AI - DRIVEN PENTESTER

- BY DEEPTHINK

ASSIGNMENT COVER SHEET

UNIVERSITY OF WOLLONGONG IN DUBAI

Assignment Cover Sheet

Subject Code:	CSIT321
Subject Name:	Project
Submission Type:	Design Report
Assignment Title:	Capstone Project
Student Name:	Abbas Bhaiji, Ansaf Sabu, Derick Reni, Mohamed Rizvan, Mohammed Uddin
Student Number:	6345438, 7390440, 6728492, 7797965, 7842430
Student Email:	ab836@uowmail.edu.au, as3623@uowmail.edu.au, djr980@uowmail.edu.au, memrr999@uowmail.edu.au, msumk596@uowmail.edu.au
Lecturer Name:	Dr. Patrick Mukala
Due Date:	12/05/2025
Date Submitted:	12/05/2025

PLAGIARISM:
The penalty for deliberate plagiarism is FAILURE in the subject. Plagiarism is cheating by using the written ideas or submitted work of someone else. UOWD has a strong policy against plagiarism.
The University of Wollongong in Dubai also endorses a policy of non-discriminatory language practice and presentation.

PLEASE NOTE: STUDENTS MUST RETAIN A COPY OF ANY WORK SUBMITTED

DECLARATION:
I/We certify that this is entirely my/our own work, except where I/we have given fully documented references to the work of others, and that the material contained in this document has not previously been submitted for assessment in any formal course of study. I/we understand the definition and consequences of plagiarism.

Signature of Student:

Optional Marks:

Comments:

Lecturer Assignment Receipt (To be filled in by student and retained by Lecturer upon return of assignment)

Subject: **Assignment Title:**
Student Name: **Student Number:**
Due Date: **Date Submitted:**
Signature of Student:

Student Assignment Receipt (To be filled in and retained by Student upon submission of assignment)

Subject: **Assignment Title:**
Student Name: **Student Number:**
Due Date: **Date Submitted:**
Signature of Lecturer

Tables Of Contents

Introduction

ArmourEye is a semi-automated penetration testing platform that integrates AI-driven decision-making with traditional scanning and exploitation tools. This system is designed to provide users with a powerful and intuitive dashboard where they can upload Docker images, initiate scans, and review detailed reports about vulnerabilities discovered during a test session.

This High-Level Design Document outlines the system's structure, flow, and components. It also serves as a guide for future development, helping identify contradictions and establish consistency across modules before implementation.

Requirement Summary

The following section outlines the primary functional and non-functional requirements for the ArmourEye system. Each requirement has been described with context and relevance to the system's goal of delivering intelligent, AI-assisted penetration testing in a containerized environment.

Functional Requirement

1. Docker Image Upload and Management

Users must be able to upload Docker images through a web interface. These images represent isolated target environments that will undergo penetration testing. The system needs to validate, store, and prepare these images for deployment within the Docker Lab Manager, ensuring proper networking and isolation.

2. Automated Network Scanning

The platform must perform reconnaissance on the deployed containers using tools like Nmap and Masscan. This includes identifying open ports, services, and technologies in use. The system is expected to automate the scan lifecycle, parse results, and pass the data to downstream modules.

3. AI-Driven Orchestration

A core functional requirement is the integration of an AI orchestration module. This module should analyze scan results, determine potential vulnerabilities, and decide which tools or techniques to execute next. The AI should leverage a language model (OpenAI) to reason about findings in a contextual manner.

4. Exploitation Engine

Following AI decision-making, the system must be capable of executing exploitation tools such as Metasploit. This involves forming correct command syntax, executing them in a controlled environment, and capturing output reliably for logging and reporting.

5. Comprehensive Reporting

After a session, ArmourEye must generate a detailed report containing all scan data, AI decisions, exploit attempts, and resulting vulnerabilities. Reports should be downloadable in PDF or JSON format and include explanations of actions taken.

6. Persistent Data Storage

All test session data must be persistently stored, including logs, tool output, AI prompts/responses, and metadata. Initially, this is handled with SQLite, with a planned migration to PostgreSQL for more robust production use.

Non-Functional Requirements

1. Responsive and Intuitive User Interface

The platform should offer a modern and responsive user interface built with Bolt UI components. Users should be able to navigate the system with minimal training, accessing complex functionality like scan configuration and report generation with ease.

2. Modular Architecture

The backend system must be structured in a modular way, currently as a monolithic Flask application with clearly separated Python modules. This organization allows for manageable codebases and supports future migration to a microservices architecture for improved scalability.

3. Performance and Scalability

The system must complete scans and reports efficiently. As workloads grow, performance must remain acceptable. Although it is built as a monolith initially, individual modules (e.g., scanning, exploitation) should be designed to scale independently once converted into microservices.

4. **Security and Isolation**

   Security is a top priority. Containers should run in isolated networks with limited permissions to prevent host compromise. Data must be stored securely, API endpoints protected, and command injection vulnerabilities prevented.

5. **Portability and Ease of Deployment**

   ArmourEye should run entirely in a containerized environment. Developers and users should be able to launch the platform using a single “docker-compose up” command, making it portable across different environments without complex setup.

6. **Extensibility**

   The system should support easy integration of new tools or features in the future. This includes adding new scanners, AI models, or report formats without major rewrites. Well documented APIs and module boundaries are crucial for this.

Assumptions and Prerequisites

To build a reliable and testable system, several key assumptions and prerequisites are considered. These clarify the foundation upon which the design is constructed and help align development efforts with real-world constraints.

Assumptions

- **User Expertise**: The system assumes that users (e.g., pentesters or security analysts) have a basic understanding of Docker, container security, and penetration testing practices.
- **Image Validity**: It is assumed that the Docker images uploaded by users are functional, properly built, and not malicious toward the host system.
- **Tool Availability**: All scanning and exploitation tools (e.g., Nmap, Hydra, Metasploit, SQLMap) are available in the containerized environment and are compatible with the system’s command execution model.
- **Internet Access**: For AI models that run in the cloud (e.g., OpenAI), internet access is assumed unless local LLMs are used.

Prerequisites

- **Docker Environment**: A fully functional Docker installation must be present on the host machine to run test containers and the ArmourEye service itself.
- **System Resources**: Minimum resource requirements must be met (e.g., 8GB RAM, 6 cores) to ensure containers run smoothly and scans complete in reasonable time.
- **Python and Flask Dependencies**: The application backend assumes a Python 3.x environment with Flask and related packages installed (if run outside Docker).
- **Database Initialization**: The initial database (SQLite or PostgreSQL) must be initialized with the required schema and permissions before use.

High-Level Design

The ArmourEye system is organized into multiple cooperating modules that interact in a sequence to perform container-based pentesting. This section outlines the overall interaction patterns, flow of data, and responsibilities across the system.

Process Flow Overview

The typical session flow proceeds as follows:

1. The user uploads a Docker image through the web dashboard.
2. The **Docker Lab Manager** initializes a container from the image in a secure, isolated network.
3. The **Scan Module** begins reconnaissance, collecting open ports and service information.
4. Data is passed to the **AI Orchestrator**, which selects appropriate exploit strategies.
5. The **Exploitation Module** executes selected tools and gathers responses.
6. All interactions and decisions are logged in the **State Store**.
7. A detailed report is compiled and presented to the user for download.

Integration and Pathways

Modules interact via internal API calls or message-passing where appropriate. Although running as a monolith, each logical module is self-contained, making eventual migration to microservices straightforward. A dedicated log and storage service ensures traceability and state persistence across the workflow.

Diagram References

- A **Sequence Diagram** is used to visualize the pentest workflow (user to report).

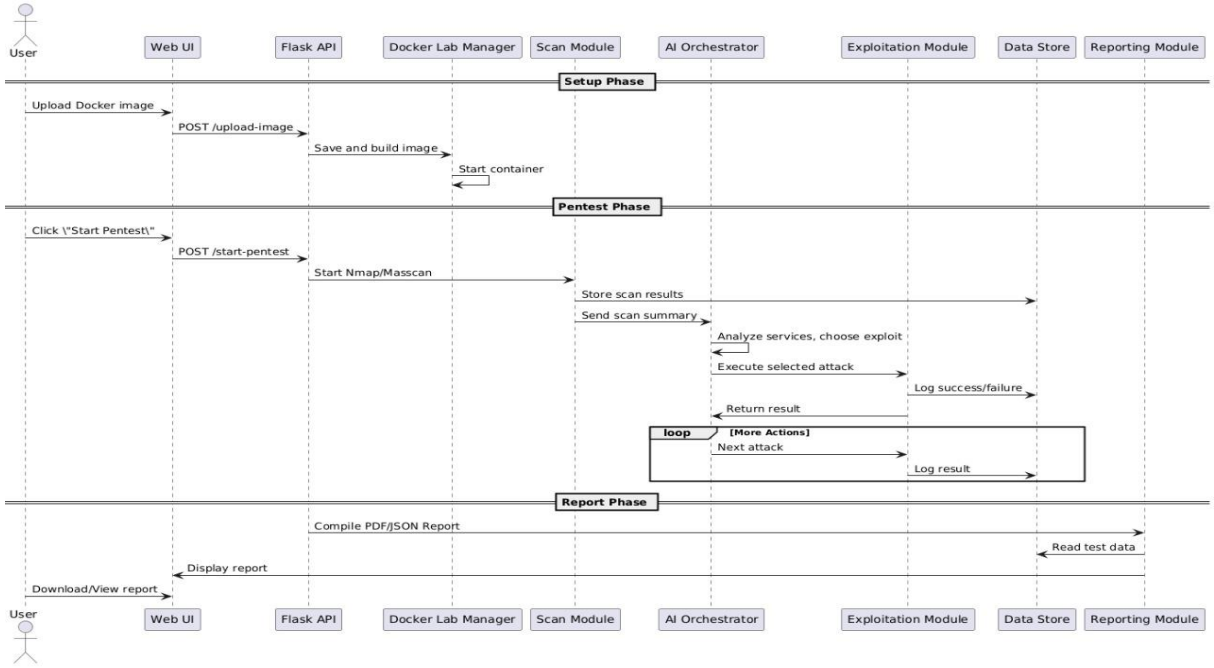

Figure 1

- A **Class Diagram** captures relationships between core data structures.

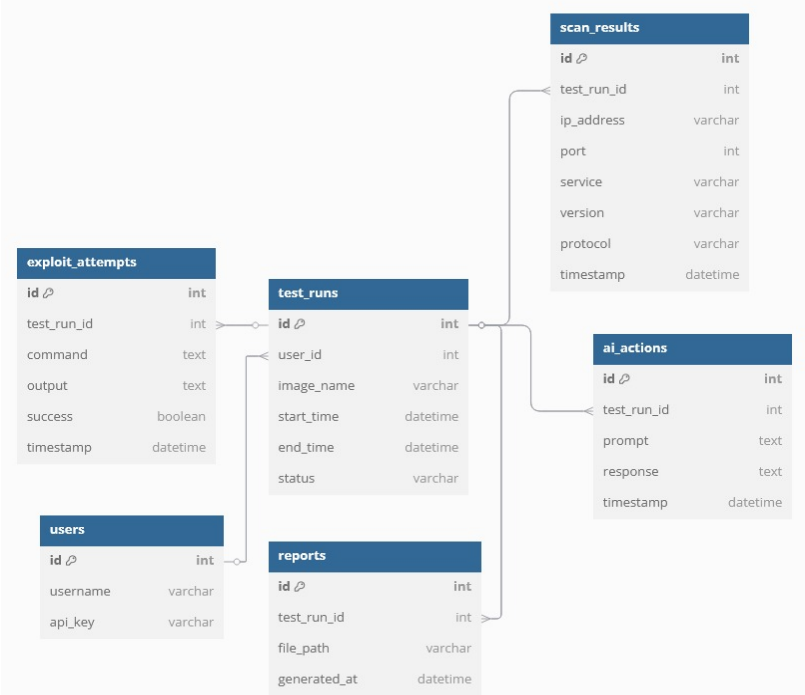

Figure 2

- A **Component Diagram** explains system packaging and interfaces.

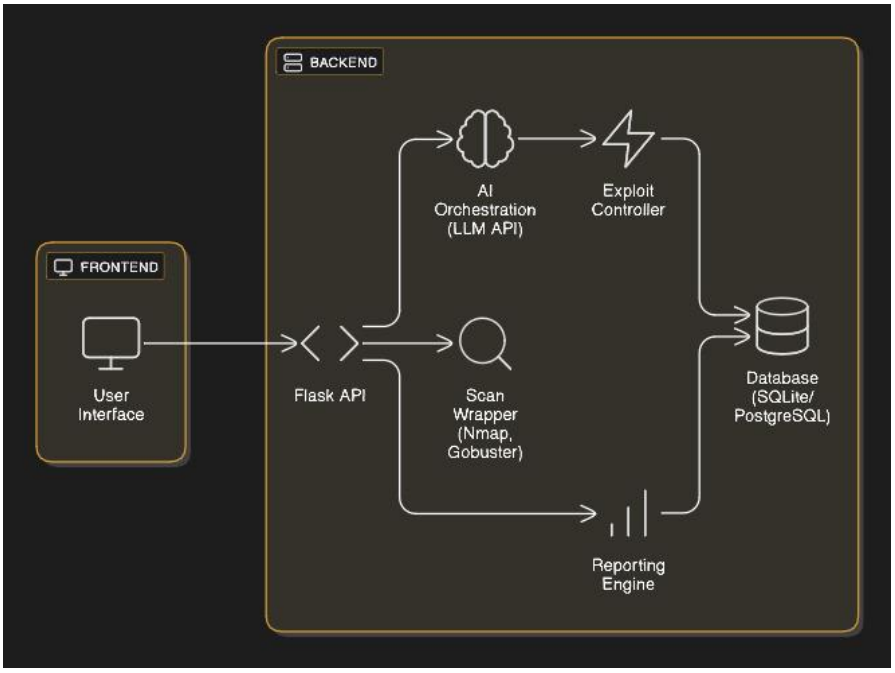

Figure 3

- The **Architecture Diagram** shows the 3-tier structure and container layout.

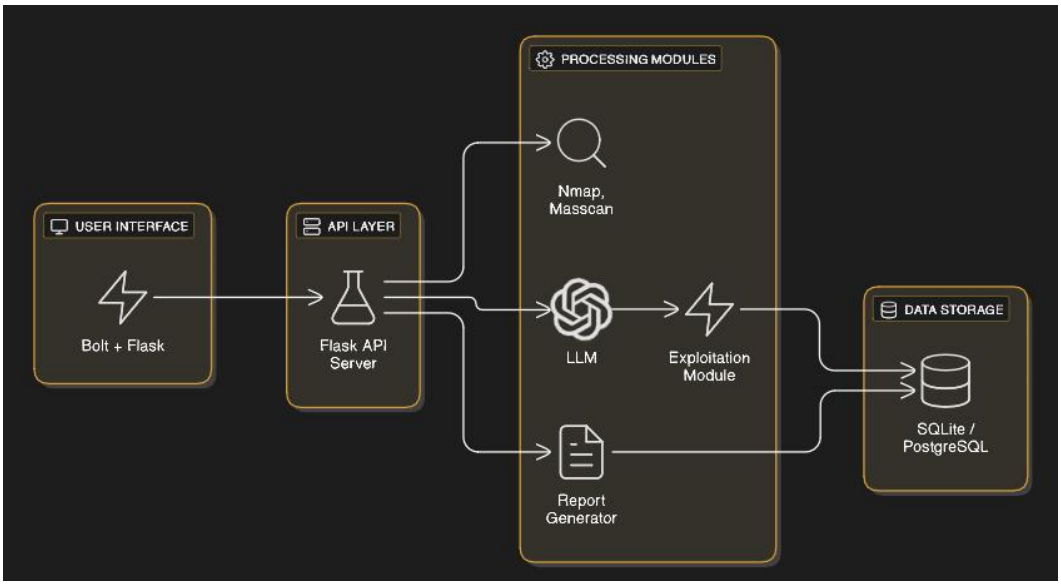

Figure 4

Architectural Plan

The **ArmourEye system architecture** is designed to support an intelligent, semi-automated penetration testing workflow, integrating traditional scanning tools with modern AI-driven decision-making. The architecture is modular, scalable, and built to evolve from a monolithic structure into a distributed microservices ecosystem as the project matures.

Major Components of the IT Solution

At its core, the system is composed of the following major components:

1. ### Frontend Web UI (Bolt + Flask) (Boundary Layer)

   Allows users to upload Docker images, configure scans, monitor progress, and view reports. It serves as the **boundary layer** for user interactions.

2. ### Flask Application Server (Backend Logic) (Control Layer)

   Hosts multiple modules that manage the core logic such as Docker Lab Manager, Scan Module, AI Orchestrator, Exploit Dispatcher, Report generator (This server orchestrates actions across the system based on user requests and internal triggers.)

3. **Containerized Target Environment (Docker Host)**

   Dynamically handles the setup and teardown of Docker containers used for testing. These containers simulate real-world attack targets.

4. **AI Decision Engine (LLM API or Local Model)**

   Communicates with OpenAI or local LLMs to process scan results and recommend or automate next actions.

5. **Tool Wrappers & Executors (Control Layer)**

   CLI tool integrations for running scanners (e.g., Nmap, Masscan), brute-force tools (Hydra), exploitation tools (Metasploit), and others.

6. Data Store (SQLite → PostgreSQL) (Entity Layer)

   A persistent storage system responsible for storing test sessions, AI interactions, scan results, and generated reports.

7. **External APIs and Services (Boundary Layer)**

   APIs like OpenAI (hosted via inference API or containerized) are accessed securely to provide AI capabilities.

How Components Fit in the IT Ecosystem

ArmourEye sits at the intersection of offensive security tooling and AI orchestration. In the broader IT ecosystem, it is designed to:

- Integrate with developer CI/CD pipelines through future API endpoints.
- Be deployed on local testing labs, student training environments, or enterprise-grade pen-testing infrastructure.
- Interface with AI inference APIs or on-premise models based on availability.
- Fit into standard DevSecOps workflows through automation and reporting.

The system, when deployed, can be either standalone or part of a larger security toolkit used by red team operators, security students, or automation platforms.

Integration & Communication Between Components

Frontend to Backend Communication

- All UI interactions (image upload, scan trigger, view report) go through REST endpoints exposed by the Flask server.

Backend Internal Module Interaction

- Python modules are structured cleanly within the monolith. Each module communicates via function calls or internal message passing.
- Example: The `ScanModule` calls `AIOrchestrator` after completing a scan, and `AIOrchestrator` then invokes the `ExploitModule`.

External Tools Invocation

- All scanning/exploitation tools are invoked as subprocesses by Python wrappers.
- Tool outputs are parsed and stored in the database for reuse.

LLM Integration

- AI interactions are performed via HTTPS requests to the OpenAI API or an internal LLM service hosted via FastAPI or Flask.

Database Access

- A central database is accessed via SQLAlchemy ORM, allowing easy migration from SQLite to PostgreSQL.

Monolith vs. Microservices

Currently implemented as a **modular monolith**, the system is structured to enable a future migration to **microservices**, such as:

- `scan-service`: Handles all scanning-related logic.
- `ai-engine`: Isolated orchestration unit that queries LLMs.
- `report-generator`: Generates final report files asynchronously.
- `exploit-runner`: Executes exploit tools and returns results.

Each of these can be deployed independently once communication protocols (REST/gRPC or message queue) are formalized.

Transactions and User Flows

The Transactions and User Flows in ArmourEye detail the critical pathways that a user follows to perform a penetration test from uploading an image to receiving an AI-driven vulnerability report. These workflows capture the system's end-to-end behavior, highlighting interactions between the user, frontend UI, backend modules, and underlying infrastructure.

Each interaction represents a business process that simulates a real-world penetration testing session, abstracted into logical flows. These can be translated into Swim Lane Diagrams, with lanes representing key actors/modules such as: User. Web UI, API Server, Docker Host, Scan Module, AI Orchestrator, Exploit Module, and Report Engine.

Flow 1: Launching a Penetration Test (User-Initiated Test Flow)

Actors: User, Web UI, Flask API, Docker Lab Manager, Scan Module, AI Orchestrator, Exploit Module, Data Store, Report Module

Flow:

1. User logs in to the system and accesses the dashboard.
2. Through the Web UI, the user uploads a Docker image of the target system.
3. The Flask API receives the image and forwards it to the Docker Lab Manager.
4. Docker Lab Manager pulls/builds and initializes the container in an isolated test network.
5. Once live, the Scan Module begins reconnaissance (Nmap, Gobuster, etc.).
6. Scan results are stored in the Data Store and forwarded to the AI Orchestrator.
7. The AI Orchestrator queries the LLM (OpenAI or local model) for analysis and action planning.
8. Based on AI output, the Exploit Module selects and executes tools (SQLMap, Metasploit).
9. All actions and outputs are logged in the Data Store.
10. The Report Module compiles a summary report and makes it available via the UI.

Flow 2: Viewing Results and Reports

Actors: User, Web UI, Flask API, Data Store, Report Module

Flow:

1. The user navigates to the **Test History** or **Report Viewer** page.
2. The **UI requests available sessions** via API.
3. The **API Server** queries the **Data Store** for past test runs and reports.
4. A list of reports is returned to the UI.
5. The user selects a report to view; the system loads its metadata and embedded results.
6. If the report is downloadable (PDF/JSON), it is streamed to the user via a secure endpoint.

User Interface Design

The User Interface (UI) of the ArmourEye system is the primary medium through which users interact with the automated penetration testing workflow. Built using a

combination of Bolt UI components and a Flask backend, the interface is designed to be clean, responsive, and task-focused, prioritizing ease of use for both novice users and experienced cybersecurity professionals.

The UI is structured around a dashboard model, offering clear navigation between stages of the pentesting lifecycle. Each key interaction is organized into modular views, promoting a logical flow from image setup to final report retrieval.

Key Interface Screens

1. Login and API Key Setup

- The entry point of the application allows users to authenticate or configure an API key.
- Security focused design ensures sensitive input is masked.
- A validation system ensures the entered key meets system requirements.

2. Docker Image Upload and Selection

- Users are presented with an image selection panel.
- Options include uploading a new Docker image or selecting from previously scanned ones.
- Progress bars and logs provide real-time feedback on image pull and build processes.

3. Scan Configuration Screen

- Allows users to define scan parameters, such as network range, service intensity, and scanning tools (e.g., Nmap, Gobuster).
- Optional modules (e.g., AI-assist or manual override) can be toggled here.
- Visual indicators confirm scan readiness.

4. Live Pentest Dashboard

- Displays real-time logs of each stage: scanning, AI analysis, exploitation, and result generation.
- Output from each module (Scan, AI, Exploit) is organized into collapsible sections.
- Users can view tool outputs like open ports, CVEs, or command logs.

5. Report Viewer

- Interactive view of the generated report (JSON or PDF format).
- Embedded viewer allows browsing through vulnerabilities, AI decisions, and exploit attempts.

- Option to download or archive the report.

6. Logs and Session History

- Shows past pentest runs with timestamps, status (e.g., completed, failed), and summary outcomes.
- Each item links to a detailed view for deeper inspection or replay.

7. Settings and API Control

- Configuration panel for system settings, external integrations (e.g., Metasploit path), and AI model selection.
- Secure fields for API tokens and authentication modules.

UI Design Principles Followed

- **Minimalist Interface:** Focused layouts reduce cognitive overload, ensuring users don't get lost navigating deep configurations.
- **Feedback Loops:** Users receive constant system feedback through loading states, progress bars, and log previews.
- **Security by Design:** All sensitive data (API keys, reports) is masked or encrypted, even at the interface level.
- **Error Prevention:** Input fields use validators and dropdowns to avoid invalid configurations.

Deployment and Environment

The deployment strategy for **ArmourEye** is structured to support flexible development, reliable testing, and scalable production use. This section outlines the system's deployment architecture, infrastructure requirements, configuration options, and environmental dependencies. It is designed to ensure smooth transitions across development, staging, and production environments.

Deployment Architecture Overview

ArmourEye adopts a **containerized architecture** based on Docker, where each major component runs in an isolated environment. The architecture supports modular monolith design in the current phase, with a future vision of migrating to microservices.

Key components deployed as containers include:

- Flask backend application

- Frontend UI (served via Flask or static Nginx)
- Vulnerability scanning tools (e.g., Nmap, Gobuster)
- AI orchestration logic
- Reporting service
- SQLite (dev) or PostgreSQL (prod) for persistent storage

The system is orchestrated using **Docker Compose**, ensuring all services are properly networked and launched in the correct order.

Environment Configurations

Development Environment

- **Architecture**: All modules run as part of a modular monolith using Flask and Docker Compose.
- **Database**: SQLite (lightweight, file-based).
- **AI Integration**: Uses local API keys to communicate with OpenAI or CodeLlama endpoints.
- **Access**: Deployed on localhost with Flask UI accessible via browser.
- **Tools**: Includes CLI-based scanners (Nmap, Hydra, SQLMap) pre-installed in base images.

Production Environment (Future Consideration)

- **Architecture**: Modular services potentially deployed separately (Scan, AI, Exploits, Reporting).
- **Database**: PostgreSQL container replaces SQLite for better concurrency and durability.
- **Web Server**: Reverse proxy (Nginx) added for HTTPS, routing, and load handling.
- **Security**: Environment variables and `.env` files for API keys, database credentials, and secret tokens.
- **Deployment Option**: Docker Compose initially; migration to Helm/Kubernetes or Docker Swarm in later stages.
- **External Integrations**:
    - AI LLM (OpenAI/CodeLlama) API
    - External repositories for Docker image pulling

Infrastructure Requirements

Resource	Development	Production (Target)
CPU	4 cores	4+ cores (especially with AI tasks)
RAM	16 GB	16 GB or higher
Disk	256 GB free space	500+ GB (reports, logs, images)
Operating System	Linux/macOS/Windows with Docker	Ubuntu Server / Cloud VM
Database	SQLite	PostgreSQL

Dependencies and Configuration

Software Dependencies

- Docker Engine & Docker Compose
- Python 3.10+ inside containers
- AI access keys for OpenAI or LLaMA
- Flask and required Python libraries

System Configuration

- All environment-specific variables (e.g., DB connection strings, LLM API keys) stored securely in .env files
- Volume mounts for persistent logs, database files, and reports

Deployment Considerations

- **Portability**: The Docker-based setup ensures consistent behavior across different environments and minimizes “it works on my machine” issues.
- **Security**: Secrets are not hardcoded; environment variables and .env files manage sensitive data.
- **Scalability**: Architecture supports both vertical scaling (more power) and horizontal scaling (service separation).
- **Monitoring and Maintenance**: Logs can be centralized, and Docker health checks can be configured to auto-restart failed services.

Security Considerations

Security is a critical aspect of the ArmourEye system due to its nature as a penetration testing automation tool. The system handles sensitive data such as vulnerability reports, AI-generated attack plans, command outputs, and potentially exploitable services. Therefore, high-level security principles must be embedded throughout the architecture, development, and deployment lifecycle.

This section outlines the key authentication, authorization, data protection, and risk mitigation mechanisms currently implemented and planned for production deployment.

Authentication & Authorization

Authentication

- **API Key-based Authentication:** Users access the system through a unique API key associated with their session. This prevents unauthorized users from uploading containers or initiating scans.

Authorization

- **Role-Based Access Control (RBAC)**: While currently single-user focused, RBAC will allow differentiated access levels in future versions. e.g., separating admin, analyst, and auditor roles.

Data Protection

At Rest

Database Security

In development, SQLite is used. For production, PostgreSQL will implement **Encrypted storage volumes** and **Database-level access controls**

Logs and Reports

Reports, tool outputs, and AI logs will be stored in structured directories with permission restrictions. Sensitive logs can be encrypted if required.

In Transit

HTTPS Everywhere

When deployed in production, all communication with the dashboard and API will be routed through HTTPS using Nginx or Traefik with TLS termination.

Secure API Communication

API requests to external services (e.g., OpenAI, CodeLlama) will use secure tokens and HTTPS, with tokens managed securely via environment variables.

Tool Execution Security

Containerisation

All scanning and exploitation tools are run inside Docker containers, with network isolation and read-only file systems to prevent host compromise.

Resource Limits

Docker resource quotas (CPU, memory, timeout) prevent denial-of-service conditions due to runaway tools or infinite loops.

Escaping Protection

Containers are launched with **"--cap-drop=ALL"** and without privileged flags to prevent escape vulnerabilities.

Vulnerabilities & Risks

Risk	Description	Mitigation
Container Escape	Tools like Metasploit may attempt to exploit targets aggressively	Strict container runtime config, no privileged containers
API Key Leakage	Hardcoded or exposed keys during development	Use .env files, never commit keys, rotate regularly
LLM Misuse or Injection	Prompt injection or unexpected LLM actions	Prompt sanitization, monitor AI output before execution
Unauthorized Access to Reports	Reports may contain sensitive test results	Limit file access, use signed URLs or tokens for download

Security Best Practices

- Follow **OWASP Top 10** recommendations in UI and API design
- Minimize attack surface by exposing only required ports and routes
- Regularly scan the system itself using tools like Clair or Trivy
- Use immutable infrastructure containers should be redeployed, not modified live
- Encrypt backups of scan logs and AI transcripts

In conclusion, ArmourEye embeds defense-in-depth across its lifecycle from containerized tool execution to secure user interaction. While currently in the development phase, future production deployment will harden all components to meet modern cybersecurity standards.

Risks and Mitigation Strategies

Despite its modular and containerized design, the ArmourEye system presents several risks due to the complexity of automated penetration testing, integration with external AI systems, and the execution of potentially dangerous tooling. This section outlines the key technical, architectural, and operational risks identified during design, along with their corresponding mitigation strategies.

System Design & Architectural Risks

Risk	Description	Mitigation Strategy
Tight coupling in modular monolith	The current design uses a modular monolith, which may lead to tight interdependencies between modules, making scaling and testing more difficult.	Adopt strict interface contracts between modules; plan early refactor toward microservices with containerized services and RESTful APIs.
Inflexible scalability	Single Flask app may struggle with concurrent users or large-scale test execution.	Containerize individual services (AI, Scanning, Reporting); deploy behind a load balancer or service mesh in production.

Data consistency issues	Simultaneous scan or AI actions may cause race conditions or inconsistent DB writes.	Implement transactional DB operations and mutex logic for shared data access; migrate to PostgreSQL with ACID guarantees.

Security & Execution Risks

Risk	Description	Mitigation Strategy
Tool misuse or unintended actions	LLM-generated actions may include unintended or harmful commands if prompt injection or AI misinterpretation occurs.	Use prompt sanitization; apply filters or human-in-the-loop review for critical exploit stages; log all actions for traceability.
Container breakout or host impact	Penetration tools run in Docker may try to exploit host or break isolation.	Run containers with **--cap-drop=ALL**, no privileged mode, use seccomp/apparmor profiles; monitor container boundaries.
External API abuse	AI orchestration using LLM APIs could be abused if API keys are leaked or rate limits exceeded.	Store keys in **.env** files; restrict scope, rotate keys periodically; enforce rate limiting at proxy/gateway level.

Data & Reporting Risks

Risk	Description	Mitigation Strategy
Loss of scan session data	Crashes or container shutdowns could lead to loss of valuable session history, logs, or AI decisions.	Implement auto-save checkpoints and periodic logging; persist data to mounted volumes or external DBs.

Unsecured report files	Generated reports may contain sensitive vulnerability data and exploit history.	Restrict access to report paths; use authenticated access or signed download tokens for report endpoints.

User Interaction & Usability Risks

Risk	Description	Mitigation Strategy
User misconfiguration	Incorrect image uploads, wrong scan parameters, or targeting invalid containers can break workflows.	Add input validation, helper tooltips, scan templates, and pre-flight checks on uploaded images.
Lack of audit trail	Without traceability, it may be unclear which user performed which action during a scan session.	Enable per-user logging with timestamps; include session metadata and command history in reports.

Operational & Deployment Risks

Risk	Description	Mitigation Strategy
Deployment complexity as features grow	As the system evolves toward microservices, deployment may become more error-prone or inconsistent.	Use Docker Compose in development, Helm for Kubernetes in production; automate environment config and container builds.
Environment mismatch	Scans or exploits may behave differently on local vs production deployments.	Standardized test environments via containerized testbeds; run regression tests on all supported stacks.

In summary, while ArmourEye is designed with modularity and security in mind, its scope and the sensitive nature of penetration testing introduce risks that must be anticipated and actively mitigated. As development progresses, these strategies should be re-evaluated and extended to ensure operational reliability, ethical tool usage, and secure data handling.

Out-of-Scope

While the ArmourEye system aims to deliver an intelligent, semi-automated penetration testing platform using AI-enhanced decision-making and real-world tools, there are several areas that are intentionally excluded from the scope of the current

implementation. This section ensures a clear understanding between stakeholders of what functionalities and features will not be addressed in this version of the project.

Non-Goals of the Current Project

- **Comprehensive Web Application Firewall (WAF) Evasion Tactics:** Although ArmourEye may attempt known exploits, it will not be tailored to bypass advanced WAFs or intrusion detection systems with obfuscation techniques.
- **Live System Exploitation of Production Environments:** The platform is intended for lab or isolated testing environments only. It will not target live production systems or perform real-world attacks without sandboxing.
- **Custom Exploit Development:** The system uses publicly available tools and databases (e.g., Metasploit, SQLMap, CVEs). It does not create or generate novel exploits beyond selecting known ones.
- **Comprehensive Post-Exploitation Tooling:** While some post-exploitation checks (e.g., file reading or privilege escalation probes) may be initiated, in-depth persistence, lateral movement, or data exfiltration workflows are out of scope.
- **Real-Time Collaboration or Multi-User Sessions:** The current platform is designed for single-user sessions and does not yet support simultaneous collaboration or shared access features.
- **Advanced AI Explainability or Auditability:** Although the system logs AI prompts and responses, it does not provide detailed explainability, confidence scores, or interpretability layers for AI-generated decisions.
- **In-Browser Terminal Emulation or Remote Shells:** The system does not provide shell access to exploited systems or allow terminal interaction from the web UI due to safety and ethical boundaries.
- **Deep Integration with CI/CD Pipelines or DevSecOps Workflows:** CI/CD hooks or integrations with enterprise-level SDLC platforms are not within the initial implementation, though they may be considered for future versions.

Appendices

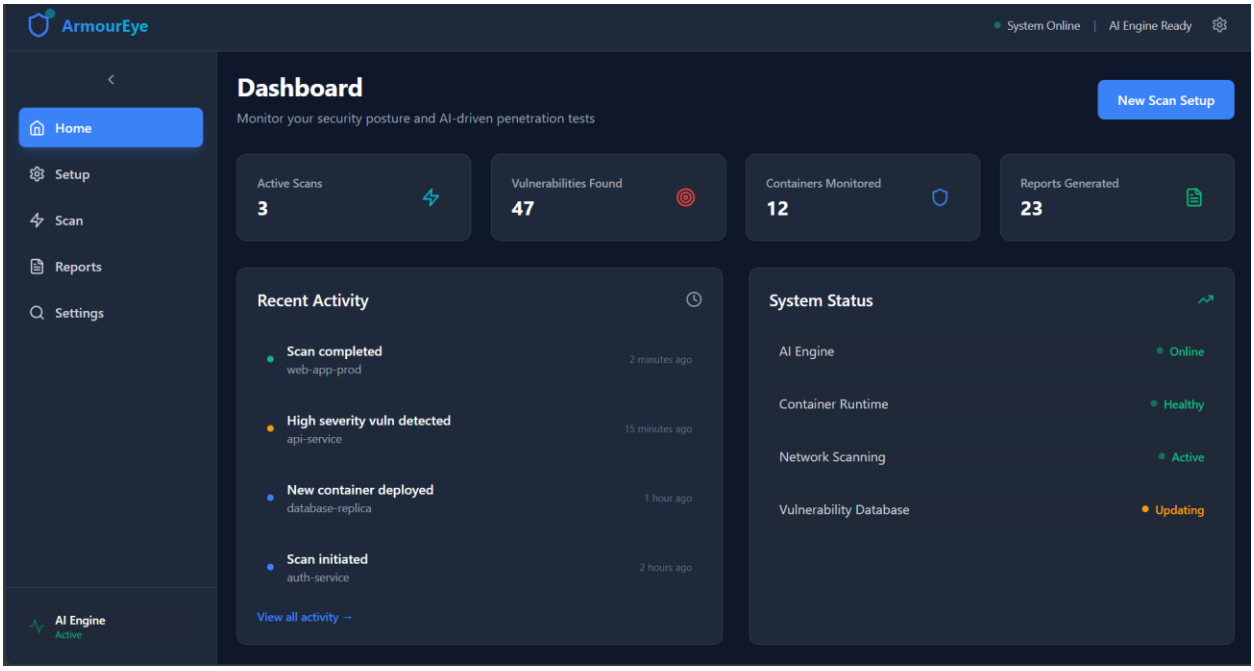

Figure 5 Dashboard

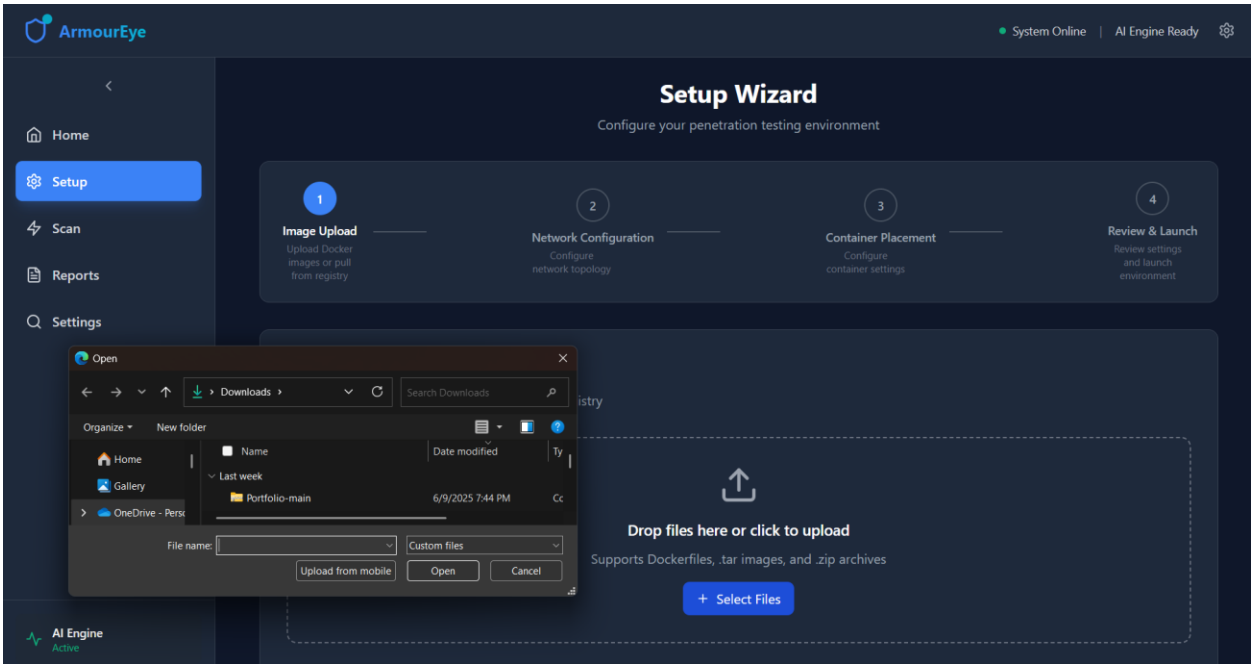

Figure 6 Upload Docker image and begin containerisation

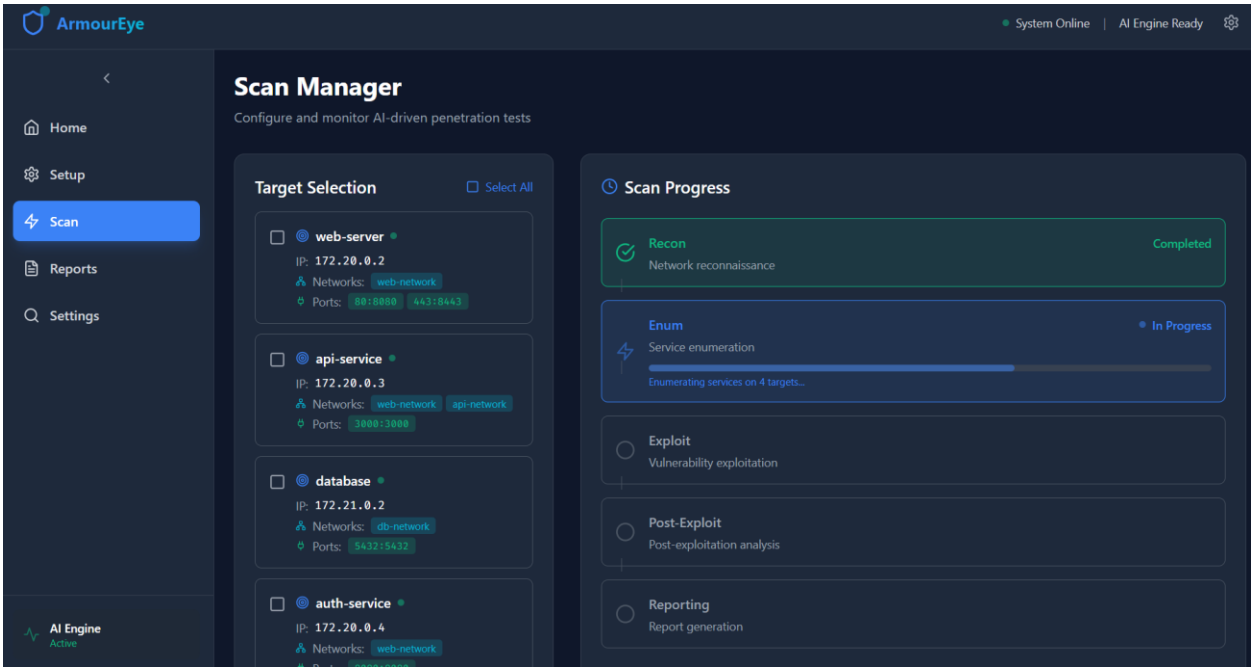

Figure 7 Scan and exploitation module

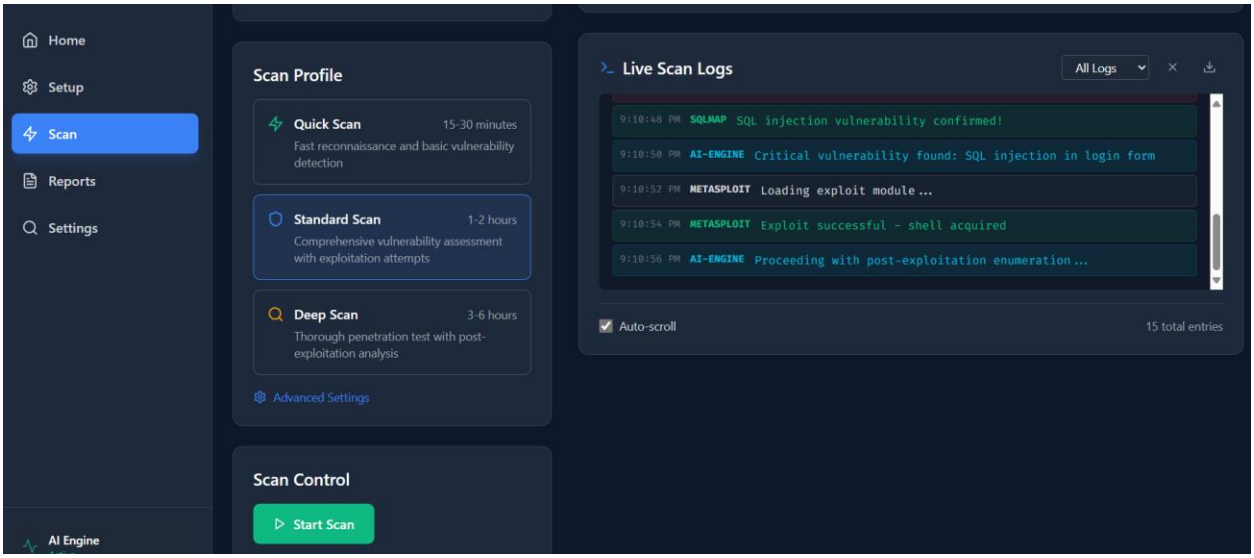

Figure 8 Scan and exploitation module

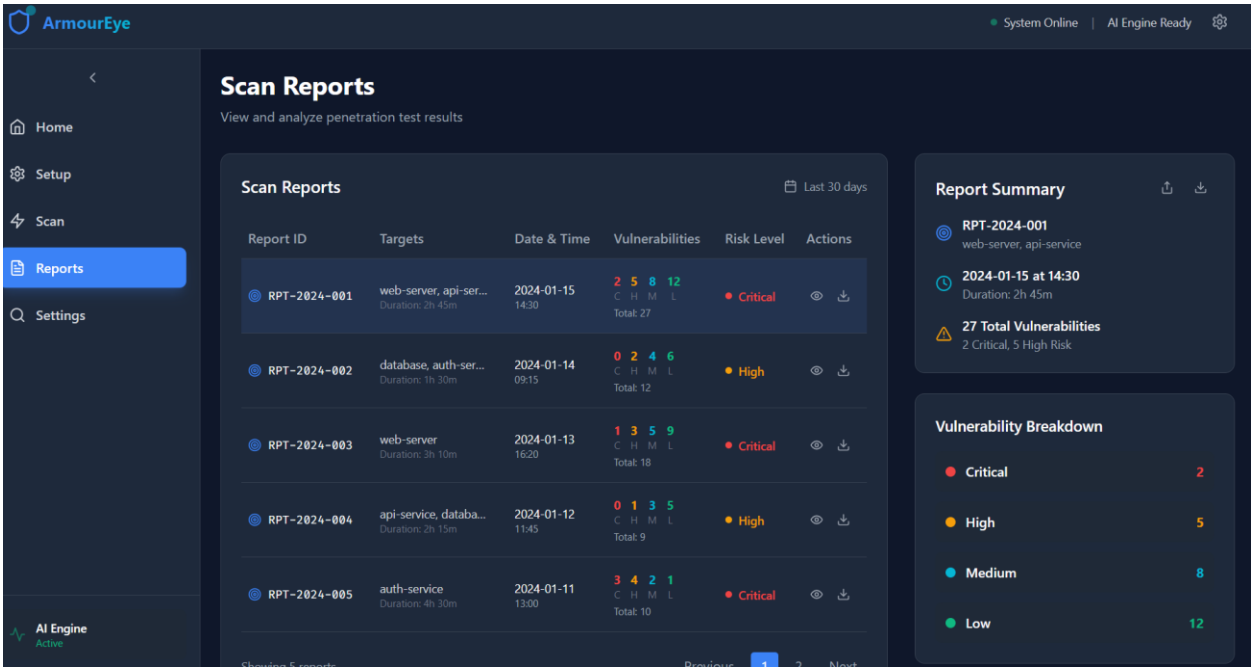

Figure 9 Report viewing

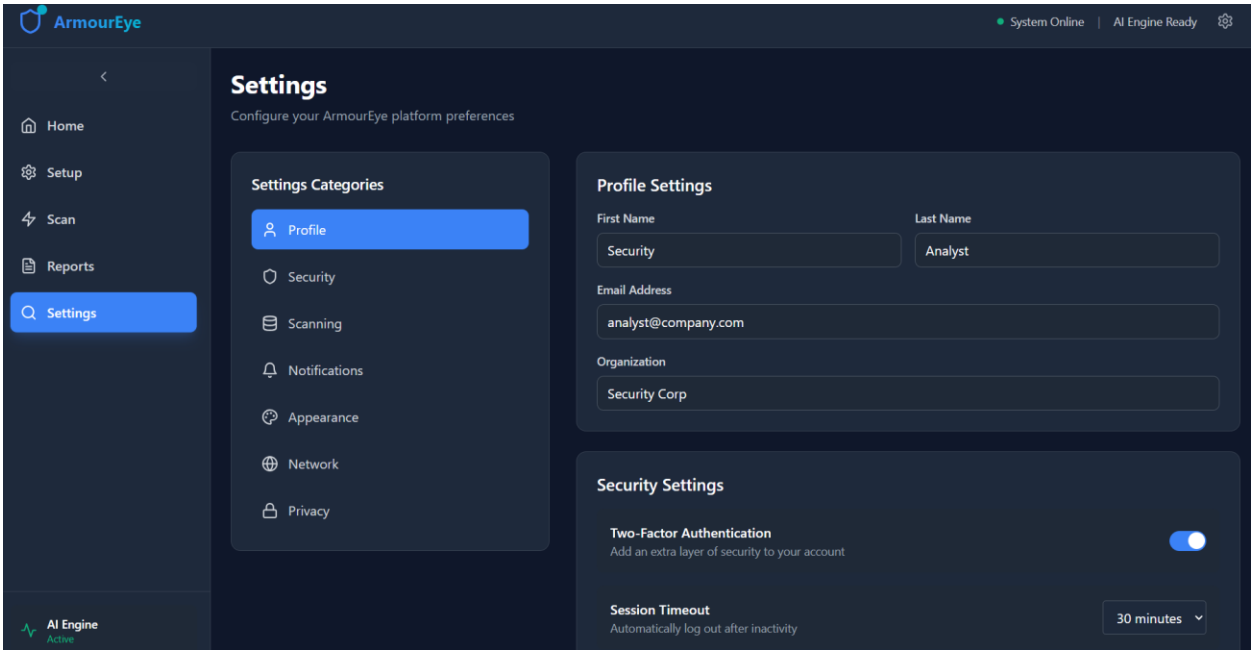

Figure 10 Settings page