PwnBoxer: A Device-Based Network Auditing Tool

Franco Mezzarapa, Robert Lee, Gugulethu Sigogo, and Abdiel Marcano

Dept. of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — PwnBoxer is a plug-and-play network security device that automatically scans a network, identifies vulnerabilities, and delivers a plain-language report suggesting fixes. The technical expertise required is minimal to operate, but can be leveraged by power users and security service providers. All of these are delivered at a lower cost than a traditional network security test carried out by a professional. This is accomplished by original code running on a combination of an ESP32 microcontroller and a Raspberry Pi Compute Module 4 single-board computer, and is controlled by the user through a capacitive touch LCD.**

I. Introduction

The PwnBoxer is a hardware network auditing device intended for home and small businesses, allowing consumers to achieve affordable automated network auditing with feedback reporting for remediation of basic to intermediate network vulnerabilities. The PwnBoxer is intended to run through an automated workflow that audits the networks to which it is connected using readily available penetration testing tools to audit and generate a report.

This tool also allows for technician connection into the network, making it a viable option for Internet Service Providers (ISPs) and Managed Service Providers (MSPs) to provide as a service, in which they can lend the device to do continuous penetration testing and, when needed, have a technician connect and continue the audit manually.

II. System Components

A. Microcontroller

The user-facing side of the system is run entirely by the ESP32-S3. This microcontroller was selected for its processing ability, large number of GPIO pins, Wi-Fi capability, and support for several peripheral communication protocols while maintaining a low cost. With its dual-core Xtensa LX7 processor running at up to 240MHz, and 45 GPIO pins, the S3 handles the capacitive touch LCD and flash memory well without significant latency issues.

The code for the board was written in Visual Studio Code and debugged using the PlatformIO extension. The code was written in C++ using the ESP-IDF framework.

B. Single Board Computer

The single board computer for this system is responsible for all networking-related computation, which requires it to have highly capable processing power. Additionally, it must have some included IO ports, such as ethernet, USB-A and HDMI for developer and technician support. Therefore, the chosen SBC for this project is the Raspberry Pi Compute Module 4, as it satisfies these requirements with the assistance of a carrier PCB to provide necessary ports.

C. LCD TFT

The display is the main avenue for the user to interact with PwnBoxer. We selected the Crystalfontz 3.5” full-color TFT LCD module with 320x480 resolution with capacitive touch. This module was chosen because it was a small enough size that was still usable as a touchscreen, it had a legible resolution with our interface, and it supported many peripheral communication protocols. The protocols the module supported were I2C for touch, and SPI, 8, 9, 16, and 18-bit parallel for display. We chose to use SPI as it was fast enough for our use case and was simpler to wire.

D. External Flash Memory

The flash memory module is intended to store user email data from the microcontroller, as well as the splash art for the startup screen. It is a 4MB module that uses the SPI peripheral protocol. This external memory module takes a significant load off of the internal memory of the ESP32-S3.

E. USB-to-UART Interface

A USB-to-UART bridge is included in the design to allow programming and debugging of the ESP32-S3 microcontroller. The selected device for this interface is the Silicon Labs CP2102 USB-to-UART controller, which converts USB signals from a host computer into UART serial communication.

III. System Concept

To understand the system as a whole, some flowcharts will be beneficial:

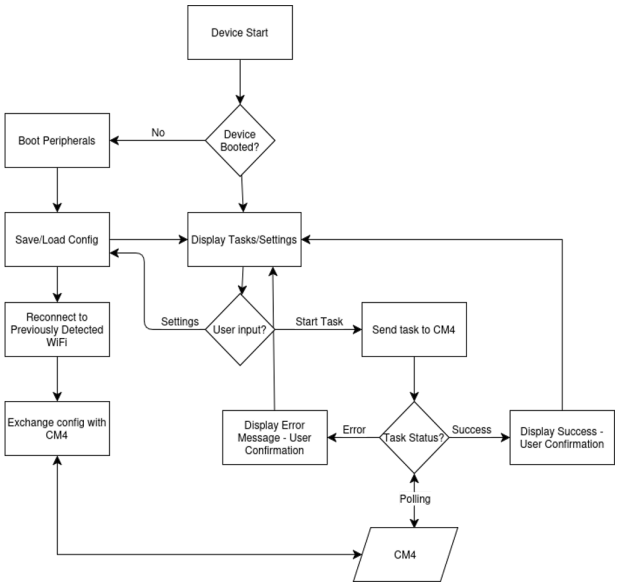

Fig. 1 Firmware Flowchart

This firmware diagram depicts the program execution for the MCU when the system boots, the microcontroller runs through the boot process and initializes the peripherals, loads the saved configuration from flash, and then attempts to reconnect to any previously detected WiFi networks.

The microcontroller and the SBC exchange information over UART — this is a two-way exchange. The microcontroller sends over saved Wi-Fi SSIDs, passwords, users, and settings; the SBC sends back the list of available tasks for the microcontroller to display.

Once both sides are synchronized, the display shows all the available tasks and settings. The user can then either select a task to run or edit the current settings. If the user decides to edit the current settings, the program will store the new configuration in flash and then return to display the tasks. If the user decides to start a task, it will send the task information to the SBC.

From there, the microcontroller starts polling the SBC for the task status. If the task comes back with an error, the display shows an error message and waits for the user to confirm before returning to the main screen. If the task succeeds, it shows a success message, waits for confirmation, and also routes back to the main display.

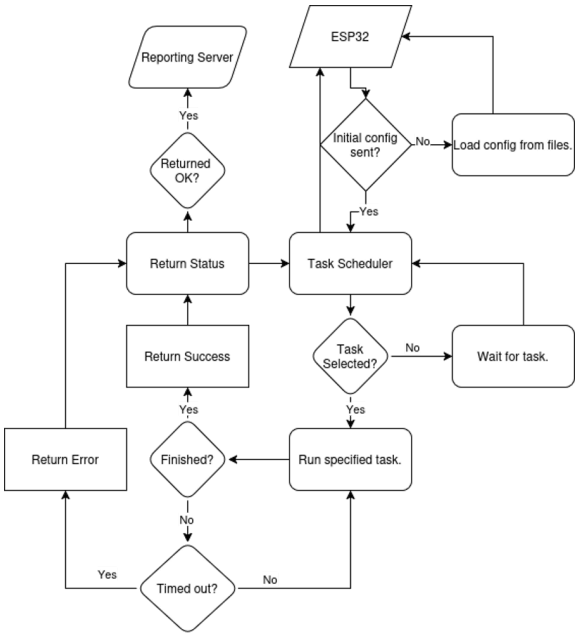

Fig. 2 Scheduler Flowchart

This software diagram depicts the software execution flow when the Raspberry Pi boots.

First, the scheduling software is loaded automatically as a system service, this allows the software to be loaded on restart. Once the software has been started, it checks whether the initial configuration has already been set. If it has not, it loads the task configuration from local files and sends the microcontroller the relevant task information so the display can be populated. Once that's done, or if the configuration was already in place, execution moves into the task scheduler.

The scheduler sits in a wait state until it receives a task from the microcontroller. When a task comes in, it pulls the parameters from that task's configuration file and starts execution. At the same time, it starts a timer and periodically checks whether the task is still running.

From here, there are three outcomes: If the task runs past its timeout window, the scheduler terminates it and returns a timeout error. If the task crashes for any reason during execution, it returns a failure error. If the task completes successfully, it returns a completion code and sends the report files to the webserver. All three of these codes are sent back to the microcontroller to be displayed on the screen.

IV. Hardware Detail

This section describes the physical hardware integration of the components introduced in Section II. While Section

II described the major devices used in the system, this section focuses on how those components are interconnected and how the hardware architecture supports the overall system functionality.

A. Hardware Architecture

The PwnBoxer hardware architecture is divided into two primary subsystems: the user interface subsystem and the network processing subsystem. The user interface subsystem is managed by the ESP32-S3 microcontroller and is responsible for handling touchscreen input, rendering the graphical interface, and managing configuration data stored in external flash memory. The network processing subsystem is managed by the Raspberry Pi Compute Module 4.

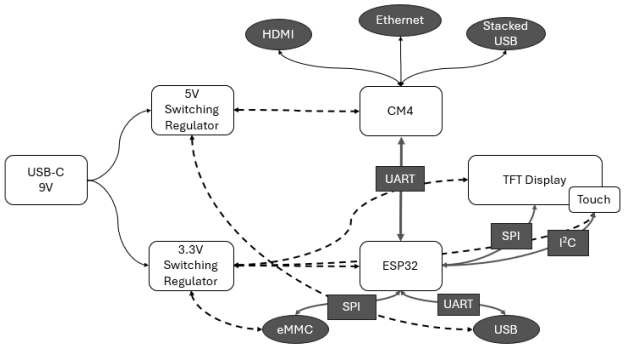

Fig. 3 Hardware Diagram

This system executes the network auditing software and performs the vulnerability analysis tasks requested by the user. Separating the system into these two subsystems allows the user interface to remain responsive while the Compute Module performs computationally intensive network scanning tasks. Figure 3 illustrates the overall hardware architecture and communication interfaces used throughout the system.

B. ESP32 to CM4 Communication

Communication between the ESP32-S3 and the CM4 occurs through a UART serial communication interface. This interface enables two-way communication between the MCU and the SBC. The ESP32 transmits configuration information such as: stored WiFi credentials, selected audit tasks, and system settings. The CM4 responds by transmitting task information and execution status updates back to the ESP32 so they can be displayed to the user.

C. Display and Touch

As introduced in Section II, the selected Crystalfontz TFT LCD module supports several communication protocols. For this implementation, the SPI interface was selected because it provides sufficient data throughput for the graphical interface while minimizing the number of required GPIO connections on the ESP32-S3.

The display module is connected to the system through two flexible flat cable connectors. The display signals are routed through a 50-position Zero Insertion Force (ZIF) connector, which carries the SPI communication lines, control signals, and power required for operation of the display panel. The capacitive touch interface is connected through a separate 8-position ZIF connector, which carries the signals used by the touch controller.

D. External Flash Memory

The W25Q32 provides 32 Mbit (4 MB) of non-volatile storage and communicates with the ESP32-S3 through the SPI bus. This flash device stores persistent system data, configuration settings, graphical assets, and user information.

E. ESP32 Programming

A USB-to-UART bridge is included in the design to allow firmware programming and debugging of the ESP32-S3 microcontroller. The selected device for this interface is the CP2102 USB-to-UART controller, which converts USB communication from a development computer into UART signals that the ESP32 can interpret.

F. Power Architecture

The PwnBoxer system receives power through a USB-C input, which negotiates a 9V supply using a USB-C power delivery controller. This input voltage provides sufficient headroom for the switching regulators used in the system. The negotiated 9V input is distributed to two switching regulators that generate the primary system voltage rails. A 5V switching regulator supplies power to the Compute Module 4 and other 5V peripherals, while a 3.3V switching regulator powers the ESP32 microcontroller and other low-voltage digital components

V. Software Detail: Web Platform and AI Integration

The software architecture of the PwnBoxer ecosystem is bifurcated into two primary domains: the device-level auditing scripts executing on the hardware, and a centralized, full-stack web platform responsible for data ingestion, artificial intelligence processing, and user management. The web platform is a comprehensive suite engineered utilizing a Next.js frontend, a Node.js/Express.js backend, and a PostgreSQL database.

A. Full-Stack Architecture and Database Design

The foundation of the web platform is a highly normalized PostgreSQL relational database, accessed and mutated via the Prisma Object-Relational Mapper (ORM). This schema strictly defines user accounts, authorized hardware devices, file upload metadata, and the lifecycle states of generated reports. The relational structure enforces cascading deletions; if a user purges a report from their dashboard, the associated file metadata and physical payloads are recursively removed from the server, thereby mitigating storage bloat.

To manage interactions between human users and headless auditing devices, the backend implements a dual-mode authentication gateway. User sessions, initiated via the frontend login portal, are secured using short-lived JSON Web Tokens (JWT). Conversely, the hardware devices rely on cryptographic API keys. To maintain zero-trust security principles, the backend stores only a bcrypt-hashed representation of the device's API secret.

B. Secure API Design and Threat Mitigation

Protecting the Express.js Application Programming Interface (API) from malicious payloads was a primary engineering constraint. The backend utilizes a modular router architecture protected by custom authorization middleware (requireJwt and requireJwtOrApiKey). When a device attempts to upload a scan, the middleware intercepts the HTTP POST request, parses the Authorization: Bearer header, and validates the hashed credential via timing-safe comparison algorithms before processing the request.

Furthermore, the system mitigates directory traversal and malicious file execution attacks during the upload phase. Instead of trusting user-provided filenames, the backend leverages Node.js's native crypto library. Incoming files are processed by the multer middleware, stripped of their original nomenclature, and reassigned a randomized 16-byte hexadecimal string (crypto.randomBytes(16)). This ensures absolute file uniqueness and sanitizes the file system against injection attacks.

C. Frontend Interface and Dynamic State Management

The user-facing web portal operates as a modern Single Page Application (SPA) built with React and the Next.js framework, utilizing Tailwind CSS for responsive, utility-first styling. The dashboard automatically scales its layout matrices using predefined CSS breakpoints, ensuring the administrative portal is equally accessible on desktop workstations and mobile devices.

The frontend heavily utilizes React's hook-based architecture (useState, useEffect, and useRef) to manage dynamic client-side states. The User Dashboard employs asynchronous API polling to fetch real-time updates from the database, hydrating the Document Object Model (DOM) with visual indicators - such as dynamically rendered status badges - that reflect the current processing stage of a vulnerability report.

Rendering the AI-generated HTML reports presented cross-origin security challenges. Rather than redirecting the user to a raw file path, the frontend initiates an authenticated GET request. The backend streams the HTML data, which the React client processes as a raw binary Blob. The application then utilizes window.URL.createObjectURL() to generate a temporary, localized URL, allowing the browser to safely render the styled vulnerability report in a sandboxed tab, effectively mitigating direct file-system execution risks.

D. Payload Processing and Context Window Management

Before incoming scan data can be analyzed by an AI model, it must be programmatically sanitized to prevent exceeding maximum token context windows. A custom utility function, safeReadText, was engineered to extract text from the stored binary files.

First, the parser implements a heuristic binary check. It scans the initial 2048 bytes of the file; if a high density of null bytes is detected (greater than a threshold of three), extraction halts, and a diagnostic warning is passed to the AI to prevent model hallucinations. For valid text files, the system enforces a strict 2000-character truncation limit. This ensures that massive log files do not overwhelm the Large Language Model's context window, thereby optimizing API latency and controlling computational costs while preserving the most critical vulnerability headers for analysis.

E. Multi-Cloud AI Routing Engine and Prompt Engineering

The analytical core of the PwnBoxer web platform is a dynamically configurable, Multi-Cloud Large Language Model (LLM) Routing Engine. This backend service abstracts the vulnerability analysis process, allowing the system to seamlessly pivot between localized and cloud-based AI providers based on environmental variables.

The truncated text is dynamically injected into a programmatic prompt engineered to include a structural file inventory, data previews, and strict behavioral boundaries instructing the LLM to adopt the persona of a cybersecurity analyst. The routing engine formats the HTTP request to match the specific schema of the selected provider. It supports localized, air-gapped execution via Ollama (utilizing Qwen 2.5 7B models), ensuring highly classified scan data never leaves the internal network. Alternatively, the router can interface with cloud APIs

such as Google Gemini, OpenAI, or Anthropic. The engine normalizes parameters across these APIs, enforcing maximum output limits (e.g., 900 tokens) and tuning the thermal parameter (temperature: 0.7) to encourage optimal HTML formatting.

F. Report Sanitization and CSS Injection

Cloud-based AI models frequently append markdown fencing (e.g., ```html) to their outputs, which prevents standard web browsers from rendering the HTML DOM natively. To combat this, the backend features a strict parsing algorithm that aggressively sanitizes the LLM's response, using regular expressions to strip extraneous markdown and whitespace.

The sanitized string is then evaluated by a conditional wrapping function. If the AI provides a fragmented response, the system constructs a complete HTML document shell. Crucially, the system injects a standardized, enterprise-grade Cascading Style Sheets (CSS) template and UTF-8 viewport meta-tags into the <head> of every report. The Express server then enforces strict Content-Type: text/html HTTP headers upon delivery, ensuring that regardless of which AI model generated the content, the final deliverable renders with consistent, professional PwnBoxer branding.

G. Asynchronous Notification Pipeline and System Resilience

To ensure the web platform remains highly responsive under load, the backend heavily utilizes JavaScript's asynchronous event loop. When a scan is uploaded, the Express server immediately responds to the client with a 200 OK status and a generated Report ID, immediately closing the HTTP connection.

The computationally expensive LLM generation and Simple Mail Transfer Protocol (SMTP) email transmission are pushed to the background via setTimeout delegations. If the user requested an email alert, the system utilizes the nodemailer library and an application-specific Google App Password to securely relay a branded HTML email containing the scan metadata and the attached report. If an external AI API fails or times out (configured to a maximum of 180,000 milliseconds), the backend catches the exception, logs the stack trace, and updates the PostgreSQL database status to "failed." This architectural decision ensures that an AI hallucination or a network timeout does not crash the main Node.js process, maintaining high availability for the PwnBoxer platform.

H. Security Operations, Environment Management, and Validation

A critical component of the web platform's architecture is its secure deployment pipeline and environment management strategy. Given the system's reliance on highly privileged credentials-including root database URIs, external Large Language Model API keys, and Simple Mail Transfer Protocol (SMTP) application passwords-hardcoding configurations within the source code posed an unacceptable security risk.To mitigate this, the platform utilizes strict environment variable injection (dotenv). All sensitive parameters are abstracted into localized, git-ignored configuration files. During runtime, the Node.js backend dynamically loads these variables into memory. This architecture ensures that the source code remains sanitized and can be safely version-controlled or distributed without exposing the underlying cryptographic infrastructure. Furthermore, this decoupling allows for rapid swapping of AI models (e.g., changing the LLM_PROVIDER variable from ollama to gemini) without requiring codebase recompilation or server downtime.

System validation testing was conducted to ensure high reliability across all integrated services. The multipart data ingestion pipeline was stress-tested against the 10-megabyte multer memory limits to verify that oversized payloads correctly trigger HTTP 413 Payload Too Large rejections rather than causing buffer overflow vulnerabilities. Additionally, the AI routing engine's error-handling matrices were validated by intentionally providing invalid API keys and simulating network timeouts; the system successfully caught these simulated exceptions, bypassed the email dispatch pipeline, and gracefully updated the PostgreSQL database with diagnostic failure codes, proving the resilience of the asynchronous event loop.

Finally, cross-origin compatibility was validated for the frontend rendering engine. By injecting standardized UTF-8 meta-tags and CSS directly into the HTML payloads, the system ensures that the dynamically generated Blob URLs render with perfect visual parity across major browser engines (Chromium, WebKit, and Gecko), fulfilling the requirement for a universally accessible reporting dashboard.

VI. Software Detail: Task Scheduling and Board Communication

A. Boot and Task Synchronization

Communication between the ESP32-S3 microcontroller and the Raspberry Pi Compute Module 4 is implemented over a UART serial link. Each frame transmitted over this link is structured into discrete fields: a magic sequence that delineates a valid PwnBoxer frame from arbitrary data on the line, a type field that identifies the class of packet being transmitted, such as status, execution, or

informational packets, a length field, a variable-length payload, and a checksum for integrity verification. At system startup, the scheduler running on the CM4 initiates a structured handshake sequence to synchronize the task list with the ESP32. The RPi first queries the local Ethernet interface for network state information, then loads all YAML-defined task configurations from a task directory residing on the RPi. An informational packet is then transmitted to the ESP32, followed by one task data packet per defined task. To conclude the initialization sequence, a task flush packet is sent, which must be acknowledged by the ESP32 via a returned acknowledgement packet before the system transitions to its operational state. Each transmission is structured in the five section packet as detailed in Fig 4.

Magic - 2	Type - 1	Len - 2	DATA - N	CRC - 1

Fig. 4 Packet Framing Diagram

B. Scheduler Architecture

The scheduler executes on the CM4 as a system service, loaded automatically at boot. Its lifecycle consists of three phases. During the initialization phase, the scheduler queries the available network adapters to determine whether connectivity is active and relays that information to the ESP32. It additionally loads task definitions and their associated parameters from YAML configuration files, transmitting each to the ESP32 for acknowledgement. Once initialization is complete, the scheduler transitions into a listening phase, blocking on the UART interface and awaiting task execution packets from the ESP32. Signal handlers for interrupt and termination signals manage graceful shutdown.

C. Task Execution

When the operator selects a task and confirms its parameters through the touch display, an execution packet is transmitted containing the task identifier and a key-value parameter string. The scheduler's task runner correlates the received task identifier with a corresponding YAML entry and substitutes the operator-supplied parameter values into the placeholder tokens defined within the YAML command templates. If any substitution cannot be resolved and no default value exists, the task is aborted prior to execution and an error status is returned to the display. Valid commands are spawned through Linux's fork() system call, executing each command template via a shell process. Task status updates are transmitted back to the ESP32 throughout execution, reporting a started, success, failed, or timeout status code along with the associated process identifier. These updates are consumed by the ESP32 each display frame and reflected in real time on the touch display.

D. Boot and Task Synchronization

When configuration changes are made by the operator through the touch display, the ESP32 constructs and transmits structured packets to relay the updated information to the CM4. Two packet types facilitate this process: a Wi-Fi connect packet and a Wi-Fi update packet. Both carry an SSID and an associated set of credentials. The ESP32 first validates the supplied credentials using its onboard Wi-Fi capabilities, and only upon successful confirmation are the credentials persisted to external flash memory and forwarded to the CM4, where nmcli is used to establish the connection to the specified network. When the connection has been confirmed to be established by the RPI the information is sent back to the ESP32 for notification.

D. ESP32 Networking

The ESP32 networking subsystem is built around a concurrent task architecture with three tasks distributed across two cores. The Wi-Fi and UART tasks run isolated from the display task to better utilize task distribution between the constantly running screen and the periodically running UART and intensive Wi-Fi operations. The Wi-Fi worker task and UART task both execute on Core 0 at lower priorities than that of the GUI task. Core pinning ensures the display remains responsive while networking operations execute in the background.

The Wi-Fi subsystem follows a single-owner worker model in which the GUI task never invokes blocking Wi-Fi operations directly. Instead, the GUI submits commands to the Wi-Fi worker through a command queue, and the worker posts results back through a separate result queue. Command types include scan, connect, and disconnect operations. Result types include scan complete, connect success, connect failed, disconnected, and periodic connection update messages carrying current signal strength levels and IP information. All queue messages are fixed-size structs, meaning no pointers are exchanged between tasks and no shared heap buffers exist between the GUI and Wi-Fi worker.

On a scan operation, the Wi-Fi worker executes a blocking network scan, collects up to 32 results, and posts a scan complete result back to the GUI queue. On a connect operation, the worker issues a disconnect to establish a clean baseline, then begins polling with the supplied SSID and credentials, checking the connection

status every 500 milliseconds for up to 15 seconds before emitting a success or failure result. This blocking connect behavior is intentionally contained within the worker context so that the GUI thread is never stalled during association attempts. A periodic health poll additionally runs while connected, outputting connection update messages with refreshed link state, and a disconnected result if the link drops unexpectedly.

VII. Software Detail: User Interface

A. Embedded Hardware Usage

The user interface was designed around the usage of a few key hardware choices: the ESP32-S3, the TFT LCD capacitive touch display, and the external flash memory module. The interface had to feel responsive and visually clean on a microcontroller platform while sharing resources with networking and serial communication. The ESP32-S3 was a strong fit because it combines adequate CPU performance, integrated wireless support, and a memory model that can be extended with PSRAM. In this system, PSRAM is especially important because the display pipeline and task buffers can consume significant memory. Rather than forcing everything into internal DRAM, the UI allocates large buffers in PSRAM so rendering and communications can coexist without instability.

The Crystalfontz TFT LCD display is driven over SPI, which keeps the wiring manageable while still delivering acceptable refresh performance for a modern-looking interface. Touch input is driven over I2C, and is calibrated and transformed in software so coordinates remain accurate across rotations and panel tolerances. This matters during runtime because the interface includes multiple interactive elements where poor touch mapping would immediately degrade usability.

An external SPI flash memory module complements the display and controller by providing persistent storage beyond volatile RAM. In this implementation, external flash is used not only for the startup image, but also for persistent user data like email and ethernet information. The design is intentionally robust, as the UI can attempt to load image data from external flash and gracefully fall back to a bundled image if needed.

B. IDE and Debugging Software

Development was centered in Visual Studio Code with PlatformIO as the build and environment manager. This combination is useful for embedded UI work because it keeps firmware configuration, dependency management, uploading, and monitoring inside one repeatable workflow. PlatformIO makes use of a very simple user interface that allows seamless building and uploading of code to the microcontroller at any time. Additionally, PlatformIO supports serial monitoring and exception decoding, both of which are essential during UI bring-up.

The user interface uses a mixed framework approach, taking the best of both ESP-IDF and Arduino. Arduino APIs simplify display and Wi-Fi integration, while ESP-IDF components provide lower-level control for drivers, FreeRTOS scheduling, and peripherals.

C. Library Selection

The central UI library is LVGL (v9.x). LVGL provides the widget system, event model, styling, layout primitives, and render scheduling used throughout the interface. It was a strong choice because it supports production-grade embedded GUIs without requiring custom rendering logic for every component. The code leverages LVGL objects for navigation bars, page containers, labels, buttons, lists, text areas, switches, and on-screen keyboards, which keeps UI behavior consistent and extensible. Build-time LVGL feature flags are tuned to reduce firmware size and memory pressure by disabling unused widgets and subsystems.

For display and touch hardware abstraction, the project uses LovyanGFX. This library provides high-performance panel/bus drivers and a clean configuration model for SPI displays and touch controllers. In this UI stack, LovyanGFX handles panel initialization, SPI transactions, backlight integration (where enabled), and touch acquisition. LVGL then renders through a flush callback into the display driver.

D. Building the UI

The UI creation process followed a staged, task-oriented approach rather than trying to render everything at once. First, the display and LVGL runtime are initialized, touch calibration is applied, and dual draw buffers are allocated in PSRAM. These buffers support partial rendering, which is a practical optimization for embedded displays because only changed regions are pushed to the panel.

After initialization, the firmware shows a loading screen. This is not just cosmetic: it gives the Compute Module 4 time to finish startup tasks while keeping user feedback immediate. The loading view also demonstrates a resilient asset strategy by preferring external-flash image data and falling back to bundled image data when necessary.

The root UI is then built with a navigation bar and multiple logical pages. A horizontal page container supports swipe and tab-style navigation between primary sections (for example, Tasks and Settings), while modal

overlays handle context-specific workflows such as network settings and email entry.

Styling is handled through a theme system with dark and light palettes. Rather than hardcoding colors at each widget call site, the implementation centralizes theme values and reapplies styles as needed.

Interactivity is event-driven. Navigation buttons, Wi-Fi controls, text inputs, list selections, and keyboard actions all use callback-based handlers. Importantly, networking operations are non-blocking from the GUI perspective; UI callbacks enqueue commands, and a separate Wi-Fi task performs scan/connect operations. Results are then pushed back through a result queue and drained by the GUI loop each frame, where labels, lists, and status icons are updated.

Persistent data ties the interface to user workflow. Email input is captured via modal and keyboard and is written to external flash, and ethernet metadata received over UART can also be stored and displayed. This turns the UI from a static demo into an operational front end with retained state across boots.

Acknowledgement

The authors wish to acknowledge the assistance and support of our review committee and advisors for Senior Design.

References

[1] Elbaz, G. (2025). Code execution vulnerability: Impact, causes, and 8 defensive measures. Oligo Security. Retrieved September 3, 2025, from https://www.oligo.security/academy/code-execution-vulnerability-impact-causes-and-8-defensive-measures

[2] IBM X-Force. (2025, April). IBM X-Force 2025 threat intelligence index. IBM. Retrieved September 3, 2025, from https://www.ibm.com/thought-leadership/institute-business-value/en-us/report/2025-threat-intelligence-index

[3] Vanhoef, M., & Piessens, F. (2017). *Key reinstallation attacks: Forcing nonce reuse in WPA2*. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (pp. 1313–1328). ACM. https://doi.org/10.1145/3133956.3134027