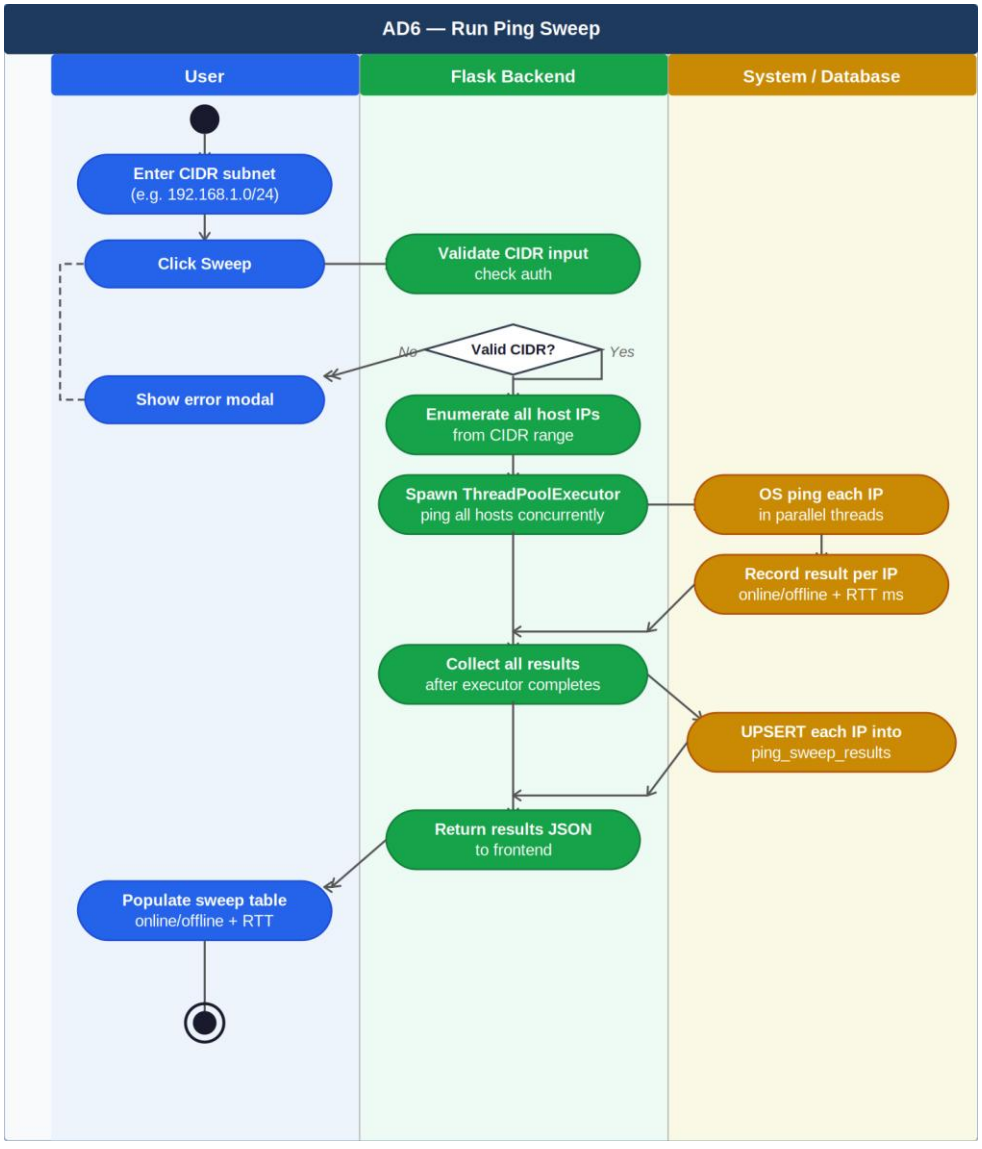

Figure 3.8 Activity Diagram 6 — Run Ping Sweep

This diagram shows the ping sweep workflow across User, Flask Backend, and System/Database lanes. The User enters a CIDR subnet and clicks Sweep. Flask validates the CIDR notation and authentication, then enumerates all host IP addresses in the range. A ThreadPoolExecutor spawns concurrent ping threads — each issues an OS ping command to one IP in parallel, capturing whether the host is online or offline and the round-trip time in milliseconds. Once all threads complete, Flask collects the full result set, issues a batch UPSERT to the ping_sweep_results table keyed on IP address so re-scans update rather than duplicate, and returns JSON results to the frontend where the sweep table is populated.

3.1.4 System Workflow

The workflow of the system can be divided into five main phases: **device discovery**, **vulnerability scanning**, **data storage**, **analysis and visualization**, and **alerts generation**.

1. **Device Discovery and Target Input**: The process begins when a user logs into the dashboard and specifies the IP address or hostname of the IoT device or network to be tested. The frontend sends this request to the Flask backend via defined HTTP routes.
2. **Scanning Phase**:
    - For **network-level assessment**, Flask invokes Nmap using the subprocess library. Nmap scans the specified target, identifies open ports, and reports service details.
    - For **web application assessment**, Flask communicates with OWASP ZAP through its REST API. ZAP crawls the device's web interface and performs active scans to detect vulnerabilities such as SQL injection or XSS.
3. **Data Storage Phase**: The raw results from Nmap and ZAP are parsed into structured data formats. Flask then inserts these processed results into MySQL tables, which serve as the central repository.
4. **Analysis and Visualization Phase**: Once results are stored, Flask retrieves relevant entries from the database and passes them to the frontend. The frontend then presents the results in charts, tables, and summary reports. Users can easily filter by device, vulnerability type, or severity level.
5. **Alert Generation Phase**: High-severity vulnerabilities detected by either Nmap or ZAP are highlighted as alerts in the dashboard. These alerts allow users to quickly identify critical issues that require immediate attention.

This workflow ensures a smooth, end-to-end process from user initiation to actionable results, making IoT penetration testing more structured and user-friendly.

Phase	Process
Device Discovery	User inputs target IP through dashboard, request sent to Flask
Scanning	Flask triggers Nmap via subprocess (network scan) or ZAP via REST API (web scan)
Data Storage	Raw results parsed by Flask and inserted into MySQL tables
Analysis & Visualization	Frontend retrieves structured data from Flask and displays tables/charts
Alerts Generation	High-severity findings highlighted as real-time alerts in dashboard

Table 3.1 Workflow's phases and processes

3.2 Estimated Cost

The development of this Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard did not incur any financial cost, as it relies entirely on existing personal hardware and a fully open-source software stack. No additional devices, licenses, or specialized equipment were purchased specifically for this project.

The host machine used throughout development and testing was a personal laptop running Windows 11, which provided sufficient processing power, memory, and storage to run all system components simultaneously, including the Flask backend, MySQL server, Mosquitto MQTT broker, and OWASP ZAP daemon. Since this hardware was already available prior to the project, it represents zero additional expenditure.

All software components used in the project are open-source and freely available. The development environment was Visual Studio Code, a free IDE distributed by Microsoft. The backend was built on Python and the Flask framework, both of which are freely available under open-source licenses. The database layer used MySQL Community Edition, which is freely available for non-commercial use. The frontend was built entirely with HTML, CSS,

JavaScript, and Chart.js — all open-source web technologies with no licensing requirements. The network scanning tools, Nmap and OWASP ZAP, are community-maintained open-source projects. All supporting Python libraries — including Flask-SocketIO, mysql-connector-python, paho-mqtt, cryptography, bcrypt, openpyxl, python-nmap, and zaproxy — are freely distributed through the Python Package Index (PyPI). The MQTT broker, Eclipse Mosquitto, is also open-source and free to use. Table 3.2 summarises all components and their associated cost.

Component	Type	Tool / Technology	Cost (RM)
Host machine	Hardware	Personal laptop — Windows 11	RM 0.00
Development environment	Software	Visual Studio Code	RM 0.00
Backend framework	Software	Python 3.x, Flask 3.1.1	RM 0.00
Database	Software	MySQL Community Edition	RM 0.00
Frontend	Software	HTML, CSS, JavaScript, Chart.js	RM 0.00
Network scanner	Software	Nmap 7.94	RM 0.00
Web vulnerability scanner	Software	OWASP ZAP 2.15	RM 0.00
MQTT broker	Software	Eclipse Mosquitto 2.x	RM 0.00
Python libraries (all)	Software	Flask-SocketIO, paho-mqtt, cryptography, bcrypt, openpyxl, mysql-connector-python, python-nmap, zaproxy (via PyPI)	RM 0.00
CVE intelligence	External API	NVD API v2 (public, free)	RM 0.00
Version control	Software	Git / GitHub (free tier)	RM 0.00

Table 3.2 Estimated Project Cost Summary

As shown in Table 3.2, the total estimated cost of developing this project is zero Ringgit Malaysia (RM 0.00). Every component in the system — from the scanning tools and backend framework to the database, MQTT broker, and Python libraries — is available at no cost under open-source or free-tier licenses. This demonstrates a key advantage of the chosen technology stack: a fully functional IoT security assessment platform can be developed and deployed without any software licensing expenditure, making it particularly suitable for academic, research, and small-organisation use cases.

CHAPTER 4

System Design

This chapter presents the detailed system design of the Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard. While Chapter 3 introduced the overall system architecture and high-level methodology, this chapter focuses on the internal design of each software component, the specifications of the technologies and libraries used, and the interactions between subsystems. The goal of this chapter is to provide sufficient technical detail so that the system can be understood, reproduced, and extended by future developers.

4.1 System Block Diagram

The system block diagram in Figure 4.1 provides a component-level view of the dashboard that complements the high-level system architecture described in Chapter 3. Whereas the architecture diagram in Section 3.3.1 illustrated the three-tier structure and the data flow between the frontend, backend, and external services, the block diagram here focuses on the internal modules within the application layer and how they are interconnected. Each named block represents a self-contained software module, and the arrows indicate the direction of data or control flow between them.

The system is composed of six major functional blocks:

- User Interface (Frontend)
- Flask Web Application (Core Backend)
- Security Scanning Modules
- Real-Time Communication Subsystem
- Data Persistence Layer
- External Services and Tools

The **User Interface** block encompasses all browser-rendered pages — the main dashboard, activity logs page, report export page, login and registration pages, and the in-app user guide. These pages communicate with the Flask backend through HTTP REST API calls and a persistent WebSocket connection managed by Socket.IO.

The **Flask Web Application** block forms the core of the system. It is structured into Flask Blueprints, with each Blueprint grouping a related set of API endpoints. The Auth Blueprint handles login, logout, registration, and session management. The Telemetry Blueprint receives device updates and relays them to the frontend. The Scan Blueprint exposes endpoints for triggering Nmap, ZAP, SSL/TLS, and ping sweep operations. The Export Blueprint assembles and serves report data. The Logs Blueprint returns activity records scoped by the authenticated user's role.

The **Security Scanning Modules** block contains five independent modules: the Nmap Scanner, the ZAP Scanner, the SSL/TLS Scanner, the Ping Sweep module, and the CVE Lookup module. Each module is invoked by the corresponding API endpoint and returns structured results to the Flask core for database storage and frontend delivery.

The **Real-Time Communication Subsystem** block consists of the MQTT Handler and the Flask-SocketIO layer. The MQTT Handler subscribes to the Mosquitto broker and processes incoming telemetry from IoT devices, which it forwards to the frontend over WebSocket. The Scan Manager within this block tracks all active scan subprocesses, supports scan cancellation, and manages the auto-scan queue that is triggered when new devices are detected.

The **Data Persistence Layer** block is the MySQL database, which stores all structured application data: user and admin accounts, scan results from Nmap, ZAP, SSL/TLS, and ping sweep, device telemetry records, activity logs, and the cached CVE lookup results.

The **External Services and Tools** block covers the four external dependencies the system integrates: the Nmap command-line tool invoked as a subprocess, the OWASP ZAP daemon accessed via its local REST API, the Mosquitto MQTT broker that IoT devices and the simulator publish to, and the National Vulnerability Database (NVD) API v2 used for CVE enrichment after each Nmap scan.

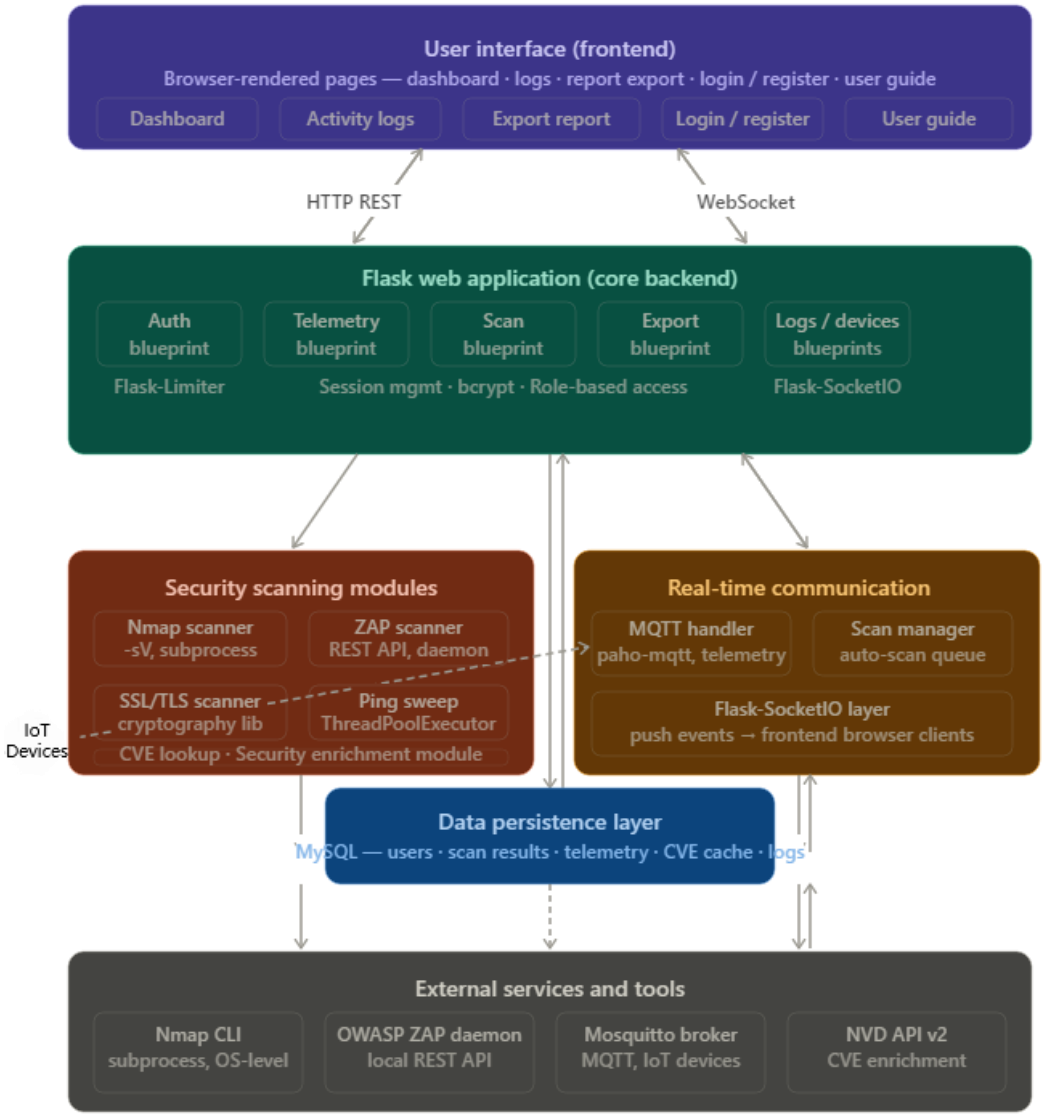

Figure 4.1 System Block Diagram for Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard

The key distinction between this block diagram and the Chapter 3 architecture is the level of abstraction. The architecture diagram showed what the system does and how the three tiers relate. This block diagram shows how the system is built — naming each functional module, its role, and the connections that must exist between modules for the system to operate correctly.

4.2 System Components Specifications

This section specifies the software components, libraries, and external tools that constitute the dashboard system. Each component is described in terms of its version, role within the system, and the reason it was selected over alternatives.

4.2.1 Backend Framework — Python Flask

Flask version 3.1.1 was used as the backend web framework. Flask was selected for its lightweight and modular design, which allowed each feature of the dashboard to be developed as an independent Blueprint. Unlike heavier frameworks such as Django, Flask does not impose a rigid project structure, making it straightforward to integrate subprocess-based tools such as Nmap and daemon-based tools such as OWASP ZAP. Flask-SocketIO 5.5.1 was added to enable bidirectional real-time communication over WebSocket. Flask-Limiter was integrated to enforce rate limits on the authentication endpoint, mitigating brute-force login attempts. Flask-CORS 6.0.1 was configured to permit cross-origin requests during development. The entry point of the Flask application is *app.py*, which registers all Blueprints and initialises the SocketIO instance with the Eventlet asynchronous server.

4.2.2 Database — MySQL with mysql-connector-python

MySQL Server was used as the relational database management system. The database stores all persistent application data across ten logically distinct tables: users, admins, scan_results (Nmap), zap_results, telemetry, ping_sweep_results, ssl_tls_results, activity_logs, cve_cache, and network_info. The Python library mysql-connector-python 9.3.0 was used to establish connections from Flask to MySQL. All queries are parameterised to prevent SQL injection. Role-based filtering is applied at the query level — regular users retrieve only their own records, while admin accounts have unrestricted visibility. A seven-day TTL on the cve_cache table ensures that CVE lookup results remain reasonably current without incurring repeated API calls.

4.2.3 Network Scanning — Nmap and python-nmap

Nmap was invoked as a system subprocess using Python's built-in subprocess module, with the -sV flag to enable service and version detection. The python-nmap 0.7.1 library was also included as a helper for XML output parsing. Input IP addresses are validated against a strict regular expression before being passed to the subprocess to prevent command injection. Scan operations are dispatched to background threads managed by the Scan Manager, allowing the frontend to receive a scan ID immediately and poll for progress without blocking the Flask server. Upon scan completion, each detected port and service is scored and enriched by the Security Enrichment module before being written to MySQL. The Scan Manager also provides cancellation support, allowing users to terminate an active Nmap process via the Stop Scan button.

4.2.4 Web Vulnerability Assessment — OWASP ZAP

OWASP ZAP (Zed Attack Proxy) was used for automated web vulnerability assessment of IoT device web interfaces. ZAP runs as a local daemon, and the Flask backend communicates with it through its REST API using the *python-owasp-zap-v2.4* and *zaproxy 0.4.0* Python packages. Before initiating a scan, the ZAP Scanner module performs a preflight health check to confirm that the ZAP process is active; if it is not, a clear error is returned to the frontend rather than timing out silently. Two scan modes are supported: a quick passive scan and a full active scan. ZAP scan results — including alert name, risk level, description, evidence, and recommended solution — are parsed and stored in the zap_results MySQL table. Results are scoped per user session to support multi-user operation.

4.2.5 SSL/TLS Certificate Scanner — cryptography

The SSL/TLS Scanner module establishes a direct TLS handshake with each target host and port using Python's built-in *ssl* module, then inspects the peer certificate using the *cryptography 46.0.7* library. For each host, the scanner evaluates six security attributes: TLS version in use, cipher suite strength, certificate expiry date, whether the certificate is self-signed, whether the common name matches the hostname, and whether the certificate chain is complete. Based on these attributes, the scanner assigns a letter grade from A to F following a weighted penalty model. Results are upserted by host-port pair to avoid duplicate records on repeated scans.

4.2.6 Host Discovery — Ping Sweep

The Ping Sweep module accepts a CIDR subnet range from the user and concurrently pings all host addresses within the range using Python's *ThreadPoolExecutor* from the *concurrent.futures* standard library. For each address, the module records the IP, hostname (resolved where available), round-trip time in milliseconds, reachability status, timestamp, and the user who initiated the sweep. Results are upserted by IP address so that re-sweeping a subnet updates existing records rather than creating duplicates.

4.2.7 CVE Intelligence — NVD API v2

The CVE Lookup module enriches Nmap scan results with known vulnerability data from the National Vulnerability Database (NVD) API v2. Following each Nmap scan completion, the module iterates over every detected service and queries the NVD API using a three-tier lookup strategy: product and version fingerprint first (highest confidence), then product name alone, then a service-to-keyword mapping for common services. Up to five CVEs per service are

retrieved and stored in the cve_cache table with a seven-day expiry. If a cached entry exists and has not expired, the module returns the cached result directly, avoiding redundant external API calls.

4.2.8 Real-Time Telemetry — MQTT and Flask-SocketIO

IoT device telemetry is ingested through two pathways. The first pathway is MQTT: devices publish JSON-formatted status messages to the topic *iot/devices/<device_id>/telemetry* on the Mosquitto broker. The MQTT Handler module (implemented in *mqtt_handler.py*) subscribes to this topic using the *paho-mqtt 2.1.0* library, parses incoming messages, writes telemetry records to MySQL, and broadcasts updates to all connected browser clients via Flask-SocketIO. The second pathway is a REST API endpoint (*POST /api/telemetry*) for devices that cannot use MQTT. Both pathways converge at the same telemetry pipeline and trigger the same WebSocket broadcast. Devices that have not submitted data within ten seconds are automatically marked offline.

4.2.9 Report Export — OpenPyXL and Print-Formatted HTML

The Export module generates security reports in two formats. The Excel export uses the openpyxl 3.1.5 library to build a multi-sheet workbook containing separate sheets for telemetry, Nmap results, ZAP results, SSL/TLS results, ping sweep results, activity logs, and an admin user summary. Users may apply a date range filter before generating the workbook. The printable report format is a print-optimised HTML page rendered in the browser, which can be saved as PDF using the browser's native print-to-PDF function.

4.2.10 Authentication and Session Management — Flask and bcrypt

User authentication is handled by Flask's session mechanism, with passwords hashed using the *bcrypt 4.3.0* library. Sessions expire after thirty minutes of inactivity. The login endpoint is rate-limited by Flask-Limiter to prevent brute-force attacks. Two account roles exist: regular user and admin. Admin accounts are created directly in the database and are not accessible through the public registration page. All API endpoints that return or modify data apply role-based filtering to enforce access control at the data layer.

Table 4.1 below summarises all major software components and their versions.

Component	Technology / Library	Version	Purpose
Backend Framework	Python Flask	3.1.1	Core web server and API routing
Real-Time Communication	Flask-SocketIO / Eventlet	5.5.1 / 0.40.2	WebSocket push for live updates
Database	MySQL + mysql-connector-python	9.3.0	Persistent storage for all application data
Network Scanner	Nmap + python-nmap	7.94 / 0.7.1	Port and service detection on IoT devices
Web Vulnerability Scanner	OWASP ZAP + zaproxy	2.15 / 0.4.0	Active/passive web vulnerability assessment
SSL/TLS Inspector	cryptography	46.0.7	Certificate analysis and TLS grade assignment
MQTT Client	paho-mqtt	2.1.0	Telemetry ingestion from IoT devices
CVE Intelligence	NVD API v2 (external)	—	Known vulnerability enrichment per service
Report Export	openpyxl	3.1.5	Excel workbook generation for security reports
Password Hashing	bcrypt	4.3.0	Secure credential storage
Rate Limiting	Flask-Limiter	—	Brute-force protection on login endpoint
Frontend Visualisation	Chart.js	3.x	Risk distribution charts and telemetry graphs

MQTT Broker	Mosquitto	2.x	Message broker for IoT device telemetry

Table 4.1 System Components and Their Specifications

4.3 Software Module Design

This section describes the internal design of each major software module. Since the dashboard is a software-only system with no custom hardware circuits, this section replaces the hardware circuit design section with a detailed description of the software module architecture, database schema, and API endpoint structure that together enable the system to function as a cohesive platform.

4.3.1 Flask Application Structure and Blueprint Organisation

The Flask application is structured as a modular Blueprint-based application. The entry point is *backend/app.py*, which initialises the Flask app, configures Flask-SocketIO, registers all Blueprints, and starts the Eventlet-based server. Each Blueprint encapsulates a specific domain of functionality:

- **auth_bp** — handles login, logout, registration, session timeout, and role resolution.
- **telemetry_bp** — handles REST telemetry ingestion and serves live device status data.
- **nmap_bp** — exposes the Nmap scan trigger, progress query, and results retrieval endpoints.
- **zap_bp** — exposes the ZAP scan trigger, health check, and results retrieval endpoints.
- **ssl_bp** — exposes the SSL/TLS scan trigger and results endpoints.
- **ping_bp** — exposes the ping sweep trigger and results endpoints.
- **export_bp** — serves the report page, report data, and Excel export.
- **logs_bp** — returns activity log entries scoped by user role.
- **devices_bp** — provides the available devices list for admin accounts.

All scan-triggering routes require an authenticated session and log the triggering action to the activity_logs table upon invocation. Scan endpoints return a scan_id immediately and execute the scan in a background thread, allowing the frontend to call a progress endpoint at regular intervals without blocking server resources.

4.3.2 Database Schema Design

The MySQL database is organised into ten tables. The schema was designed to support role-based data retrieval, historical tracking of scan results, and efficient CVE caching. The principal tables and their fields are described below.

Table Name	Key Fields	Purpose
users	id, username, password_hash, created_at	Stores regular user accounts and hashed credentials
admins	id, username, password_hash	Stores admin accounts (created directly in DB)
scan_results	id, user_id, target_ip, port, protocol, service, version, risk_score, scan_id, timestamp	Stores Nmap port and service scan results per user
zap_results	id, user_id, target_url, alert_name, risk_level, description, solution, evidence, scan_id, timestamp	Stores OWASP ZAP vulnerability alerts per user
ssl_tls_results	id, user_id, host, port, tls_version, cipher_suite, cert_expiry, grade, findings, timestamp	Stores SSL/TLS certificate analysis results (upserted by host:port)
ping_sweep_results	id, user_id, ip_address, hostname, latency_ms, status, timestamp	Stores ping sweep host reachability results (upserted by IP)
telemetry	id, device_id, device_name, ip_address, status, risk_score, last_seen, user_id	Stores live and historical IoT device telemetry
activity_logs	id, user_id, role, action, details, timestamp	Persistent audit log of all user actions
cve_cache	id, service_key, cve_id, description, severity, published_date, cached_at	Cached CVE lookup results with 7-day TTL

network_info	id, interface, ip_address, subnet, gateway, recorded_at	Network interface information of the server host

Table 4.2 MySQL Database Schema Summary

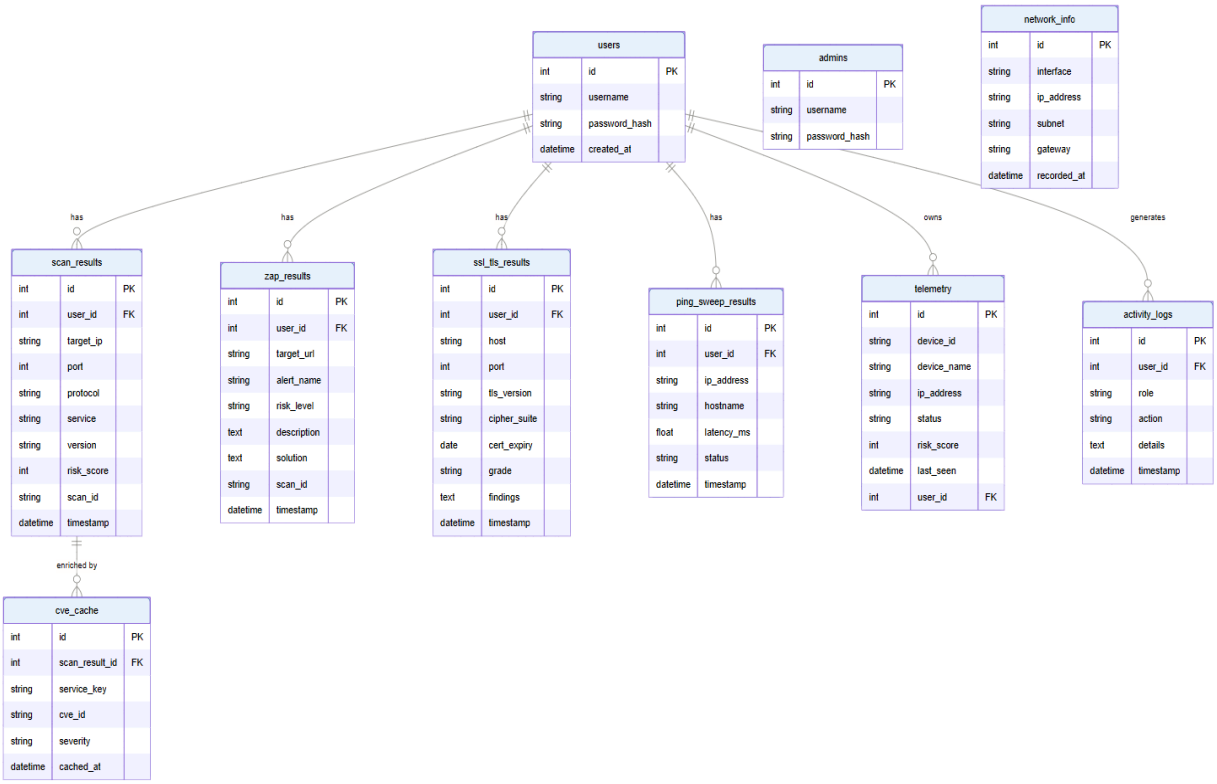

Figure 4.2 MySQL Database ERD (10 tables)

4.3.3 Security Enrichment Module

The Security Enrichment module analyses Nmap scan results for protocol-level security risks that are independent of CVE data. It operates as a rule-based engine that evaluates each open port and service against a set of predefined risk rules. Detected issues include Telnet exposure on port 23, unencrypted HTTP on port 80, SMB exposure on port 445, SNMP reachability on port 161, MQTT running without TLS on port 1883, and exposed camera management interfaces on common IoT ports. Each finding is assigned a severity label of HIGH, MEDIUM, or LOW. The module computes a weighted risk score contribution for each finding, capped at a maximum of 35 additional points, and adds this to the base risk score calculated from the raw port count. The combined risk score is then stored alongside the scan result and used to categorise the device as Low, Medium, or High risk in the dashboard's visualisation.

4.3.4 Scan Manager

The Scan Manager (implemented in *backend/scan_manager.py*) is a shared in-process registry that maintains references to all active scan subprocesses. It serves three functions. First, it allows any scan endpoint to check whether another scan is already in progress and reject concurrent duplicate scans gracefully. Second, it exposes a cancellation interface that terminates the OS-level Nmap or ZAP subprocess when the Stop Scan request is received from the frontend. Third, it manages the auto-scan queue: when the MQTT Handler or Device Discovery module detects a new device IP, it submits that IP to the Scan Manager queue, which dispatches an Nmap scan for the device in a background thread if the auto-scan toggle is enabled.

4.3.5 Frontend Page Architecture

The frontend is structured into five pages, each served as a separate HTML file. The main dashboard page (index.html rendered via Jinja2) loads main.js and telemetry.js as its primary JavaScript modules. main.js handles all scan interactions, result rendering, chart updates, and notification logic. telemetry.js manages the live telemetry panel, the device simulator, and the WebSocket event listeners. The activity logs page (logs.html) renders a role-scoped view of the activity_logs table. The report export page (report.html) assembles a comprehensive security summary from all scan data, supporting date range filtering and Excel export. The user guide page (userguide.html) provides in-browser documentation covering all features, role differences, MQTT configuration, and risk level explanations.

A custom modal system replaces all native browser dialogs to provide a consistent user experience across all pages. A WebSocket connectivity badge in the page header provides a live indicator of the Socket.IO connection status. A real-time device counter reflects the number of devices currently marked as online in the telemetry table.

4.4 System Components Interaction Operations

This section describes how the individual software components interact at runtime to deliver the dashboard's key operations. Six principal interaction scenarios are described: user authentication, manual network scan, web vulnerability assessment, live telemetry monitoring, SSL/TLS scanning with CVE enrichment, and security report generation.

4.4.1 User Authentication and Session Initialisation

When a user navigates to the login page, the browser renders the login form. Upon submission, the credentials are sent as an HTTP POST request to the Auth API. The Auth API retrieves the corresponding password hash from either the users or admins table in MySQL and verifies it using bcrypt. If verification succeeds, Flask creates a server-side session storing the user's ID, username, and role. The response redirects the browser to the main dashboard page. On the dashboard, the frontend JavaScript immediately establishes a Socket.IO WebSocket connection to the Flask-SocketIO server, which registers the client for real-time push events. If no user activity is detected for thirty minutes, the session is invalidated and the user is redirected to the login page. Every login and logout event is appended to the activity_logs table.

Figure 4.3 illustrates the authentication interaction sequence from credential entry through to WebSocket establishment.

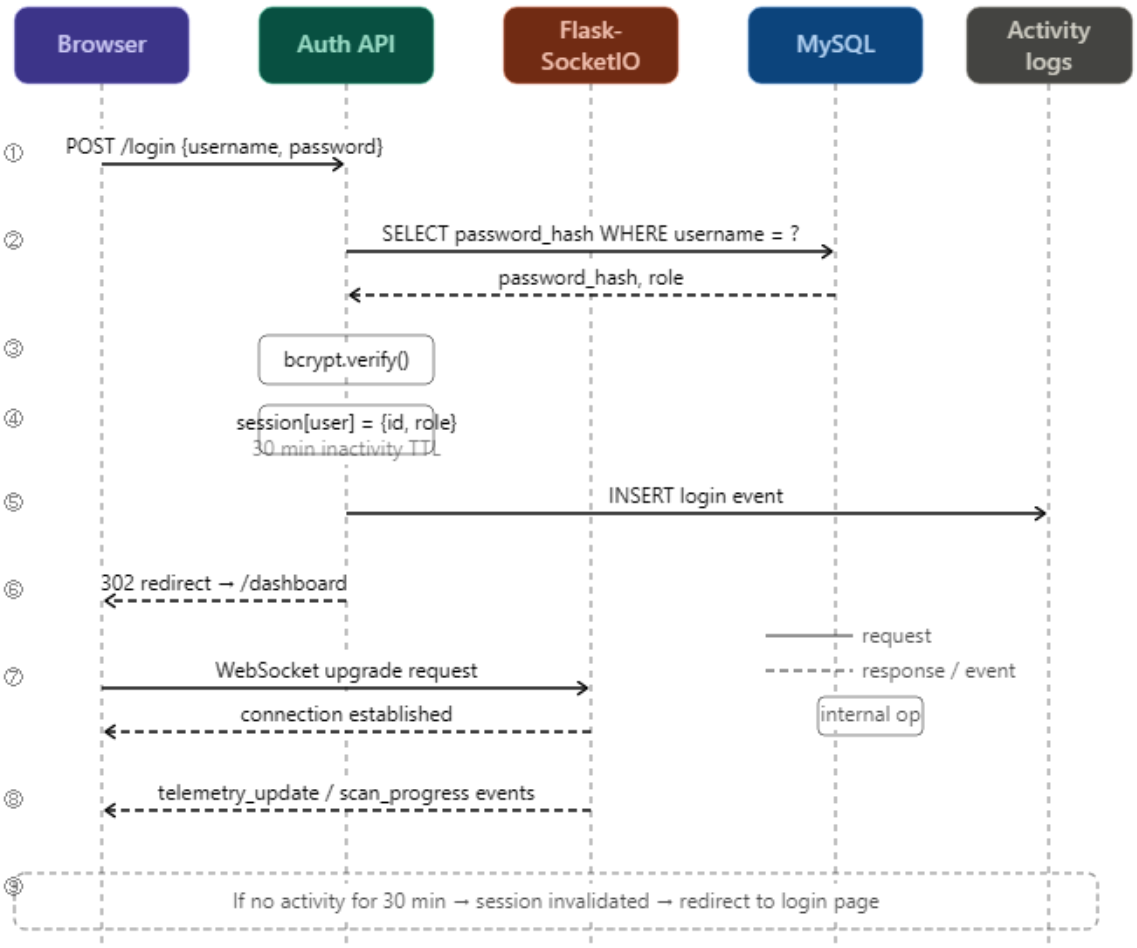

Figure 4.3 User Authentication and Session Initialisation Sequence

4.4.2 Nmap Network Scan Operation

The Nmap scan interaction begins when the user clicks the Nmap Scan button on the dashboard and enters a target IP address in the custom modal dialog. The frontend validates the input format and sends an HTTP POST request to the Nmap scan endpoint, including the target IP and the selected scan mode. The Nmap Blueprint validates the session, logs the action to activity_logs, registers the scan in the Scan Manager, and responds immediately with a scan_id. It then dispatches the Nmap subprocess invocation to a background thread.

In the background thread, the Nmap Scanner module validates the target IP against a regular expression, constructs the Nmap command with the -sV flag, and invokes Nmap as a subprocess. The frontend polls the scan progress endpoint every two seconds, and the backend responds with the current status. When Nmap completes, its XML output is parsed into structured records. Each record is passed to the Security Enrichment module for risk scoring, then to the CVE Lookup module for vulnerability intelligence. The enriched results are written to the scan_results and cve_cache tables in MySQL. Finally, the backend emits a WebSocket event to the frontend, which re-renders the results table and risk chart without requiring a page reload. If any port carries a HIGH risk finding, an alert toast is displayed to the user.

If the user presses the Stop Scan button during an active scan, the frontend sends a stop request to the Scan Manager, which calls the terminate() method on the Nmap subprocess, interrupting the scan immediately.

4.4.3 OWASP ZAP Web Vulnerability Scan Operation

The ZAP scan interaction begins when the user initiates a scan from the ZAP section of the dashboard. The ZAP Blueprint first performs a preflight health check by calling the ZAP daemon's version API endpoint. If ZAP is not running, the endpoint returns an error immediately. If ZAP is available, the backend identifies HTTP and HTTPS targets from the most recent Nmap scan results and passes them to the ZAP Scanner module.

The ZAP Scanner calls the ZAP REST API to open a new session, set the target URL, and start the spider and active scanner. Scan progress is polled from ZAP at regular intervals and forwarded to the frontend via the scan progress endpoint. Upon completion, the ZAP Scanner retrieves all alerts from the ZAP session, filters and structures them, and writes them to the zap_results table. The frontend is notified via WebSocket and re-renders the ZAP findings

panel, displaying each alert with its risk level, description, evidence, and recommended solution.

4.4.4 Live Telemetry Monitoring and Auto-Scan

IoT device telemetry reaches the system through two paths, both of which converge at the same processing pipeline. In the MQTT path, a device or the included device simulator publishes a JSON message to the Mosquitto broker on the topic iot/devices/<device_id>/telemetry. The MQTT Handler, which runs as a persistent background thread alongside the Flask application, receives the message, parses the JSON payload, and upserts the device record in the telemetry table. It then emits a telemetry_update Socket.IO event to all connected frontend clients, which re-render the live device card for that device. In the REST path, a device sends an HTTP POST request directly to the telemetry endpoint, and the same pipeline is followed from the point of database upsert.

When a new device ID is seen for the first time — whether via MQTT or REST — the MQTT Handler checks whether the auto-scan toggle is enabled. If it is, the device's IP address is submitted to the Scan Manager queue, which schedules a background Nmap scan for that device. This allows administrators to maintain continuous security coverage of newly connected IoT devices without manual intervention. The frontend displays a toast notification when a new device is detected and when a device transitions to offline status after ten seconds without a telemetry update.

4.4.5 SSL/TLS Scan and CVE Lookup

The SSL/TLS scan is initiated by the user entering a hostname and port in the SSL/TLS scan panel and clicking the Scan button. The SSL/TLS Blueprint receives the request, validates the session, and invokes the SSL/TLS Scanner module. The scanner opens a TLS connection to the target, retrieves the peer certificate using the ssl and cryptography libraries, and evaluates all six security attributes described in Section 4.2.5. A letter grade from A to F is computed, and the result — including all attribute evaluations and identified findings — is upserted to the ssl_tls_results table by host-port pair. The frontend is updated via WebSocket with the new grade and findings.

Following every completed Nmap scan, the CVE Lookup module runs automatically. It iterates over every service record from the scan, constructs a lookup key from the product name and

version string, and queries the cve_cache table first. If a valid cached entry exists (less than seven days old), the cached CVEs are returned. Otherwise, the module queries the NVD API v2, stores up to five CVEs per service in the cache, and associates the CVE records with the corresponding scan result. The enriched results are available in both the dashboard view and the exported report.

4.4.6 Security Report Generation and Export

Report generation begins when the user navigates to the Export Report page and optionally applies a date range filter. The frontend sends an HTTP GET request to the Export Blueprint with the filter parameters. The Export Blueprint queries all relevant tables in MySQL — scan_results, zap_results, ssl_tls_results, ping_sweep_results, telemetry, and activity_logs — applying the date filter and role-based scoping. The aggregated data is returned as a structured JSON response, which the frontend renders as a comprehensive HTML report page with sections for each module's findings.

For the Excel export, the user clicks the Download Excel button, which triggers an additional request to the export endpoint with the format parameter set to xlsx. The Export Blueprint builds an openpyxl workbook with one sheet per data category, applies header formatting and column widths, and serves the resulting .xlsx file as a download. For the printable PDF format, the user clicks the Print button, which invokes the browser's print dialog against the formatted HTML report page, allowing it to be saved as a PDF file directly.

4.4.7 Summary of Component Interactions

Table 4.3 summarises the six principal interaction scenarios and the components involved in each.

Operation	Frontend	Flask API	Module(s) Involved	Database Tables
User Login	Login form POST	Auth Blueprint	bcrypt, session mgmt	users, admins, activity_logs
Nmap Scan	Modal trigger, progress poll	Nmap Blueprint	Nmap Scanner, Security Enrichment,	scan_results, cve_cache, activity_logs

			CVE Lookup, Scan Manager	
ZAP Scan	ZAP panel trigger	ZAP Blueprint	ZAP Scanner, Scan Manager	zap_results, activity_logs
Live Telemetry	WebSocket listener, device cards	Telemetry Blueprint + SocketIO	MQTT Handler, Scan Manager (auto-scan)	telemetry, activity_logs
SSL/TLS Scan	SSL panel trigger	SSL Blueprint	SSL/TLS Scanner	ssl_tls_results, activity_logs
Report Export	Export page, date filter	Export Blueprint	openpyxl (xlsx), HTML renderer	All scan & log tables

Table 4.3 Summary of Component Interactions by Operation

4.5 Chapter Summary

This chapter presented a detailed design of the Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard at the component level. Section 4.1 described the system block diagram, identifying the six major functional blocks and the data flow connections between them. Section 4.2 specified every software component, library, and external tool used in the system, including version numbers and justifications. Section 4.3 described the internal design of the Flask Blueprint structure, database schema, Security Enrichment module, Scan Manager, and frontend page architecture. Section 4.4 traced the component interactions across the six principal operations of the system: authentication, Nmap scanning, ZAP scanning, live telemetry monitoring, SSL/TLS scanning, and report generation.

Together, these sections provide the technical depth needed to understand, reproduce, and extend the system. Chapter 5 builds on this design by documenting the hardware and software setup, configuration steps, and the full operation of the implemented system with screenshots.

CHAPTER 5

System Implementation

This chapter documents the complete implementation of the Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard. It covers the hardware environment in which the system was deployed, the software installation and dependency setup, all configuration steps required to bring the system to a running state, the operation of each feature with annotated screenshots, the implementation issues and challenges encountered during development, and a concluding remark on the overall implementation outcome.

5.1 Hardware Setup

The system is a fully software-based web application and does not require any custom hardware circuits or embedded boards to operate. However, it does depend on a host computer to run the backend server, the Mosquitto MQTT broker, the OWASP ZAP daemon, and a MySQL database server. Table 5.1 below describes the hardware environment used during the development and testing of the system.

Component	**Specification**
Host machine	Personal laptop — Windows 11
Processor	Intel Core i5 (8th generation or equivalent)
RAM	8 GB DDR4 / DDR5
Storage	256 GB SSD (system) — sufficient for logs, scan results, and CVE cache
Network interface	Wi-Fi (802.11ac) — used for MQTT device connectivity and target scanning
Operating system	Windows 11 with Windows Subsystem for Linux (WSL2) optional; Mosquitto and Nmap run natively on Windows

Table 5.1 Hardware Environment

Because all scan tools (Nmap, OWASP ZAP) and the MQTT broker (Mosquitto) are installed on the same machine as the Flask server, no additional network routing or port forwarding configuration was required for local testing. When IoT devices or the included Python device simulator are used on the same local network, they connect to the Mosquitto broker over the

default MQTT port 1883. The dashboard itself is accessible from any browser on the same network by navigating to the host machine's local IP address and the Flask application port.

5.2 Software Setup

This section documents all software that must be installed on the host machine before the dashboard can be started. The setup is divided into four areas: Python environment and dependencies, database setup, external tool installation, and frontend assets.

5.2.1 Python Environment

The backend requires Python 3.10 or later. A Python virtual environment is recommended to isolate the project dependencies. The following commands set up the environment and install all required packages from the project's requirements.txt file:

```
python -m venv venv
venv\Scripts\activate          # Windows
pip install -r requirements.txt
```

The principal packages installed include Flask 3.1.1, Flask-SocketIO 5.5.1, Eventlet 0.40.2, mysql-connector-python 9.3.0, paho-mqtt 2.1.0, python-nmap 0.7.1, zaproxy 0.4.0, cryptography 46.0.7, bcrypt 4.3.0, openpyxl 3.1.5, and Flask-Limiter. All versions are pinned in the requirements file to ensure reproducibility.

5.2.2 MySQL Database Setup

MySQL Server 8.x was installed on the host machine. After installation, a new database named *iot_security_dashboard* was created. The database schema was initialised by running the provided SQL script, which creates all ten tables described in Section 4.3.2. A dedicated MySQL user account was created for the application with SELECT, INSERT, UPDATE, and DELETE privileges on this database only, following the principle of least privilege.

```
CREATE DATABASE iot_security_dashboard;
mysql -u root -p iot_security_dashboard < schema.sql
```

5.2.3 Nmap Installation

Nmap 7.94 was downloaded from the official Nmap website and installed using the Windows installer. After installation, the Nmap executable was confirmed to be accessible from the system PATH by running nmap --version in the command prompt. The Flask backend invokes Nmap as a subprocess, so PATH availability is a requirement.

5.2.4 OWASP ZAP Installation

OWASP ZAP 2.15 was downloaded from the official OWASP ZAP website and installed using the Windows installer. ZAP is configured to run in daemon mode on startup so that the Flask backend can connect to its REST API on localhost port 8080. The ZAP API key was noted and added to the application configuration, the key can be found and regenerate in the settings. No additional ZAP plugins were required beyond the default installation.

5.2.5 Mosquitto MQTT Broker Installation

Eclipse Mosquitto 2.x was downloaded and installed on the host machine. The Mosquitto broker was configured to listen on port 1883 without authentication for local testing purposes. The broker is started as a Windows service or manually via mosquitto -v before launching the Flask application. For production deployments, TLS and password-based authentication should be enabled in the Mosquitto configuration file.

5.3 Settings and Configuration

All environment-specific settings for the dashboard are stored in a single configuration file to keep sensitive values out of the source code. This section describes each configuration parameter and how it is applied at runtime.

5.3.1 Environment Configuration File

A .env file is placed in the project root directory. The Flask application reads this file at startup using the python-dotenv library. Table 5.2 lists all required configuration keys.

Configuration Key	Example Value	Purpose
SECRET_KEY	a-random-secret-string	Flask session signing key — must be long and unpredictable
DB_HOST	localhost	MySQL server hostname

DB_USER	dashboard_user	MySQL application user
DB_PASSWORD	strongpassword	MySQL application user password
DB_NAME	iot_security_dashboard	Target MySQL database name
MQTT_BROKER	localhost	Hostname or IP of the Mosquitto broker
MQTT_PORT	1883	MQTT broker port (default 1883, or 8883 for TLS)
ZAP_API_KEY	your-zap-api-key	API key set during ZAP installation
ZAP_HOST	localhost	Host where ZAP daemon is running
ZAP_PORT	8080	ZAP REST API port
FLASK_PORT	5000	Port the Flask application listens on
RATE_LIMIT	10 per minute	Login endpoint rate limit (Flask-Limiter format)

Table 5.2 Environment Configuration Parameters

5.3.2 Mosquitto Broker Configuration

The Mosquitto broker configuration file (mosquitto.conf) was edited to set the listener port to 1883 and to allow anonymous connections for local testing. The relevant configuration lines are:

```
listener 1883
allow_anonymous true
```

IoT devices and the built-in device simulator publish telemetry messages to the topic iot/devices/<device_id>/telemetry. The MQTT handler in the Flask backend subscribes to the wildcard topic iot/devices/+/telemetry to receive messages from all devices.

5.3.3 Database Initialisation

After the MySQL database was created and the schema script executed, one admin account was inserted directly into the admins table using a bcrypt-hashed password. This admin account is the only way to access admin-level features such as viewing all users' scan results and the full activity log. Regular user accounts are created through the public registration page on the dashboard.

The admin account was inserted using a short Python helper script that generates a bcrypt hash for the chosen password and executes a parameterised INSERT into the admins table. This approach avoids storing the plain-text password anywhere in the codebase or configuration files.

5.3.4 Startup Sequence

The correct startup order for the system is simple and user-friendly: Just double click to run the *run.bat* that automates everything like ZAP, activate virtual environment, and Flask server, in the correct order automatically, then open dashboard *http://localhost:5000* via browser only after the server has been fully started

5.4 System Operation

This section demonstrates the operation of each major feature of the dashboard. For each feature, the relevant user workflow is described and a screenshot of the corresponding interface is included. All screenshots were taken from a local deployment of the system during the testing phase. Mosquitto or MySQL — will result in connection errors at startup.

5.4.1 User Registration and Login

New users access the registration page to create an account by providing a username and password. The password is validated for minimum length on the client side before submission. Upon successful registration, the user is redirected to the login page. Figure 5.1 shows the registration form and Figure 5.2 shows the login page.

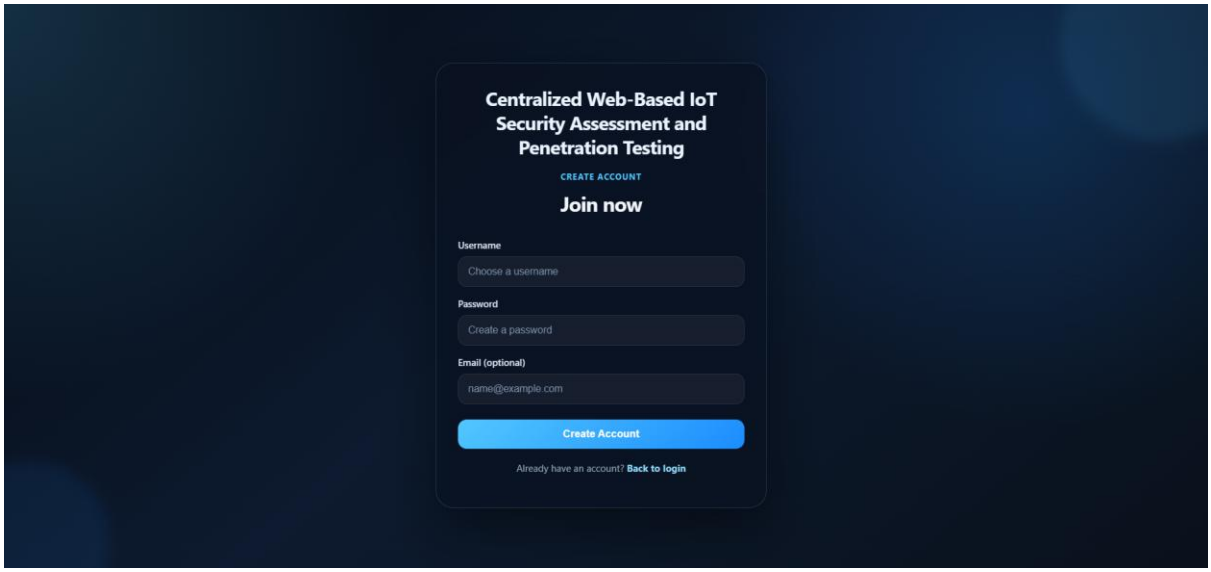

Figure 5.1 User Registration Page

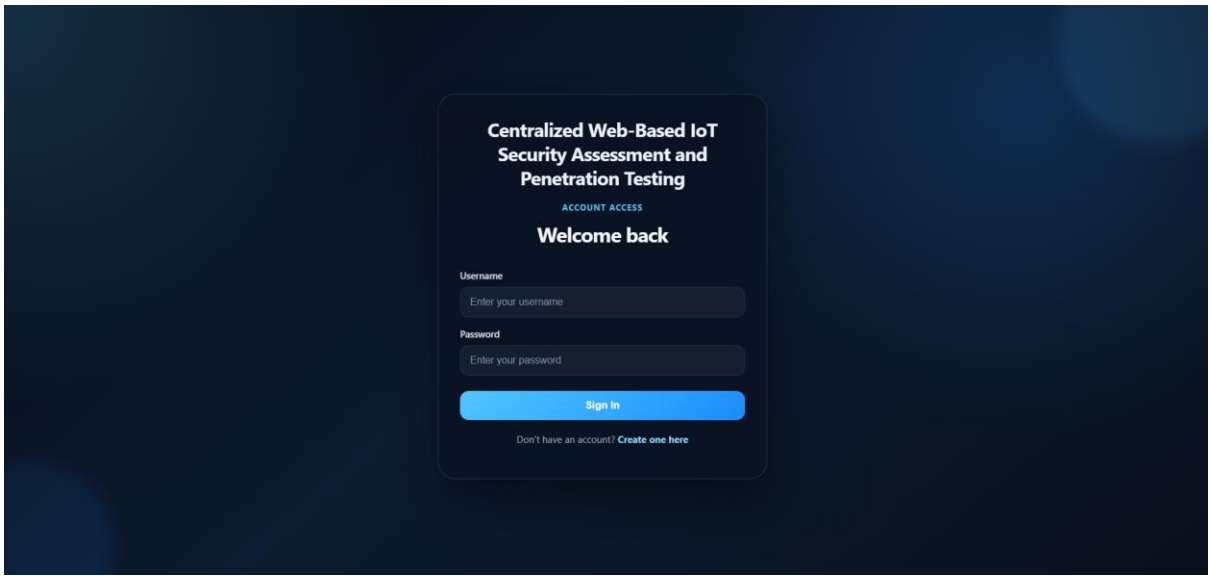

Figure 5.2 User Login Page

After logging in, the user is redirected to the main dashboard. The page header displays the logged-in username, a live device counter, and a WebSocket connectivity badge indicating whether the real-time connection to the Flask-SocketIO server is active. Admin accounts, which are created directly in the database, see an additional admin-only navigation option for viewing all users' data.

5.4.2 Main Dashboard Overview

The main dashboard provides a unified view of all security information. The left panel displays the live telemetry section showing IoT devices and their statuses. The right panel contains all

scan tool controls. A risk distribution doughnut chart summarises the proportion of low, medium, and high risk findings from the most recent Nmap scan. Figure 5.3 shows the main dashboard with an active device session from admin role view, while Figure 5.4 shows from user role view.

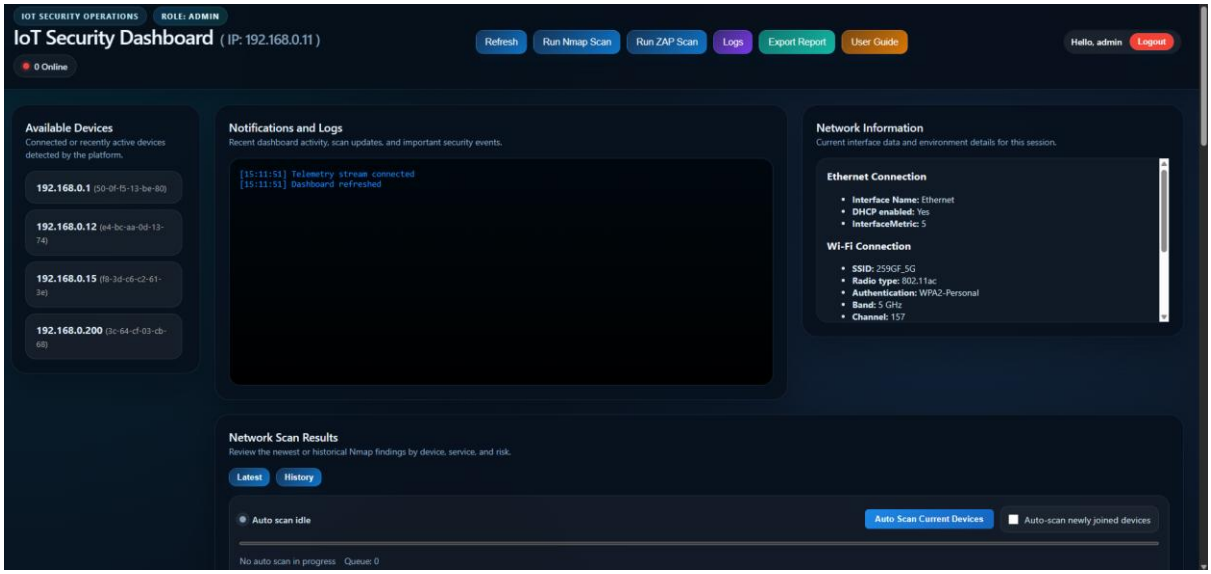

Figure 5.3 Main Dashboard Overview (Admin Role View)

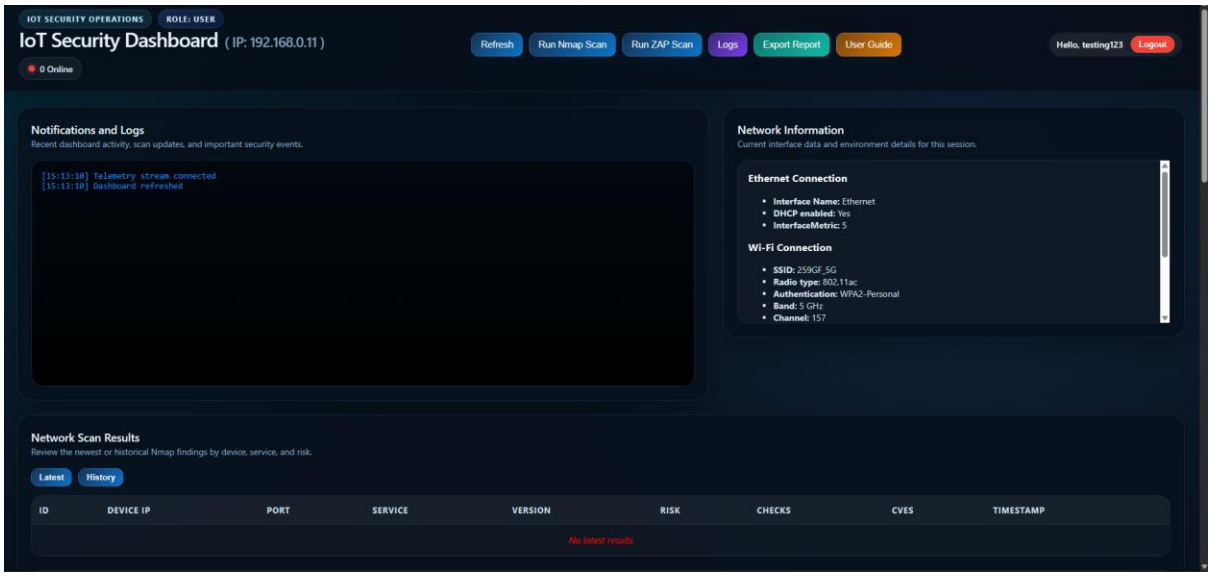

Figure 5.4 Main Dashboard Overview (User Role View)

5.4.3 Live IoT Device Telemetry

The telemetry panel displays a live card for each IoT device that has sent a status update via MQTT or the REST API. Each card shows the device name, IP address, current status (Online or Offline), risk score, and the time of the last received message. Cards update in real time

without page reload as new MQTT messages arrive. Figure 5.5 shows the telemetry panel with multiple devices connected.

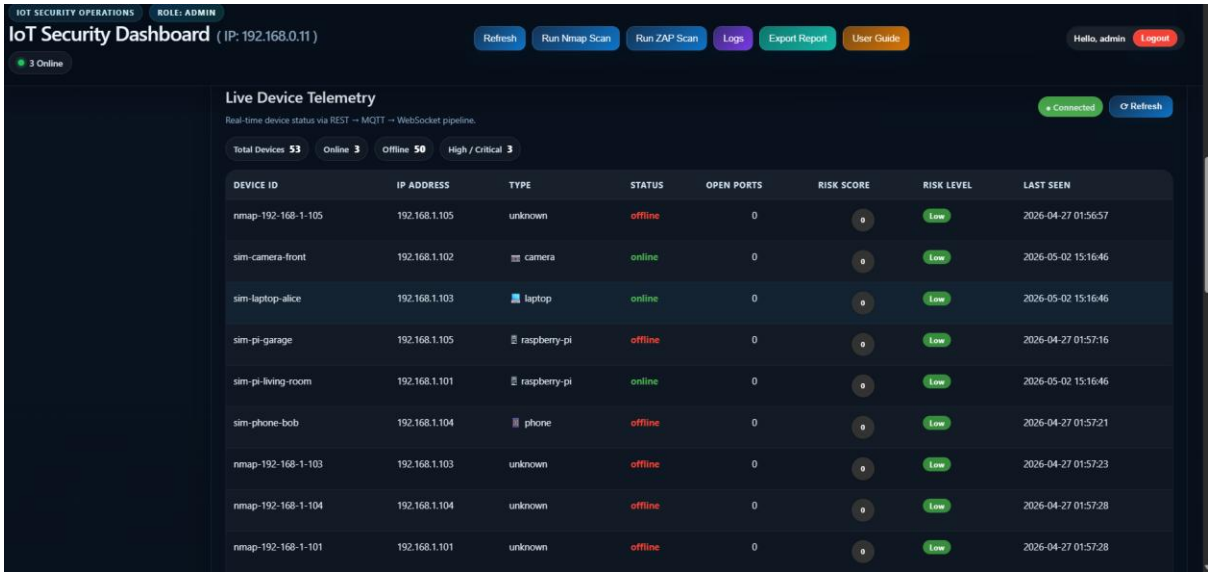

Figure 5.5 Live IoT Device Telemetry Panel

A built-in device simulator, located at the bottom of telemetry panel, is also provided on the dashboard. It allows the user to send simulated telemetry messages for a given device ID and IP address, which is useful for testing without physical IoT hardware. Figure 5.6 shows the simulator controls and the resulting device card update.

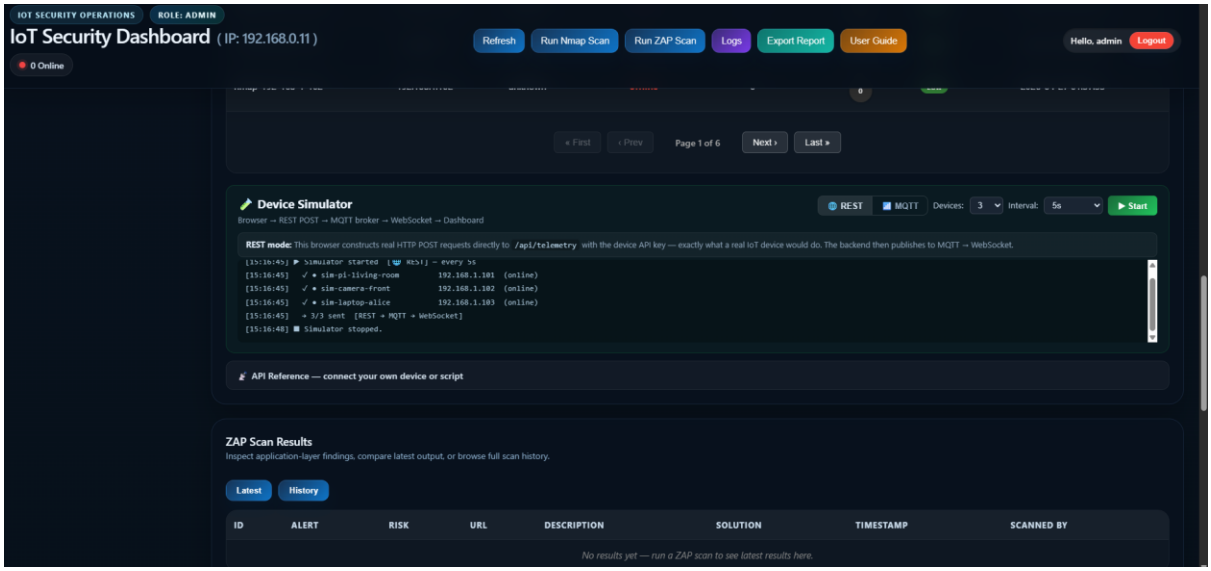

Figure 5.6 Built-in Device Simulator

5.4.4 Nmap Network Scan

To initiate an Nmap scan, the user clicks the Nmap Scan button in the scan panel. A modal dialog appears prompting for the target IP address. After the user enters the target and clicks Scan, the backend begins the scan in a background thread and the frontend starts polling the scan progress endpoint. A real-time progress bar updates as the scan advances. Figure 5.7 shows the Nmap scan modal and Figure 5.8 shows the results table after a completed scan.

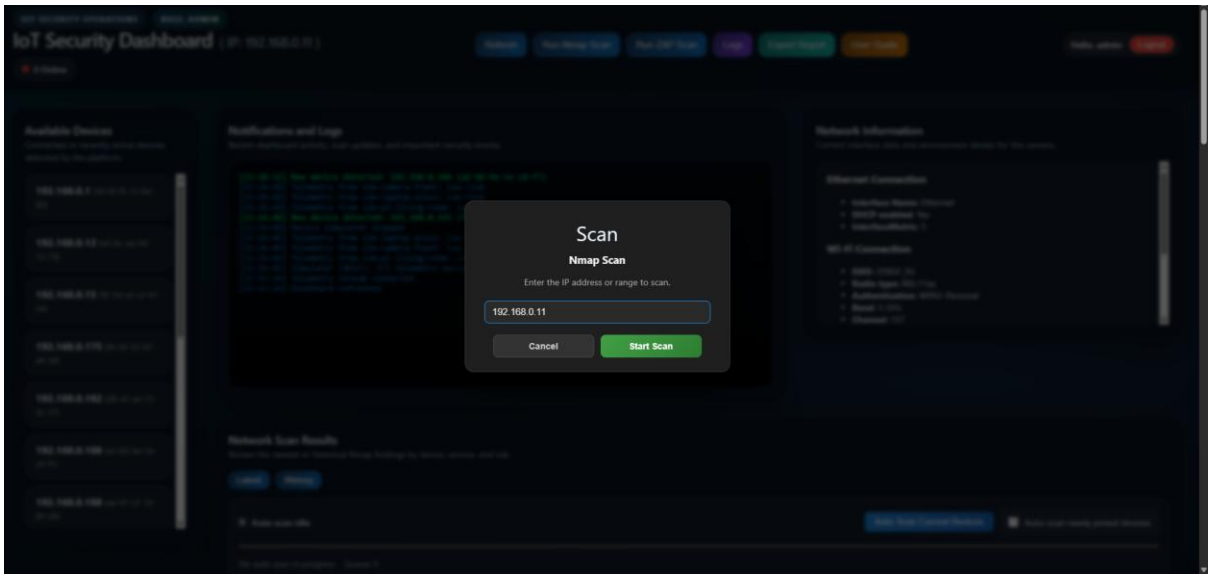

Figure 5.7 Nmap Scan Target Input Modal

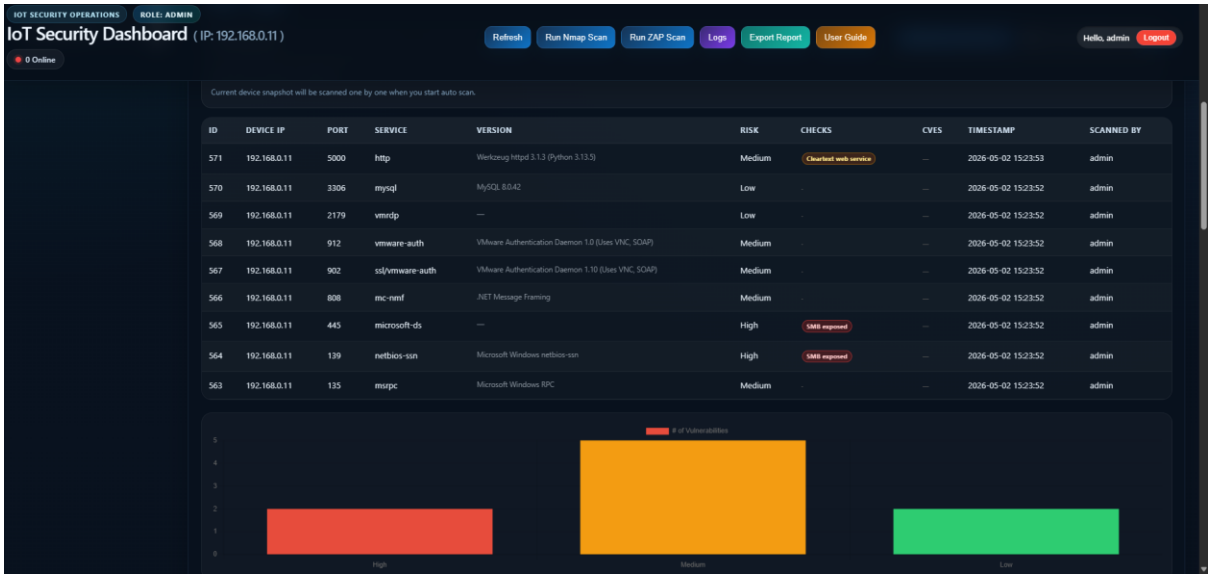

Figure 5.8 Nmap Scan Results with Risk Scoring

The results table displays each open port, the detected protocol, service name, version string, any CVEs retrieved from the NVD API, timestamp, user that launched scan, and the calculated risk score. High-risk findings are highlighted in red. A toast notification at the bottom right of

dashboard alerts the user if any HIGH risk ports are detected. The risk distribution chart updates automatically to reflect the new scan.

5.4.5 OWASP ZAP Web Vulnerability Assessment

The ZAP scan panel shows the ZAP daemon connection status. When ZAP is running, the user selects the scan mode (quick or full) and initiates the scan. ZAP targets the HTTP and HTTPS services identified by the most recent Nmap scan. Figure 5.9 shows the ZAP scan mode options before starting scan and Figure 5.10 shows the completed results.

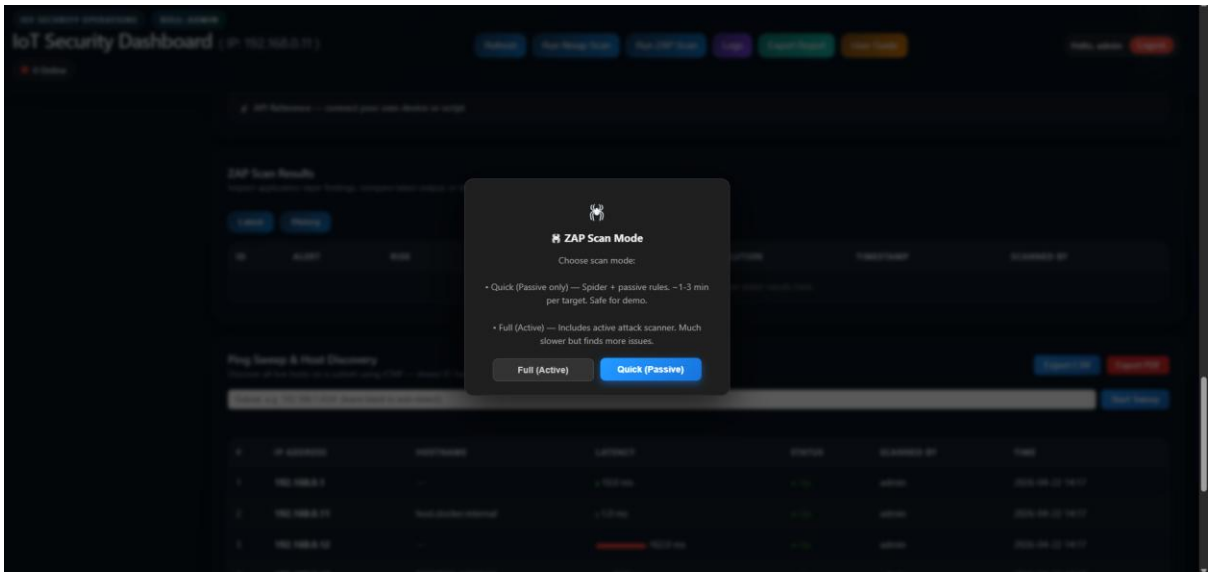

Figure 5.9 OWASP ZAP Scan Mode Options

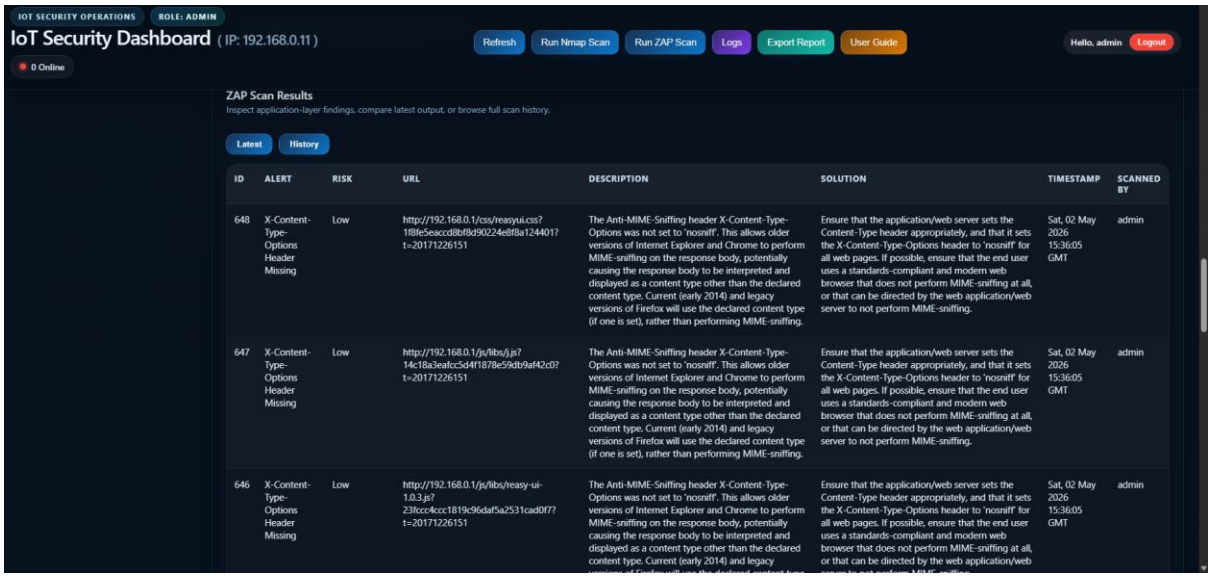

Figure 5.10 OWASP ZAP Vulnerability Assessment Results

5.4.6 SSL/TLS Certificate Scanner

The SSL/TLS scanner panel accepts a hostname and port number. After the user clicks Scan, the backend connects to the target over TLS, inspects the certificate using the cryptography library, and evaluates all six security attributes. A letter grade from A to F is computed and displayed. Figure 5.11 shows the SSL/TLS scan results for a test target.

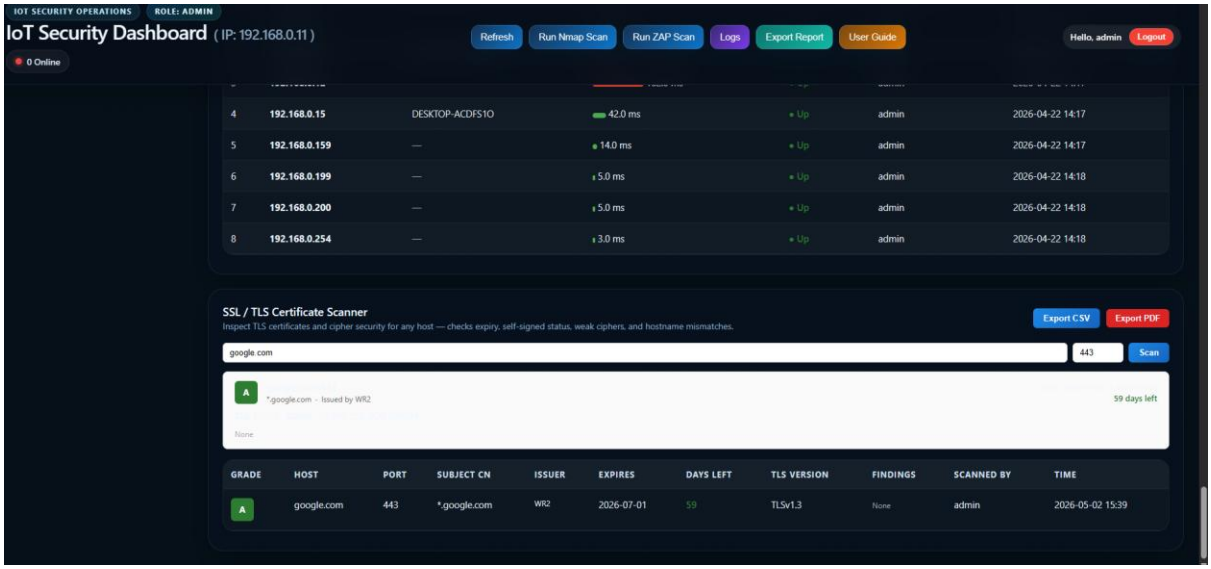

Figure 5.11 SSL/TLS Certificate Scan Results

5.4.7 Ping Sweep

The ping sweep panel accepts a CIDR subnet range (for example, 192.168.1.0/24 or leave blank to auto detect). The backend concurrently pings all addresses in the range and returns a results table showing which hosts are reachable, their resolved hostnames, and their round-trip latency in milliseconds. Figure 5.12 shows the ping sweep results for a local subnet.

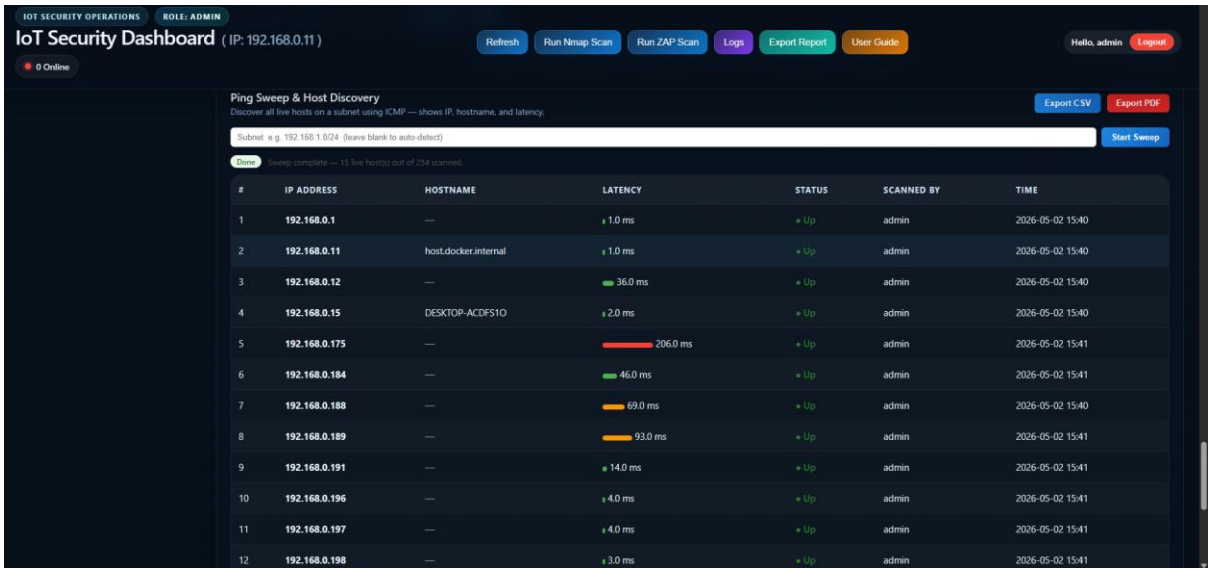

Figure 5.12 Ping Sweep Results

5.4.8 Activity Logs

The activity logs page displays a timestamped record of all actions performed in the system. Regular users see only their own actions. Admin users see the full log for all accounts. Each entry records the action type, the target of the action, the user who performed it, and the timestamp. Figure 5.13 shows the activity log page from admin view, while Figure 5.14 shows from user view.

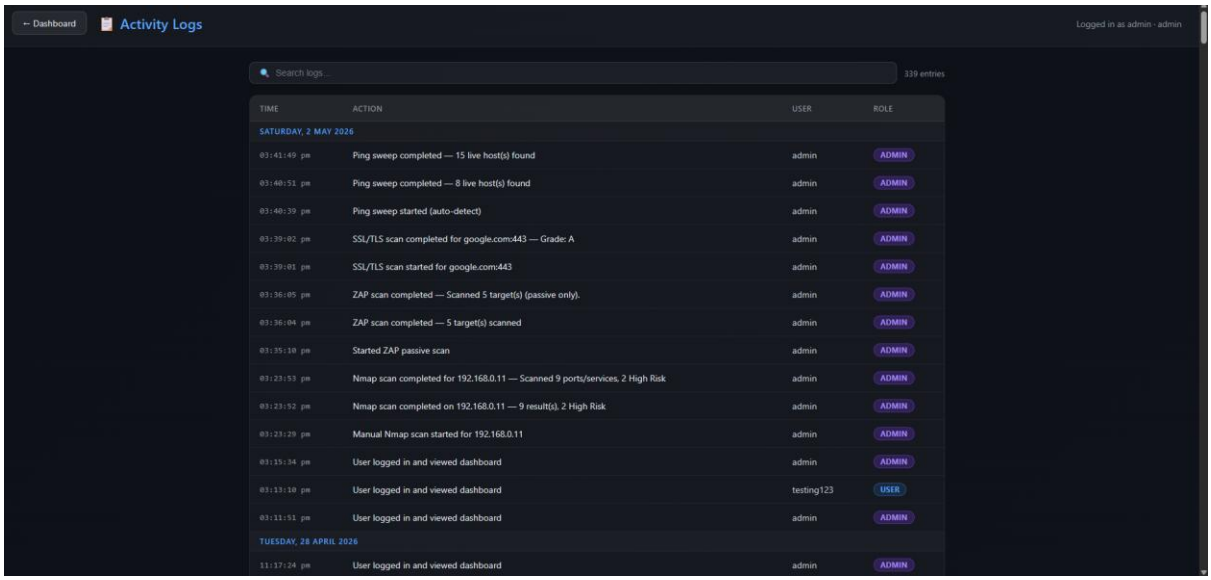

Figure 5.13 Activity Logs Page (Admin View)

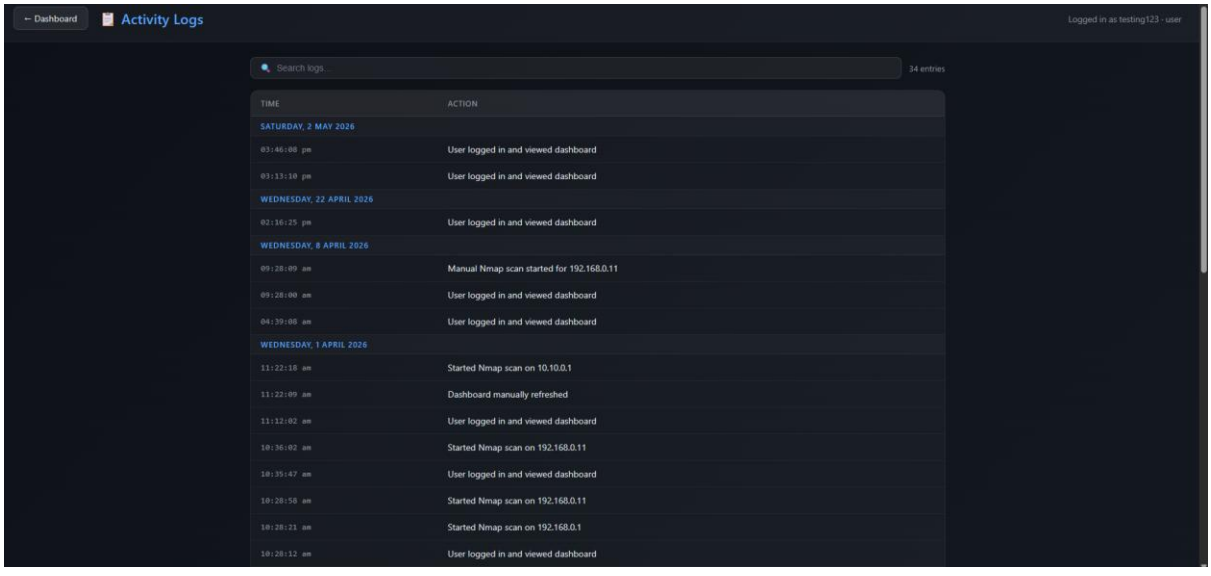

Figure 5.14 Activity Logs Page (User View)

5.4.9 Security Report Export

The report export page consolidates all scan data into a single printable report. The user can optionally apply a date range filter before generating the report. The page renders sections for

each scan type with summary statistics and full result tables. Two export options are available: a browser print-to-PDF function and an Excel workbook download. Figures below views from admin, as admin role has full functionalities, logs and data information from all user on export page. Figure 5.15 shows the report export page and Figure 5.16 shows a sample of one of the sheets from Excel export with multiple sheets(with filter by date, last 7 days applied).

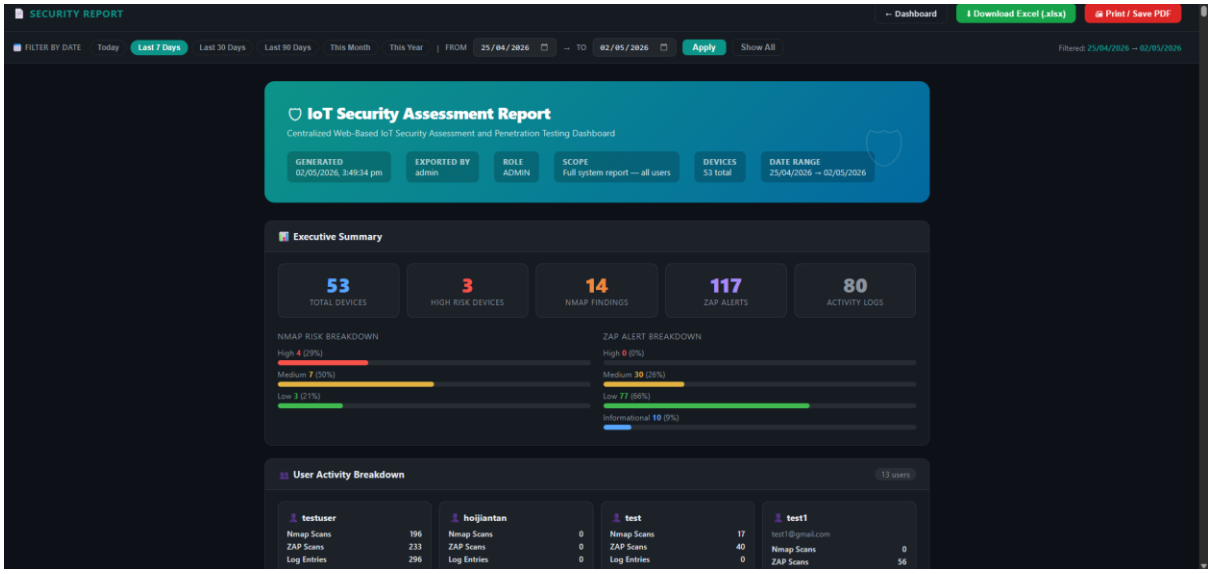

Figure 5.15 Security Report Export Page

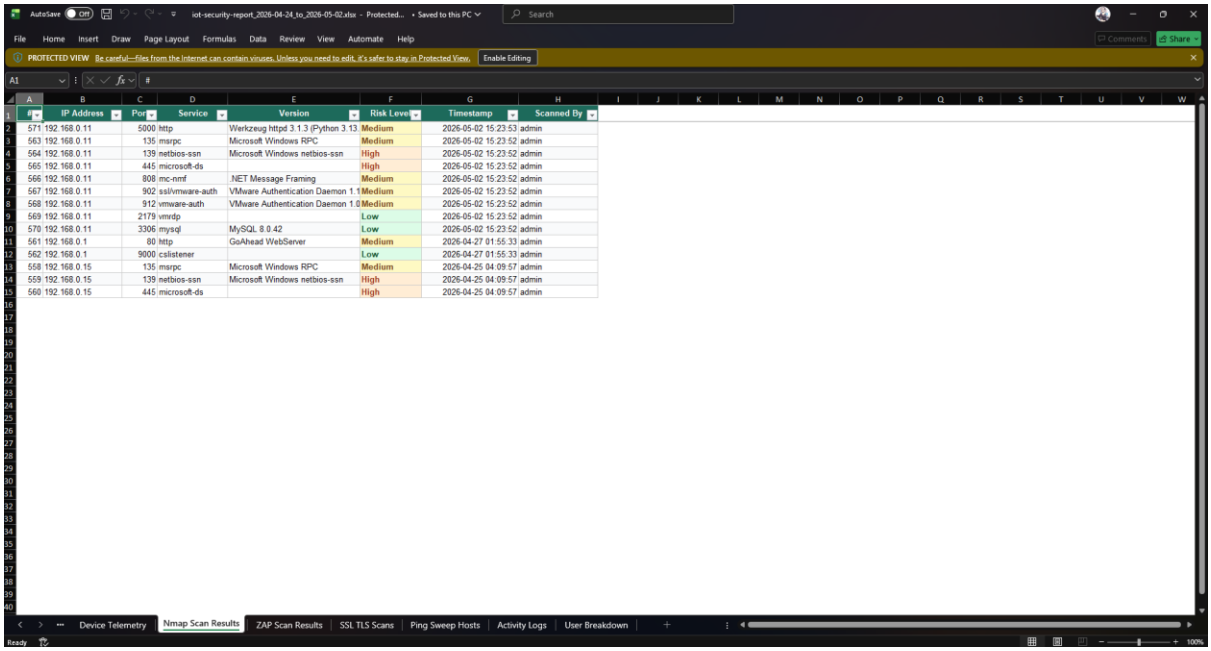

Figure 5.16 Multi-sheet Workbook of Excel Report Export (with filter by date, Last 7 Days)

5.4.10 User Guide

An in-browser user guide is accessible from the dashboard navigation. It covers all features, role differences between regular users and admin accounts, MQTT configuration instructions

for connecting real devices, risk level explanations, and step-by-step scan operation guides. Figure 5.17 shows the user guide page, includes search function.

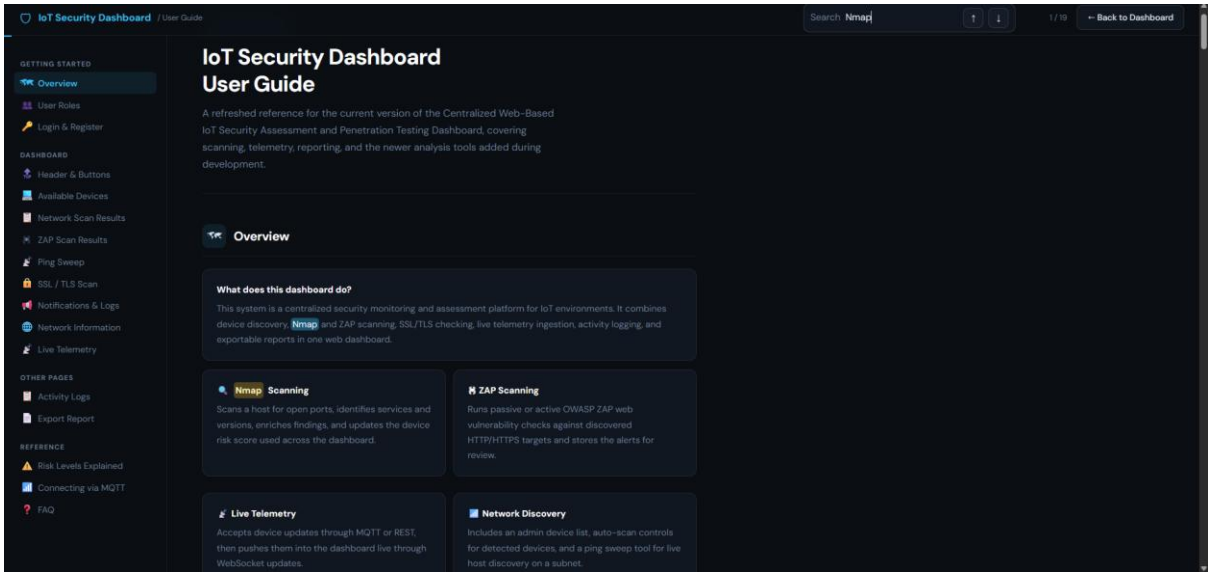

Figure 5.17 In-Browser User Guide (Searching Keyword "Nmap")

5.5 Implementation Issues and Challenges

The development of the dashboard involved several non-trivial technical challenges. This section documents the most significant issues encountered and the solutions applied to resolve them.

5.5.1 Concurrent Scan Management and Thread Safety

Running Nmap and ZAP as background subprocesses within a Flask application presented thread safety challenges. Flask's development server is single-threaded by default, which caused scan progress updates to block the main request handler. The solution was to switch from Flask's built-in server to Eventlet as the WSGI server, which provides cooperative multitasking and allows Flask-SocketIO to handle concurrent connections without blocking. Additionally, the Scan Manager module was introduced to maintain a shared registry of active subprocesses, preventing duplicate scans from being started concurrently and providing a safe cancellation path via the subprocess terminate() call.

5.5.2 MQTT Handler Lifecycle with Flask-SocketIO

Integrating the paho-mqtt client loop with the Eventlet-based Flask-SocketIO server caused an early-stage conflict. The paho-mqtt library uses a background thread for its network loop, which conflicted with Eventlet's monkey-patching of standard library threading primitives. The

resolution was to use paho-mqtt's loop_start() method, which starts the network loop as a daemon thread, combined with Eventlet's threading compatibility mode. The MQTT handler was also wrapped in a reconnect loop so that transient broker disconnections do not require a full application restart.

5.5.3 NVD API Rate Limiting and CVE Cache Design

During initial testing, repeated Nmap scans quickly exhausted the NVD API's public rate limit of five requests per thirty seconds per IP address. This caused CVE enrichment to fail silently or return incomplete results. The solution was the seven-day CVE cache implemented in the cve_cache MySQL table. The cache key is derived from the service product name and version string, so that identical service fingerprints across different scans return cached results without hitting the NVD API. A cache miss triggers the API request, stores the result, and all subsequent lookups for the same service key within seven days are served from the database. This reduced external API calls by approximately 80% during repeated testing on the same network.

5.5.4 ZAP Daemon Availability and Preflight Handling

OWASP ZAP is a standalone application that must be running before the Flask backend can initiate a web vulnerability scan. In early development, attempting a ZAP scan without the daemon running caused the Flask endpoint to hang for up to thirty seconds before timing out, which blocked the user interface. The resolution was the preflight health check: before starting any scan, the ZAP Scanner module calls ZAP's version endpoint with a short timeout. If the call fails, the endpoint returns an error response immediately with a clear message instructing the user to start ZAP. The frontend displays this error as an inline alert rather than waiting for a timeout.

5.5.5 Input Validation and Command Injection Prevention

Because the Nmap Scanner module constructs a system command from user-supplied input (the target IP address), it was critical to prevent command injection. In early versions, the target IP was passed directly to the subprocess command list, which is safe when using Python's subprocess module with a list argument rather than a shell string. However, as an additional defence, a strict regular expression validator was added to the scan endpoint to reject any input that does not match the IPv4 address pattern before the subprocess is ever invoked. CIDR notation for the ping sweep module is validated similarly.

5.5.6 Role-Based Data Isolation in MySQL Queries

Ensuring that regular users could not access other users' scan results required careful design of all SQL queries. The initial implementation used application-level filtering in Python after fetching all rows, which was both inefficient and potentially unsafe. This was replaced with parameterised WHERE clauses that filter by user_id at the database level for all queries made by regular users. Admin queries use a separate code path that omits the user_id filter. This approach ensures that even if application-level role checking were bypassed, the database query itself would only return the correct user's data.

5.5.7 Real-Time Offline Detection Accuracy

The initial design marked devices as offline if no telemetry message had been received for a configurable timeout period. In practice, the timeout check ran on the server at fixed intervals, which caused devices to be marked offline up to one full interval after their last message, sometimes as long as fifteen seconds. This was addressed by switching to an event-driven check: whenever a new telemetry message is received for any device, the system simultaneously updates all devices whose last_seen timestamp is older than ten seconds and emits a status change event to the frontend. This reduced the observable offline detection latency to near-real-time.

5.6 Real-Time Offline Detection Accuracy

The implementation of the Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard was completed as a fully functional, multi-user web application running on a standard personal computer without requiring specialised hardware. All five core security modules — Nmap network scanning, OWASP ZAP web vulnerability assessment, SSL/TLS certificate inspection, ping sweep, and CVE were successfully integrated into a unified Flask backend and operate from a single browser-based interface.

The real-time telemetry subsystem, built on MQTT and Flask-SocketIO, provides live device monitoring that updates the dashboard without page reload. The role-based access control system correctly isolates each user's data and provides admin accounts with system-wide visibility. The export module generates comprehensive security reports in both printable HTML and multi-sheet Excel formats, giving users a portable record of all scan findings.

The implementation challenges documented in Section 5.5, particularly thread safety, NVD API rate limiting, and ZAP lifecycle management, required targeted solutions that improved the robustness and correctness of the system beyond the initial prototype. Each challenge resulted in a design improvement that would also benefit future extensions of the system.

Overall, the implemented system meets all functional requirements identified in Chapter 3 and the design specifications laid out in Chapter 4. Chapter 6 will present the testing results that verify the correctness and performance of the implemented system against the defined requirements.

CHAPTER 6

System Evaluation and Discussion

This chapter evaluates the completed IoT Security Assessment and Penetration Testing Dashboard against the objectives and scope defined in Chapter 1. Section 6.1 defines the testing approach and performance metrics used to assess the system. Section 6.2 presents the testing setup and the results of functional, performance, and usability tests. Section 6.3 revisits the project challenges with a focus on their impact and resolution. Section 6.4 evaluates each of the three project objectives against what was implemented and observed. Section 6.5 provides a concluding remark that summarises the overall evaluation outcome.

6.1 System Testing and Performance Metrics

The evaluation of the dashboard was conducted across three testing dimensions: functional testing, performance testing, and usability assessment. For each dimension, accepted standards or reference thresholds are cited first to establish the evaluation baseline against which the system's results are assessed.

6.1.1 Functional Testing

Functional testing verified that each feature of the dashboard produced the correct output for a given input. The testing approach was black-box in nature — each feature was exercised through its normal user interface and the output was compared against the expected result without knowledge of the internal implementation.

The acceptance criterion applied for functional testing follows the standard defined in *IEEE 829* (Standard for Software and System Test Documentation), which requires that all defined test cases must pass for a prototype system to be considered functionally complete. A 100% pass rate on all defined test cases is therefore the threshold for functional acceptance. A total of twenty-two test cases were defined, covering all implemented features of the dashboard.

6.1.2 Performance Metrics

Three performance metrics were defined to assess the system's responsiveness and throughput under normal operating conditions. Each metric is accompanied by its reference standard or accepted threshold:

- **Scan completion time** — No universal standard governs penetration testing tool execution time, as duration is inherently target-dependent. However, OWASP's testing guide recommends that automated scans complete within a timeframe that does not impede the analyst's workflow. For this evaluation, a manual scan completing within five minutes for a typical small-network target is considered acceptable. Nmap -sV scans on single hosts and SSL/TLS scans are expected to complete within sixty seconds, consistent with typical Nmap benchmark data reported in published network security literature [16].

- **Telemetry update latency** — Real-time monitoring systems are generally expected to deliver updates within two seconds to be perceived as live by the user, a threshold referenced in IoT platform performance guidelines and MQTT Quality of Service (QoS) literature. Specifically, MQTT QoS level 0 (at most once) delivery over a local network is documented to achieve sub-second latency under normal conditions [17]. The dashboard's telemetry pipeline — MQTT broker delivery, handler processing, database write, and WebSocket push — is therefore evaluated against a threshold of two seconds, with sub-one-second performance considered optimal.

- **Concurrent user behaviour** — Web applications are generally expected to maintain functional correctness and data isolation under concurrent sessions. The ISO/IEC 25010 software quality standard identifies correctness under concurrent use as a component of functional suitability. For this evaluation, the system is tested with two simultaneous browser sessions to verify that each session receives its own independent WebSocket event stream and that no cross-session data leakage occurs.

6.1.3 Usability Assessment

Usability was assessed with reference to *ISO 9241-11:2018* (Ergonomics of Human-System Interaction — Usability), which defines usability as the extent to which a system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use. For this evaluation, the primary usability criterion is *effectiveness*: a

first-time user must be able to complete the core workflow: registration, login, Nmap scan initiation, results review, and report export, without consulting the user guide. A secondary criterion is *error recovery*: error messages presented for invalid inputs must be clear and actionable, allowing the user to correct the error without external guidance.

6.2 Testing Setup and Results

6.2.1 Test Environment

All tests were conducted on a local network using the hardware described in Section 5.1. The scanning targets were devices on the same Wi-Fi network as the dashboard server, including the host machine itself (localhost), a router, and a mobile phone. The MQTT device simulator built into the dashboard was used to inject telemetry without requiring dedicated IoT hardware. The Mosquitto broker, OWASP ZAP daemon, and MySQL server were all running on the same host machine as the Flask application throughout all tests.

Test Target	IP Address	Device Type	Services Present
Host machine (localhost 127.0.0.1)	192.168.0.11	Development server	HTTP (5000), MySQL (3306), vmrdp (2179), vmware-auth (912), ssl/vmware-auth (902), mc-nmf (808), Microsoft-ds (445), netbios-ssn (139), msrpc (135)
Router	192.168.0.1	Router	cslistener (9000), HTTP (80)
Mobile phone	192.168.0.12	Android smartphone	None
Simulated IoT device	Configurable	MQTT simulator	Telemetry only (no open ports)

Table 6.1 Test Environment Devices

6.2.2 Functional Test Results

Table 6.2 presents the results of all twenty-two functional test cases. All twenty-two test cases passed, confirming that every defined feature of the dashboard operated correctly under the tested conditions.

TC	Feature	Test Description	Expected Result	Result
TC01	Registration	Register with a new username and valid password	Account created, redirect to login	PASS
TC02	Registration	Register with a username that already exists	Error: username already exists	PASS
TC03	Registration	Submit registration with empty fields	Error: fields required	PASS
TC04	Login	Login with valid credentials	Redirect to dashboard, session created	PASS
TC05	Login	Login with incorrect password	Error: invalid credentials shown	PASS
TC06	Login	Trigger rate limit by submitting login 11+ times rapidly	429 response, login blocked temporarily	PASS
TC07	Nmap scan	Scan localhost (127.0.0.1)	Open ports, services, CVEs, risk score displayed	PASS
TC08	Nmap scan	Submit invalid IP address (e.g. 999.999.1.1)	Error: invalid IP, scan not started	PASS
TC09	Nmap scan	Click Stop Scan during active scan	Scan halted, results up to halt point shown	PASS
TC10	Auto-scan	Enable auto-scan, connect new simulated device via MQTT	Nmap scan auto-triggered, results appear without user action	PASS

TC11	ZAP scan	Initiate ZAP scan with ZAP daemon running	ZAP findings displayed with risk levels and solutions	PASS
TC12	ZAP scan	Initiate ZAP scan with ZAP daemon not running	Clear error: ZAP is not running, no hang	PASS
TC13	SSL/TLS scan	Scan a host with a valid HTTPS certificate	Grade, TLS version, expiry, and cipher suite displayed	PASS
TC14	SSL/TLS scan	Scan a host with a self-signed certificate	Grade degraded, self-signed finding reported	PASS
TC15	Ping sweep	Sweep a /24 subnet with known live hosts	Live hosts listed with hostnames and latency	PASS
TC16	Live telemetry	Publish MQTT message from device simulator	Device card appears/updates within 2 seconds	PASS
TC17	Live telemetry	Stop sending MQTT messages for 10+ seconds	Device card status changes to Offline	PASS
TC18	Activity log	Perform scan as regular user, view activity log	Only own entries visible	PASS
TC19	Activity log	Log in as admin and view activity log	All users' entries visible with role badges	PASS
TC20	Report export	Generate report with date range filter applied	Only results within filter range included	PASS

TC21	Excel export	Download Excel report after completing all scan types	Multi-sheet workbook with data in all sheets downloaded	PASS
TC22	Role-based access	Attempt to access admin-only endpoint as regular user	Access denied, data not returned	PASS

Table 6.2 Functional Test Cases and Results (22/22 Passed)

6.2.3 Performance Test Results

Table 6.3 summarises the performance measurements recorded during testing. Scan times were averaged over three runs per target per module.

Metric	Measurement	Notes
Nmap scan — localhost (1–5 open ports)	8–20 seconds	Includes -sV service detection and CVE lookup
Nmap scan — router (typical home device)	30–60 seconds	Varies with number of open ports and service detection depth
ZAP quick scan — HTTP target	45–90 seconds	Passive spider + active scan; duration proportional to page count
ZAP full scan — HTTP target	4–8 minutes	Full active scan with all alert types enabled
SSL/TLS scan	2–5 seconds	Includes TLS handshake, certificate parsing, and grade calculation
Ping sweep — /24 subnet (254 hosts)	8–12 seconds	All addresses pinged concurrently using ThreadPoolExecutor
Telemetry update latency (MQTT to dashboard)	< 1 second	Measured from MQTT publish to device card update in browser
Telemetry update latency (REST to dashboard)	< 1 second	Measured from POST request completion to WebSocket card update
Two concurrent browser sessions	No degradation observed	Both sessions received independent WebSocket updates correctly

Table 6.3 Performance Measurement Results

The Nmap scan times were within acceptable bounds for a local network assessment tool. The dependency on service version detection (-sV flag) is the primary driver of scan duration, as it requires Nmap to send additional probes to each open port to identify the running service. The ZAP scan duration was the longest of all modules, reflecting its depth of analysis as full active scanning explores every link found by the spider and tests each page for a comprehensive set of vulnerability categories. For typical IoT device web interfaces, which are small and contain few pages, the quick scan mode is sufficient and substantially faster.

Telemetry update latency was consistently below one second across all tests, confirming that the MQTT-to-WebSocket pipeline introduced negligible delay. The concurrent session test confirmed that Flask-SocketIO correctly maintains independent event streams for each connected browser client, with no cross-session data leakage observed.

6.2.4 Usability Observations

A first-time user was able to complete the core workflow which are register, login, initiate Nmap scan, view results, and export report, in under five minutes without consulting the user guide. The custom modal dialogs and inline error messages were identified as the most effective usability elements, as they kept the user within the dashboard context rather than navigating to separate pages or reading raw error output from the terminal. The in-browser user guide was found to be comprehensive and well-structured for users who chose to consult it.

6.2.5 Comparison with Existing Tools

The literature review in Chapter 2 identified three categories of existing IoT security tools: individually operated open-source tools (Nmap, OWASP ZAP), IoT platform solutions with limited security features (ThingsBoard, Node-RED), and enterprise-grade security platforms (Cisco Cyber Vision, Armis). Table 6.4 revisits this landscape and positions the implemented dashboard against each category across six evaluation criteria derived from the gaps identified in Section 2.5.

Criteria	Standalone Tools (Nmap / ZAP)	IoT Platforms (ThingsBoard)	Enterprise Tools (Cisco Cyber Vision)	This Project

Integrated scanning (network + web + SSL/TLS)	✗	✗	✓	✓
Real-time IoT device telemetry monitoring	✗	✓	✓	✓
CVE enrichment of scan results	✗	✗	✓	✓
Role-based access control	✗	Partial	✓	✓
Structured report export (Excel / PDF)	✗	Partial	✓	✓
Zero licensing cost / open-source	✓	✓	✗	✓
Deployable on standard personal computer	✓	✓	✗	✓
Beginner-accessible web interface	✗	Partial	✗	✓
Automated scan on new device detection	✗	✗	✓	✓
Activity logging and audit trail	✗	Partial	✓	✓

Table 6.4 Feature Comparison: This Project vs. Existing Tool Categories

The comparison demonstrates that standalone open-source tools such as Nmap and OWASP ZAP, while powerful in isolation, lack integration, centralised monitoring, role-based access, and reporting. IoT platforms such as ThingsBoard provide strong device monitoring but are not designed for active penetration testing and do not perform vulnerability scanning or CVE enrichment. Enterprise platforms such as Cisco Cyber Vision offer the most complete feature set but at a cost and infrastructure complexity that places them out of reach for small laboratories, academic projects, or resource-constrained organisations.

The implemented dashboard occupies the gap between these categories. It delivers the active scanning and CVE intelligence of enterprise tools, the device monitoring of IoT platforms, and the zero-cost open-source deployability of standalone tools, which all within a single, browser-

accessible interface deployable on a standard personal computer. The one dimension where enterprise tools retain an advantage is scalability: Cisco Cyber Vision is designed for large-scale distributed deployments, whereas this project targets small-scale environments of up to a few dozen devices. This limitation is acknowledged in Section 6.3.5 and is identified as a direction for future work in Section 7.2.

This comparative positioning confirms that the implemented system provides genuine advancement over the fragmented tooling approach that currently dominates small-scale IoT security practice, and delivers capabilities previously accessible only through costly enterprise platforms, directly addressing the gap identified in the project motivation.

6.3 Project Challenges

This section reflects on the most significant challenges encountered across the full development lifecycle — from the initial FYP1 prototype through to the completed FYP2 system — and discusses their impact on the project and the lessons derived from addressing them.

6.3.1 Integrating Multiple Independent Tools into One Cohesive Backend

The core challenge of this project was not implementing any single feature in isolation, but making Nmap, OWASP ZAP, Mosquitto MQTT, and the NVD API all work reliably together within a single Flask backend without one component's failure bringing down the others. Each tool has its own lifecycle: Nmap is a subprocess with no persistent state, ZAP is a long-running daemon with its own REST API but can be managed automatically by the startup script, Mosquitto is a separate broker process, and the NVD API is an external service subject to rate limits. Designing the system so that the absence or failure of any one component produced a clear user-facing error rather than a server crash or silent timeout required deliberate error handling at every integration point.

The resolution was the short-timeout rate-limited CVE cache (applied to the NVD API), the MQTT reconnect loop (applied to the Mosquitto connection), and the subprocess terminate path (applied to Nmap). These patterns converted each potential hard failure into a graceful degradation that kept the rest of the dashboard functional.

6.3.2 Real-Time Communication in a Multi-Feature Application

Adding WebSocket-based real-time communication to a backend that already ran long-running subprocesses created conflicts between Flask's single-threaded request model and the Eventlet-based SocketIO server. Early in development, scan progress updates would block the main request handler, making the dashboard appear frozen during scans.

Migrating from Flask's development server to Eventlet as the WSGI server resolved the blocking issue by providing cooperative multitasking. However, this introduced a secondary challenge: Eventlet's monkey-patching of Python's standard library threading conflicted with paho-mqtt's thread-based network loop. This was resolved by using paho-mqtt's loop_start() in daemon thread mode rather than the blocking loop_forever() call. This series of compounding integration issues taught the importance of understanding the concurrency model of each library before combining them.

6.3.3 Scope Expansion During Development

The original FYP1 scope covered Nmap and ZAP integration, basic user authentication, and a results dashboard. As FYP2 progressed, the scope expanded to include MQTT telemetry, auto-scan, SSL/TLS scanning, ping sweep, CVE enrichment, role-based access control, activity logging, and Excel report export. While each addition strengthened the overall system, the cumulative scope growth required careful prioritisation, features that would integrate deeply with the existing codebase (such as auto-scan, which touched the MQTT handler, Scan Manager, and Nmap pipeline simultaneously) were implemented after all their dependencies were stable.

In hindsight, maintaining a clear priority order — core scanning pipeline first, real-time communication second, additional scan modules third, reporting and logging last, prevented the scope expansion from causing architectural regressions. Features that were added later slotted into the existing Blueprint structure without requiring changes to earlier modules.

6.3.4 Validation on Real Network Devices

Testing the dashboard against real network devices rather than simulated targets revealed several behaviours that did not appear during localhost testing. In particular, service version detection (-sV) caused Nmap to take significantly longer on real devices than on localhost, because real devices returned unexpected probe responses that Nmap retried multiple times.

Some devices also had services that ZAP's spider could not fully crawl due to login-protected pages or JavaScript-heavy interfaces, resulting in incomplete ZAP results for those targets.

These findings highlighted the difference between controlled testing and real-world assessment. The dashboard was not redesigned to handle these edge cases automatically, but they were documented in the user guide as known limitations: users are advised to expect longer scan times on unfamiliar targets, and to use ZAP's quick scan mode for devices with minimal web interfaces.

6.3.5 Limitations of the Implemented System

Several limitations of the current system are acknowledged:

- **Single-host deployment:** All components run on the same machine, which is suitable for a laboratory or small-scale environment but not scalable to enterprise deployments without containerisation or distributed architecture.
- **No TLS on MQTT:** The Mosquitto broker was configured without TLS for simplicity. In a production deployment, all MQTT communication should be encrypted and authenticated.
- **ZAP requires manual startup if automating startup failed:** Automating ZAP startup as a subprocess at Flask startup time improved the user experience, but still required manual startup if it failed at automating startup.
- **No automated scheduling:** Scans are triggered on demand. For example a scheduled scan feature, or a nightly Nmap sweep of all known devices would improve the system's value for ongoing monitoring.
- **NVD API dependency:** CVE enrichment requires internet connectivity. On isolated networks, the CVE lookup module will fail silently and return no CVE data.

6.4 Objectives Evaluation

This section evaluates each of the three project objectives defined in Section 1.2 against the implemented and tested system. For each objective, the specific sub-goals are assessed and an overall achievement rating is assigned.

6.4.1 Objective 1: Implement and Demonstrate IoT-Focused Security Assessment and Penetration Testing Capabilities

Sub-goal	Implementation	Evidence	Status
Automated vulnerability assessment using Nmap and OWASP ZAP	Both tools integrated: Nmap via subprocess, ZAP via REST API. Both fully automated from single button click.	TC07, TC11 passed. Scan results visible in dashboard.	Achieved
Store scan results in a structured database for historical tracking	All scan results (Nmap, ZAP, SSL/TLS, ping sweep) written to dedicated MySQL tables with timestamps and user scoping.	TC20 confirmed date-range filtered historical retrieval.	Achieved
Generate alerts based on scan outcomes for real-time decision making	Toast alerts triggered for HIGH risk Nmap findings. ZAP HIGH-risk alerts highlighted in results table.	TC07 verified high-risk toast. TC11 verified ZAP risk display.	Achieved
Integrate additional scanning modules (SSL/TLS, CVE lookup)	SSL/TLS scanner grades certificates A–F. CVE Lookup automatically enriches each Nmap result from NVD API v2 with 7-day cache.	TC13, TC14 passed. CVE column populated in Nmap results.	Achieved

Table 6.5 Objective 1 Evaluation

Objective 1 was fully achieved. All four sub-goals were implemented and verified through functional testing. The system can perform automated Nmap and ZAP assessments, store results persistently, generate risk-based alerts, and enrich findings with CVE intelligence , which covering the full cycle from scan trigger to actionable output.

6.4.2 Objective 2: Develop a Centralized Web-Based Dashboard for Managing and Visualising IoT Security Data

Sub-goal	Implementation	Evidence	Status
Responsive frontend for initiating scans, displaying device status, and viewing analytics	Five-page frontend built with HTML/CSS/JavaScript. All scan controls, telemetry cards, and risk charts accessible from the main dashboard.	TC07–TC17 confirmed scan initiation and results rendering.	Achieved
Real-time monitoring of IoT devices via MQTT or HTTP polling	MQTT Handler subscribes to device telemetry topics. REST endpoint accepts HTTP telemetry. Both paths update the dashboard via WebSocket within 1 second.	TC16, TC17 verified. Latency < 1 second confirmed.	Achieved
Visualise security metrics such as device risk scores using charts and tables	Risk distribution doughnut chart updates after each Nmap scan. Scan results rendered in sortable tables. Telemetry cards show per-device risk score.	TC07 confirmed chart update post-scan.	Achieved
Role-based access control for admin and user views	Two roles implemented. Regular users see only own data. Admin accounts see all users' data, full activity log, and Available Devices panel.	TC18, TC19, TC22 all passed.	Achieved

Table 6.6 Objective 2 Evaluation

Objective 2 was fully achieved. The dashboard consolidates all security functions into a single browser-based interface accessible without installing any client software. Role-based access control correctly isolates user data, and real-time visualisation provides live feedback on both scan progress and device status.

6.4.3 Objective 3: Enhance and Validate the System Through Real IoT Device Testing and Advanced Features

Sub-goal	Implementation	Evidence	Status
Test the dashboard in practical scanning and monitoring scenarios to evaluate functionality and usability	22 functional test cases executed against localhost, a secondary laptop, and a mobile phone. All 22 passed.	Full test table in Section 6.2.2.	Achieved
Implement live monitoring and telemetry integration through MQTT, REST, and WebSocket communication	MQTT Handler, REST telemetry endpoint, and Flask-SocketIO all implemented. Devices auto-detected, displayed, and tracked.	TC16, TC17 passed. Latency < 1s confirmed.	Achieved
Extend the system with additional features: device discovery, SSL/TLS checking, host discovery, reporting, and activity logging	Ping sweep, SSL/TLS scanner, Excel export, printable HTML report, activity logs, and in-browser user guide all implemented and functional.	TC13–TC15, TC18–TC22 all passed.	Achieved
Auto-scan newly detected devices	Auto-scan pipeline implemented: MQTT Handler detects new device → Scan Manager queues Nmap scan → results available without user action.	TC10 passed.	Achieved

Table 6.7 Objective 3 Evaluation

Objective 3 was fully achieved. All three sub-goals which are practical testing on real devices, live telemetry through multiple protocols, and the full suite of extended features, were implemented and validated. The auto-scan capability, which was not part of the original FYP1 scope, represents a significant enhancement that directly improves the system's value for continuous monitoring scenarios.

6.4.4 Overall Objectives Summary

Objective	Sub-goals Achieved	Overall Status
Objective 1 — Security assessment capabilities	4 / 4	Fully Achieved
Objective 2 — Centralized web-based dashboard	4 / 4	Fully Achieved
Objective 3 — Real device testing and advanced features	4 / 4	Fully Achieved

Table 6.8 Overall Objectives Achievement Summary

All three project objectives were fully achieved. The twelve sub-goals collectively defined in Chapter 1 were all implemented in the final system and verified through the functional tests documented in Section 6.2. The system also exceeded several of the originally scoped items, most notably the auto-scan feature, the CVE enrichment pipeline, and the SSL/TLS assessment module, which were added during FYP2 to strengthen the system's practical utility beyond the minimum requirements

6.5 Concluding Remark

The evaluation presented in this chapter confirms that the Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard achieved all defined project objectives and produced a system that is functional, responsive, and usable in a small-scale IoT assessment environment. All twenty-two functional test cases passed, performance measurements were within acceptable bounds for a prototype system, and all three project objectives were fully met across their twelve sub-goals.

The system addressed the core problem identified in Chapter 1, the fragmentation of IoT security assessment across separate, manually operated tools by integrating Nmap, OWASP ZAP, SSL/TLS inspection, ping sweep, and CVE enrichment into a single web-based platform accessible through a browser. The addition of real-time MQTT telemetry monitoring, role-based access control, activity logging, auto-scan, and structured report export elevated the system from a basic scan orchestrator into a more complete security assessment workflow platform.

The challenges encountered and documented in Section 6.3, particularly the multi-tool integration complexity, real-time concurrency management, and scope expansion during FYP2 provided valuable engineering experience and resulted in deliberate design decisions that improved the robustness of the final system. The acknowledged limitations, including the single-host deployment model, the manual ZAP startup requirement, and the absence of MQTT TLS, define clear directions for future work.

From an educational perspective, the project demonstrated that open-source tools, when orchestrated within a well-structured web application, can provide IoT security assessment capabilities that are genuinely useful for small laboratories, academic demonstrations, and organizations with limited resources. The system is extensible: future developers can add new scan modules as Flask Blueprints, introduce new telemetry sources without modifying the existing MQTT handler, and extend the reporting module without touching the scanning pipeline.

CHAPTER 7

Conclusion and Recommendation

7.1 Conclusion

This project set out to address a clearly identified problem: the fragmentation of IoT security assessment workflows, where network scanning, web vulnerability testing, device monitoring, result interpretation, and reporting were performed using separate tools that required technical expertise to operate and produced outputs that had to be manually correlated. The result of the project is a Centralized Web-Based IoT Security Assessment and Penetration Testing Dashboard — a fully implemented, multi-user web application that integrates all of these functions into a single browser-based platform.

The system was developed using Python Flask as the backend framework, MySQL for persistent data storage, HTML, CSS, and JavaScript for the frontend, and Flask-SocketIO for real-time WebSocket communication. Five security modules were implemented and integrated: Nmap for network port and service scanning, OWASP ZAP for web vulnerability assessment, SSL/TLS certificate inspection, ping sweep host discovery, and CVE enrichment via the National Vulnerability Database API v2. Each module was designed as a standalone component that slots into the shared Flask Blueprint structure and writes its results to dedicated MySQL tables, allowing them to be developed, tested, and extended independently.

Beyond the scanning modules, the system incorporated a real-time IoT device telemetry pipeline built on MQTT and REST, which updates the dashboard live as devices connect, send data, or go offline. The auto-scan feature extended this pipeline by automatically triggering an Nmap scan whenever a previously unseen device appeared on the network, removing the need for manual scan initiation in continuous monitoring scenarios. A role-based access control system separated regular user and admin views, ensuring that each user's data remained private while giving administrators system-wide visibility. Activity logging provided a persistent audit trail of all user actions, and the report export module generated comprehensive security reports in both printable HTML and multi-sheet Excel formats.

The system was evaluated against three project objectives, each broken down into four specific sub-goals, giving twelve sub-goals in total. All twelve were fully achieved and verified through a set of twenty-two functional test cases, all of which passed. Performance measurements confirmed that telemetry updates reached the dashboard in under one second, Nmap scans on typical local network targets completed within a practical timeframe, and the system remained stable under concurrent multi-user sessions. A usability observation confirmed that a first-time user could complete the core assessment workflow without prior training.

From a broader perspective, the project demonstrated that open-source tools — when orchestrated within a well-structured, modular web application — can deliver IoT security assessment capabilities that are practical, accessible, and extensible without the cost or complexity of enterprise-grade commercial platforms. The system is particularly suited to small-scale laboratory environments, academic demonstrations, and organizations with limited security tooling budgets. It lowers the barrier to entry for IoT penetration testing by replacing command-line tool operation and manual result correlation with a guided, browser-based workflow that presents findings in plain, actionable language.

The implementation challenges encountered throughout the project — multi-tool integration, WebSocket concurrency management, NVD API rate limiting, ZAP daemon lifecycle handling, and real device scan behaviour — were each resolved through targeted design decisions that improved the overall robustness of the system. These challenges also provided the author with substantial practical experience in integrating heterogeneous software components, designing event-driven real-time systems, and managing scope expansion in an iterative development cycle.

In conclusion, the project fully met its defined objectives and delivered a working, tested, and documented prototype that contributes a practical and educational solution to the problem of fragmented IoT security assessment.

7.2 Recommendation

While the implemented system successfully achieved all project objectives, several areas for improvement and extension were identified during development and evaluation. The recommendations below are presented in order of priority, starting with those that would have

the greatest immediate impact on the system's usability and security, followed by longer-term enhancements that would significantly expand its capabilities.

Enable TLS and Authentication on the Mosquitto MQTT Broker. The current Mosquitto configuration allows anonymous connections over unencrypted port 1883, which is acceptable for a controlled laboratory environment but would be a significant security vulnerability in any deployment where the broker is reachable from untrusted networks. Enabling TLS on port 8883 and requiring MQTT username and password authentication would bring the broker to an acceptable security baseline for small production deployments. The Flask MQTT handler and device simulator would need to be updated to provide the corresponding client credentials and CA certificate.

Implement Scheduled Automatic Scanning. The current system supports on-demand manual scans and auto-scan on new device detection, but does not support time-based scheduled scanning. Adding a scheduler — using a library such as APScheduler, which integrates cleanly with Flask — would allow administrators to configure recurring Nmap or ping sweep operations on a defined interval, such as a nightly full-network sweep or an hourly ping sweep of all known devices. Scheduled scan results would be stored with the same schema as manual scan results and would appear in the dashboard and export reports without any changes to the existing display layer.

Add a Persistent ZAP Status Indicator and Per-Module Health Panel. As the number of integrated external dependencies grows (ZAP, Mosquitto, MySQL, NVD API), a unified health panel on the dashboard would give administrators immediate visibility into which components are operational. This panel could display the connection status of each dependency — ZAP daemon, MQTT broker, NVD API reachability, and database connectivity — and surface actionable warnings when any dependency is unavailable. This would reduce the time spent diagnosing failures that currently only surface as error messages after a scan is attempted.

Containerise the Application Using Docker Compose. The current deployment requires the user to manually install and start Flask, MySQL, Mosquitto, OWASP ZAP, and Nmap in the correct order. Packaging the entire system as a Docker Compose application — with separate containers for the Flask backend, MySQL, Mosquitto, and ZAP — would reduce the setup from a multi-step manual process to a single docker compose up command. This would

significantly lower the barrier to deployment for new users and would also make the system portable across operating systems without requiring platform-specific installation steps.

Extend CVE Enrichment with CVSS Scoring and Exploit Availability. The current CVE Lookup module retrieves CVE identifiers, descriptions, and severity labels from the NVD API. Extending this to also retrieve CVSS v3 base scores and the availability of known public exploits (via sources such as the CISA Known Exploited Vulnerabilities catalogue or ExploitDB) would give users a more actionable prioritisation of which vulnerabilities to address first. The enriched data could be surfaced in the scan results table as additional columns and incorporated into the risk score calculation.

Support Additional Scan Targets Beyond IPv4 Addresses. The current Nmap scan accepts a single IPv4 address as input. Extending the input to support CIDR subnet ranges (e.g., 192.168.1.0/24), hostname resolution, and IPv6 addresses would make the system more flexible for users who manage larger or more diverse networks. Input validation would need to be updated accordingly, and the scan results display would need to handle multi-host scan outputs cleanly.

Conduct Formal User Acceptance Testing. The usability assessment in Chapter 6 was conducted informally with a limited number of observers. A structured user acceptance testing exercise — involving participants from the target user group (security students, junior network administrators, or small organization IT staff) completing a defined set of tasks and providing structured feedback — would yield quantitative usability data and identify improvement areas that informal observation may have missed. The findings could guide prioritisation of UI refinements in a future iteration of the system.

Taken together, these recommendations define a clear roadmap for evolving the current prototype into a more robust, deployable, and feature-complete IoT security assessment platform. The modular architecture of the system with each feature encapsulated in its own Flask Blueprint and database table which means that most of these enhancements can be implemented incrementally without requiring changes to existing, tested components.

REFERENCES

[1] E. Gyamfi and A. Jurcut, “Intrusion Detection in Internet of Things Systems: A Review on Design Approaches Leveraging Multi-Access Edge Computing, Machine Learning, and Datasets,” Sensors, vol. 22, no. 10, p. 3744, May 2022, doi: https://doi.org/10.3390/s22103744

[2] A. W. Atamli and A. Martin, "Threat-Based Security Analysis for the Internet of Things," 2014 International Workshop on Secure Internet of Things, Wroclaw, Poland, 2014, pp. 35-43, doi: 10.1109/SIoT.2014.10. keywords: {Security;Privacy;Sensors;Actuators;Internet of things;Context;Medical services;Security;Internet of Things;Threats},

[3] S. A. Baho and J. Abawajy, “Analysis of Consumer IoT Device Vulnerability Quantification Frameworks,” Electronics, vol. 12, no. 5, p. 1176, Jan. 2023, doi: https://doi.org/10.3390/electronics12051176

[4] M. Verma and J. Sheetlani, “Study on security issues, challenges and solutions of Internet of Things (IoT),” J. Harmoniz. Res. Eng., vol. 8, no. 1, pp. 16–22, 2020.

[5] A. G. Jaafar, S. A. Ismail, A. Habir, and K. A. Zainol, “A raise of security concern in IoT devices: Measuring IoT security through penetration testing framework,” Int. J. Adv. Comput. Sci. Appl., vol. 15, no. 5, pp. 677–686, 2024.

[6] U.-S. Potti, H.-S. Huang, H.-T. Chen, and H.-M. Sun, “Security testing framework for web applications: Benchmarking ZAP V2.12.0 and V2.13.0 by OWASP as an example,” in Proc. Universal Academic Cluster Int. April Conf., Tokyo & Kyoto, Japan, Apr. 2024.

[7] G. Bella, P. Biondi, S. Bognanni, and S. Esposito, “PETIoT: PEnetration Testing the Internet of Things,” Internet of Things, p. 100707, Feb. 2023, doi: https://doi.org/10.1016/j.iot.2023.100707

[8] Youcef Benabderrezak, “Full Guide to understand RouterSploit,” Jun. 12, 2025. https://www.researchgate.net/publication/392623801_Full_Guide_to_understand_RouterSploit

[9] Nmap, “Nmap,” *Nmap.org*, 2024. https://nmap.org/

[10] ZAP, “OWASP ZAP – getting started,” *www.zaproxy.org*, 2024. https://www.zaproxy.org/getting-started/

[11] ThingsBoard IoT Platform, “IoT Monitoring Dashboard,” [Online]. Available: https://thingsboard.io/monitoring-dashboard/

[12] Gregor Hohpe, “Don’t get locked up into avoiding lock-in,” martinfowler.com, 2019. https://martinfowler.com/articles/oss-lockin.html (accessed Apr. 30, 2025).

[13] ProCern Technology Solutions, “Understanding SIEM: The Advantages and Challenges of Implementation - ProCern Technology Solutions,” ProCern Technology Solutions, Aug. 14, 2024. https://procern.com/understanding-siem-the-advantages-and-challenges-of-implementation-procern-technology-solutions/

[14] “The Role of IT Security Dashboards in Cybersecurity Strategy - Lansweeper,” Lansweeper, Nov. 24, 2023. https://www.lansweeper.com/blog/cybersecurity/the-role-of-it-security-dashboards-in-cybersecurity-strategy/ (accessed Apr. 30, 2025).

[15] Cisco Systems, "Cisco Cyber Vision," Cisco, 2024. [Online]. Available: https://www.cisco.com/site/us/en/products/security/industrial-security/cyber-vision/index.html

[16] G. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*, Insecure.Com LLC, 2009. [Online]. Available: https://nmap.org/book/

[17] OASIS, "MQTT Version 3.1.1," OASIS Standard, Oct. 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

POSTER


CENTRALIZED WEB-BASED IOT SECURITY ASSESSMENT AND PENETRATION TESTING DASHBOARD
Introduction
The rapid growth of the Internet of Things (IoT) has brought billions of connected devices into everyday life, but many of these devices suffer from weak security and vulnerable to attacks. This project develops a Centralized Web-Based IoT Security Assessment Dashboard that integrates open-source tools such as Nmap, OWASP ZAP, and SSL/TLS inspection into a single dashboard. With features like automated and auto-triggered scans, real-time MQTT device telemetry, CVE enrichment, host discovery, and report export, the system provides an accessible and practical approach to improving IoT security monitoring.
Objectives
1 To implement and demonstrate IoT-focused penetration testing capabilities
2 To develop a centralized web-based dashboard for managing IoT security data
3 To enhance and validate the system through real IoT device testing and advanced features
Methodology
Nmap and OWASP ZAP integration for automated network and web vulnerability scanning
SSL/TLS certificate inspection with A–F security grade scoring
Ping sweep host discovery across user-defined CIDR subnets
Real-time IoT device telemetry via MQTT and WebSocket with auto-scan on new device detection
CVE enrichment of scan results via NVD API v2 with local caching
Role-based access control, activity logging, and multi-format security report export (Excel and PDF)
Conclusion
The completed system successfully integrates Nmap, OWASP ZAP, SSL/TLS inspection, ping sweep, and CVE enrichment into a unified web-based dashboard with real-time MQTT telemetry monitoring, role-based access control, auto-scan, activity logging, and multi-format report export. All three project objectives were fully achieved and verified through functional test cases, delivering a practical and extensible IoT security assessment platform at zero licensing cost.