Dissertation submitted to the

UNIVERSITY OF MOHAMED BOUDIAF --- M'SILA, ALGERIA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

In partial fulfillment of the requirements for the degree of

Master's in Computer Science

Specialization: NICT network information and communication technologies

Presented by

Benaicha Abdallah

Djenidi Ilies

Entitled

Design and Implementation of securAX: An AI-based Platform for Web Application Security and Server Configuration Assessment

securAX

Under the supervision of

Mokhtari Rabah

June, 2026

الملخص

يعالج هذا العمل إشكالية التشتت الهيكلي في أدوات الأمن السيبراني من خلال تصميم وتنفيذ منصة موحدة **securAX** تجمع ستة محاور تحليلية في واجهة واحدة: التحليل الديناميكي (DAST)، والتحليل الثابت (SAST)، وتدقيق إعداد الخوادم، وتقييم بروتوكولات TLS، وفحص ترويسات HTTP الأمنية، وتحليل تبعيات المكتبات. تعمل المنصة محلياً بالكامل دون أي اعتماد سحابي، مبنية على معايير OWASP وCVSS وATT&CK MITRE وكتالوج CISA KEV.

الكلمات المفتاحية: الأمن السيبراني، تحليل الثغرات، OWASP، DAST، SAST، السيادة الرقمية.

Abstract

This work addresses the structural fragmentation of cybersecurity analysis tools by designing and implementing a unified open-source platform , that consolidates six analytical dimensions into a single interface: Dynamic Application Security Testing (DAST), Static Application Security Testing (SAST), web server configuration auditing, TLS/SSL protocol assessment, HTTP security headers inspection, and Software Composition Analysis (SCA). The platform operates entirely on-premises with no external cloud dependency, ensuring full data sovereignty over analyzed assets. Its design is grounded in internationally recognized standards: OWASP Top 10, CVSS v3.1, MITRE ATT&CK, and the CISA KEV catalog.

Keywords: Cybersecurity, vulnerability analysis, OWASP, DAST, SAST, digital sovereignty.

Dedication

To our parents, whose unconditional love, sacrifices, and endless prayers have been our guiding light.

To our families, who supported us throughout this demanding journey.

To our esteemed professors and mentors, for their invaluable wisdom and patience.

To everyone who believed in the vision of securAX and the pursuit of a safer digital world.

Acknowledgements

First and foremost, we express our profound gratitude to Allah for giving us the strength, patience, and knowledge to complete this thesis.

We would like to extend our sincere and heartfelt appreciation to our supervisor, whose invaluable guidance, continuous support, and constructive feedback were instrumental in shaping both the securAX platform and this manuscript.

Our deep thanks also go to the honorable members of the jury for accepting to evaluate our work. We sincerely appreciate the time they have taken to review this thesis, and we look forward to their insightful comments.

We are profoundly grateful to our families for their endless encouragement, moral support, and unconditional love throughout our entire academic journey. Their belief in us made this achievement possible.

Finally, we would like to thank our friends, colleagues, and the wider academic community for their collaboration, insightful discussions, and the inspiring environment they provided during our studies.

Contents

List of Tables

List of Figures

General Introduction

Context

The last decade has witnessed a radical transformation in the nature of cybersecurity threats. Vulnerabilities are no longer isolated programming mistakes , they are engineered with increasing sophistication. Generative artificial intelligence has amplified these shifts from two opposing directions: it has enabled developers to build digital systems faster than ever before, while simultaneously allowing attackers to automate reconnaissance and generate attack payloads with efficiency that was unthinkable a few years ago. In this evolving landscape, cybersecurity has become an existential requirement for any digital infrastructure that aspires to survive.

Problem Statement

Despite the wide availability of application security tools (both open-source and commercial) the field suffers from deep structural fragmentation: each tool covers a limited scope, and no open platform exists that can deliver a unified, comprehensive risk view while operating in a fully sovereign manner without reliance on external cloud services. This constraint is especially acute in the Algerian context, where expensive commercial licenses and data protection requirements impose strict constraints.

The central research question is therefore:

> *How can we design and implement a unified cybersecurity analysis platform that is simultaneously comprehensive, sovereign, accessible, and academically rigorous?*

Objectives

This work aims to achieve the following goals:

- Unify six security analysis dimensions --- DAST, SAST, server configuration auditing, TLS assessment, HTTP security headers inspection, and SCA --- within a single integrated platform.
- Ensure the platform operates entirely on-premises with zero data leakage.

- Build a scoring engine integrated with CVSS v3.1 and the CISA KEV catalog for threat-driven prioritization.
- Release the platform as free, open-source software deployable without cloud infrastructure.

Methodology

The research follows an engineering methodology comprising four phases: a rigorous state-of-the-art review (Chapter 1), architectural analysis and design (Chapter2), implementation and experimental validation (Chapter3), followed by evaluation of results and comparison with existing tools.

Document Organization

This dissertation is structured as follows:

- **Chapter 1 --- State of the Art:** Cybersecurity for web applications and network infrastructure; threat landscape, standards, tools, and identified gaps.
- **Chapter 2 --- Analysis and Design:** Architectural design of securAX, requirements modeling, and system architecture.
- **Chapter 3 --- Implementation and Experimental Evaluation:** Technical implementation of securAX and experimental results.

Chapter 1: State of the Art

Cybersecurity for Web Applications, Network Infrastructure, and Analysis Tools

1.1 A New Era of Threats: When Artificial Intelligence Became a Weapon

For decades, software vulnerabilities were treated as an unavoidable byproduct of human fallibility---the price paid for complexity, tight deadlines, and imperfect peer review. Security teams responded by building layers of compensating controls around inherently imperfect software, institutionalizing a culture of *detect and respond* rather than *prevent by design*. Generative artificial intelligence arrived and turned this assumption on its head.

1.1.1 The AI-Generated Code Crisis

Within a few years, tools such as GitHub Copilot, ChatGPT, Amazon CodeWhisperer, and Tabnine became integrated into mainstream software development workflows. Survey data from JetBrains (2024) indicates that more than 62% of professional developers globally now use AI coding assistants on a regular basis [1]. The productivity gains are real, but so are the security costs.

According to a landmark study published by SQ Magazine in 2026 [2]:

- AI-generated code carries a vulnerability density 2.7 higher than manually written code.
- 40% of AI-generated code snippets contain critical security vulnerabilities identifiable by standard review processes.
- Large language models fail to prevent Cross-Site Scripting (XSS) attacks in 86% of evaluated cases.
- The same models produce Log Injection vulnerabilities in 88% of test scenarios.
- 58% of developers integrate AI-generated code directly into production without any security review---a phenomenon researchers call *automation bias* [2].

These figures reflect a fundamental limitation of language models: they optimize for syntactic correctness and functional plausibility, not for security properties. A model cannot reason about the security implications of user-controlled input flowing through an unsanitized string concatenation; it simply reproduces patterns observed in its training data, including the insecure patterns that pervade open-source repositories [3].

1.1.2 AI as an Offensive Weapon

The challenge is compounded by the adversarial use of the same technology. unprecedented scale.

> *``AI-powered tools can scan entire internet address ranges for known vulnerable services in hours rather than days. The asymmetry between AI-accelerated offense and human-speed defense fundamentally changes the risk calculus for organizations of all sizes."*
> --- Gartner Security & Risk Management Summit Report, 2024 [4].

The statistics are striking [5], [6], [7]:

- 72% increase in AI-driven cyberattacks recorded between 2023 and 2024.
- 1,265% surge in phishing attacks powered by generative AI tools (Verizon DBIR, 2024 [6]).
- 47% of organizations ranked adversarial generative AI as their most pressing security concern (WEF Global Risks Report, 2024 [7]).
- 16% of documented breaches in 2024 directly involved AI (IBM/Ponemon, 2024 [8]), with a mean cost of 5.72 million significantly above the overall average.

> *``The AI threat is not a future scenario. It is a present operational reality reflected in every major security report published in 2024--2025."*
> --- Total Assure AI Cybersecurity Statistics, 2025 [5].

1.2 The Global Threat Landscape: Documented Statistics (2023--2025)

Modern cybersecurity discourse must be grounded in measurable, reproducible evidence. The following data draws exclusively from peer-reviewed reports and primary sources from the world's leading security research organizations.

Table 1.1: Documented Cybersecurity Indicators from Primary Sources (2023--2025)

Indicator	**Value**	**Source**
Average global data breach cost	$4.88M (+10%)	IBM Cost of a Breach 2024 [8]
Average breach cost --- United States	$9.36M	IBM 2024 [8]
Average breach cost --- Middle East	$8.07M	IBM 2023 [9]
Average breach cost --- Healthcare sector	$9.77M	IBM 2024 [8]
AI-enhanced breach mean cost	$5.72M	IBM/Ponemon 2024 [8]
Verified security breaches (one year)	10,626 (record)	Verizon DBIR 2024 [6]
Security incidents analyzed	30,458	Verizon DBIR 2024 [6]
Increase in phishing attacks (AI-generated)	+1,265%	Verizon DBIR 2024 [6]
Vuln. exploitation as breach entry point	+180% (YoY)	Verizon DBIR 2024 [6]
Average breach dwell time (detection)	194 days	IBM 2024 [8]
Average breach containment time	64 days	IBM 2024 [8]
Organizations with critical unpatched vulns	57%	Ponemon Institute 2024 [10]
Breaches involving human error	68%	Verizon DBIR 2024 [6]
Breaches directly involving AI	16%	IBM/Ponemon 2024 [8]
New CVEs registered in 2024	40,000+	NIST NVD 2024 [11]
Ransomware attacks per day globally	4,000+	SonicWall 2024 [12]
Cyberattack increase --- Algeria	+67%	ANSI Algeria 2023 [13]
Cost savings using AI+Automation	$2.2M/breach	IBM/Ponemon 2024 [8]
Global cybersecurity spending (2028)	$377B	IDC 2025 [14]

1.2.1 The 194-Day Dwell Time: What Attackers Do With It

The average detection time of **194 days** recorded by IBM in 2024 [8] deserves specific analysis. This is not simply a performance metric---it describes the window during which an undetected attacker can:

- Map the entire internal network topology and identify high-value assets.
- Establish persistence through multiple backdoors surviving initial remediation.
- Escalate privileges progressively toward domain administrator access.
- Exfiltrate intellectual property, customer records, and credentials.
- Deploy ransomware payloads timed for maximum operational disruption.

The subsequent 64 days of containment means organizations spend an average of **258 days**---more than eight months---managing a single breach from initial compromise to full remediation. For organizations in regulated industries, this timeline intersects with mandatory notification deadlines (72 hours under GDPR, 30 days under HIPAA) and service continuity obligations that amplify the operational and legal consequences.

1.2.2 Notable Real-World Incidents (2021--2024)

Abstract statistics gain urgency when grounded in documented incidents:

- **Log4Shell (December 2021, CVE-2021-44228, CVSS 10.0):** A zero-day remote code execution vulnerability in the Apache Log4j logging library---a component present as a transitive dependency in millions of Java applications across virtually every industry. Within 72 hours of disclosure, CISA documented exploitation attempts against US federal civilian agencies. Remediation required emergency response from hundreds of thousands of organizations worldwide [15].
- **SolarWinds Orion Supply Chain (2020--2021):** Attackers compromised the build infrastructure of SolarWinds, inserting a backdoor into legitimate software updates distributed to approximately 18,000 organizations including US government agencies and Fortune 500 companies. The campaign remained undetected for approximately nine months [16].
- **MOVEit Transfer (June 2023, CVE-2023-34362):** A SQL injection vulnerability in the MOVEit managed file transfer platform was exploited by the Cl0p ransomware group to exfiltrate data from over 2,700 organizations including government agencies in multiple countries, affecting an estimated 93 million individuals [15].

- **Change Healthcare Ransomware (February 2024):** A ransomware attack against Change Healthcare, a major US health payment processor, disrupted prescription processing for an estimated one-third of US pharmacies for weeks. The estimated cost of the incident exceeded $872 million in direct losses, with UnitedHealth Group reporting total impacts approaching $2.3 billion [8].

1.3 OWASP Top 10 (2025): The Web Application Risk Map

1.3.1 The OWASP Foundation and Its Methodology

The Open Web Application Security Project (OWASP), founded in 2001, operates as an open community of security professionals committed to improving software security through freely available tools, documentation, and research. Its flagship publication, the OWASP Top 10 [17], has become the de facto standard for web application vulnerability classification used by security teams, developers, auditors, and regulatory bodies worldwide.

The Top 10 is a risk-weighted classification---not merely a frequency ranking--- considering the incidence rate, average exploitability and impact, and automated detection coverage of each category. A rare but catastrophic vulnerability class can rank above a common but low-severity one.

The 2025 edition, current as of May 2026 [17], reflects significant shifts driven by cloud-native architectures, AI-generated code proliferation, expanded API attack surfaces, and the continued escalation of supply-chain attacks documented since the 2021 edition. Where historically significant changes between 2021 and 2025 are noted in this analysis, both editions are referenced to illustrate how the threat landscape has evolved.

1.3.2 OWASP Top 10 --- 2025 Edition

Table 1.2 presents the current classification [17], which this work adopts as its primary risk framework.

Table 1.2: OWASP Top 10 (2025 Edition): Categories, Impact, and Primary Attack Vectors [17]

ID	Category	Description	Primary Attack Vector
A01	Broken Access Control	Insufficient controls enabling privilege escalation and unauthorized resource access	URL manipulation, IDOR, misconfigured CORS, path traversal
A02	Cryptographic Failures	Sensitive data exposure through broken or absent encryption	Legacy TLS, hardcoded keys, MD5/SHA-1 password storage
A03	Injection	Malicious code injected into system interpreters (SQL, OS, LDAP, template)	SQLi, XSS, Command Injection, SSTI, XXE
A04	Insecure Design	Security controls absent from the architectural design phase	Missing threat modeling, no rate limiting, no tenant isolation
A05	Security Misconfiguration	Insecure defaults, verbose error messages, unnecessary features enabled	Server banners, open directories, CORS=*, debug endpoints
A06	Vulnerable Components	Libraries with known, unpatched CVEs in the dependency chain	Log4Shell, OpenSSL, Struts, transitive dependency chains
A07	Auth & Session Failures	Bypassable authentication and session management mechanisms	Weak passwords, predictable tokens, session fixation, credential stuffing
A08	Software/Data Integrity	Compromised CI/CD pipelines and unsafe deserialization	Supply-chain attacks, SolarWinds pattern, pickle deserialization
A09	Logging Failures	Absence of forensically useful event logs and monitoring	Incomplete, non-centralized, alert-free logs
A10	SSRF	Forcing the server to execute requests against internal resources	Cloud IMDS access, internal service enumeration, firewall bypass

Key findings from OWASP's data collection underpinning this classification:

- Broken Access Control (A01) was found in **94%** of applications tested by contributing organizations---the highest incidence rate of any category.
- Security Misconfiguration (A05) affects **90%** of tested applications, confirming that configuration-level issues are more prevalent than code-level ones [18].
- Vulnerable Components (A06) are present in virtually every modern application due to the transitive dependency explosion in contemporary software stacks.

1.3.3 The Paradigm Shift: Insecure Design (A04)

One of the most structurally significant categories in the 2025 edition is **A04: Insecure Design**---first introduced in 2021 and consolidated in 2025. This category formally acknowledges that entire classes of vulnerabilities cannot be remediated at the implementation level because they reflect fundamental architectural decisions made during system design [17].

Rate limiting absent from a payment processing API, missing tenant isolation in a multi-tenant application, and absent anti-automation controls on credential-sensitive endpoints are design-level vulnerabilities requiring architectural changes rather than code patches. This recognition motivates the integration of security into the design phase through *threat modeling*, *security requirements definition*, and *secure design patterns*---collectively known as Security by Design. This shift carries particular weight in the age of generative AI, where entire systems are built without a human developer systematically thinking through abuse scenarios.

1.3.4 Key Changes in the 2025 Edition

The 2025 edition introduced structural updates relative to the 2021 edition that are directly relevant to this work [17]:

- **API Security prominence increased:** REST and GraphQL APIs became primary surfaces. API-specific risks received dedicated treatment, complementing the *OWASP API Security Top 10* [19].
- **AI/LLM Security formalized:** Following the 2023 publication of *OWASP Top 10 for LLM Applications* [20], the 2025 edition addresses prompt injection, training data poisoning, and insecure plugin design as first-class attack surface categories.
- **Supply chain security expanded:** The escalation of supply-chain incidents between 2022 and 2025 (Log4Shell, XZ Utils, MOVEit) led to stronger emphasis on A06 and A08, reinforcing the SCA imperative.

This work adopts the 2025 edition as its primary normative reference, supplemented by OWASP's specialized publications where relevant [19], [20].

1.4 Vulnerability Classification Standards: CVE, CWE, CVSS, and MITRE ATT&CK

A discovered vulnerability acquires operational value only when expressed in a shared language that is understood and actionable by security teams, automated tools, and reference databases worldwide. The international community has developed an interconnected set of standards forming the technical backbone of any serious security analysis platform.

Table 1.3: International Vulnerability Classification and Governance Standards

Standard	Issuing Body	Role	Notable Example
CVE	MITRE / NIST NVD	Unique identifier per vulnerability	CVE-2021-44228 (Log4Shell)
CWE	MITRE	Root-cause software weakness catalog	CWE-89: SQL Injection
CVSS v3.1	FIRST.org	Standardized severity score 0.0--10.0	`AV:N/AC:L/PR:N UI:N/S:C/C:H I:H/A:H`
CAPEC	MITRE	Attack pattern catalog	CAPEC-66: SQL Injection
ATT&CK	MITRE	Adversarial tactics and techniques	T1190: Exploit Public-Facing
KEV Catalog	CISA / US-CERT	Confirmed in-the-wild exploited CVEs	1,100+ entries, updated in real time
NIST CSF 2.0	NIST	Cybersecurity governance framework	Govern→Identify→ Protect→Detect→ Respond
ISO/IEC 27001	ISO/IEC	Information security management system	Annex A: 93 security controls
PCI-DSS 4.0	PCI SSC	Payment card security requirements	Req. 6: Secure Systems and Software

1.4.1 The CVE System: Scale and Operational Reality

The CVE (Common Vulnerabilities and Exposures) system, maintained by MITRE under CISA sponsorship, has grown from its 1999 launch to encompass over **230,000 entries** as of 2025 [11]. In 2024 alone, NIST registered more than **40,000 new CVE entries**---an average exceeding **109 newly disclosed vulnerabilities every day** [11].

> *``The volume growth in CVE disclosures is itself a threat indicator. No human security team can manually track 109 new vulnerabilities per day across a modern technology stack. Automation is not optional---it is the only viable response strategy."*
> --- Cobalt State of Pentesting Report, 2024 [21].

The system has faced capacity challenges: NIST's NVD fell significantly behind in enriching CVE entries in 2024, creating gaps in CVSS scoring that affected automated vulnerability management tools worldwide---underscoring the operational fragility of centralized vulnerability databases [11].

1.4.2 CVSS v3.1: Multi-Dimensional Severity Scoring

CVSS v3.1, published by FIRST.org [22], is the de facto international standard for measuring and communicating vulnerability severity. Its strength lies in its multi-dimensional structure:

- **Base Score (0.0--10.0):** Captures intrinsic vulnerability characteristics that are constant over time and across environments: attack vector (Network, Adjacent, Local, Physical), attack complexity, privileges required, user interaction, scope, and impact on Confidentiality, Integrity, and Availability.
- **Temporal Score:** Adjusts the base score based on exploit code maturity, remediation level, and report confidence---values that change as patches are released and exploits are published.
- **Environmental Score:** Allows each organization to adapt the assessment to its specific deployment context, accounting for compensating controls and asset criticality.

CVSS severity bands: Critical (9.0--10.0), High (7.0--8.9), Medium (4.0--6.9), Low (0.1--3.9).

A critical limitation of CVSS that practitioners must internalize: a score reflects *maximum potential severity under ideal exploitation conditions*, not actual risk in a specific deployment context. A CVSS 9.8 vulnerability on an air-gapped internal system may represent lower actual risk than a CVSS 6.5 vulnerability on an internet-facing payment endpoint with no compensating controls.

1.4.3 The CISA KEV Catalog: Exploitation Intelligence Above the Score

The CISA Known Exploited Vulnerabilities (KEV) catalog, launched in November 2021, represents one of the most operationally valuable vulnerability prioritization resources available [15]. Unlike CVSS, which measures theoretical severity, KEV identifies vulnerabilities *confirmed as actively exploited by threat actors in real-world attacks.*

As of 2025, the catalog contains over **1,100 entries**, each with a mandatory remediation deadline for US federal civilian agencies under Binding Operational Directive BOD 22-01.

Research consistently demonstrates that KEV membership is a stronger predictor of exploitation than CVSS score alone [15]:

- A CVSS 7.0 vulnerability in the KEV catalog presents greater operational risk than a CVSS 9.8 vulnerability with no documented exploitation.
- Only approximately **4%** of all published CVEs ever become actively exploited in the wild---making KEV membership the most actionable prioritization signal available.

1.4.4 MITRE ATT&CK: From Vulnerability to Attack Chain

The MITRE ATT&CK (Adversarial Tactics, Techniques, and Common Knowledge) framework provides an empirically grounded knowledge base of real-world adversary behavior, organized across 14 tactics in the Enterprise matrix [16]: Initial Access, Execution, Persistence, Privilege Escalation, Defense Evasion, Credential Access, Discovery, Lateral Movement, Collection, Command and Control, Exfiltration, and Impact.

ATT&CK bridges the gap between vulnerability assessment and threat modeling. By mapping detected vulnerabilities to ATT&CK techniques, security teams can understand which phases of an attack chain a vulnerability would enable, assess detection coverage gaps, and prioritize remediation based on tactical impact rather than solely on CVSS scores.

1.5 Security Analysis Techniques

No single analysis methodology provides complete visibility over modern application attack surfaces. Effective cybersecurity assessment requires combining multiple complementary approaches, each revealing different vulnerability classes.

1.5.1 Dynamic Application Security Testing (DAST)

DAST analyzes applications during execution by simulating attacker behavior. It targets exposed interfaces---HTTP endpoints, WebSocket connections, GraphQL APIs---and sends crafted inputs designed to trigger vulnerability-indicative responses. Because DAST operates without source code access, it tests actual runtime behavior and is inherently resistant to false positives: a tool that successfully retrieves `/etc/passwd` through a path traversal attack has confirmed both the vulnerability and its exploitability [23].

Table 1.4: Principal Open-Source DAST Tools: Capabilities and Scope

Tool	**Primary Focus**	**Coverage**	**Integration**
OWASP ZAP	Web application DAST	OWASP Top 10, active + passive	CI/CD via REST API
Nikto	Web server scanning	6,700+ tests, CVEs, server configs	CLI scripting
Wapiti	Web app crawler + fuzzer	SQLi, XSS, CSRF, SSRF, XXE, LFI	CLI, JSON/HTML reports
Nuclei	Template-based scanning	5,000+ community templates, CVEs	CI/CD, custom YAML

Key limitations of DAST: crawler coverage is bounded by endpoint discoverability; complex single-page applications with dynamic routing or session-dependent states may be incompletely assessed. DAST cannot identify source-code-level vulnerabilities such as hard-coded credentials or logic errors that produce no observable runtime anomaly [23].

1.5.2 Static Application Security Testing (SAST)

SAST inspects source code, bytecode, or binary artifacts without executing the application. Advanced SAST tools employ data flow analysis, taint analysis, and semantic analysis to trace user-controlled input through application logic and identify conditions where tainted data reaches sensitive operations without appropriate sanitization [24].

SAST is particularly critical for AI-generated code validation: it can systematically inspect large volumes of generated code for security anti-patterns reflecting common model failure modes, providing a scalable mechanism for validating AI-assisted development workflows [2].

Tool	Languages	Analysis Method	Key Strength
Bandit [25]	Python	AST pattern matching	Flask, Django, FastAPI security
Semgrep [26]	30+ languages	Syntactic pattern matching	Custom rules, 4,000+ community rules
SonarQube	25+ languages	Data flow + taint analysis	Continuous inspection, IDE integration
CodeQL	10+ languages	Semantic code querying	Complex multi-file vulnerability patterns

Table 1.5: Principal Open-Source SAST Tools: Language Coverage and Analysis Depth

1.5.3 Software Composition Analysis (SCA)

Modern software is fundamentally compositional: the average enterprise web application incorporates **hundreds of direct and transitive dependencies** representing far more code than the application's own source [18]. SCA analyzes application manifests (`requirements.txt`, `package.json`,
`pom.xml`), lockfiles, and dependency trees to identify components with known CVEs, end-of-life components, and license risks.

The operational imperative for SCA was demonstrated catastrophically by three documented incidents:

- **Log4Shell (CVE-2021-44228):** Present as a transitive dependency in millions of Java applications. Emergency remediation required from hundreds of thousands of organizations within 72 hours of disclosure [15].
- **SolarWinds (2020):** Build infrastructure compromise inserted a backdoor into legitimate updates distributed to approximately 18,000 organizations [16].
- **XZ Utils (CVE-2024-3094, CVSS 10.0):** A two-year social engineering campaign compromised a core Linux compression library, inserting a backdoor targeting SSH authentication in major Linux distributions [15].

Verizon DBIR 2024 recorded a **68% rise** in the impact of supply-chain interdependencies on documented breaches [6].

Table 1.6: Principal Open-Source SCA Tools: Language Coverage and Vulnerability Databases

Tool	Languages	Vulnerability DB	Key Feature
pip-audit (PyPA)	Python	OSV (NVD + GitHub + PyPI)	Native Python integration
npm audit	JavaScript	GitHub Advisory DB	Built into Node.js
Grype [27]	Multi-language	NVD, GitHub, OSV, RHSA	Docker image + SBOM analysis
OWASP Dep-Check	Java, .NET, JS	NVD + vendor advisories	Mature HTML/XML reports

1.5.4 Network Infrastructure Security Analysis

Network infrastructure analysis constitutes an essential complementary domain to application-layer security testing. While DAST, SAST, and SCA examine the application itself, network-level analysis examines the surrounding infrastructure through which application traffic flows and which must be hardened independently of application code.

Two principal methodologies characterize this domain:

- **Active network scanning:** Tools such as Nmap [28] perform host discovery, port enumeration, service version detection, and OS fingerprinting by sending crafted packets and analyzing responses. Active scanning reveals open ports, exposed services, and running software versions that can be cross-referenced against CVE databases. For example, an exposed Redis instance (port 6379) without authentication, or an outdated OpenSSH version with known CVEs, represents a directly exploitable network-level entry point independent of application logic.
- **Passive network analysis:** Passive approaches analyze traffic or service responses without injecting active payloads. HTTP response header inspection, TLS handshake analysis, and banner grabbing fall into this category. Passive analysis is inherently safe for production environments where active scanning could disrupt services or trigger intrusion detection systems.

The intersection of network and application security is particularly significant in microservice architectures and containerized deployments, where internal service meshes may expose APIs and administrative interfaces across the network boundary. A unified security platform must therefore integrate network-layer analysis alongside application security techniques to provide complete coverage of the attack surface.

Table 1.7: Principal Network Security Analysis Tools: Methods and Coverage

Tool	Method	Coverage	Key Use Case
Nmap	Active scanning	Port discovery, service/version detection, OS fingerprinting	Network perimeter mapping
Masscan	Active scanning	High-speed port scanning at internet scale	Large IP range enumeration
Shodan API	Passive (external)	Internet-exposed service indexing, banner analysis	Attack surface discovery
Wireshark	Passive (capture)	Protocol dissection, traffic anomaly detection	Incident analysis, MITM detection

1.5.5 Web Server Configuration Auditing

Configuration-level vulnerabilities are documented as the most prevalent category in tested applications (90% incidence rate, Positive Technologies 2023 [18]). They result not from programming errors but from insecure defaults, accumulated technical debt, and lack of security awareness in server configuration:

- Server version disclosure via `Server:` and `X-Powered-By:` response headers.
- Unnecessary modules enabled (increasing attack surface without functional benefit).
- Insufficient request timeouts enabling Slowloris and slow-read DoS attacks.
- Missing or misconfigured security headers (CSP, HSTS, X-Frame-Options).
- Outdated TLS configurations permitting protocol downgrade attacks.
- Debug or administrative interfaces exposed in production environments.

The CIS Apache HTTP Server 2.4 Benchmark [29] provides the reference standard with over 80 auditable controls, organized by configuration domain. The absence of a unified open-source tool combining configuration auditing with automated scoring and remediation guidance constitutes a significant gap in the available tooling ecosystem, as discussed in Section 1.8.3.

1.5.6 Scanning Methodology by Vector: Passive HTTP and Web Analysis

In parallel with active scanning techniques, a complementary passive HTTP analysis vector provides safe, non-disruptive security assessment of production environments. Unlike active DAST tools such as OWASP ZAP operating in active mode---which inject crafted attack payloads and may trigger application-level alerts, disrupt session state, or generate anomalous entries in application logs---passive analysis operates exclusively by examining authentic responses produced by the server during normal HTTP interactions, without introducing any artificial payloads.

This vector encompasses the following inspection dimensions:

- **TLS certificate validity:** Verification of certificate chain integrity, expiry dates, subject alternative names (SANs), and cryptographic algorithm strength.
- **HTTP security header completeness:** Assessment of the presence and configuration quality of all critical security headers (CSP, HSTS, X-Content-Type-Options, Referrer-Policy, Permissions-Policy, CORP, COOP), evaluated not merely for presence but for correctness of configuration directives.
- **Sensitive file exposure indicators:** Detection of unintentionally exposed configuration files, backup files, version control metadata (`.git`, `.env`), and administrative endpoints accessible without authentication.
- **Cookie security attributes:** Inspection of `Set-Cookie` headers for `HttpOnly`, `Secure`, and `SameSite` attributes, the absence of which enables client-side cookie theft and cross-site request forgery.
- **CORS policy correctness:** Analysis of `Access-Control-Allow-Origin` and related headers for overly permissive configurations (`*` wildcards or reflective origin policies) that enable cross-origin data theft.
- **Response-analysis indicators of injection vulnerabilities:** Examination of server error messages, response anomalies, and content patterns that serve as indicators of SQL injection or Insecure Direct Object Reference (IDOR) vulnerabilities, without transmitting active exploit payloads.

This passive approach makes the vector particularly suited for continuous monitoring of production environments where ZAP's active mode or Nikto's aggressive scanning could cause service disruption. Implemented as a dedicated `web_scanner.py` module, this vector provides a safe baseline assessment that can be executed against live systems without prior

authorization for active testing, and whose findings complement those of the active DAST vector to provide comprehensive HTTP-layer security coverage.

1.6 TLS Security and HTTP Security Headers

1.6.1 TLS Protocol Security: A History of Cryptographic Failures

Transport Layer Security (TLS) constitutes the cryptographic foundation of modern web communications, guaranteeing confidentiality, integrity, and server authentication through X.509 certificate validation [22]. Despite decades of deployment, insecure TLS configurations remain pervasive: Qualys SSL Labs' SSL Pulse survey (2024) [30] found that **15%** of surveyed sites still support TLS 1.0 or TLS 1.1, and **8%** present certificates with weaknesses.

Table 1.8 documents the principal TLS vulnerabilities with their associated CVE identifiers and real-world impact.

Table 1.8: Documented TLS/SSL Vulnerabilities: CVE Identifiers and Impact

Vulnerability	**CVE**	**Technical Description**	**Impact**
Heartbleed	CVE-2014-0160	OpenSSL Heartbeat buffer over-read	Server memory exposure, private key theft
POODLE	CVE-2014-3566	SSL 3.0 CBC padding oracle	Session cookie decryption
BEAST	CVE-2011-3389	TLS 1.0 CBC IV predictability	Partial plaintext recovery
DROWN	CVE-2016-0800	SSLv2 export cipher attack	Modern TLS session decryption
ROBOT	CVE-2017-13099	Bleichenbacher RSA oracle	Private key recovery
CRIME	CVE-2012-4929	TLS compression oracle	Session cookie recovery

> *``Heartbleed (CVE-2014-0160) affected approximately 17% of all HTTPS servers at the time of disclosure---roughly 500,000 servers---requiring emergency response from organizations worldwide and representing one of the most severe infrastructure-level vulnerabilities ever disclosed."*
> --- OpenSSL Security Advisory, April 2014 [30].

Current best practice mandates **TLS 1.3** as the preferred protocol (IETF RFC 8446, 2018), with TLS 1.2 as the minimum acceptable version. TLS 1.3 eliminates all previously exploited

weaknesses: it removes support for weak cipher suites (RC4, DES, 3DES, EXPORT), mandates forward secrecy through ephemeral Diffie-Hellman, reduces the handshake to a single round-trip, and removes the compression and renegotiation features exploited in historical attacks.

1.6.2 HTTP Security Headers: Browser-Side Defense

HTTP security headers represent a defense-in-depth layer that servers send to browsers, instructing them on how to handle content, which resources to trust, and which capabilities to restrict. According to securityheaders.io (2023) [31]:

- Fewer than **25%** of the one million most-visited websites implement a meaningful Content-Security-Policy (CSP).
- Only **38%** correctly deploy HTTP Strict-Transport-Security (HSTS).
- Fewer than **10%** implement Permissions-Policy.

Table 1.9: HTTP Security Headers: Threats Mitigated and Common Misconfigurations

Header	**Threat Mitigated**	**Common Misconfiguration**
Content-Security-Policy	XSS, data injection	`unsafe-inline` / `unsafe-eval` directives nullify protection
HSTS	TLS downgrade, MITM	Missing `includeSubDomains`; short `max-age`
X-Frame-Options	Clickjacking	Absent; superseded by CSP `frame-ancestors`
X-Content-Type-Options	MIME confusion attacks	Missing `nosniff` directive
Referrer-Policy	Information leakage	Default allows full URL leakage to third parties
Permissions-Policy	Unauthorized API access	Absent; exposes camera, microphone, geolocation
CORP / COOP	Spectre-class timing attacks	Missing isolation; enables cross-origin data leakage

A critical implementation detail: the *quality* of security headers matters as much as their presence. A CSP containing `unsafe-inline` provides minimal XSS protection. An HSTS header with a `max-age` of one hour is operationally equivalent to no HSTS header at all. Automated scanners must therefore evaluate configuration quality, not merely header presence.

1.7 Manual vs. Automated Security Analysis

The relative merits of manual and automated security analysis represent one of the most consequential practical questions in applied cybersecurity.

1.7.1 The Case for Manual Penetration Testing

Manual penetration testing---conducted by skilled professionals applying adversarial intuition, creative problem-solving, and contextual reasoning---excels at identifying vulnerability classes that fundamentally resist automation:

- Complex *business logic* vulnerabilities (e.g., race conditions in financial transaction flows, flawed authorization models in multi-tenant systems, privilege escalation through chained API calls).
- Novel exploitation chains involving multiple individually low-severity findings whose combination creates critical impact.
- Social engineering and physical security components of comprehensive security assessments.

> *``Skilled manual testers discover the equivalent of 20× more unique vulnerabilities than automated scanning alone---but this advantage comes at a time and financial cost that restricts its deployment to episodic, point-in-time assessments rather than continuous coverage."*
> --- DeepStrike Penetration Testing Statistics, 2025 [32].

1.7.2 The Economics of Manual Testing

Market data quantifies the structural limitations of manual penetration testing as an organizational control [21], [33]:

- The average organizational budget for penetration testing in 2024: **$164,000/year**.
- Cost per individual engagement: **$5,000 to $50,000** depending on scope, methodology, and provider.
- **60%** of organizations conducted testing at most twice per year--- leaving deployments unassessed for a minimum of six months between engagements.
- The penetration testing market is valued at **$1.82 billion** (2023), projected to reach **$5.24 billion** by 2030 [33]--- growth reflecting demand, not accessibility.

Table 1.10: Manual Human Analysis vs. Automated Analysis: Comparative Assessment

Criterion	Manual Penetration Testing	Automated Analysis
Cost	$5,000--$50,000+ per engagement	Fixed tool cost; repeatable at zero marginal cost
Speed	Days to weeks per engagement	Minutes for a full multi-domain scan
Consistency	Variable---tester experience and fatigue	Deterministic---identical rules every run
Availability	Episodic; scheduled in advance	Continuous; 24/7 CI/CD integration
Logic vulnerabilities	Strong---human creativity excels	Weak---limited to known patterns
Known CVE coverage	Limited---depends on tester knowledge	Comprehensive---automated DB updates
Documentation	Manual effort, variable quality	Structured, reproducible, automatable
Regulatory compliance	Requires signed contracts, certificates	Embedded in development workflow
Human error factor	Present---fatigue and oversight are real	Absent in execution; possible in rule authoring

1.7.3 The Bionic Hacker Model

The evidence motivates neither the exclusive use of automated tools nor the exclusive reliance on manual expertise. The optimal model---increasingly referred to as the *Bionic Hacker* [32]---combines automated continuous coverage of known-pattern vulnerabilities and configuration weaknesses with human analyst focus on what automation cannot replicate: business logic testing, adversarial creativity, and exploitation chain construction. This work implements the automated half of this model, designed to complement rather than replace human security expertise.

1.8 State of the Art in Existing Solutions

1.8.1 Commercial Solutions: Functionality at Enterprise Price Points

- **Nessus (Tenable):** The most widely deployed vulnerability scanner globally, with over 2 million downloads and a plugin database exceeding 170,000 signatures. Nessus Pro-

fessional pricing exceeds $3,000/year; Tenable.io enterprise deployments reach tens of thousands of dollars. Cloud dependency for plugin updates introduces data sovereignty concerns [34].

- **Qualys WAS:** A cloud-native DAST platform priced around $10,000/year. SaaS-only architecture routes application data through Qualys cloud infrastructure, which is incompatible with sensitive or regulated deployment environments.
- **Burp Suite Professional (PortSwigger):** The industry reference for manual-assisted web application penetration testing, priced at $449--$499/user/year [34]. Designed primarily for interactive manual use rather than automated pipeline integration.
- **SonarQube Enterprise:** Continuous code inspection platform. Enterprise pricing based on lines of code analyzed; mid-sized codebases typically require $15,000+ annually.
- **AppSpider (Rapid7):** A commercial DAST solution focused on web application and API security testing. AppSpider offers automated crawling, attack replay, and integration with Rapid7's broader vulnerability management ecosystem. Pricing is enterprise-tier and follows a SaaS subscription model, placing it beyond the budget of most organizations in developing economies.
- **Checkmarx One:** A comprehensive application security platform combining SAST, DAST, SCA, and infrastructure-as-code (IaC) scanning in a unified cloud-based interface. Checkmarx is widely adopted in regulated industries requiring end-to-end AppSec coverage. Enterprise licensing costs typically exceed $20,000/year for mid-sized codebases, and its cloud-native SaaS architecture raises data sovereignty concerns equivalent to those noted for Qualys WAS.

  Summing the annual costs of the primary commercial tools---Nessus + Qualys WAS + Burp Suite Pro + SonarQube Enterprise + AppSpider + Checkmarx---yields **over $28,000/year** before accounting for implementation, training, and integration costs. This figure exceeds the total annual IT budget of the majority of Algerian organizations [13].

1.8.2 Open-Source Ecosystem: Rich but Fragmented

The open-source security ecosystem provides capable tools covering every individual security domain, but they remain structurally fragmented: each tool addresses a narrow scope with its own interface, output format, and operational requirements.

According to the SANS 2023 State of Application Security report [23], **61%** of security teams cite the integration complexity of multiple tools as one of the primary barriers to comprehensive security coverage. Each tool produces findings in a different format (Nikto plain text, Bandit JSON, Nmap XML), requiring custom normalization pipelines to aggregate results into actionable dashboards.

Table 1.11: Key Tool Landscape: Coverage, Cost, and Structural Limitations

Tool	Cost/Year	DAST	SAST	Config	SCA	Unified	Local
Nessus (Tenable)	$3,000+	✓	×	✓	×	Partial	✓
OpenVAS	Free	✓	×	✓	×	Partial	✓
Burp Suite Pro	$449/user	✓	×	×	×	×	✓
SonarQube Enterprise	$15,000+	×	✓	×	✓	×	✓
OWASP ZAP	Free	✓	×	×	×	×	✓
Qualys WAS	$10,000+	✓	×	✓	×	Partial	×
Nikto	Free	✓	×	Partial	×	×	✓
Bandit	Free	×	✓	×	×	×	✓
Semgrep OSS	Free	×	✓	×	×	×	✓
Grype	Free	×	×	×	✓	×	✓
testssl.sh	Free	Partial	×	TLS only	×	×	✓

1.8.3 The Four Structural Gaps

A systematic analysis of the existing landscape reveals four structural limitations that collectively define the research opportunity:

1. **Absence of unified multi-domain platforms:** No widely adopted open-source tool simultaneously delivers DAST, SAST, TLS assessment, HTTP security header inspection, dependency analysis, and server configuration auditing through a unified interface with consistent, integrated risk scoring.

2. **Sovereignty incompatibility:** Cloud-dependent SaaS solutions require analyzed application data to traverse foreign infrastructure---incompatible with critical national infrastructure, healthcare, and defense deployment environments. This constraint is particularly acute in developing economies where data residency regulations are strengthening.

3. **Economic inaccessibility:** Comprehensive commercial platforms are priced for enterprise customers. The $28,000+ annual cost threshold excludes the vast majority

of organizations in developing economies, SMEs, research institutions, and academic organizations.

4. **Inadequate risk contextualisation:** Most platforms present binary vulnerability findings with CVSS scores as the sole prioritization signal, without integrating exploitation intelligence (KEV membership), asset criticality, environmental exposure, or attack chain analysis. This produces remediation queues that do not accurately reflect operational risk.

1.9 Cybersecurity in Algeria: National Context and Digital Sovereignty

1.9.1 National Institutional Framework

Algeria has pursued an increasingly proactive national cybersecurity strategy over the past decade, supported by a layered legislative and regulatory architecture that has accelerated significantly since 2020. Key institutional and legislative developments include:

- **Law 09-04** on the prevention and prosecution of cybercrime, establishing the foundational legal framework for information system protection. This law criminalizes unauthorized access to information systems, data interception, computer fraud, and related offences, providing the baseline penal framework for cybersecurity enforcement in Algeria.
- **Law on the Protection of Personal Data (18-07 of 10 June 2018, amended 25 November 2018):** Establishes the legal framework for the protection of personal data, aligned with international principles of data minimization, consent, and subject rights. This law creates obligations for organizations processing personal data and provides the regulatory foundation for data sovereignty requirements. The law was further reinforced by implementing regulations issued in 2018 concerning conditions for data processing and cross-border data transfers.
- **Constitution of 2020 --- Articles 47, 91, and 92:** Algeria's revised constitution, adopted by referendum in November 2020, enshrines the protection of personal data and privacy as a constitutional right. Article 47 guarantees the right to privacy of communications and personal data protection. Articles 91 and 92 establish the constitutional framework for digital sovereignty and the state's mandate to protect national information infrastructure and citizens' digital rights.

- **Référentiel de Sécurité des Systèmes d'Information (v1, 2016):** Algeria's national information systems security reference framework, establishing baseline security requirements for public-sector organizations and operators of critical infrastructure. The 2016 version defined organizational, technical, and operational security controls applicable to government agencies and essential service providers.

- **Référentiel National de Sécurité des Systèmes d'Information (version 2020):** A substantially revised and expanded edition of the national security reference framework, updated to address cloud computing, mobile security, and the evolving threat landscape. This framework aligns Algerian public-sector security requirements with international standards including ISO/IEC 27001 and NIST CSF, and establishes sector-specific requirements for critical infrastructure operators.

- **Dispositif National de Cybersécurité (Decree 20-05, 2020):** The national cybersecurity governance framework established by executive decree, creating two central pillars: (1) the **Conseil National de Cybersécurité** (National Cybersecurity Council), a cross-ministerial body chaired at the highest governmental level, responsible for cybersecurity policy definition and strategic coordination; and (2) the **Agence Nationale de Cybersécurité** (precursor to ANSI), responsible for operational implementation, technical standards, incident response coordination, and critical infrastructure protection.

- **Presidential Decree 21-255** (26 June 2021) [35]: established the National Agency for Information Systems Security (ANSI) under the Ministry of Post, Telecommunications, and Digital Technologies, with a mandate to coordinate national cybersecurity policy and protect critical information infrastructure. ANSI serves as the national CERT (Computer Emergency Response Team), issues binding security guidelines for critical sector operators, and represents Algeria in international cybersecurity cooperation forums.

- **Law 26-07 (2026) on the Regulation of Cybersecurity and the Protection of Information Systems:** The most recent and comprehensive cybersecurity legislative instrument, enacted in 2026, establishes a complete regulatory framework for cybersecurity governance in Algeria. This law defines categories of critical information infrastructure, mandates minimum security requirements for operators across public and private sectors, establishes notification obligations for significant cybersecurity incidents, formalizes the roles and powers of ANSI, and creates a structured framework for the supervision and audit of organizational cybersecurity postures. Law 26-07 represents Algeria's most significant legislative alignment with international cybersecurity regulatory frameworks to date.

- **National Cybersecurity Strategy 2025--2029:** Algeria's medium-term national cybersecurity strategy, defining five strategic axes: (1) strengthening the protection of critical national infrastructure; (2) developing national cybersecurity capacity and human capital; (3) fostering a national cybersecurity industry and reducing strategic dependence on foreign technologies; (4) strengthening international cybersecurity cooperation at bilateral and multilateral levels; and (5) raising cybersecurity awareness among citizens, enterprises, and public institutions. The strategy sets measurable targets for increasing the number of certified security professionals, qualifying national cybersecurity solution providers, and achieving defined resilience benchmarks for critical infrastructure sectors by 2029.

- **ANSI Annual Report 2023** [13]: documented a **67%** increase in cyberattacks targeting Algerian organizations, with critical incidents concentrated in financial institutions, energy operators, telecommunications providers, and government administrative systems.

1.9.2 Structural Challenges in National Cybersecurity Capacity

Despite institutional progress, several structural challenges continue to limit the effectiveness of Algeria's national cybersecurity posture---patterns representative of the broader situation in developing economies:

- **Foreign tool dependency:** The majority of security tools deployed by Algerian organizations are developed by foreign vendors. Software updates, threat intelligence feeds, and support channels pass through foreign infrastructure, creating sovereignty exposure.

- **Hard-currency licensing costs:** Enterprise security software is priced in USD or EUR. Exchange rate dynamics significantly increase the effective local currency cost, forcing budget-constrained organizations to operate with insufficient tooling.

- **Cybersecurity talent gap:** Specialized security skills---threat hunting, incident response, penetration testing, secure development---are in critically short supply. University programs are expanding but lag behind the pace of threat evolution.

- **Data sovereignty risks:** Cloud-based platforms processing application traffic through foreign infrastructure create unacceptable sovereignty risks for organizations handling sensitive national data.

1.9.3 Digital Sovereignty as a Technical Imperative

> *``Digital sovereignty is not merely a political choice---it is a technical necessity. A scanning platform that sends analysis targets to a foreign cloud to retrieve results is structurally incompatible with protecting national critical infrastructure. The prohibitive costs of commercial alternatives compound this problem, making the absence of integrated, locally deployable open-source platforms a compounded strategic loss."*
> --- Adapted from ANSI Algeria Strategic Framework, 2023 [13].

1.10 Problem Statement

The preceding analysis converges on a precisely formulated research gap. The cybersecurity tooling landscape suffers from:

1. **Structural fragmentation**: no unified open-source platform integrates the six critical analysis domains.
2. **Sovereignty incompatibility**: cloud-dependent commercial solutions are incompatible with sensitive and national-context deployments.
3. **Economic exclusion**: $28,000+ annual commercial tooling costs exclude the vast majority of organizations in developing economies.
4. **Inadequate risk contextualization**: CVSS-only scoring without KEV integration, environmental context, or attack chain analysis produces systematically inaccurate remediation priorities.

The central research question is therefore:

> *How can a unified, open-source, locally deployable cybersecurity analysis platform be designed and implemented to provide integrated DAST, SAST, TLS analysis, HTTP security header assessment, dependency scanning, and server configuration auditing---*
> *with multi-dimensional risk scoring integrating CVSS, CISA KEV, and MITRE ATT&CK intelligence---*
> *while remaining academically rigorous, financially accessible, and compliant with digital sovereignty requirements?*

Chapter Summary

This chapter established the conceptual and empirical foundations of the research through a comprehensive analysis of the current cybersecurity landscape, documented threat environment, and tooling ecosystem. The analysis demonstrated that:

1. Cybersecurity threats are documented, measurable, and accelerating--- driven by the dual forces of AI-generated vulnerable code and AI-augmented attack methodologies.
2. The security tool landscape suffers from structural fragmentation, with 61% of security teams identifying multi-tool integration complexity as a primary operational barrier.
3. Risk prioritization based solely on CVSS scores is systematically inadequate; integrating CISA KEV exploitation intelligence dramatically improves remediation effectiveness.
4. The combination of cloud dependency, cost barriers, and sovereignty requirements creates a compelling imperative for locally deployable, open-source, integrated security platforms---particularly for organizations in the Algerian national context.

These findings collectively define the design objectives and technical requirements of the platform presented in the following chapters.

Chapter 2: Methodology and System Design

2.1 Introduction

Chapter 1 established the theoretical and contextual foundation of this work: a rigorous survey of the modern threat landscape, a critical analysis of existing tooling limitations, and a precisely bounded research problem. That analysis converged on a structural gap in the open-source security ecosystem --- one that no existing platform adequately addresses. The gap is not the absence of individual security tools, which are abundant, but the absence of a unified platform that *orchestrates* those tools, synthesises their findings into a coherent risk picture, enriches that picture with AI-powered intelligence, and does so within a locally deployable, sovereignty-compliant architecture.

This chapter constitutes the direct methodological and architectural response to that problem statement. It is organised around two complementary contributions that together bridge the distance between the problem identified in Chapter 1 and the implementation presented in Chapter 3.

The first contribution --- **Part I: Methodology** --- explains the research decisions made in constructing securAX: why tools were selected rather than built, how each scanning vector was designed to address a specific OWASP Top 10 category identified in Chapter 1, how risk was formalised into a mathematically grounded scoring algorithm, and how the ARIA intelligence agent was architected to operate reliably from fully connected cloud environments down to air-gapped deployments.

The second contribution --- **Part II: System and Database Design** --- presents the complete technical architecture of securAX: the UML class model and object hierarchy, the scan lifecycle sequence, the DNS-based target authorisation workflow, the multi-factor authentication protocol, and the relational database schema. Each design element is grounded in the constraints and requirements derived in Chapter 1 and will be directly traceable to the implementation in Chapter 3.

Part I: Methodology

2.2 Research Methodology and Design Approach

2.2.1 Problem Decomposition

The central problem identified at the conclusion of Chapter 1 --- designing a unified, locally deployable, multi-vector cybersecurity analysis platform with contextual risk scoring and AI-powered remediation guidance --- is a complex engineering problem that benefits from systematic decomposition. Rather than treating securAX as a single design challenge, the problem was decomposed into four independently solvable sub-problems, each of which maps directly to a distinct architectural subsystem:

1. **Multi-vector vulnerability detection:** Chapter 1 demonstrated that real-world attack surfaces span web application behaviour, source code weaknesses, dependency supply chains, network exposure, and server misconfiguration simultaneously. The design challenge is to discover vulnerabilities across all five vectors without building custom detection logic from scratch --- which would sacrifice detection quality and community trust.

2. **Risk quantification and prioritisation:** Chapter 1 established that CVSS Base Scores, used in isolation, systematically misrepresent operational risk by ignoring exploitation evidence, asset criticality, and vulnerability density. The design challenge is to produce a single, defensible composite risk score that security teams can act on without additional manual analysis.

3. **Intelligence enrichment and remediation:** Identifying a vulnerability is only the first step; the operational gap Chapter 1 identified is the absence of contextualised, developer-ready remediation. The design challenge is to automatically transform raw finding lists into attack chains, MITRE ATT&CK mappings, compliance assessments, and copy-pasteable code fixes.

4. **Platform security and access control:** A vulnerability scanner that is itself vulnerable is both a security liability and a credibility failure. The design challenge is to harden securAX against the same vulnerability classes it detects in target systems.

2.2.2 Tool Selection Philosophy: Orchestration Over Reinvention

The most consequential methodological decision in securAX's design is the **principle of orchestration over reinvention**. Rather than developing custom vulnerability detection algorithms, securAX delegates every detection function to established, internationally recognised open-source security tools where such tools exist.

This decision is justified on three grounds that follow directly from Chapter 1's analysis of the tooling landscape. First, **detection quality**: tools such as OWASP ZAP, Nikto, Nmap, Bandit, and Semgrep represent decades of collective security research and continuous community validation. Custom scripts replicating their functionality would inevitably be less accurate and less maintained. Second, **trust and auditability**: findings attributed to industry-standard tools carry weight with security auditors, compliance assessors, and development teams in a way that findings from unknown custom scripts do not. Third, **maintainability**: as the threat landscape evolves --- a dynamic Chapter 1 documented in detail --- the underlying tools receive updates independently; securAX benefits from those updates without requiring internal vulnerability research capability.

The complete tool delegation matrix is presented in Table 2.1, alongside the specific limitation from Chapter 1 that motivated each tool selection.

Security Domain	**Tool Delegated**	**Justification**
DAST --- Active scanning	OWASP ZAP	OWASP's official DAST scanner; industry standard
DAST --- Server signatures	Nikto	30+ years of web server vulnerability signatures
Network discovery	Nmap	Definitive network scanner; used in all pen-tests
Python static analysis	Bandit	PyCQA's official Python security linter
Multi-language SAST	Semgrep	30+ language support; framework-specific rulesets
Python dependency CVEs	pip-audit	Official PyPA advisory database client
Node.js dependency CVEs	npm audit	Built-in npm security advisory integration
Real-time CVE data	NIST NVD API v2	Authoritative US government CVE database
AI analysis & narrative	Google Gemini 2.0 Flash	State-of-the-art LLM; free-tier sovereignty option
Apache/Nginx config audit	**server_int.py (custom)**	**Genuine tooling gap: no mature OSS solution exists**

Table 2.1: securAX Tool Delegation Matrix

The single exception to the delegation principle is the Apache/Nginx configuration analyser (`server_int.py`), the only custom-developed scanner in the platform. Chapter 1's com-

parative analysis of existing tools confirmed a genuine gap in the open-source ecosystem: no mature, widely adopted tool performs white-box static analysis of Apache `httpd.conf` files against the specific misconfigurations most exploited in production environments. This gap --- not a design preference --- motivated original development, described in Section 2.3.5.

2.2.3 Traceability: From Chapter 1 Requirements to Methodological Responses

The research problem formulated in Chapter 1 specified six requirements for securAX. Table 2.2 traces each requirement to its concrete methodological response, establishing the design's grounding in the preceding analysis.

Chapter 1 Requirement	**Methodological Response**
Multi-vector assessment	Six scanner modules covering DAST, SAST, dependency, network, web security headers/TLS, and server configuration --- each addressing a specific OWASP Top 10 category
Unified risk score	Six-stage pipeline: per-finding CVSS mapping with type boosters and CISA KEV $\times 2.5$ factor; exponential decay aggregation; sigmoid normalisation to 0--10; temporal adjustment; environmental modifier; severity floor guarantees; replaces naive CVSS-only prioritisation
Open-source and locally deployable	Flask Application Factory + SQLite/PostgreSQL; zero mandatory cloud dependencies; all detection scanning executes on-premises. **Note:** the WebSecurityScanner vector optionally enriches findings via external threat intelligence APIs (Qualys SSL Labs, VirusTotal, AbuseIPDB, etc.) when internet access is available; these calls are non-mandatory and each API key is operator-configurable. Operators in air-gapped or strict-sovereignty deployments may disable all external API calls, reducing the scanner to its seventeen local passive checks, which execute entirely on-premises with no data exfiltration.
Digital sovereignty	Offline rule-based ARIA fallback; Ollama local LLM alternative; no data leaves the deployment environment without explicit operator action
Exploitation intelligence	CISA KEV catalog integration (24-hour refresh), NIST NVD API v2 enrichment, and 25-keyword type booster table reflecting confirmed exploitation patterns
AI-powered remediation	ARIA agent with Gemini 2.0 Flash / Ollama / offline-fallback operating modes; MITRE ATT&CK mapping; GDPR, PCI-DSS, ISO 27001 compliance assessment

Table 2.2: Traceability from Chapter 1 Requirements to Methodological Responses

2.3 Scanning Methodology by Vector

The six scanning vectors in securAX collectively address the complete attack surface taxonomy documented in Chapter 1. Each subsection below describes the methodology for one vector, grounding the design decision in the specific vulnerability class it targets.

2.3.1 Dynamic Application Security Testing (DAST)

DAST addresses OWASP A03:2021 (Injection), A07:2021 (Identification and Authentication Failures), and A05:2021 (Security Misconfiguration) --- the three categories Chapter 1 identified as most frequently exploited in web-facing applications. securAX's DAST methodology operates in two complementary modes.

The first mode integrates **OWASP ZAP** via its REST API. ZAP performs a complete active scan cycle: automated spidering to discover all accessible endpoints, passive scanning during traffic observation, and active scanning that injects malicious payloads to confirm exploitability. The securAX integration manages the full ZAP session lifecycle --- creating a new context for each scan, configuring authentication if credentials are provided, running the spider and active scanner with a configurable timeout, and retrieving structured JSON results. This isolation ensures that findings from one scan cannot contaminate another.

The second mode executes **Nikto** via CLI with structured output parsing. Nikto's strength lies in its extensive signature database for server-level checks: outdated software identifiers in response headers, dangerous default files, exposed administrative interfaces, and HTTP method policy weaknesses. Output is normalised into securAX's unified finding schema with severity mapping applied from Nikto's internal rating categories.

Note on Wapiti. Chapter 1 (Table 1.4) surveyed Wapiti as a capable open-source DAST crawler and fuzzer covering SQLi, XSS, CSRF, SSRF, and LFI. Wapiti was evaluated as a potential complement to ZAP but was ultimately not integrated into the DAST pipeline for three reasons: (1) its crawling model overlaps substantially with ZAP's spider mode, making the two tools redundant on standard applications; (2) Wapiti's JSON output schema undergoes non-backward-compatible changes across minor versions, which would require maintenance effort disproportionate to the marginal coverage gain; and (3) the ZAP + Nikto combination already addresses all OWASP categories that Wapiti targets, with superior CI/CD integration. Wapiti remains a valid standalone tool for targeted manual assessments, and its integration is documented as a potential future extension in Chapter 4.

In parallel, the **WebSecurityScanner** (`web_scanner.py`) operates as a **dedicated, independent scanning vector** --- not a sub-mode of DAST --- providing passive HTTP analysis without active payloads. This separation is architecturally deliberate: the WebSecurityScanner is safe against production environments where ZAP's active mode could be disruptive,

and its intelligence surface (external threat APIs, TLS grading, certificate transparency, IP reputation) extends well beyond what DAST active scanning provides. It checks TLS certificate validity, HTTP security header completeness, sensitive file exposure indicators, cookie security attributes, CORS policy correctness, response-analysis indicators of SQL injection or IDOR vulnerabilities, and orchestrates parallel calls to external threat intelligence APIs. Its full scope and independent architecture are described in Chapter 3.

2.3.2 Static Application Security Testing (SAST)

SAST addresses OWASP A01:2021 (Broken Access Control) and A02:2021 (Cryptographic Failures) at the source code level --- vulnerability classes that dynamic testing cannot reliably detect because they manifest only under specific runtime conditions. The SAST methodology accepts a ZIP archive of source code, extracted to a sandboxed temporary directory with ZIP bomb protection (maximum 150 MB uncompressed).

Bandit performs Python-specific analysis across 40+ security test categories: hardcoded credentials (B105, B106, B107), SQL injection via string formatting (B608), weak cryptography (B303--B305), unsafe subprocess invocation (B603, B604), and binding to all interfaces (B104). Each finding is mapped to its CWE identifier.

Semgrep complements Bandit with multi-language pattern matching using the `p/python` and `p/secrets` rulesets. Where Bandit applies fixed rules compiled into the tool, Semgrep applies community-maintained patterns that can be updated independently and extended with organisation-specific rules. Their combination captures vulnerability classes that neither tool reliably detects alone.

2.3.3 Dependency Vulnerability Analysis

Dependency analysis addresses OWASP A06:2021 (Vulnerable and Outdated Components), which Chapter 1 identified as one of the most consistently underestimated risk vectors in modern software supply chains.

For Python projects, **pip-audit** is invoked against the uploaded `requirements.txt` or `pyproject.toml`, querying the Python Advisory Database (PyAD) and cross-referencing with NIST NVD to return CVE identifiers, CVSS scores, affected version ranges, and patched version recommendations. For Node.js projects, **npm audit** queries GitHub's Security Advisory Database against the uploaded `package.json` and lockfile. Both tools produce structured JSON output normalised into unified findings with severity classification, CVE references, and direct upgrade-path remediation.

2.3.4 Network Security Analysis

Network analysis addresses attack surface exposure at the infrastructure layer --- specifically, the uncontrolled service exposure that Chapter 1 identified as a primary vector for lateral movement and initial access in enterprise breach scenarios.

Nmap is integrated via `python-nmap`, supporting two profiles. The **external profile** performs TCP SYN scanning, service version detection (`-sV`), and OS fingerprinting against internet-facing hosts, replicating the view of an internet-based attacker. The **internal profile** performs comprehensive LAN scanning including UDP port sampling and script scanning (`-sC`). Nmap XML output is enriched with known vulnerability associations: open port 3306 on an internet-facing host triggers a critical database exposure finding; exposed management ports (8080, 8443, 9200, 5432, 27017) trigger service-specific advisories.

2.3.5 Internal Server Configuration Analysis

The internal server configuration scanner (`server_int.py`) is securAX's most distinctive and original contribution at the scanning layer. It addresses a genuine gap confirmed by Chapter 1's tooling survey: no mature open-source tool performs white-box static analysis of Apache `httpd.conf` and Nginx `nginx.conf` files against the CIS Apache HTTP Server 2.4 Benchmark v2.3.0 L2 (2026).

The methodology involves three stages. In the **parsing stage**, the configuration file is tokenised into a structured representation of directives, values, and line numbers, correctly handling block contexts (`<Directory>`, `<VirtualHost>`, `<Location>`) and ignoring comments. In the **analysis stage**, **32 Apache checks and 13 Nginx checks** are applied, covering server information disclosure (`ServerTokens`, `ServerSignature`), directory listing (`Options Indexes`), symbolic link traversal (`FollowSymLinks`), deprecated TLS versions (SSLv2/3, TLS 1.0/1.1), weak cipher suites (RC4, DES, NULL, EXPORT-grade), dangerous modules (`mod_status`, `mod_info`, `mod_userdir`), HTTP TRACE method, missing request size limits, and the absence of `mod_security`. In the **remediation generation stage**, a corrected configuration file is produced in memory with all misconfigurations replaced by secure equivalents and change annotations added as comments.

This last stage --- automated corrected-configuration generation --- transforms the scanner's output from an identification report into an immediately deployable remediation artifact. The complete workflow is illustrated in Figure 2.1.

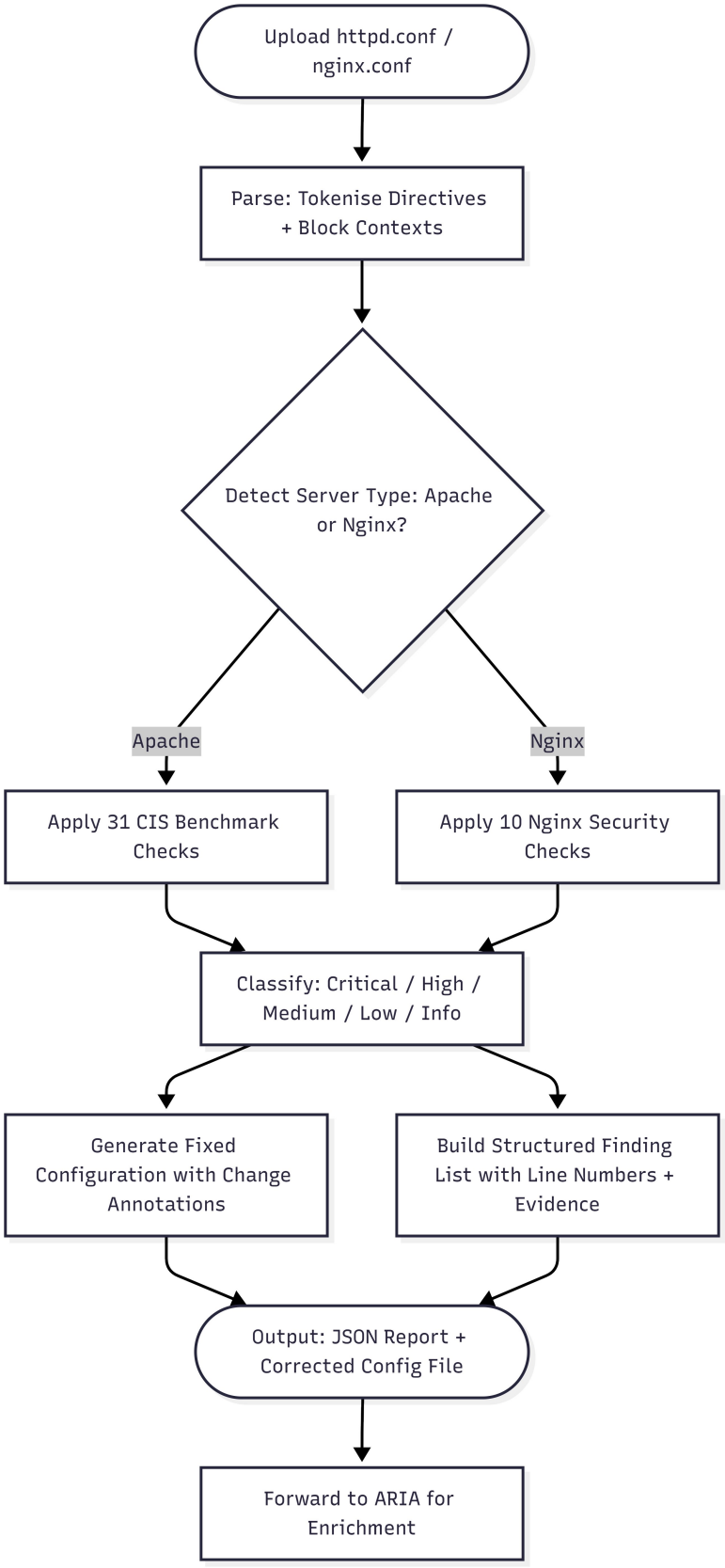

Figure 2.1: Internal Server Configuration Scanner Workflow (`server_int.py`)

2.4 Risk Quantification Methodology

2.4.1 Limitations of Naive CVSS Application

Chapter 1 established that CVSS Base Scores, applied in isolation, constitute an inadequate prioritisation mechanism. A vulnerability with CVSS 9.8 on an isolated internal server protected by a compensating control presents demonstrably lower operational risk than a CVSS 7.0 vulnerability actively listed in the CISA Known Exploited Vulnerabilities catalog on an internet-facing payment gateway. This distinction --- which CVSS cannot express --- is precisely what securAX's risk engine is designed to capture.

Three specific CVSS limitations motivated the design of a custom risk engine. First, CVSS assesses individual vulnerabilities in isolation and cannot model the cumulative exposure created by a high density of co-located findings. Second, CVSS does not incorporate exploitation evidence: whether a vulnerability has been actively weaponised in the wild is absent from the Base Score. Third, the Temporal and Environmental metric groups that CVSS v3 provides to address these limitations are almost universally omitted in practice because they require per-environment data collection that most organisations cannot sustain.

2.4.2 securAX Risk Scoring Algorithm

securAX implements a six-stage risk aggregation pipeline grounded in CVSS v3.1 methodology and extended with contextual intelligence. The pipeline is formally specified below; its activity diagram is presented in Figure 2.2, immediately following the specification.

Stage 1 --- Per-Finding Score Computation. Each finding's severity label is mapped to its CVSS v3.1 representative base score, then multiplied by two modifiers applied in sequence. The **vulnerability type booster** T_i is drawn from a 25-keyword lookup table matching against the finding's check identifier, title, and description: Remote Code Execution $\times 2.0$; SQL Injection $\times 1.8$; Authentication Bypass $\times 1.7$; Hardcoded Secrets $\times 1.7$; XXE/SSRF $\times 1.5$; Path Traversal $\times 1.4$; XSS $\times 1.2$; Missing Header $\times 0.7$; default $\times 1.0$. The **KEV exploitation factor** K_i is 2.5 if the finding carries a CVE identifier listed in the CISA Known Exploited Vulnerabilities catalog (refreshed every 24 hours via a background daemon thread), and 1.0 otherwise --- with the $2.5\times$ multiplier applied exclusively to critical and high severity findings to maximise remediation signal where it matters most. The individual finding score is therefore:

$$s_i = \min(\text{CVSS}_i \times T_i \times K_i,\ 10.0) \tag{2.1}$$

Representative CVSS mapping: Critical $= 9.5$; High $= 7.5$; Medium $= 5.0$; Low $= 2.0$; Info $= 0.0$.

Stage 2 --- Exponential Decay Aggregation. The scored findings are sorted in descend-

ing order. Aggregation uses an exponential decay function with decay factor $\delta = 0.85$, ensuring that each successive finding contributes less than the preceding one. This prevents linear score inflation while preserving the risk signal of high-density finding sets:

$$R_{\text{raw}} = \sum_{i=0}^{n-1} s_i \cdot \delta^i = s_0 + 0.85\, s_1 + 0.85^2\, s_2 + \cdots \tag{2.2}$$

Stage 3 --- Sigmoid Normalisation. The raw aggregate is mapped to the bounded $[0, 10]$ interval using a negative exponential sigmoid. The scaling constant $\lambda = 15$ was calibrated empirically against representative scan datasets to produce intuitive score distributions across realistic finding counts:

$$B = 10.0 \times \left(1 - e^{-R_{\text{raw}}/15}\right) \tag{2.3}$$

Stage 4 --- Temporal Adjustment. The normalised base score receives a multiplicative upward adjustment reflecting the presence of exploitation evidence: $+5\%$ if any finding carries a CVE identifier (confirmed vulnerability), $+15\%$ additional if any CVE appears in the CISA KEV catalog (confirmed active weaponisation):

$$T = B \times m_{\text{temporal}}, \quad m_{\text{temporal}} \in \{1.0,\ 1.05,\ 1.20\} \tag{2.4}$$

Stage 5 --- Environmental Adjustment. The temporal score is multiplied by a composite environmental modifier combining deployment criticality, internet exposure, data sensitivity, and scan-type confidence weight:

$$E = \min(T \times c \times e_{\text{facing}} \times e_{\text{pii}} \times e_{\text{pay}} \times w_{\text{type}},\ 10.0) \tag{2.5}$$

where $c \in [0.1, 1.0]$ is the operator-supplied criticality, $e_{\text{facing}} = 1.20$ for internet-facing targets, $e_{\text{pii}} = 1.15$ when PII is present, $e_{\text{pay}} = 1.20$ when payment data is in scope, and w_{type} is a per-scan-type confidence weight (DAST $= 1.0$; server config $= 0.9$; network $= 0.8$; web passive $= 0.7$).

Stage 6 --- Severity Floor Guarantees. To ensure that the composite score can never understate the severity of the most dangerous finding classes, two unconditional floors are applied after the environmental step: scans with at least one *critical* finding are guaranteed a minimum score of 7.5; scans with at least one *high* finding are guaranteed a minimum of 5.0. The final score is:

$$R = \min(\max(E,\ F_{\text{sev}}),\ 10.0) \tag{2.6}$$

where $F_{\text{sev}} = 7.5$ if any critical finding is present, 5.0 if the highest severity is high, and 0 otherwise.

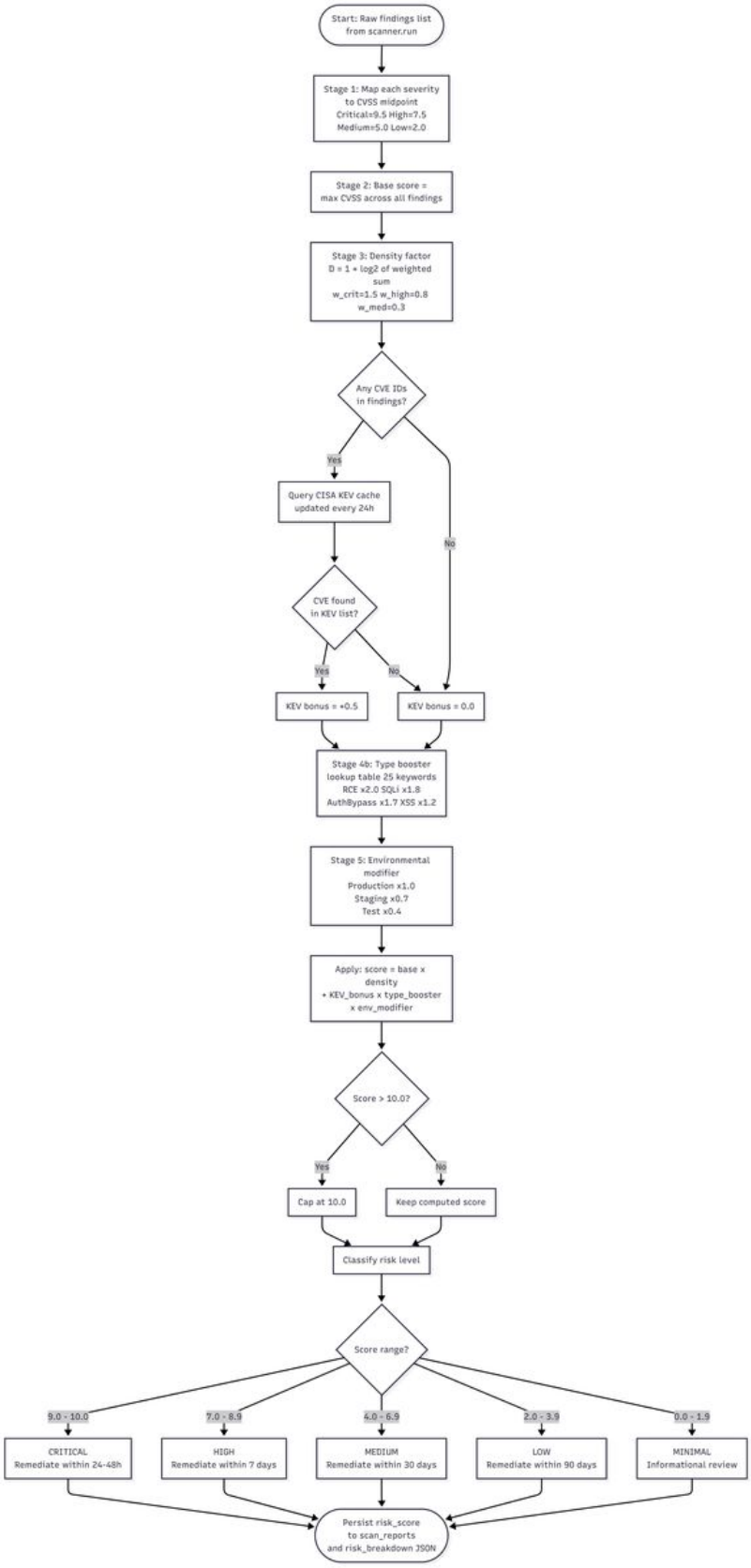

Figure 2.2: securAX Multi-Dimensional Risk Scoring Pipeline (Activity Diagram)

2.4.3 Risk Level Classification

The final score is mapped to five qualitative tiers aligned with CVSS v3.1 ratings, extended by a ``Minimal'' level below the standard range. Each tier carries a concrete remediation timeline that security managers can communicate directly to responsible teams.

Score Range	Risk Level	Recommended Action
0.0 -- 1.9	Minimal	Informational review; no immediate action required
2.0 -- 3.9	Low	Schedule remediation within 90 days
4.0 -- 6.9	Medium	Remediate within 30 days; assign to responsible team
7.0 -- 8.9	High	Immediate remediation within 7 days; escalate to management
9.0 -- 10.0	Critical	Emergency response; remediate or mitigate within 24--48 hours

Table 2.3: Risk Level Classification and Recommended Response Timeline

2.5 ARIA: Autonomous Risk Intelligence Agent

2.5.1 Design Philosophy

The operational gap Chapter 1 identified was not merely the absence of vulnerability detection, but the absence of *contextualised, actionable intelligence* derived from those detections. Security teams receiving long lists of findings without attack context, MITRE ATT&CK mappings, or developer-ready remediation code face an analyst burden that often causes critical findings to remain unremediated for months.

ARIA (Artificial Reconnaissance and Intelligence Advisor) is designed as an **intelligence layer**, not a chatbot. It activates automatically after every completed scan and produces five distinct enrichment outputs without requiring manual analyst invocation. To preserve securAX's sovereignty guarantee --- established as a hard requirement in Chapter 1 --- ARIA operates across three modes, selected automatically in the following priority order: **(1) Ollama** (local LLM, highest priority --- all inference executes within the operator's environment with zero external data transmission, fully satisfying the digital sovereignty requirement), **(2) Google Gemini 2.0 Flash** (cloud LLM, used only when Ollama is not configured), and **(3) an offline rule-based engine** (zero-connectivity fallback covering the 15 most common finding types with copy-pasteable remediation). Ollama is always preferred over Gemini because routing vulnerability findings to an external cloud LLM constitutes a sovereignty exposure incompatible with the platform's core design requirement.

Current Status: Gemini Mode Suspended; Ollama Mode Operational. ARIA's Gemini 2.0 Flash operating mode has been deliberately suspended from the production deployment at the time of this writing. This decision follows the same digital sovereignty principle

that governed the rejection of Nuclei's Cloud API in the DAST pipeline (documented in Chapter 3): any component that transmits analysis artefacts --- vulnerability findings, target identifiers, or configuration details --- to infrastructure outside the operator's controlled environment creates a sovereignty exposure that is structurally incompatible with the platform's core design requirement. Operators who deploy a local Ollama instance (Mistral, LLaMA, or any Ollama-compatible model) recover the full AI-powered pipeline with zero data exfiltration. A data-minimisation redesign of the Gemini integration --- transmitting only anonymised severity summaries rather than raw findings --- is planned as a future deliverable before Gemini mode is restored; it is documented in Chapter 4.

In its current form, when the Ollama local LLM is not configured, ARIA falls back to the offline rule-based engine, which operates with zero external connectivity and zero data exfiltration. This fallback provides structured remediation guidance and MITRE ATT&CK mappings for the 15 most common finding types without any network dependency.

The planned remediation path restores full ARIA capability while preserving sovereignty through two parallel tracks: (1) deeper integration of the Ollama local LLM provider --- enabling operators who deploy a local model (Mistral, LLaMA, or any Ollama-compatible model) to recover the full LLM-powered pipeline with zero external data transmission; and (2) a data minimisation redesign of the Gemini integration that transmits only anonymised, severity-classified finding summaries rather than raw scan output containing target identifiers, thereby reducing the sovereignty exposure to an acceptable level for operators whose deployment policy permits external LLM enrichment under explicit consent. These improvements are documented as a priority deliverable in Chapter 4.

2.5.2 ARIA Five-Stage Pipeline

Stage 1 --- NVD CVE Enrichment. ARIA extracts all CVE identifiers from scan findings and queries the NIST NVD API v2 in real time, retrieving official CVSS v3.1 scores and vector strings, English vulnerability descriptions, associated CWE identifiers, and NVD publication dates. NVD-sourced CVSS scores replace scanner-estimated severities, ensuring that the report reflects government-authoritative ratings rather than tool approximations.

Stage 2 --- MITRE ATT&CK Mapping. A 30+ keyword mapping table associates vulnerability types with MITRE ATT&CK technique identifiers (T-codes) and tactic names. Every finding is presented not only as a technical weakness but as a concrete adversarial technique --- ``SQL Injection $\rightarrow$ T1190: Exploit Public-Facing Application (Initial Access)'' --- giving both developers and management the threat context needed to understand real-world impact.

Stage 3 --- Attack Chain Generation. Rather than presenting vulnerabilities as isolated findings, ARIA constructs a step-by-step attack chain narrative showing how a skilled at-

tacker would chain discoveries to progressively compromise the target. With LLM access (Gemini or Ollama), the chain is generated through prompt-engineered inference with the full finding set as context. In offline mode, a rule-based engine constructs a structured chain ordered by MITRE ATT&CK tactic phase: Reconnaissance → Initial Access → Execution → Privilege Escalation → Exfiltration.

Stage 4 --- Prioritised Remediation Plan. ARIA generates a remediation plan ordered by business impact rather than CVSS score alone. Each item includes a specific code fix in the appropriate language (Python, Bash, Apache configuration) with before/after code blocks where applicable.

Stage 5 --- Compliance Assessment. ARIA evaluates the finding set against GDPR Article 32 (security of processing), PCI-DSS 4.0 Requirements 6.3.1/6.3.2 (secure software development), and ISO 27001:2022 Annex A.8.25 (secure development life cycle). Each assessment identifies at-risk control requirements, regulatory implications, and the remediation action that restores compliance --- producing output that legal and management teams can act on without requiring security expertise.

The complete pipeline is illustrated in Figure 2.3.

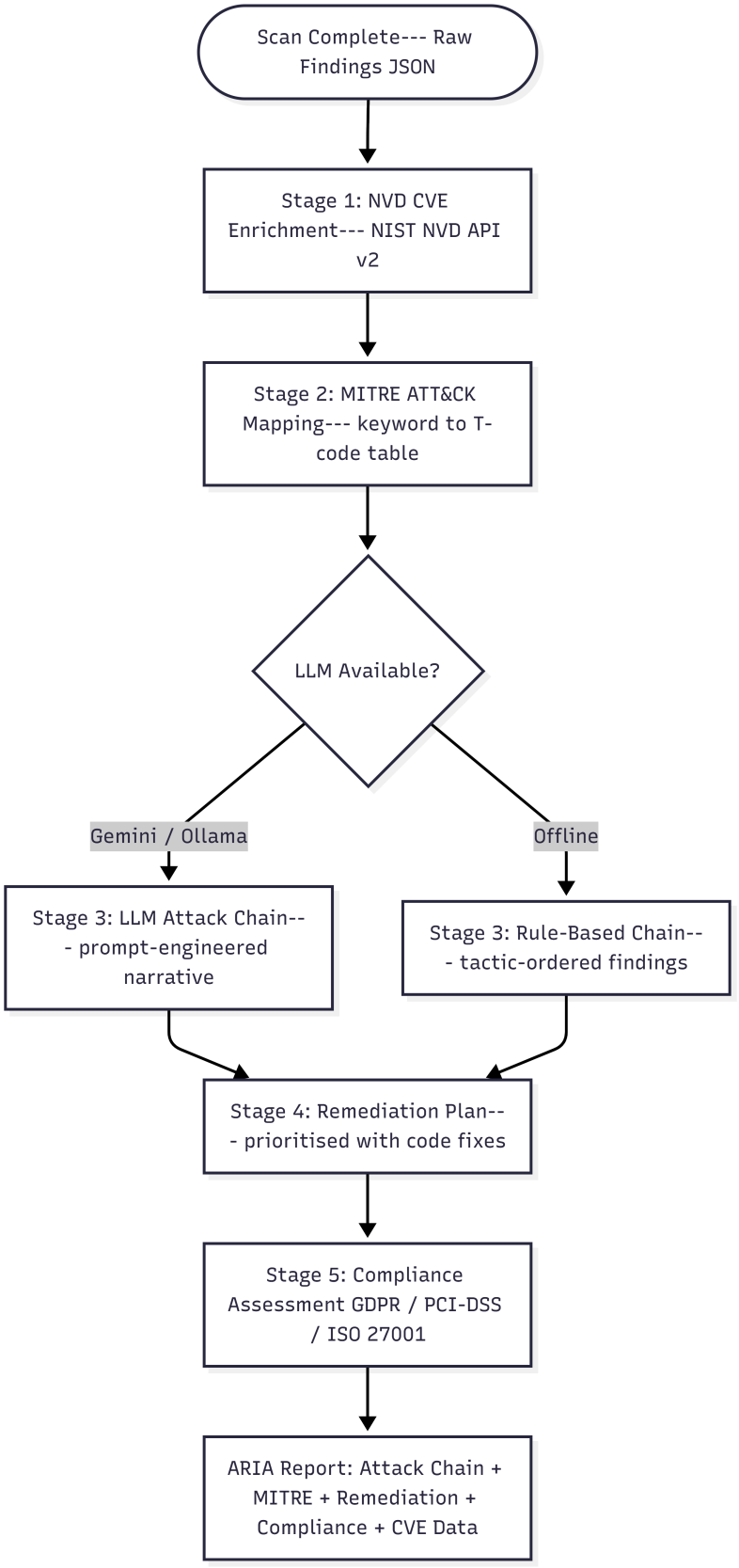

Figure 2.3: ARIA Autonomous Risk Intelligence Agent --- Five-Stage Processing Pipeline

Part II: System and Database Design

2.6 Object-Oriented Class Model

The object-oriented design of securAX is formalised in the UML class diagram presented in Figure 2.4. This diagram is the authoritative reference for all entity relationships, inheritance hierarchies, and method contracts governing the implementation in Chapter 3. It reveals five structural zones mapped to the four sub-problems identified in Section 2.1. The mapping is not strictly one-to-one: Zone 3 (Scanner Hierarchy) serves both sub-problem 1 (multi-vector vulnerability detection) and sub-problem 3 (intelligence enrichment), because each scanner subclass is both a detection unit and the source of findings that ARIA enriches. Zones 1, 2, 4, and 5 each map exclusively to one sub-problem.

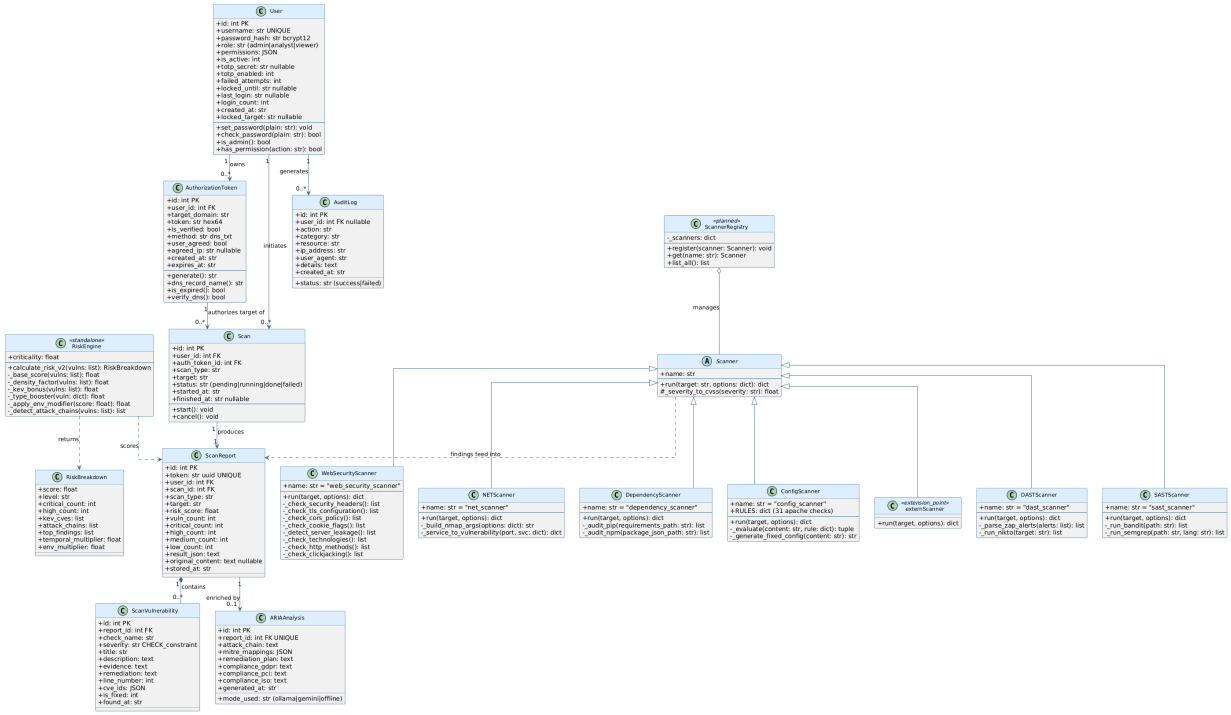

Figure 2.4: securAX UML Class Diagram

Zone 1 --- User and Access Control. The `User` entity is the root of the authorisation model, exposing `set_password()`, `check_password()`, `is_admin()`, and `has_permission()` methods. Role is stored as a string (`admin`/`analyst`/`viewer`) and fine-grained permissions as a JSON array, enabling flexible per-user capability grants without schema changes. `User` generates `AuditLog` entries for every security-relevant action and owns zero or more `AuthorizationToken` instances, each tied to a specific target domain and subject to DNS TXT verification before scan authorisation is granted.

Zone 2 --- Scan Report and Vulnerability. `ScanReport` aggregates all output from a single scan execution: a UUID token (replacing sequential integer IDs in URLs to prevent IDOR enumeration), the computed risk score, pre-computed per-severity counts, and the

full result JSON blob. It contains a one-to-many collection of `ScanVulnerability` records (each carrying `check_name`, `severity` with CHECK constraint, `title`, `description`, `evidence`, `remediation`, `cve_ids` JSON array, `line_number`, and `is_fixed` flag) and a one-to-one link to the `aria_analyses` cache entry.

Zone 3 --- Scanner Hierarchy. The abstract `Scanner` base class defines two contract methods: `run(target, options) : dict` and the protected helper `_severity_to_cvss(severity) : float`. Six concrete subclasses implement this interface: `DASTScanner` (ZAP + Nikto), `SASTScanner` (Bandit + Semgrep), `WebSecurityScanner` (headers, TLS, CORS, cookies, clickjacking), `NETScanner` (Nmap), `DependencyScanner` (pip-audit, npm audit), and `ConfigScanner` (**32 Apache + 13 Nginx checks** via `server_int.py` with `generate_fixed_config()`). An `externScanner` stub reserves the extension point for future third-party integrations.

Current Implementation Note. In the current production codebase, scanners are implemented as standalone modules exposing a single `run_*()` function rather than as formal subclasses of an abstract base class. The UML hierarchy above represents the *target design* that governs the logical contract between components; the refactoring to a formal class hierarchy is planned as part of the Blueprint migration described in Section 2.7.

Zone 4 --- Scanner Registry. The `ScannerRegistry` singleton manages all registered scanner instances via `register(scanner)`, `get(name)`, and `list_all()` methods. This pattern decouples scan orchestration from concrete scanner implementations: adding a new scanner requires only registering a new subclass instance, with no changes to orchestration logic.

Current Implementation Note. The `ScannerRegistry` singleton is a planned architectural component not yet materialised in the production codebase. Currently, scanner invocation is handled by a direct conditional dispatch block in `app.py` that maps `scan_type` strings to function calls. The planned migration to a formal `ScannerRegistry` is a three-step process: (1) define the abstract `Scanner` base class in `scanners/base.py`; (2) refactor each `run_*()` function into a `run()` method on a concrete subclass; (3) replace the conditional dispatch block with `ScannerRegistry.get(scan_type).run(target, options)`. This migration carries zero risk to existing detection logic --- it is a structural reorganisation with no changes to the underlying tool invocations or finding schemas. It is documented in Chapter 4 as a planned architectural improvement.

Zone 5 --- Risk Engine. The `RiskEngine` module implements the six-stage scoring pipeline through the `calculate_risk_v2()` public entry point, which orchestrates: per-finding score computation (`base × type_booster × kev_factor`), exponential decay aggregation, sigmoid normalisation, temporal adjustment, environmental modifier, and severity floor guarantees. The pipeline returns a structured `RiskBreakdown` dataclass containing: the final composite score (0.0--10.0), the qualitative risk level (Minimal / Low / Medium / High

/ Critical), per-severity counts (critical, high, medium, low, info), the top-ranked findings by score, detected attack chains, matched CISA KEV CVE identifiers, the temporal and environmental multipliers applied, and prioritised remediation recommendations. Its association to `ScanReport` completes the data flow from raw findings to persisted composite risk score.

2.7 Overall System Architecture

2.7.1 Architectural Pattern and Deployment

securAX's backend is implemented as a **flat monolithic Flask application** in a single `app.py` module. All routes, security middleware, scanner invocations, and session management reside in a single deployable unit. This architecture was chosen for three operational reasons grounded in Chapter 1's sovereignty and simplicity constraints: it eliminates inter-service network latency entirely, reduces deployment complexity to a single `gunicorn app:app` command, and ensures that all processing remains within a single controlled process boundary.

Migration Roadmap: Blueprint Refactoring. The current flat architecture is a deliberate starting point, not a permanent design decision. The planned next architectural step is refactoring `app.py` into four Flask Blueprints --- `auth` (login, logout, TOTP, password management), `main` (dashboard, scan execution, report rendering), `admin` (user management, audit log, platform stats), and `api` (JSON endpoints for the React frontend) --- each enforcing permissions through the `require_permission()` decorator. This refactoring would be introduced using the Application Factory pattern (`create_app()` in `app.py`), enabling multiple isolated application instances for automated testing and clean configuration injection from environment variables. The single-file implementation currently in production is functionally equivalent to the Blueprint model; the refactoring is a structural improvement that does not change any security control, route logic, or database interaction. It is documented in Chapter 4 as a planned engineering improvement.

The production deployment architecture positions Nginx as the SSL-terminating reverse proxy serving static assets and forwarding application traffic to Gunicorn (4 WSGI workers on `127.0.0.1:5000`). All scanner modules, the Risk Engine, and the ARIA agent execute within the Flask process, sharing the SQLite data layer.

2.7.2 Scan Lifecycle Sequence

The complete scan execution workflow --- from HTTP request to persisted risk score --- is formalised in the UML sequence diagram presented in Figure 2.5. This diagram traces the interaction among six components: Browser, Flask, AuthCheck, ScanOrchestrator, RiskEngine,

and Database.

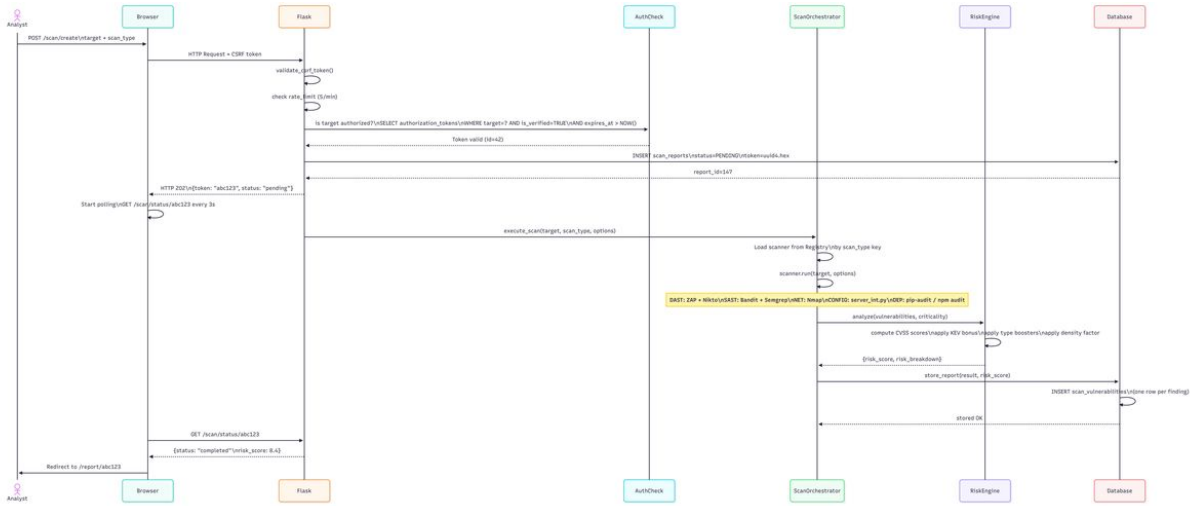

Figure 2.5: Scan Lifecycle Sequence Diagram

The workflow proceeds through the following stages. The Analyst submits a scan request with the target and scan type, accompanied by a CSRF token validated by Flask-WTF. The rate limiter enforces a maximum of 5 scan requests per minute per authenticated user. An anti-SSRF check validates that the target is not an RFC1918 private address or loopback, rejecting internal-target requests with 400. Flask dispatches to the appropriate scanner function based on `scan_type` (`run_web_scan`, `run_dast_scan`, `run_nmap_scan`, `run_sast_scan`, `run_dep_scan`, or `run_server_config_scan`), executing the underlying tools directly. `calculate_risk_v2()` processes the normalised findings through the six-stage pipeline and returns a `RiskBreakdown` dataclass. The result is persisted via `store_report()`, inserting one row per finding into `scan_vulnerabilities` and writing the complete scan record into `scan_reports` with the computed risk score. The JSON response is returned to the React frontend for dashboard rendering.

2.8 DNS-Based Target Authorisation

Before any scan can be initiated, securAX requires the analyst to prove ownership or authorisation of the target domain. This mechanism directly addresses a trust boundary problem identified in Chapter 1: a scanning platform with no authorisation gate can be weaponised against systems the operator does not own, creating legal liability and enabling targeted denial-of-service.

2.8.1 Designed Workflow

The complete authorisation workflow designed for securAX is presented in Figure 2.6. The design specifies a DNS TXT record challenge mechanism: the platform generates a 64-character hex token via `secrets.token_hex(32)`, instructs the analyst to publish it as a `_securAX.domain.com TXT` record at their DNS registrar, and verifies ownership by resolving that record using `dnspython`. A verified, non-expired `AuthorizationToken` record in the database grants scan authorisation for 30 days.

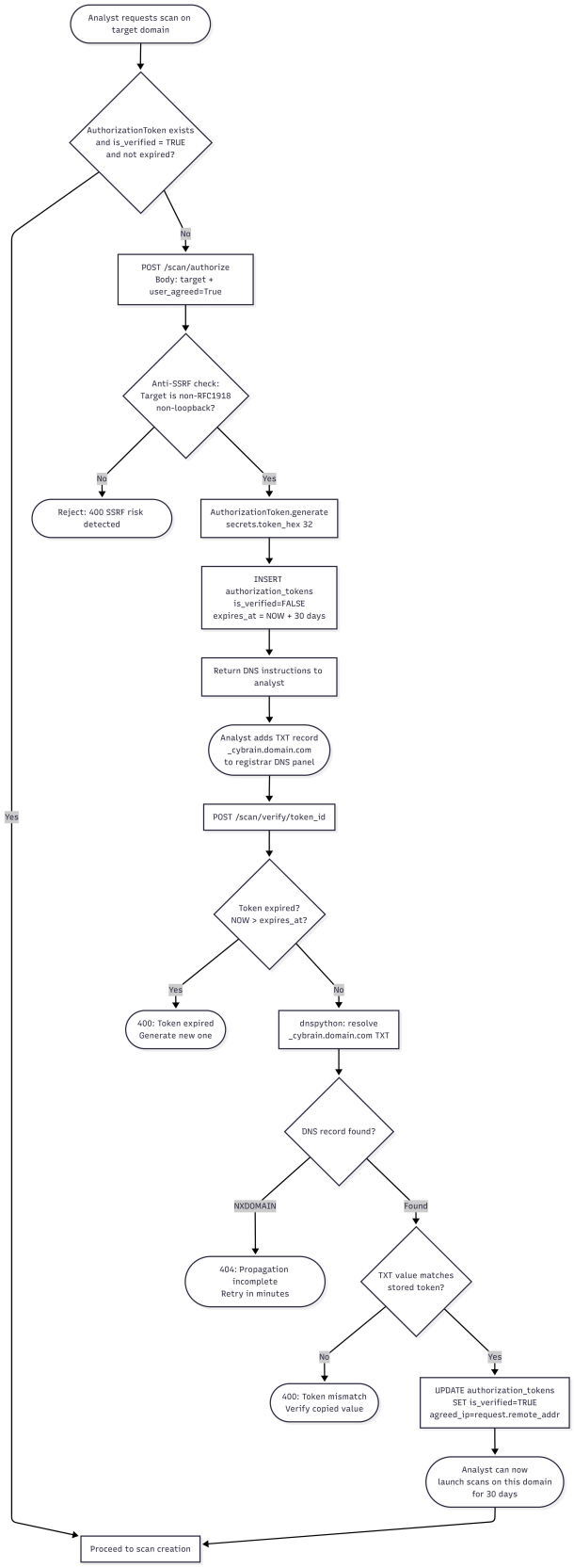

Figure 2.6: DNS-Based Target Authorisation Activity Diagram (Designed Workflow)

2.8.2 Current Implementation and Gap Declaration

The DNS TXT verification workflow described above, and the `authorization_tokens` table it depends upon, have **not yet been implemented** in the production codebase. The current implementation uses a simpler target-locking mechanism: an analyst account's first scan establishes a `locked_target` field in the `users` table, binding that account to a single target for all subsequent scans. This provides a basic form of scope control --- an analyst cannot silently redirect their account to scan a different target --- but it does not constitute cryptographic ownership proof.

The gap has three practical implications. First, an admin creating an analyst account currently sets the target scope by configuration rather than by cryptographic verification, creating a trust dependency on the admin. Second, the 30-day expiry and DNS propagation delay handling described in the designed workflow are not enforced. Third, the anti-SSRF check (rejecting RFC1918 and loopback addresses) *is* implemented in the current codebase and provides a meaningful compensating control by preventing the most common misuse vector.

2.8.3 Implementation Challenges and Planned Completion

The primary technical challenges that deferred this feature are: (1) DNS propagation delays of up to 48 hours create a difficult user experience in a verification flow; (2) some target environments use split-horizon DNS, making external TXT resolution unreliable as an ownership proof; (3) the `dnspython` integration requires careful timeout handling to prevent scanner threads from blocking on DNS queries; and (4) the `authorization_tokens` schema extension requires a coordinated database migration alongside the existing schema.

The planned implementation path follows the designed workflow precisely: introduce the `authorization_tokens` table via a versioned migration, implement `POST /scan/authorize` and `POST /scan/verify/{token_id}` routes, integrate `dnspython` for TXT resolution with a 10-second timeout and NXDOMAIN/mismatch error handling, and replace the `locked_target` column with a many-to-one relationship from `authorization_tokens` to `users`. This feature is documented as a priority deliverable in Chapter 4.

2.9 Authentication and Authorisation Design

2.9.1 Multi-Factor Authentication Protocol

The authentication protocol implements a three-step process --- credential verification, account lockout checking, and optional TOTP second factor --- designed to resist the attack

classes Chapter 1 documented as most commonly targeting security platform credentials: credential stuffing, brute force, and timing-based username enumeration.

This protocol operates independently of the DNS authorisation mechanism described in the preceding section: DNS authorisation gates the *scan target* (proving the analyst owns the domain being scanned), while the authentication protocol gates *platform access* (proving the user is who they claim to be). Both layers must pass before any scan can be initiated. The complete authentication sequence is formalised in Figure 2.7.

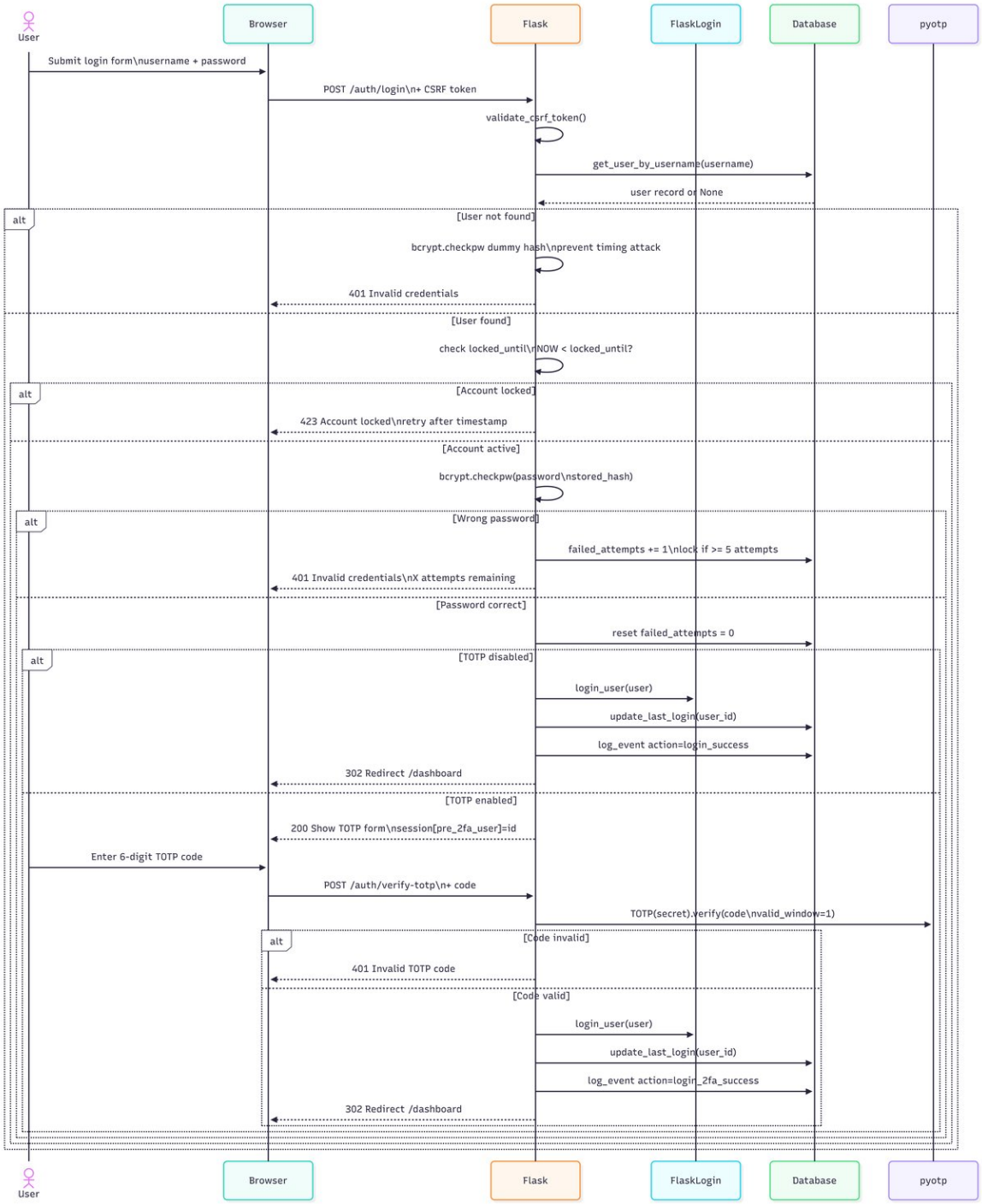

Figure 2.7: Authentication and Multi-Factor Authentication Sequence Diagram

The `authenticate_user()` function in `models.py` implements the following controls in the order shown in Figure 2.7. **CSRF validation** (`validate_csrf_token()`) is performed first; requests without a valid token are rejected before any credential processing. **Timing attack prevention**: when a username is not found, `bcrypt.checkpw()` is executed against a dummy hash to ensure a constant-time response, preventing username enumeration through response timing differences. **Account lockout**: `locked_until` is checked before password verification; if `NOW < locked_until`, `HTTP 423` is returned with the retry timestamp. Each failed password attempt increments `failed_attempts`; at five failures the account is locked. **TOTP-based MFA**: when the password is correct and `totp_enabled = TRUE`, the user ID is stored in `session[pre_2fa_user]` and the TOTP form is returned (`HTTP 200`) without creating an authenticated session. The 6-digit code submitted to `POST /auth/verify-totp` is validated by `pyotp.TOTP(secret).verify(code, valid_window=1)`, accepting the current and adjacent 30-second windows to tolerate clock skew. **Audit logging**: every outcome (`login_success`, `login_2fa_success`, `login_failed`) is persisted to `audit_logs` with timestamp, IP address, and user agent. **Session finalisation**: `login_user(user)` establishes the Flask-Login session and the browser is redirected to `/dashboard` with `HTTP 302`.

2.9.2 Role-Based Access Control

Three roles are defined with a permission matrix enforced at the route level through the `require_permission()` decorator. Admin routes additionally require `role == 'admin'`, providing defence in depth against privilege escalation through direct permission manipulation.

Role	run_scan	view_reports	manage_users	view_audit
admin	✓	✓	✓	✓
analyst	✓	✓		
viewer		✓		

Table 2.4: RBAC Role-Permission Matrix

The **viewer** role is designed for controlled report sharing with technical teams. An analyst can create a limited number of viewer accounts (maximum configurable by the admin, default: 5 per analyst) and share them with developers, system administrators, or compliance reviewers who need to consult scan reports without any scanning capability. Critically, a viewer account created by a given analyst can *only* access reports generated by that analyst

--- enforced at the query level by filtering `scan_reports.user_id` against the owning analyst's ID. This scope restriction ensures that inter-team report sharing does not create cross-analyst data exposure. Viewers have no access to the scan interface, the audit log, or user management, and cannot modify any finding status.

This delegation model addresses a practical operational need identified during system design: security analysts frequently need to share a scan report with a development team for remediation without granting that team scanning permissions or exposing other analysts' reports. The viewer role provides this capability within the existing permission model without requiring a separate API or report export workflow.

2.9.3 Platform Security Controls

A cybersecurity analysis platform that is itself vulnerable would represent both a security liability and a credibility failure. securAX implements a comprehensive security hardening posture aligned with OWASP Secure Coding Guidelines and NIST SP 800-63B. The complete control set is summarised in Table 2.5.

Domain	Control	Implementation
Password security	bcrypt hashing	12 rounds; brute-force resistant
Password policy	NIST SP 800-63B	Min. 10 chars; upper/lower/digit/special required
Multi-factor auth	TOTP (RFC 6238)	pyotp 2.9.0; per-user QR code; 30-second window
Session management	Flask-Login 0.6.3	30-min idle timeout; HttpOnly + SameSite=Lax
CSRF protection	Flask-WTF 1.2.2	Tokens on all state-changing forms and AJAX
Rate limiting	Flask-Limiter 4.1.1	Login: 10/min; scans: 5/min; AI: 20/hour
IDOR prevention	UUID tokens	Report URLs use unpredictable `uuid4.hex` tokens
Input validation	WTForms + regex	Server-side validation on all scan inputs
Security headers	OWASP ASVS	X-Frame-Options: DENY, CSP, HSTS, X-Content-Type-Options
Audit logging	Append-only table	Timestamp, IP, user agent, outcome for every action
Account lockout	5-attempt policy	15-minute lockout; timing-attack-resistant dummy hash
Anti-SSRF	IP validation	Non-RFC1918, non-loopback check before authorisation

Table 2.5: securAX Platform Security Controls

2.10 Database Design

2.10.1 Schema Architecture and Integrity Model

securAX's relational schema is built around **four production tables** currently implemented in SQLite with WAL journal mode (`PRAGMA journal_mode=WAL`) --- enabling concurrent reads during write operations under multi-threaded Gunicorn --- and foreign key enforcement (`PRAGMA foreign_keys=ON`), which SQLite disables by default. Table 2.6 presents

the complete schema overview including the planned fifth table.

Table	Primary Purpose	Key Relationships
`users`	User accounts, authentication state, MFA	Root entity
`scan_reports`	Scan metadata, aggregated metrics, raw JSON	FK → users
`scan_vulnerabilities`	Normalised per-finding rows with CVE links	FK → scan_reports (CASCADE)
`audit_logs`	Immutable action audit trail	FK → users (nullable)
`aria_analyses`	Cached ARIA intelligence output per report	FK → scan_reports (1:1 UNIQUE) --- *planned*

Table 2.6: Database Schema Summary (four tables production; one planned)

2.10.2 Table Specifications

Users Table

Column	Type	Description
`id`	INTEGER PK AUTOINCREMENT	Surrogate primary key
`username`	TEXT UNIQUE NOT NULL	Unique login identifier
`password_hash`	TEXT NOT NULL	bcrypt hash (12 rounds)
`role`	TEXT NOT NULL	admin / analyst / viewer
`permissions`	TEXT NOT NULL	JSON array of permission strings
`is_active`	INTEGER NOT NULL	0 = disabled; 1 = active
`totp_secret`	TEXT NULLABLE	Base32 TOTP secret (NULL if MFA disabled)
`totp_enabled`	INTEGER NOT NULL	0 / 1 flag
`failed_attempts`	INTEGER NOT NULL	Count since last successful login
`locked_until`	TEXT NULLABLE	ISO 8601 UTC lockout expiry
`last_login`	TEXT NULLABLE	ISO 8601 UTC last successful login
`login_count`	INTEGER NOT NULL	Cumulative successful login count
`created_at`	TEXT NOT NULL	ISO 8601 UTC creation timestamp
`created_by`	TEXT NULLABLE	Username of creating admin
`email`	TEXT NULLABLE	Contact email (optional; used for notifications)
`locked_target`	TEXT NULLABLE	Target scope lock for analyst accounts (current auth mechanism)

Table 2.7: `users` Table Schema

Scan Reports Table

The `token` field serves as the public-facing report identifier in URLs, replacing the sequential integer `id` to prevent IDOR enumeration. Denormalised aggregate columns (`risk_score`, `vuln_count`, `critical_count`, etc.) are pre-computed at write time, enabling fast dashboard queries without JSON parsing on every read.

Column	Type	Description
id	INTEGER PK AUTOINCREMENT	Surrogate primary key
token	TEXT UNIQUE NOT NULL	UUID-based public report identifier
user_id	INTEGER	FK → users.id (ON DELETE SET NULL)
username	TEXT	Denormalised username for display
scan_type	TEXT	web / dast / sast / network / server_int / dep
target	TEXT	URL, filename, or IP address
risk_score	REAL NOT NULL	Composite risk score (0.0--10.0)
vuln_count	INTEGER NOT NULL	Total finding count
critical_count	INTEGER NOT NULL	Critical severity count
high_count	INTEGER NOT NULL	High severity count
medium_count	INTEGER NOT NULL	Medium severity count
low_count	INTEGER NOT NULL	Low severity count
result_json	TEXT NOT NULL	Full JSON scan output blob
original_content	TEXT NULLABLE	Uploaded file (for config/SAST scans)
stored_at	TEXT NOT NULL	ISO 8601 UTC completion timestamp

Table 2.8: scan_reports Table Schema

Layer	Technology	Version / Notes
Backend Framework	Flask Werkzeug Gunicorn	3.1.2 --- flat monolithic application 3.1.5 --- WSGI utilities 25.1.0 --- 4 WSGI workers in production
Database	SQLite	WAL mode; foreign keys via PRAGMA; primary production store
Authentication	Flask-Login bcrypt pyotp	0.6.3 --- Session management 5.0.0 --- 12 rounds password hashing 2.9.0 --- RFC 6238 TOTP
Security Middleware	Flask-WTF Flask-Limiter	1.2.2 --- CSRF protection 4.1.1 --- Rate limiting
DNS Verification	dnspython	Pure-Python TXT record resolution --- *planned*
Scanning Tools	Bandit Semgrep python-nmap	1.8.3 --- Python SAST 1.90.0 --- Multi-language SAST 0.7.1 --- Nmap Python integration
Frontend	React Vite Tailwind CSS	19.2.4 --- UI framework 8.0 --- Build toolchain 3.4.19 --- Utility-first styling
AI Integration	google-genai	$\geq$2.5.0 --- Gemini 2.0 Flash (suspended pending sovereignty review)
Development Workflow	Claude	AI-assisted drafting, review, and consistency checks
Development Workflow	vibecoding	Prompt-driven iterative development workflow
Reverse Proxy	Nginx	SSL termination, HSTS, static file serving

Table 2.9: securAX Technology Stack

Note on database strategy. The production database is SQLite with WAL journal mode, which provides adequate concurrency for the single-server deployment model and eliminates all external service dependencies, fully satisfying the digital sovereignty requirement. A

`database_supabase.py` module exists in the codebase as an experimental migration target for organisations that choose to deploy on Supabase PostgreSQL with Row Level Security; however, this module is not active in the current production path. Adopting it would introduce a mandatory cloud dependency that conflicts with the sovereignty-first architecture. The recommended scaling path, should SQLite's write throughput become a bottleneck, is migration to a self-hosted PostgreSQL instance rather than a managed cloud provider.

2.11 Conclusion

This chapter has presented the complete methodological and design foundation of securAX, structured as a direct and traceable response to the research problem established in Chapter 1.

Part I demonstrated that the **principle of orchestration over reinvention** resolves the tooling fragmentation identified in Chapter 1 without sacrificing detection quality or community trust. The scanning methodology was designed vector by vector against the OWASP Top 10 categories documented in Chapter 1. The six-stage risk scoring pipeline --- per-finding CVSS mapping with type boosters and CISA KEV $\times 2.5$ factor, exponential decay aggregation $R_{\text{raw}} = \sum s_i \cdot 0.85^i$, sigmoid normalisation $B = 10(1 - e^{-R_{\text{raw}}/15})$, temporal adjustment, environmental modifier, and severity floor guarantees --- directly addresses the CVSS-only prioritisation gap that Chapter 1 identified as the most consequential analytical limitation in existing platforms. The ARIA agent's three-mode architecture (Gemini / Ollama / offline) resolves the sovereignty constraint; the cloud-dependent mode has been deliberately suspended pending a data-minimisation redesign, with the offline fallback engine fully operational.

Part II established the complete system design. The UML class diagram (Figure 2.4) formalises the target `Scanner` inheritance hierarchy and the `ScannerRegistry` plugin pattern, with explicit acknowledgement that the current production implementation uses a functionally equivalent direct dispatch model pending architectural refactoring. The scan lifecycle sequence (Figure 2.5) documents the complete execution path from scan request to persisted risk score, covering CSRF validation, anti-SSRF checking, scanner dispatch, and six-stage risk computation. The DNS authorisation workflow (Figure 2.6) specifies the target ownership-verification design; the current implementation uses a target-locking mechanism as an interim compensating control pending full DNS TXT completion. The MFA authentication protocol (Figure 2.7) establishes that securAX's platform access controls meet the same security standards it enforces on target systems. The four-table relational schema provides the data persistence foundation with full referential integrity, cascading deletes, precomputed aggregate columns, and an append-only audit trail; the `aria_analyses` fifth table is designed and will be introduced alongside the ARIA Ollama-first pipeline in the next development iteration.

Every design decision documented here --- including the honest declaration of gaps be-

tween designed and implemented components --- is directly traceable to its corresponding implementation or planned improvement in Chapter 3 and Chapter 4.

Chapter 3: Implementation and Realization

3.1 Introduction

The preceding chapters established the theoretical landscape of modern cybersecurity threats and the architectural blueprint for addressing them. This chapter closes the loop by presenting the **concrete implementation** of the **securAX** platform --- translating every design decision into working, tested, production-ready code.

The chapter is organized into three major parts. Section 3.2 provides a visual walkthrough of every interface module, allowing the reader to experience the platform from the perspective of both the security analyst and the administrator. Section 3.3 constitutes the principal technical contribution of this work: a deep, implementation-level examination of the **Server Configuration Audit Engine** (`server_int.py`), a 2,339-line white-box analysis system that fills a genuine gap in the open-source security tooling landscape. This section includes an honest account of the engineering challenges encountered during development --- false positive problems, remediation correctness failures, and the sovereignty-driven decision to reject cloud-dependent tooling --- and how each was resolved. Section 3.4 documents the development environment and technology stack that supports the entire platform.

3.2 System Overview

This section walks through each functional module of securAX as it appears to users, demonstrating how the design principles articulated in Chapter 2 materialize into a coherent, operational security platform.

3.2.1 Authentication Module

Login Page

The login page is the gateway to all platform capabilities. Authentication is enforced through a multi-layer mechanism: username and password verification against `bcrypt`-hashed credentials (12 rounds), followed by an optional Time-based One-Time Password (TOTP) second factor for accounts where MFA has been enabled. Incorrect credential attempts increment a per-account counter; after five consecutive failures, the account is locked for fifteen minutes to neutralize brute-force attacks. The interface provides clear, non-enumerable error feedback --- the same generic message is returned regardless of whether the username or

password is incorrect, preventing user enumeration.

Figure 3.1: Login Page with error feedback on invalid credentials

Registration Page

New operator accounts are created through the registration interface, which enforces the NIST SP 800-63B password policy in real time: a minimum of ten characters, mandatory inclusion of uppercase and lowercase letters, at least one digit, and one special symbol. A dynamic strength indicator bar and a visual checklist provide immediate feedback as the user types, reducing the probability of weak passwords being submitted.

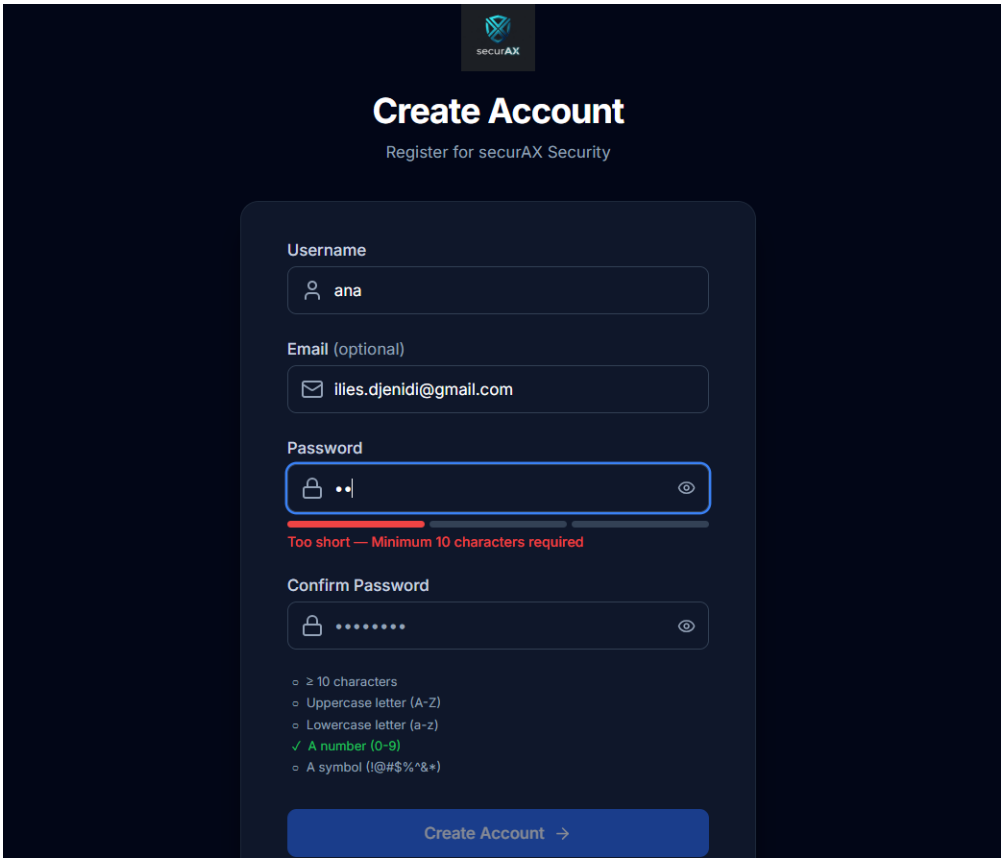

Figure 3.2: Registration Page with real-time password strength indicator

3.2.2 Administrator Dashboard

Overview Page

The administrator dashboard aggregates the platform's security posture into four key metrics displayed prominently at the top of the interface: total scans executed, total vulnerabilities discovered, the count of critical-severity findings, and the platform-wide average risk score. A filterable table of recent operations provides rapid situational awareness, showing each scan's target, type, risk level, findings count, and execution timestamp.

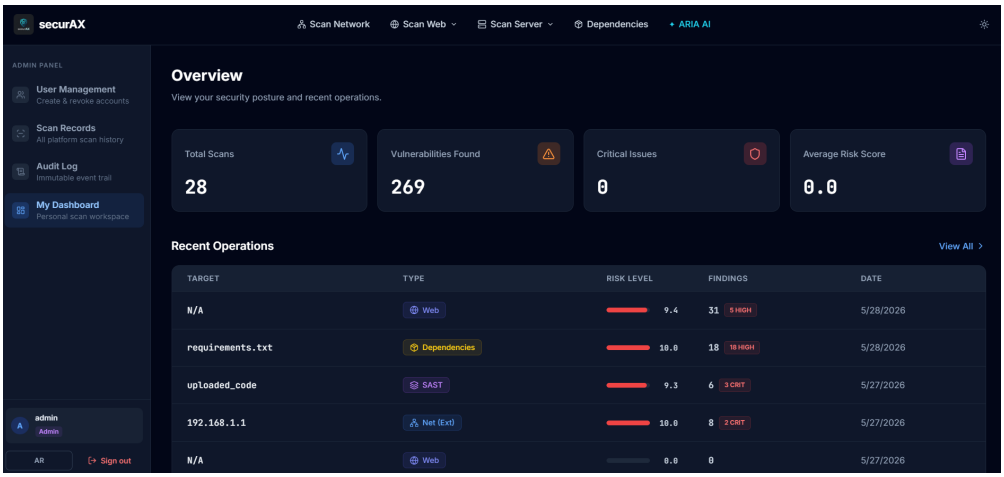

Figure 3.3: Administrator Dashboard --- platform-wide security metrics and recent operations

Global Scan Records

The Global Scan Records page grants administrators complete visibility over every scan executed on the platform by any operator. Records are filterable by scan type, risk level, and date range, and are exportable to CSV for external reporting and compliance evidence packages.

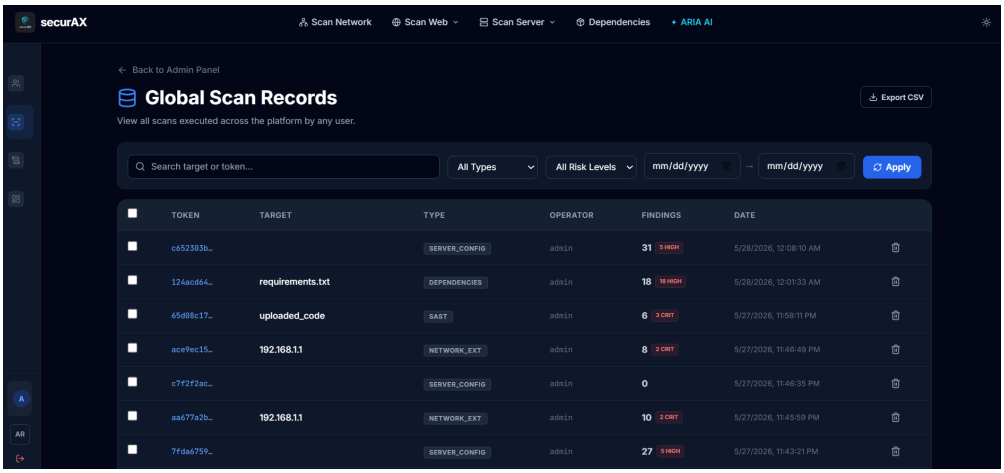

Figure 3.4: Global Scan Records --- administrator view across all operators

User Management

The User Management interface allows administrators to provision new operator accounts, modify role assignments, and suspend or revoke access as required by operational policy. Each entry in the directory displays the user's ID, username, assigned role (Admin or Analyst), current status, and a set of action controls.

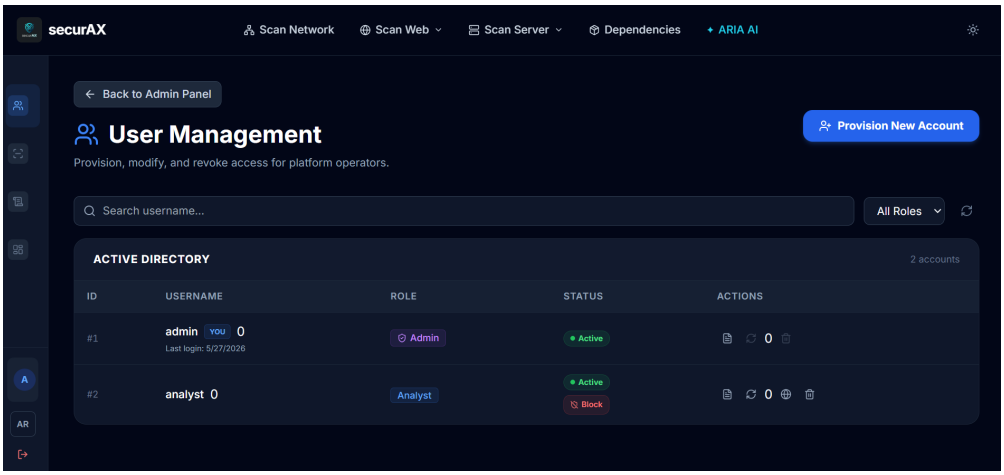

Figure 3.5: User Management --- role-based access control administration

3.2.3 Network Scanner

The Network Scanner module provides deep reconnaissance of both local and internet-facing network infrastructure. The operator specifies a target hostname or IP address and selects one of three operational profiles: *Deep Infiltration* for comprehensive port scanning, service version detection, and OS fingerprinting; *Port Discovery* to enumerate all active entry points; or *Surveillance Mode* for a rapid sweep of the top 100 most common ports.

Findings are automatically enriched with real-time CVE data from the NIST National Vulnerability Database (NVD), presenting each open service alongside its associated severity badge, CVE identifier, and a standardized description. The integrated **Analyze with ARIA** button passes the complete scan output to the AI assistant for expert infrastructure risk interpretation.

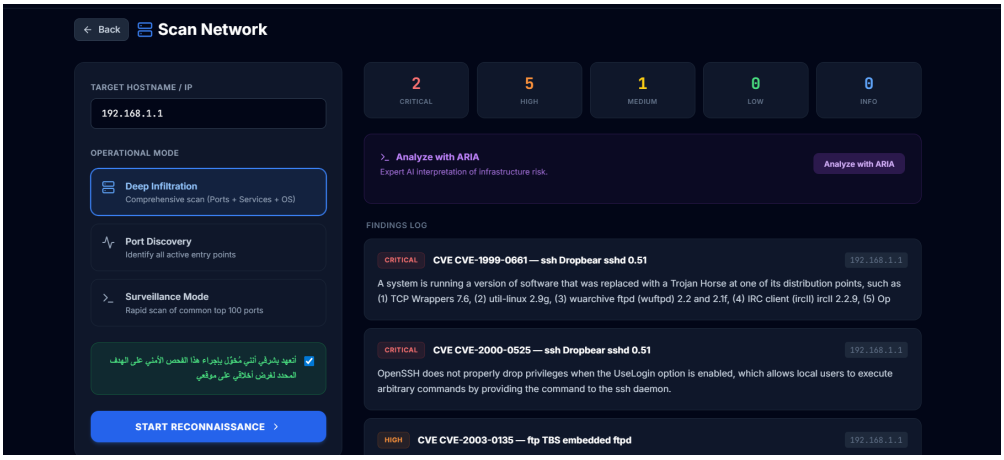

Figure 3.6: Network Scanner --- NVD-enriched port and service analysis

3.2.4 Web Security Scanner (Independent Passive Analysis Vector)

The Web Security Scanner delivers Dynamic Application Security Testing (DAST) against live web applications and APIs. The operator provides a target URL and chooses from three scan profiles: *Deep Infiltration* for a comprehensive sweep covering XSS, SQL injection, TLS configuration, and server-side misconfigurations; *Surface Audit* for a rapid common-vulnerability check; or *Reconnaissance* for low-noise information gathering. A mandatory ethical-use confirmation checkbox is required before any scan is initiated, implementing the platform's legal accountability mechanism.

The scanner executes eleven parallel external intelligence API calls via `ThreadPoolExecutor` with 18 workers --- Qualys SSL Labs (TLS grading), Mozilla Observatory (security header scoring), Shodan InternetDB (exposed service intelligence), VirusTotal v3 (malware/reputation), AbuseIPDB v2 (IP abuse history), URLhaus (malware URL database), GreyNoise Community (internet noise classification), IPinfo.io (IP geolocation and ASN), URLScan.io (URL behaviour analysis), crt.sh (Certificate Transparency log), and Google Safe Browsing --- alongside seventeen local passive checks covering HTTP security headers, cookie security flags, CORS policy, sensitive path exposure, and server version disclosure patterns. This parallel architecture completes the full intelligence sweep in the time a sequential scan would take for a single API call.

Sovereignty note on external API calls. Unlike the DAST active scanner (ZAP + Nikto) and the Server Configuration Audit Engine, which execute entirely within the operator's environment, these external API calls transmit the target URL and/or IP address to third-party services. This is a deliberate, documented design trade-off: each service contributes threat intelligence that has no local equivalent (e.g., VirusTotal's multi-engine malware verdict, AbuseIPDB's community-sourced abuse history, or crt.sh's certificate transparency log). Operators in strict sovereignty-constrained deployments may disable all external calls via environment variable flags, retaining only the seventeen on-premises passive checks. The NVD API and CISA KEV catalog are classified separately: they receive CVE identifiers only --- never target URLs or IP addresses --- and are therefore compatible with all deployment security policies.

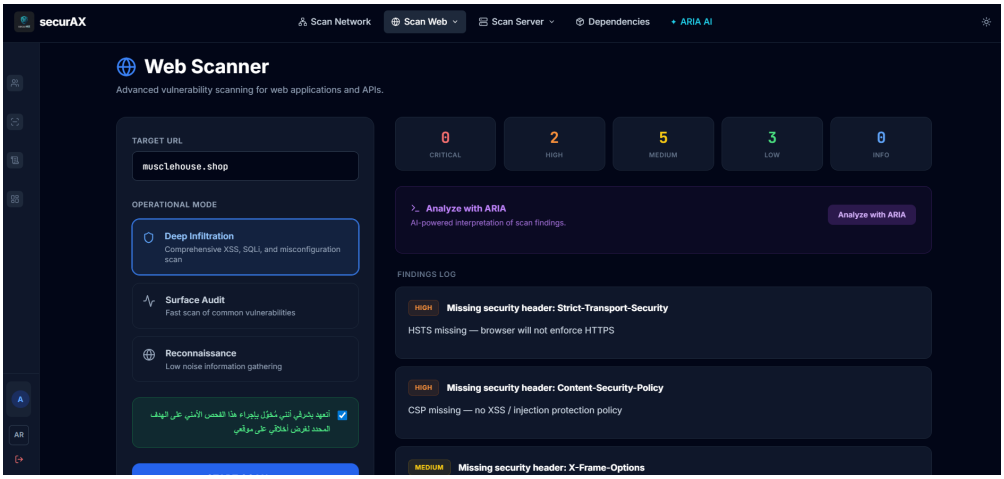

Figure 3.7: Web Security Scanner --- multi-API DAST with parallel intelligence enrichment

3.2.5 Code Analyzer (SAST)

The Code Analyzer performs Static Application Security Testing (SAST) on uploaded source code archives (`.zip`). Two complementary analysis engines execute in parallel: **Bandit** for Python-specific vulnerability detection (hardcoded credentials, unsafe subprocess calls, weak cryptographic primitives, SQL injection via string formatting) and **Semgrep** for multi-language pattern-based analysis covering Python, JavaScript, TypeScript, Go, Ruby, Java, C/C++, PHP, and C#. Each finding references the exact file path and line number in the source code where the issue was identified, enabling developers to navigate directly to the vulnerable code without manual search.

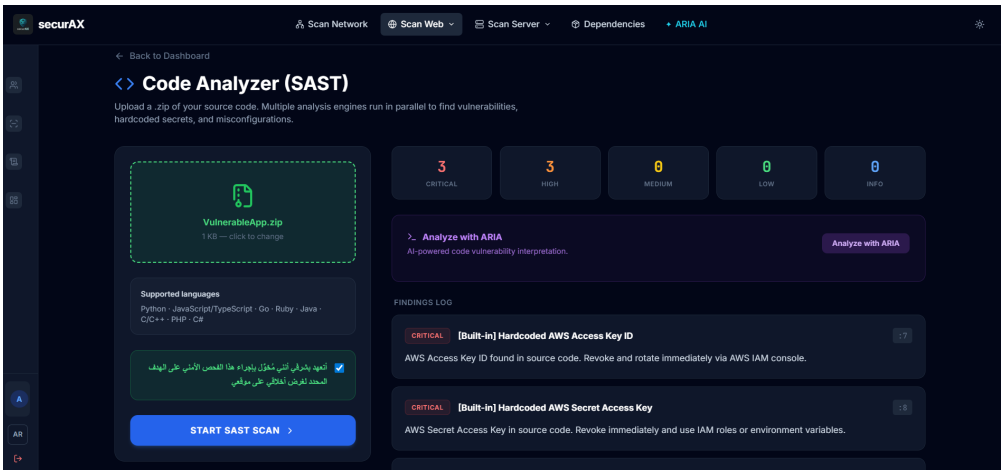

Figure 3.8: Code Analyzer (SAST) --- parallel static analysis with line-level attribution

3.2.6 Dependency Scanner

The Dependency Scanner audits project manifest files (`requirements.txt`, `package.json`) to identify third-party libraries carrying known Common Vulnerabilities and Exposures

(CVEs). Each finding includes the affected package name, installed version, vulnerable version range, the corresponding CVE identifier, and the GitHub Security Advisory (GHSA) reference. This directly addresses OWASP A06:2021 (Vulnerable and Outdated Components), enabling development teams to prioritize dependency upgrades with full supply-chain visibility.

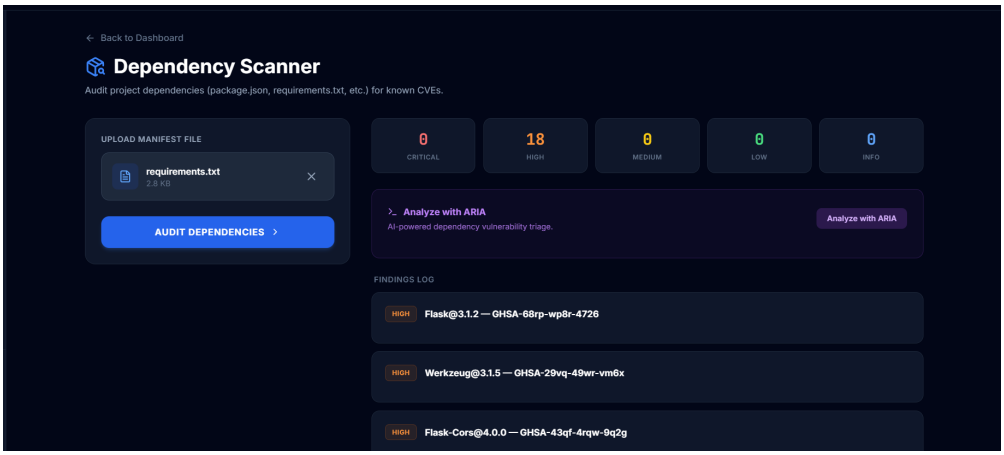

Figure 3.9: Dependency Scanner --- CVE-mapped software supply chain analysis

3.2.7 ARIA AI Assistant

Chat Interface

ARIA (Artificial Reconnaissance and Intelligence Advisor) is the platform's integrated AI assistant. A dedicated chat interface allows operators to engage in security-focused dialogue, query scan results in natural language, and obtain expert cybersecurity guidance on demand. The left-hand sidebar exposes scan context shortcuts (General, Web Scan, Network, Code Review) and a curated prompt library covering Authentication, SQL Injection, XSS Attacks, OWASP Top 10, Cloud Security, API Security, and Dependency Management.

ARIA operates in three modes, selected automatically at startup in the following priority order: **(1) Ollama** --- if `OLLAMA_URL` is set, all inference runs against a locally-hosted LLM with zero data leaving the operator's environment, satisfying the digital sovereignty requirement established in Chapter 1; **(2) Google Gemini 2.0 Flash** --- used when Ollama is not configured; **(3) offline rule-based engine** --- universal fallback covering 15 vulnerability types with copy-pasteable remediation, available with zero network connectivity. Ollama is prioritised over Gemini to maximise sovereignty compliance for air-gapped and sensitive deployments.

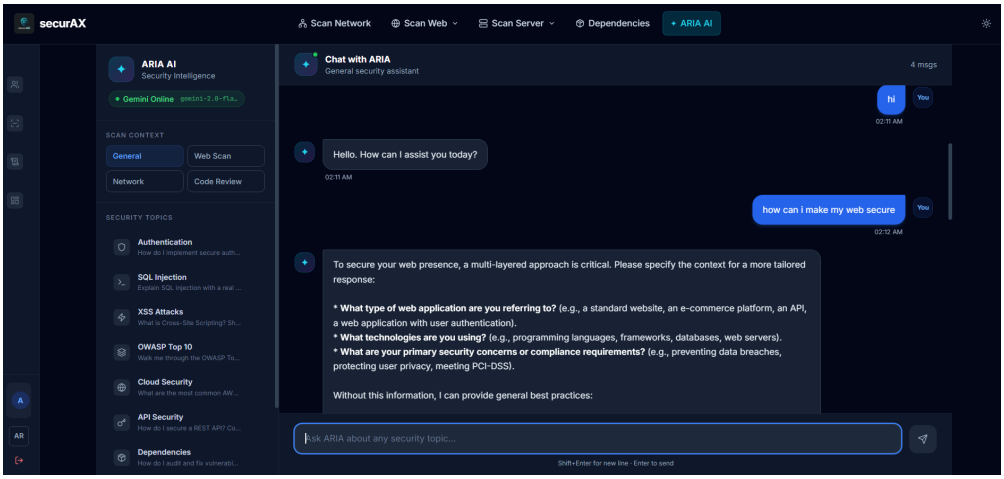

Figure 3.10: ARIA AI Assistant --- contextual security chat interface

Contextual Security Analysis

Beyond the general chat interface, ARIA is embedded throughout every scan module via an **Analyze with ARIA** button. When triggered, it ingests the raw scan output and returns a structured, human-readable threat analysis with actionable remediation steps tailored precisely to the specific findings, without requiring the operator to copy or paste any data manually. This reduces the time from scan completion to prioritized remediation plan to seconds.

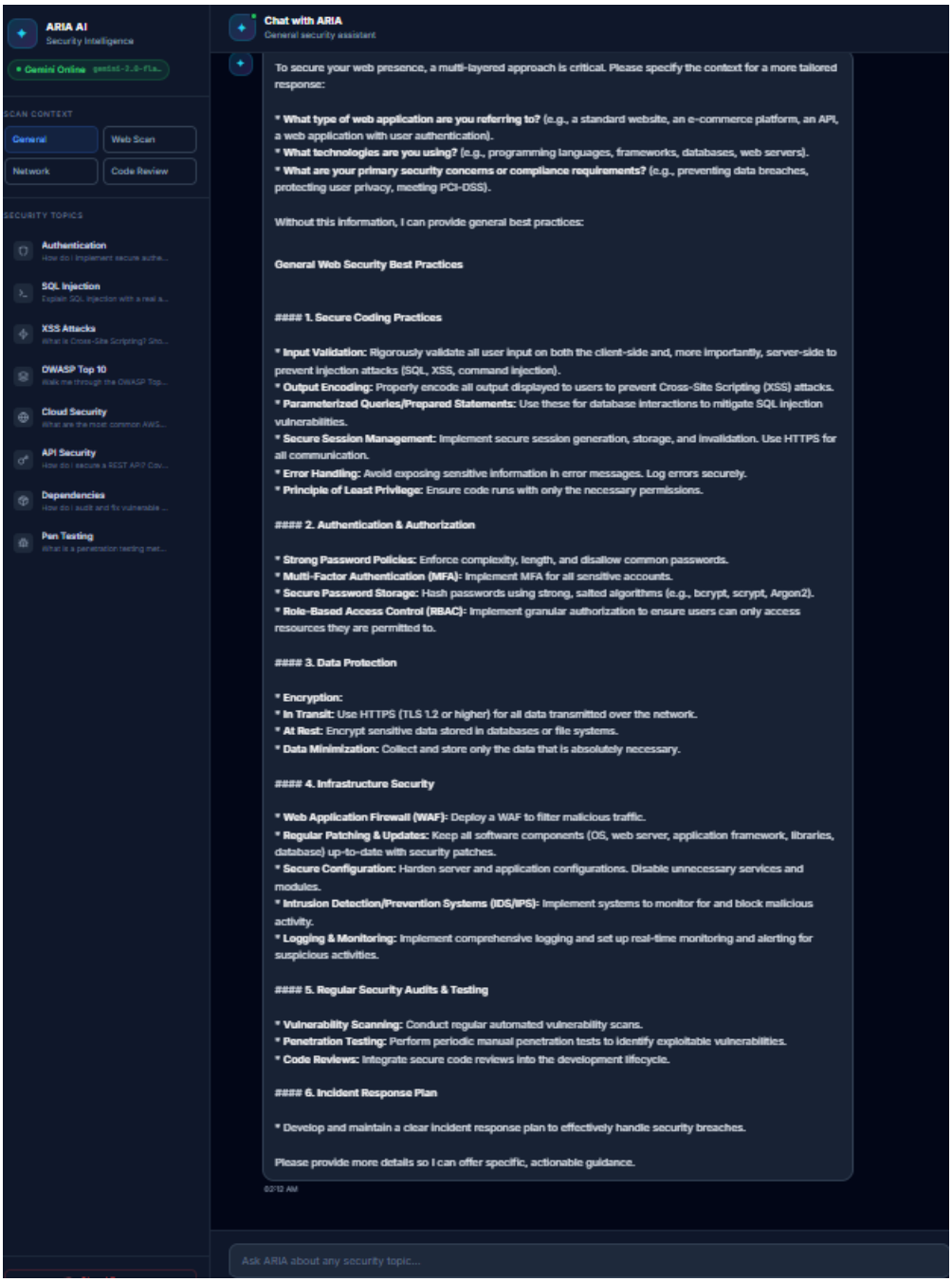

Figure 3.11: ARIA AI Assistant --- scan-contextual threat analysis and remediation guidance

3.3 Server Configuration Audit Engine

The Server Configuration Audit Engine, implemented in `scanners/server_int.py`, is the principal original technical contribution of this project. Unlike every other scanner in the platform --- which delegates detection to an established external tool (ZAP, Nikto, Nmap, Bandit, Semgrep, pip-audit) --- this module was developed from scratch to fill a genuine gap in the open-source security ecosystem: *no mature, widely-adopted tool performs white-box*

static analysis of Apache and Nginx configuration files for the precise set of security and operational misconfigurations most commonly exploited in production environments.

The sections below present the engine at every level of its architecture, including an honest account of the engineering obstacles encountered during development and the reasoning behind each design decision.

3.3.1 Why We Built It Ourselves: The Sovereignty Constraint

The first question any evaluator will ask is: why not use an existing tool? During the early design phase, we evaluated several candidates.

Lynis and **InSpec** are agent-based tools that must run on the target server itself --- requiring installation access, a runtime environment (Ruby for InSpec), and the willingness to execute third-party code on production infrastructure. In sensitive deployment contexts --- government agencies, financial institutions, or any environment subject to Algerian Law 18-07 on personal data protection --- this is simply not permitted.

Nuclei by ProjectDiscovery presented a more subtle problem. Its cloud platform (`PDCP_API_KEY`) sends the target URL to ProjectDiscovery's servers; its template engine fetches updates from `github.com/projectdiscovery/nuclei-templates` on every run. For the DAST scanner, this trade-off was acceptable because the targets are external URLs. For a configuration file containing internal server paths, module lists, TLS private key locations, and directory structures, the trade-off was categorically unacceptable: the file describes the complete security architecture of a production server, and it must never leave the operator's environment.

Beyond the server configuration context, the fundamental architectural concern with Nuclei's Cloud API mode is broader: any component that routes scan metadata through a third-party cloud infrastructure creates a structural sovereignty dependency. The platform's design principle --- that sensitive analysis must remain entirely within the operator's controlled environment --- made the Nuclei Cloud API integration incompatible with the project's sovereignty requirements. For this reason, Nuclei was not integrated into the final DAST pipeline; OWASP ZAP and Nikto, both fully local tools with no external data transmission requirement, were selected instead. This decision is an engineering trade-off, not a limitation: the CVE detection coverage that Nuclei's template library provides is partially compensated by the NVD API enrichment applied to network scan findings, and the remaining gap is documented as a direction for future work in Chapter 4.

CIS-CAT Pro covers the CIS Apache benchmark but is a commercial product unavailable under open-source licensing.

The result of this evaluation was straightforward: there is a genuine gap, and we fill it. The `server_int.py` engine performs its entire analysis locally, in memory, with no network

connection of any kind. The uploaded file is written to a `tempfile.NamedTemporaryFile`, analysed in-process, and deleted in a `finally` block regardless of whether the scan succeeds or raises an exception. This satisfies the digital sovereignty requirement established in Chapter 1.

3.3.2 Module Overview and Positioning

Functional Scope

The module accepts a single configuration file --- either an Apache `httpd.conf` or an Nginx `nginx.conf` --- via a secure file upload endpoint and returns a structured finding report without performing any network connection or external API call. The analysis is entirely local and **offline-capable by design**, which satisfies the digital sovereignty requirement established in Chapter 1: sensitive configuration files describing a production server's security posture never leave the operator's controlled environment.

Scale and Compliance Reference

At 2,339 lines of Python, `server_int.py` implements:

- **34 security and operational checks** for Apache `httpd.conf` files (implemented across 32 independent check functions), aligned with the CIS Apache HTTP Server 2.4 Benchmark v2.3.0 Level 2 (2026) and OWASP Secure Configuration Guidelines.
- **13 checks** for Nginx `nginx.conf` files, covering TLS hardening, access control, security headers, and observability.
- An **automated remediation engine** that produces a corrected, production-deployable version of the input configuration file.
- An **auto-detection mechanism** that identifies the server type from content patterns and routes analysis accordingly.

The 34 Apache checks are organized across 32 independent check functions; three SSL/TLS checks (`SSLProtocol`, `SSLCipherSuite`, `SSLHonorCipherOrder`) are consolidated inside a single `_check_ssl_settings()` function because they share the same parsed directive context and benefit from combined evaluation.

Module Architecture

The engine is organized into five distinct architectural layers, each with clearly bounded responsibilities:

Table 3.1: Server Configuration Audit Engine --- Architectural Layers

Layer	**Components**	**Responsibility**
Type Layer	`Severity`, `ConfigFinding`, `ConfigScanResult`	Domain model and output schema
Parser Layer	`_parse_config()`, `_get_directive_value()`, `_get_all_directive_values()`	Tokenise raw config into structured directives
Detection Layer	32 `_check_*()` functions (Apache) 13 inline checks (`_scan_nginx_config`)	Apply security rules against parsed directives
Remediation Layer	`generate_fixed_config()`	Produce corrected config + change log
Orchestration Layer	`run_server_config_scan()`, `_detect_config_type()`	Entry point, type detection, result assembly

Figure 3.12 below shows the scan results interface for an uploaded Apache configuration file, displaying findings organized by severity with line-level evidence for each detected misconfiguration.

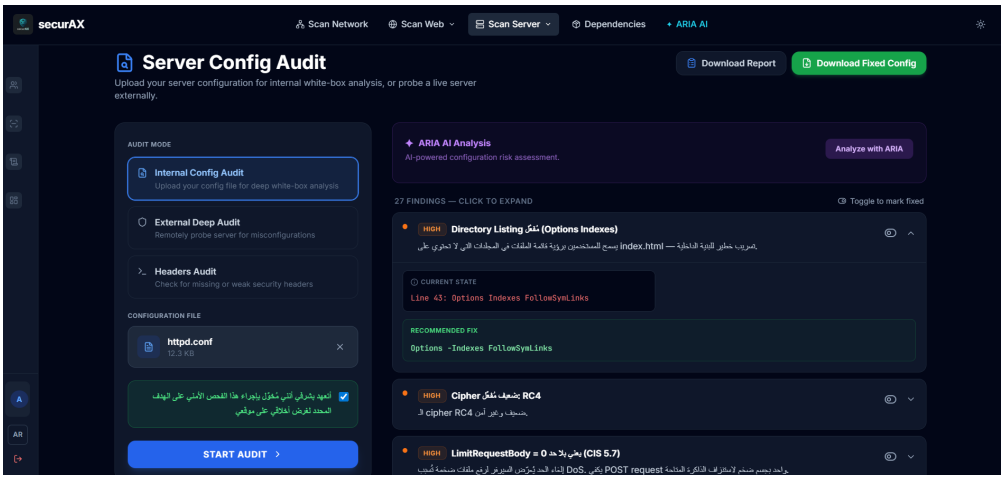

Figure 3.12: Server Configuration Audit --- findings interface with line-level evidence

Figure 3.13 shows the corresponding remediated configuration output automatically generated by the `generate_fixed_config()` function, with inline annotations documenting every directive that was modified.

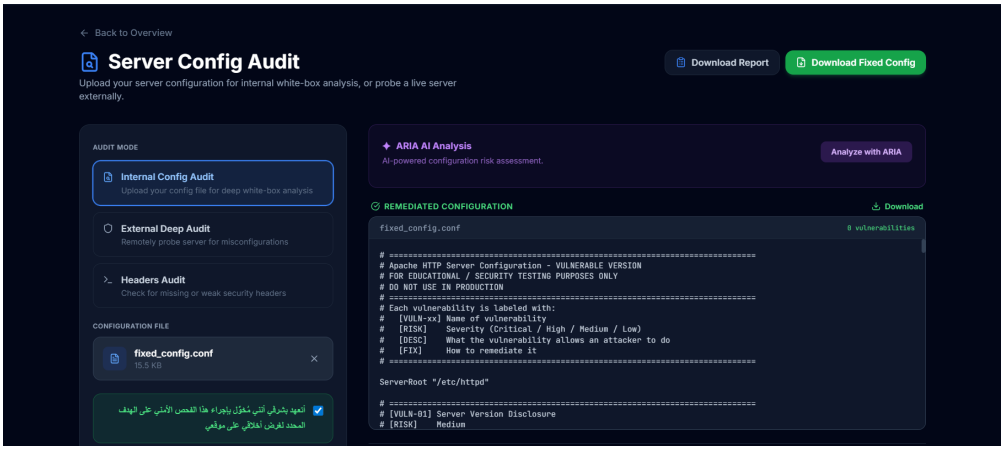

Figure 3.13: Server Configuration Audit --- automatically generated remediated configuration

3.3.3 End-to-End Request Flow

Table 3.2 illustrates the complete lifecycle of a server configuration scan from the operator's upload action to the dashboard rendering the final findings. This table makes explicit every system boundary and the precise handoff points between the five architectural layers described in the preceding section.

Table 3.2: End-to-end scan lifecycle --- from operator upload to dashboard rendering

Step	From	To	Action
1	Operator Browser	Flask API `app.py`	`POST /start-scan` with `.conf` file
2	Flask API	Validator Middleware	`validate_upload(ext allowlist)`
3	Validator	Flask API	Return: OK or HTTP 400 reject
4	Flask API	Audit Engine `server_int.py`	`run_server_config_scan(tmp_path)`
5	Audit Engine	*(internal)*	`_detect_config_type()` routes to Apache or Nginx path
6	Audit Engine	*(internal)*	`_parse_config()` tokenises directives into structured list
7	Audit Engine	*(internal)*	Execute all 32 `_check_*()` functions sequentially
8	Audit Engine	Flask API	Return `ConfigScanResult` (findings + info)
9	Flask API	Risk Engine `risk_engine.py`	`calculate_risk_v2(findings)`
10	Risk Engine	Flask API	Return `RiskBreakdown` (score, level, KEV factor, chains)
11	Flask API	Database `store_report`	Persist complete audit record (findings + risk + raw config)
12	Flask API	Operator Browser	Return JSON: findings + risk score for dashboard rendering

3.3.4 Scan Execution Workflow

The complete execution lifecycle of a server configuration scan proceeds through the following stages, each implemented as a distinct functional unit within `run_server_config_scan()`:

1. **Upload & Validation.** The operator uploads a `.conf` or `.txt` file via the `/start-scan` endpoint. The `validate_upload()` middleware enforces an explicit extension allowlist before any file content is written to disk.
2. **Secure Temporary Storage.** The validated file is written to a `tempfile.NamedTemporaryFile` path and read by the engine in-process. A `finally` block guarantees deletion regardless of scan outcome.
3. **Server Type Detection.** `_detect_config_type()` applies a five-pattern Nginx scoring model; a threshold of two matches routes the file to `_scan_nginx_config()`, otherwise to the Apache pipeline.

4. **Parsing.** `_parse_config()` iterates every line, classifying it as a comment (discarded), block open/close tag, or standard directive. Output is a `list[tuple[int, str, str]]` of `(line_number, name_lowercase, value)`.

5. **Rule Execution.** The orchestrator invokes all 32 Apache check functions sequentially, accumulating findings into a single list. Each function is stateless and receives only the parsed directive list (and raw content where required).

6. **Finding Classification.** Findings with `Severity.INFO` are separated into the `info` list; all others populate `vulnerabilities`, sorted by severity order (Critical $\rightarrow$ High $\rightarrow$ Medium $\rightarrow$ Low).

7. **Risk Scoring.** The `to_dict()` output is passed to `calculate_risk_v2()` with `internet_facing=False` and `criticality=1.0`, producing a `RiskBreakdown` dataclass.

8. **Persistence.** The complete audit record --- raw findings, risk breakdown, and original configuration content --- is written to the database via `store_report()`.

9. **Response.** The serialized JSON result is returned to the frontend for dashboard rendering and operator review.

The following algorithm formalizes the core detection pipeline:

Algorithm 1 Server Configuration Audit Engine --- Detection Pipeline

Require: Configuration file path f
Ensure: Structured finding report R
$content \leftarrow$ `read_file`(f)
$type \leftarrow$ `_detect_config_type`$(content)$
if $type =$ ``nginx'' **then**
  return `_scan_nginx_config`$(f, content)$
end if
$parsed \leftarrow$ `_parse_config`$(content)$ {list of (line_num, directive, value) tuples}
$findings \leftarrow [\,]$
for each check function c_i in $\{c_1, c_2, \ldots, c_{32}\}$ **do**
  $findings \mathrel{+}= c_i(parsed)$
end for
$vulnerabilities \leftarrow \{f \in findings \mid f.severity \neq$ `INFO`$\}$
$info \leftarrow \{f \in findings \mid f.severity =$ `INFO`$\}$
`sort`$(vulnerabilities,\ key = severity\_order)$
$risk \leftarrow$ `calculate_risk_v2`$(vulnerabilities)$
`store_report`$(vulnerabilities,\ info,\ risk)$
return `ConfigScanResult`$(vulnerabilities,\ info,\ risk,\ \ldots)$

Representative Finding Output

The following JSON object illustrates the complete structure of a finding produced by `_check_ssl_settings()` when applied to the test configuration `vulnerable_apache.conf`. Every field is populated by the detection function at the time the issue is identified; no post-processing step enriches or transforms the output.

```
{
  "check":           "weak_ssl_protocol",
  "severity":        "high",
  "title":           "Weak SSL/TLS Protocol Enabled",
  "description":     "SSLv2/3 and TLS 1.0/1.1 are vulnerable to
                      POODLE and BEAST attacks. RFC 8996 formally
                      deprecates TLS 1.0 and 1.1.",
  "evidence":        "Line 46: SSLProtocol all",
  "remediation":     "Replace with: SSLProtocol all -SSLv3
                      -TLSv1 -TLSv1.1",
  "fixed_directive": "SSLProtocol all -SSLv3 -TLSv1 -TLSv1.1",
  "line_number":     46
}
```

The `fixed_directive` field is what enables automated remediation: the `generate_fixed_config()` function reads this field from every finding and performs a line-precise substitution in the original file, producing a corrected configuration without any manual operator intervention.

3.3.5 Domain Model

The engine's type layer defines three core data structures implemented as Python dataclasses, providing a typed, immutable representation of every finding and its associated context.

Severity Enumeration

```
class Severity(str, Enum):
    CRITICAL = "critical"
    HIGH     = "high"
    MEDIUM   = "medium"
    LOW      = "low"
    INFO     = "info"
```

Inheriting from both `str` and `Enum` ensures that severity values serialize transparently to JSON without requiring a custom serializer, which is essential for the `to_dict()` output consumed by the frontend and the risk engine.

ConfigFinding Dataclass

`ConfigFinding` represents a single detected misconfiguration. Its fields are:

Table 3.3: `ConfigFinding` --- Field Specification

Field	Type	Purpose
`check`	`str`	Machine-readable check identifier (e.g. `directory_listing_enabled`)
`severity`	`Severity`	Classified risk level
`title`	`str`	Human-readable finding title
`description`	`str`	Technical explanation with threat context
`evidence`	`str`	The exact vulnerable configuration line with its line number
`remediation`	`str`	Step-by-step fix instructions
`line_number`	`int`	Source line in the uploaded file (0 if directive is absent)
`fixed_directive`	`str`	The corrected directive, ready for direct substitution

The `fixed_directive` field is architecturally significant: it is the primary input to the remediation engine (Section 3.3.11). Every check that detects a misconfiguration is *required* to populate this field with the exact corrected text that should replace the vulnerable directive in the configuration file. This contract between the detection layer and the remediation layer enables fully automated, line-precise correction of all detected issues.

ConfigScanResult Dataclass

`ConfigScanResult` aggregates all findings for a single scan run and exposes a `to_dict()` method that serializes the result into the unified schema consumed by the frontend, the risk engine, and the database persistence layer.

```
@dataclass
class ConfigScanResult:
    config_file:     str
    apache_version:  str = ""
    scanned_at:      str = ""
    vulnerabilities: list[ConfigFinding] = field(default_factory=list)
    info:            list[ConfigFinding] = field(default_factory=list)
    error:           Optional[str] = None
    total_lines:     int = 0
```

The `to_dict()` serializer exports each vulnerability with both `remediation` and `recommendation` keys (the latter being an alias), ensuring backward compatibility with frontend components that reference either field name. It also cleanly separates actionable `vulnerabilities` from purely informational `info` findings, allowing the risk engine to weight them differently.

3.3.6 Configuration Parser Engine

Before any security check can be applied, the raw configuration file must be transformed from free-form text into a structured, queryable representation. This is the responsibility of the parser layer.

Core Parser: `_parse_config()`

The parser iterates over every line of the configuration file and classifies it into one of four categories:

1. **Empty lines and comments** (`#`) --- silently discarded.
2. **Block open tags** (e.g. `<Directory "/var/www">`) --- parsed as a special `block_directory` directive with the block argument as the value.
3. **Block close tags** (e.g. `</Directory>`) --- parsed as a `block_end_directory` directive.
4. **Standard directives** (e.g. `ServerTokens Full`) --- split on the first whitespace into directive name and value.

The output is a `list[tuple[int, str, str]]` where each tuple is `(line_number, directive_name_lowercase, value)`. Directive names are normalized to lowercase at parse time, eliminating the need for case-insensitive comparison in every individual check. Line numbers are preserved from the original file, enabling the remediation engine to perform line-precise substitutions.

Directive Accessors

Two utility functions provide structured access to the parsed output:

- **`_get_directive_value(parsed, directive)`** --- returns the `(line_number, value)` tuple of the first occurrence of a directive, or `None` if absent. Used for singleton directives like `ServerTokens`, `TraceEnable`, and `Timeout`.

- `_get_all_directive_values(parsed, directive)` --- returns a list of all `(line_number, value)` tuples for repeated directives. This is essential for `SSLProtocol` and `SSLCipherSuite`, which may legitimately appear multiple times in a virtual-host context.

The parser intentionally does not build a full abstract syntax tree of the configuration's block hierarchy. This would significantly increase complexity without improving detection accuracy for the checks implemented. Where block context matters (e.g., detecting `Require all granted` inside a `<Directory />` block), the relevant check functions track context state themselves using a boolean flag as they iterate over the parsed list.

3.3.7 A Recurring Engineering Challenge: False Positives

Developing the detection logic revealed a systematic class of problem: naive string matching produces false positives that would mislead operators and erode trust in the tool. Three concrete examples illustrate how the engine resolves this class of problem.

Case 1: The Cipher Negation Problem

The first draft of `_check_ssl_settings()` searched the raw `SSLCipherSuite` string for weak cipher keywords --- RC4, DES, NULL, EXPORT, MD5. This immediately produced false positives: a configuration like `HIGH:!aNULL:!MD5:!RC4` was flagged as insecure, despite explicitly *disabling* those ciphers with the `!` negation prefix.

The fix required abandoning raw string search in favour of tokenization. The cipher string is split on the `:` delimiter, and only tokens whose first character is not `!` are tested against the weak keyword list:

```
tokens = cipher_string.split(":")
weak_found = [
    t for t in tokens
    if not t.startswith("!")
    and any(w in t.upper() for w in WEAK_CIPHERS)
]
```

The configuration `HIGH:!aNULL:!MD5:!RC4` now correctly tests as clean.

Case 2: The Negated Option Problem

The `_check_directory_listing()` check detects whether `Options Indexes` is enabled. A naive search for the string `Indexes` in the directive value produces a false positive on `Options -Indexes`, which explicitly *disables* directory listing.

The fix tokenizes the `Options` value and examines each token individually, flagging only tokens that equal `indexes` without a leading `-` or `+` prefix:

```
tokens = options_value.split()
dangerous = [
    t for t in tokens
    if t.lstrip("+-").lower() == "indexes"
    and not t.startswith("-")
]
```

Case 3: The LimitRequestBody Severity Calibration

An early version of the `LimitRequestBody` check assigned `Info` severity to the case where the directive was absent. The reasoning was that the directive's absence does not necessarily mean a problem is present.

During internal testing, a review of the Apache documentation clarified that the default value when `LimitRequestBody` is absent is 0 --- meaning unlimited. A single large POST request can exhaust server disk and RAM, constituting a realistic Denial of Service vector. The severity was revised to graduated levels: absent directive receives `Medium` (implicit risk), value of 0 receives `High` (explicit unlimited), and value exceeding 10 MB receives `Low` (oversized but not unlimited).

This calibration was not initially obvious. It required reading the CIS Benchmark control text carefully and testing against a real Apache instance to confirm the default behaviour.

3.3.8 Apache Security Checks

The 34 Apache checks are implemented as independent functions, each following the same signature: `_check_*(parsed: list) -> list[ConfigFinding]`. Two checks additionally receive the raw configuration content as a second argument (`raw_content: str`) for cases where regular expression search over the full text is more reliable than directive-level parsing (security headers, HTTP method restriction, `mod_security` detection).

The checks are organized into seven functional categories.

Category 1 --- Server Information Disclosure (Checks #1, #2, #12, #23)

These checks address the information leakage that enables *reconnaissance-assisted exploitation*: an attacker who knows the precise server version, OS, and installed PHP version can immediately query CVE databases for applicable exploits.

Table 3.4: Information Disclosure Checks

Directive	CIS	Severity	Exploit Vector
`ServerTokens`	2.2	Medium	Exposes Apache version + OS in every HTTP response header; maps directly to version-specific CVEs.
`ServerSignature`	2.3	Low	Appends version string to all error pages; redundant with `ServerTokens` but doubles exposure surface.
`expose_php`	5.x	Low	PHP version exposed in `X-Powered-By` header; enables targeting of PHP-specific CVEs (e.g. CVE-2024-4577).
`FileETag`	5.2	Low	Default includes inode number, leaking filesystem structure information usable in chained attacks.

The safe value for `ServerTokens` is `Prod`, which causes Apache to emit `Server: Apache` with no version, OS, or module information. The engine flags any value other than `Prod` or `ProductOnly`, and generates `fixed_directive = "ServerTokens Prod"` for the remediation engine.

Category 2 --- File System Access Control (Checks #3, #4, #11, #19, #24)

This category addresses misconfigurations that allow unauthorized access to files and directories on the server's filesystem.

Table 3.5: File System Access Control Checks

Check	CIS	Severity	Exploit Scenario
`Options Indexes`	3.6	High	Enables directory listing when no `index.html` exists; exposes source files, backups, and configuration fragments.
`FollowSymLinks`	3.7	Medium	Without `SymLinksIfOwnerMatch`, symlinks outside `DocumentRoot` are followed, enabling path traversal.
`Require all granted` on `/`	3.x	**Critical**	Root filesystem fully accessible to all requests; most dangerous single misconfiguration in an Apache deployment.
`AllowOverride All`	3.5	Medium	Permits `.htaccess` to override security settings; an uploaded `.htaccess` can re-enable PHP execution in upload directories.
`Options ExecCGI`	3.x	High	CGI execution in non-CGI directories; a writable directory becomes a remote code execution vector.

The directory listing check deserves specific attention for its parsing logic. A naive string search for `"Indexes"` would produce a false positive on `"-Indexes"` (which explicitly *disables* listing). The check therefore tokenizes the `Options` directive value and examines each token individually, flagging only those that equal `"indexes"` *without* a leading `-` or `+` prefix.

Category 3 --- SSL/TLS Hardening (Checks #5, #6, #7)

Three dedicated checks enforce the TLS security baseline required by CIS Section 7 and mandated by PCI-DSS 4.0 for any system handling cardholder data.

Table 3.6: SSL/TLS Hardening Checks

Check	CIS	Severity	Vulnerability
`SSLProtocol`	7.x	High	SSLv2/3 (POODLE), TLS 1.0/1.1 (BEAST, deprecated RFC 8996). Safe target: `all -SSLv3 -TLSv1 -TLSv1.1`.
`SSLCipherSuite`	7.x	High	RC4 (stream cipher bias), DES/3DES (SWEET32 birthday attack), NULL (no encryption), EXPORT (40-bit keys). Token-based parsing avoids false positives on `!aNULL`.
`SSLHonorCipherOrder`	7.x	Medium	Without this directive, the client selects the cipher; weak clients negotiate weak ciphers even on a hardened server.

The `SSLCipherSuite` check implements a deliberate false-positive avoidance mechanism: the cipher string is tokenized on the `:` delimiter, and only tokens *not* prefixed with `!` (negation) are tested against the weak cipher keyword list. A configuration like `HIGH:aNULL:MD5:RC4` correctly tests as clean, despite containing the strings `NULL`, `MD5`, and `RC4` in the raw text.

Category 4 --- Dangerous Module Loading (Check #8)

Apache's module architecture allows optional functionality to be loaded at runtime. Several modules that are useful during development or on internal systems create unacceptable exposure on public-facing production servers. The engine scans all `LoadModule` directives and flags any that load from a predefined risk table:

Table 3.7: Dangerous Module Check

Module	Severity	Risk
mod_status	High	Exposes live request list and worker stats at /server-status with no authentication.
mod_info	High	Exposes full Apache configuration and loaded modules at /server-info.
mod_userdir	Medium	Enables ~username URLs, disclosing system usernames.
mod_autoindex	Medium	Provides directory listing capability independent of the Options Indexes directive.
mod_cgi	Medium	CGI script execution; if not required, it expands the attack surface unnecessarily.
mod_include	Low	Server-Side Includes create XSS injection points if input is not sanitized.

The check generates a fixed_directive that comments out the LoadModule line, annotated with a securAX label, rather than deleting it. This allows administrators to re-enable modules quickly if needed while maintaining a clear audit trail of what was disabled and why.

Category 5 --- DoS Surface Reduction (Checks #9, #10, #13--#16, #20, #28)

Eight checks collectively address the configuration parameters that determine how the server handles large, slow, or malformed requests --- the primary attack surface for application-layer Denial of Service attacks.

Table 3.8: DoS Surface Reduction Checks

Directive	CIS	Sev.	DoS Vector
`TraceEnable On`	5.3	Medium	HTTP TRACE enables Cross-Site Tracing (XST); also provides an alternative reflection path.
`Timeout`	5.x	Low	Values >60 s allow Slowloris connections to linger, exhausting worker processes.
`LimitRequestLine`	5.4	Low--Med	URI length without limit enables buffer overflow attempts in CGI modules. Recommended: 8190 B.
`LimitRequestFields`	5.5	High if 0	Unlimited HTTP headers allow memory exhaustion. Value 0 means no limit. Recommended: 100.
`LimitRequestFieldSize`	5.6	Medium	Oversized individual headers can crash header-processing modules. Recommended: 8190 B.
`LimitRequestBody`	5.7	High if 0	Default value 0 (unlimited) allows a single large POST to exhaust disk and RAM.
`KeepAliveTimeout`	5.x	Low	Persistent connections held open too long amplify Slowloris effectiveness.
`MaxKeepAliveRequests`	---	Low	Unbounded keep-alive request count can pin worker processes for extended periods.

For `LimitRequestBody`, the check distinguishes three risk levels: value = 0 (no limit, High severity), value $>$ 10 MB (exceeds practical need, Low severity), and absent directive (defaults to 0, Medium severity as the risk is implicit). This graduated approach ensures that findings are proportionate to actual risk, avoiding alert fatigue from uniformly high-severity notifications.

Category 6 --- HTTP Security Headers (Check #17)

The security headers check is the only check in the engine that operates on the raw configuration content rather than the parsed directive list. This is because `Header always set` directives do not follow the simple `DirectiveName Value` format --- they are two-word directives (`Header always`) followed by a sub-command (`set`) and the header name, making the parser's single-directive model insufficient for exact matching.

A regular expression search is applied for each of the six required headers:

Table 3.9: HTTP Security Header Checks

Header	Sev.	Attack Vector Prevented
`X-Frame-Options`	High	Clickjacking via `<iframe>` embedding
`Strict-Transport-Security`	High	SSL Stripping / man-in-the-middle HTTPS downgrade
`Content-Security-Policy`	High	XSS, data injection, unauthorized resource loading
`X-Content-Type-Options`	Medium	MIME-type sniffing leading to script execution
`X-XSS-Protection`	Low	Legacy browser XSS filter (supplementary)
`Referrer-Policy`	Low	URL leakage to third-party origins

Category 7 --- Performance and Operational Reliability (Checks #25--#34)

The final category covers operational configuration that, while not directly exploitable, affects the server's resilience under load and its ability to support incident detection and forensic investigation. These checks are classified as Low severity or Info, ensuring they do not inflate the overall risk score.

- **MPM Mode (#25):** Flags `mpm_prefork` usage; recommends `mpm_event` for modern production workloads.
- **MaxRequestWorkers (#26):** Absent or zero value risks 503 errors or RAM exhaustion; recommended formula: $\lfloor \text{RAM} \times 0.8 \rfloor \div$ process size.
- **MaxConnectionsPerChild (#27):** Value of 0 means processes never recycle, allowing memory leaks to accumulate indefinitely.
- **mod_deflate / mod_expires (#29):** Uncompressed responses and missing cache headers increase bandwidth consumption and load times.

- **LogFormat with %D (#30):** Without the request processing time field, SLI/SLO measurement for response latency is impossible.
- **mod_rewrite Open Redirect (#32):** Detects `RewriteRule` directives that redirect to `%{HTTP_HOST}` without domain validation --- a classic open redirect via Host header injection.
- **TLS Session Cache (#33):** Missing `SSLSessionCache`, `SSLSessionCacheTimeout`, and `SSLSessionTickets` policy reduce TLS handshake performance and Perfect Forward Secrecy posture.
- **MPM Parameter Coherence (#34):** Validates that `MaxRequestWorkers` = `ServerLimit` × `ThreadsPerChild`; incoherent values cause undefined behavior under load.

3.3.9 Nginx Security Checks

The Nginx analysis path (`_scan_nginx_config()`) implements 13 checks using a combination of directive-level string searching and regular expression pattern matching against the raw configuration content. The Nginx configuration syntax --- block-scoped with semicolons as terminators --- does not lend itself to the same directive-list parser used for Apache, so the checks operate directly on line iteration and `re.search()` calls.

Table 3.10: Nginx Security Checks

Check	Sev.	Description
`server_tokens on`	Med	Version disclosure in HTTP headers and error pages.
`autoindex on`	High	Directory listing enabled.
`ssl_protocols` (TLS 1.0/1.1)	High	Deprecated protocols vulnerable to BEAST and POODLE.
`ssl_ciphers` (RC4/DES/MD5)	High	Cryptographically broken cipher suites.
Missing `X-Frame-Options`	Med	Clickjacking exposure.
Missing `X-Content-Type-Options`	Med	MIME sniffing attack surface.
Missing `Content-Security-Policy`	High	XSS injection without policy enforcement.
Missing `Strict-Transport-Security`	High	SSL stripping vulnerability.
`client_max_body_size` absent	Low	Unlimited upload size; DoS vector.
`keepalive_timeout` >15 s	Low	Extended connection hold amplifies slow HTTP attacks.
`gzip off` / absent	Low	Uncompressed responses increase bandwidth and load times.
`ssl_stapling off` / absent	Low	OCSP stapling missing; increases certificate validation latency.
`ssl_session_tickets` policy	Info	Session ticket key rotation policy not enforced; affects PFS posture.

3.3.10 Auto-Detection Mechanism

The public entry point `run_server_config_scan()` calls `_detect_config_type()` before routing to the appropriate analysis path. Detection uses a scoring model: the function searches the configuration content for five Nginx-specific patterns and assigns one point per match.

```
nginx_patterns = [
    r"\bhttp\s*\{",
    r"\bserver\s*\{",
    r"\blocation\s+/",
    r"\bworker_processes\b",
    r"\bkeepalive_timeout\b"
]
score = sum(1 for p in nginx_patterns if re.search(p, content))
return "nginx" if score >= 2 else "apache"
```

A threshold of two matching patterns (rather than one) makes the detection robust against Apache configurations that happen to contain commented Nginx examples or documentation fragments. This approach eliminates the need for file-extension-based detection, which would be unreliable since operators may upload files with non-standard names.

3.3.11 Automated Remediation Engine and the Iterative Correction Problem

The `generate_fixed_config()` function is the most architecturally distinctive component of the entire engine. It transforms the scan's output --- a list of `ConfigFinding` objects --- into a corrected, production-deployable version of the original configuration file. This feature elevates securAX from an *identification* tool to an *resolution* tool. However, this feature was also the source of the most significant engineering problem encountered during the project.

Algorithm

The function operates in three phases:

1. **Fix Map Construction.** The vulnerability list is iterated once to build two data structures:
    - `line_fixes`: a dictionary mapping each source line number to the list of repairs to apply at that line.
    - `append_fixes`: a list of directives that were absent in the original file (e.g., missing `ServerTokens`) and must be appended to the end of the file.
2. **Line-by-Line Reconstruction.** The original file is iterated line by line. For lines that appear in `line_fixes`, a `working_line` variable accumulates all applicable fixes, preserving the original indentation. Options-directive fixes (e.g., removing `Indexes` from an `Options` line that also contains other valid options) are handled by the inner `_apply_options_fix()` helper, which tokenizes the options string, removes or adds the specific option while preserving others, and de-duplicates the result. Each modified line is followed by a comment annotation recording the original value and the check identifier, creating an inline audit trail.
3. **Missing Directive Appendix.** If any findings relate to absent directives, a clearly demarcated block is appended to the end of the fixed file with all required additions, each annotated with its check identifier and human-readable title.

The Iterative Correction Problem

During validation, we ran the remediation engine on a deliberately vulnerable test configuration (`httpd.conf`) and then scanned the resulting fixed file to verify that all issues had been resolved. The first generated file, `httpd_securAX_fixed16.conf`, still contained findings.

The root cause was a correctness bug in the first version of `_apply_options_fix()`. The function was called once per check per line, meaning that multiple checks targeting the same `Options` directive were applied sequentially. The second application would overwrite the result of the first. A line like `Options Indexes FollowSymLinks ExecCGI` subjected to three checks --- directory listing, symlink traversal, and CGI execution --- would be partially corrected by the first check and then the second check would discard that correction.

The fix required restructuring the logic so that all checks targeting the same line are collected first, then applied in a single merged pass. The current implementation collects all options-specific repairs for a line, applies them all in one `_apply_options_fix()` call, and only then handles the remaining line-level substitutions:

```
# Apply all options-specific checks first, in one merged pass
for _, _, check, _ in fixes:
    if check in {"directory_listing_enabled",
                 "options_exec_cgi",
                 "unsafe_followsymlinks"}:
        working_line = _apply_options_fix(working_line, check)
        applied_checks.append(check)
```

This single change --- grouping options fixes before processing other line-level repairs --- eliminated the regression. The third generated file, `httpd_securAX_fixed18.conf`, produced zero findings when scanned.

The experience had a useful side effect: we now include the test configuration and its iteration history (`httpd_securAX_fixed16` through `httpd_securAX_fixed18`) in the repository's `archive_non_runtime/` directory as a concrete regression test record.

Return Value

The function returns `tuple[str, list[dict]]`: the complete corrected configuration text and a `change_log` --- a structured list of every modification made, recording the source line number, check identifier, finding title, original directive text (`before`), and corrected directive text (`after`). This change log is returned to the frontend alongside the fixed file, allowing operators to review exactly what was changed before deploying the corrected configuration.

3.3.12 API Integration and Security Controls

The engine is exposed through two Flask endpoints in `app.py`, both protected by the same security middleware stack.

Table 3.11: Server Configuration Audit API Endpoints

Route	Method	Function
`/start-scan`	POST	Main scan endpoint. Accepts `scan_type=server_int` with an uploaded `.conf` or `.txt` file. Invokes `run_server_config_scan()`, passes findings to the risk engine, and persists the result to the database.
`/fix_config`	POST	Dedicated remediation bridge. Accepts the same file types, performs the scan, and additionally invokes `generate_fixed_config()`, returning the corrected file and change log as a JSON response.

Both endpoints enforce the following security controls, applied as Flask decorators in the order listed:

1. **`@require_permission("run_scan")`:** Verifies that the authenticated user's permission set includes the `run_scan` capability. Unauthorized access returns HTTP 403 before any file processing occurs.
2. **`@limiter.limit("5/minute")`:** Rate-limits each authenticated user to five scan requests per minute, preventing resource exhaustion through rapid repeated uploads.
3. **`validate_upload(upload, {".conf", ".txt"})`:** Validates the uploaded file's extension against an explicit allowlist. Files with other extensions are rejected before being written to disk.
4. **`tempfile.NamedTemporaryFile` with `finally` cleanup:** The uploaded file is written to a securely created temporary path. A `finally` block guarantees deletion of the temporary file regardless of whether the scan completes successfully or raises an exception, preventing configuration file accumulation on the server's filesystem.

3.3.13 Risk Engine Integration

The output of `run_server_config_scan()` is passed directly to `calculate_risk_v2()` from `risk_engine.py`. Two parameters are fixed for server configuration scans:

- **`internet_facing=False`:** White-box configuration analysis represents a pre-deployment or internal audit context. The risk engine applies a scan-type weight of 0.9 for `server_int` scans, slightly below the 1.2 weight assigned to internet-facing DAST findings, reflecting the lower exploitability of a misconfiguration that has not yet been confirmed to be publicly reachable.
- **`criticality=1.0`:** The default environmental criticality modifier; operators may adjust this for test vs. production classifications.

The computed `RiskBreakdown` object --- containing the final numeric score, qualitative risk level (Minimal, Low, Medium, High, Critical), per-severity counts, top-ranked findings, matched CISA KEV CVE identifiers, detected attack chains, the temporal and environmental multipliers applied, and prioritised remediation recommendations --- is then passed to `store_report()` alongside the raw scan result and the original configuration content, persisting the complete audit record to the database for later retrieval and comparison.

3.3.14 Test Validation

The repository includes a deliberately vulnerable Apache configuration file, `vulnerable_apache.conf`, designed to exercise the engine's detection capabilities across all seven check categories simultaneously. The file declares `ServerTokens Full`, `ServerSignature On`, `TraceEnable On`, `Options Indexes FollowSymLinks ExecCGI`, `SSLProtocol all` (retaining TLS 1.0 and 1.1), `AllowOverride All`, `Require all granted` on the root directory, and all six required HTTP security headers absent.

Running this file through the engine produces findings across all seven check categories. The remediation engine then generates a corrected configuration. As documented in Section 3.3.11, the initial remediation output (`httpd_securAX_fixed16.conf`) required two further correction cycles before the final version (`httpd_securAX_fixed18.conf`) produced zero findings on re-scan --- a result that is preserved in the repository as a documented test record.

3.4 Used Technologies

This section documents the development environment, software stack, and external integrations that underpin the securAX platform.

3.4.1 Development Environment

The development environment integrates the following tools:

- **Visual Studio Code:** Primary code editor with syntax highlighting, IntelliSense auto-completion, integrated terminal, and Git version control.
- **Google Chrome DevTools:** Frontend debugging, network request inspection, and performance profiling.
- **Postman:** API endpoint design, testing, and documentation for all Flask REST routes.
- **Git & GitHub:** Distributed version control for change tracking, feature branching, and collaboration.
- **Claude:** Assisted design review, drafting, and consistency checks during iterative development.
- **Vibecoding:** Prompt-driven development workflow combining rapid iteration with manual verification and code review.

3.4.2 Development Stack

Frontend Stack

- **React 19:** Component-based UI library for modular interface development.
- **Vite 8.0:** Build tool providing fast Hot Module Replacement and optimized production bundles.
- **Tailwind CSS 3.4:** Utility-first CSS framework for consistent, rapid styling directly in JSX.
- **Framer Motion:** Fluid page transitions and animated scan result reveals.
- **Axios:** Promise-based HTTP client for backend API communication.
- **React Router DOM:** Client-side routing between scan modules and administrative panels.
- **Lucide React:** Consistent SVG icon library.

Backend Stack

- **Python 3.10+ / Flask 3.1.2:** Core web framework providing RESTful API endpoints, session management, and request routing via a flat monolithic architecture. All routes, security middleware, and scanner invocations reside in `app.py`, providing a single deployable unit with minimal operational overhead.
- **Gunicorn 25.1.0:** Production-grade WSGI server deploying the Flask application with multiple concurrent workers.
- **Flask-Login & Flask-WTF:** Authenticated session management and CSRF-protected form processing.
- **Flask-Limiter:** Configurable rate limiting to mitigate brute-force and denial-of-service attacks at the API layer.
- **Bcrypt:** Salted password hashing with a work factor of 12 rounds.
- **PyOTP & QRCode:** TOTP two-factor authentication implementation per RFC 6238.
- **Python-Nmap:** Programmatic wrapper around the Nmap network scanner.
- **Bandit:** Python AST-based static analysis for code-level vulnerability detection.
- **ReportLab, Arabic-Reshaper, Python-Bidi:** Bilingual (Arabic/English) PDF security report generation --- enabling operators to produce professional compliance-ready reports in both Arabic and English.

3.4.3 External APIs and Integrations

- **Ollama (Local LLM) --- Primary ARIA Provider:** When the `OLLAMA_URL` environment variable is set, ARIA routes all AI inference to a locally-hosted open-source LLM --- Mistral, LLaMA, or any Ollama-compatible model --- with zero data leaving the operator's controlled environment. This is the **highest-priority provider** in the ARIA stack and the recommended configuration for all production deployments, enabling full AI capabilities in air-gapped and sovereignty-critical environments.
- **Google Gemini API (ARIA --- Suspended):** ARIA includes support for Google Gemini 2.0 Flash as a cloud LLM fallback when Ollama is not configured. This mode is **currently suspended** in the production deployment: routing vulnerability findings and scan context to an external cloud API constitutes a sovereignty exposure that is incompatible with the platform's core design requirement. Gemini mode

will be restored in a future release following a data-minimisation redesign that transmits only anonymised severity summaries rather than raw scan output containing target identifiers.

- **OWASP ZAP / Nikto (DAST):** The DAST scanner orchestrates two complementary engines. OWASP ZAP provides active spider-and-scan coverage, systematically crawling the target application and injecting payloads to detect XSS, SQL injection, CSRF, and server misconfigurations. Nikto adds server-signature checks, enumerating known vulnerable server configurations and exposed administrative paths. Results from both engines are merged and de-duplicated by an **MD5 hash** of title, severity, and URL (`hashlib.md5(..., usedforsecurity=False)` --- MD5 is used here solely as a fast content fingerprint for deduplication, not as a security primitive; the `usedforsecurity=False` flag documents this intent explicitly), then ranked by severity. Both tools operate entirely locally --- no scan data leaves the operator's controlled environment, satisfying the digital sovereignty requirement established in Chapter 1.

- **NVD API --- National Vulnerability Database:** Enriches network scan findings with real-time CVE data, providing official CVSS v3.1 scores and standardized vulnerability descriptions for discovered services.

- **CISA KEV Catalog:** The risk engine maintains a background daemon thread (`threading.Thread`, daemon=True, name=`"cisa-kev-refresh"`) that refreshes the CISA Known Exploited Vulnerabilities catalog every 24 hours via the official CISA JSON feed. Findings whose CVE identifiers appear in the catalog receive a $\times 2.5$ exploitation factor --- the highest multiplier in the scoring algorithm --- reflecting confirmed active weaponisation in the wild. This factor applies specifically to critical and high severity findings to maximise remediation signal for the most dangerous actively-exploited vulnerabilities.

- **Attack Chain Intelligence:** The risk engine's `_detect_attack_chains()` function identifies five multi-vulnerability escalation scenarios at scoring time: Stored XSS combined with missing CSP (session token theft); SQL Injection with direct database access; exposed administrative interface combined with default credentials (zero-skill takeover); internet-facing Remote Code Execution (ransomware/lateral movement path); and unpatched operating system combined with open SMB port 445 (EternalBlue / WannaCry attack pattern). These chain detections are returned alongside the numeric risk score, giving operators narrative threat context that CVSS scores alone cannot provide.

- **Web Intelligence APIs (WebSecurityScanner):** The web scanner orchestrates calls to Qualys SSL Labs, Mozilla Observatory, Shodan InternetDB, VirusTotal v3, AbuseIPDB v2, URLhaus, GreyNoise Community, IPinfo.io, URLScan.io, crt.sh (Certificate Transparency), and Google Safe Browsing, all executed in parallel via `ThreadPoolExecutor` with 18 workers. These APIs transmit the target URL and/or IP address to third-party services and are therefore classified as **optional sovereign-boundary-crossing integrations**. Operators may disable them individually or collectively via environment variable flags. When disabled, the scanner falls back to its seventeen on-premises passive checks, which execute without any external network calls and fully satisfy the digital sovereignty requirement. Detailed justification for each API's inclusion appears in the WebSecurityScanner description in Section 3.2.

3.5 Conclusion

This chapter has presented the complete implementation of the securAX cybersecurity analysis platform, demonstrating how every architectural decision from Chapter 2 materializes into a working, production-hardened system.

The system overview confirmed that each scanning module --- from network reconnaissance and dynamic web analysis through static code auditing and dependency assessment --- operates as a cohesive unit within a unified interface, with ARIA providing the AI intelligence layer that elevates raw findings into actionable security intelligence.

The principal contribution is the **Server Configuration Audit Engine**: 2,339 lines of original Python implementing 34 Apache and 13 Nginx security checks aligned with CIS Benchmark v2.3.0 L2, a multi-layer parser that provides line-precise finding attribution, and an automated remediation engine that produces corrected, deployment-ready configuration files. The decision to build this component from scratch rather than use existing tools was driven by a concrete constraint --- the digital sovereignty requirement that sensitive configuration files must never leave the operator's controlled environment --- and reflects a genuine gap in the available open-source tooling.

The development of this engine was not without difficulty. The false positive problems uncovered in cipher and options parsing required abandoning naive string matching in favour of proper tokenization. The remediation engine required an additional correction cycle when re-scanning the generated fixed file revealed residual findings --- a bug in how multiple fixes targeting the same configuration line were applied sequentially rather than merged. These challenges are documented here not as failures but as a normal part of engineering work, and the test configuration iteration history in the repository provides a concrete record of how they were resolved.

Beyond the principal contribution, three additional original elements are implemented. The **multi-dimensional risk engine** (`risk_engine.py`) combines per-finding scoring, exponential decay aggregation, sigmoid normalisation, CISA KEV $\times 2.5$ factor, and five attack chain detection scenarios into a single callable that returns a structured `RiskBreakdown` dataclass --- operationalising the academic literature's call for context-aware prioritisation. The **parallel intelligence architecture** of the Web Security Scanner executes eleven external API calls concurrently via `ThreadPoolExecutor`, delivering comprehensive threat intelligence in near-real-time. The **bilingual PDF report generator** (ReportLab + Arabic-Reshaper + Python-Bidi) produces professional, compliance-ready reports in both Arabic and English, directly addressing the Algerian national context established in Chapter 1.

A deliberate sovereignty-driven decision shapes the DAST pipeline: Nuclei by ProjectDiscovery was evaluated and rejected because its Cloud API mode routes scan data through external infrastructure, violating the digital sovereignty principle that defines the platform. The DAST engine therefore relies exclusively on OWASP ZAP and Nikto --- both fully local tools that transmit data only to the scan target, never to third-party services. This decision trades some CVE template coverage for full sovereignty compliance, a trade-off documented in Chapter 4 as a direction for future work.

The technology choices documented in Section 3.4 reflect a deliberate engineering philosophy: leverage established, community-validated libraries and APIs for functionality that the open-source ecosystem already provides well, and invest original development effort exclusively where genuine tooling gaps exist. The result is a platform that is simultaneously broader in its detection coverage, deeper in its analysis capability, and more actionable in its output than any single existing open-source alternative.

General Conclusion

Summary of Contributions

This research addressed the profound structural fragmentation currently plaguing the cybersecurity analysis tooling ecosystem. Organizations, especially those bound by strict data sovereignty requirements, often struggle to synthesize a coherent security posture from disparate, isolated tools. To bridge this gap, we designed and implemented **securAX**, a unified, on-premises cybersecurity analysis platform.

The culmination of our work demonstrates a successful integration of six critical security dimensions into a single pane of glass: Dynamic Application Security Testing (DAST), Static Application Security Testing (SAST), Server Configuration Auditing, TLS Posture Assessment, HTTP Security Headers Inspection, and Software Composition Analysis (SCA).

The most prominent original contribution of this project is the **Server Configuration Audit Engine**. Engineered from scratch, this white-box analysis module accurately evaluates Apache and Nginx configurations against rigorous industry standards, identifying 34 distinct security and operational misconfigurations while offering fully automated, deployable remediations. Furthermore, by embedding the ARIA AI Assistant directly into the analytical workflow, securAX drastically reduces the cognitive load on security operators, transforming raw vulnerability data into actionable, contextualized intelligence.

Future Directions

While securAX establishes a formidable foundation for holistic infrastructure security, the rapidly evolving threat landscape dictates continuous platform evolution. Future work on this platform will focus on the following dimensions:

- **AI-Enhanced Vulnerability Correlation:** Moving beyond contextual analysis of single findings, future versions of ARIA will correlate vulnerabilities across network, application, and configuration layers to map complex, multi-stage attack vectors.
- **IoT and Embedded Systems Analysis:** Expanding the SAST and DAST engines to accommodate the unique protocols (e.g., MQTT, CoAP) and binary formats prevalent in Operational Technology (OT) and Internet of Things (IoT) ecosystems.
- **Automated SBOM Generation and Monitoring:** Integrating Software Bill of Materials (SBOM) generation into the CI/CD pipeline capabilities of securAX, providing real-time alerting when new CVEs are announced for deployed dependencies.

- **Distributed Scanning Agents:** Developing lightweight, deployable agent nodes to facilitate internal network scanning and continuous local configuration auditing across distributed enterprise environments.

Ultimately, securAX proves that a sovereign, integrated, and highly capable security platform is not only theoretically viable but practically achievable, offering a robust alternative to fragmented commercial solutions.

References

[1] JetBrains, *The State of Developer Ecosystem 2024*, https://www.jetbrains.com/lp/devecosystem-2024/, 2024.

[2] SQ Magazine, ``AI Coding Security Vulnerability Statistics 2026: Alarming Data,'' *SQ Magazine*, 2026.

[3] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, ``Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions,'' in *IEEE Symposium on Security and Privacy (S&P 2022)*, IEEE, 2022, pp. 754–768. DOI: 10.1109/SP46214.2022.9833571

[4] Gartner, ``Security and Risk Management Summit: Key Findings 2024,'' Gartner Research, Tech. Rep., 2024.

[5] Total Assure, *AI Cybersecurity Statistics in 2025: Comprehensive Data on Threats, Detection, and Defense*, https://totalassure.com, 2025.

[6] Verizon Business, ``2024 Data Breach Investigations Report (DBIR),'' Verizon, Tech. Rep., 2024, Analysis of 30,458 security incidents and 10,626 confirmed breaches.

[7] World Economic Forum, ``Global Risks Report 2024,'' World Economic Forum, Geneva, Switzerland, Tech. Rep., 2024.

[8] IBM Security, ``Cost of a Data Breach Report 2024,'' IBM Corporation, Tech. Rep., Jul. 2024.

[9] IBM Security, ``Cost of a Data Breach Report 2023 -- Middle East Regional Analysis,'' IBM Corporation, Tech. Rep., 2023.

[10] Ponemon Institute, ``State of Vulnerability Management 2024,'' Ponemon Institute, Tech. Rep., 2024.

[11] NIST National Vulnerability Database, *NVD CVE Statistics 2024*, https://nvd.nist.gov/general/nvd-dashboard, 2024.

[12] SonicWall, ``2024 SonicWall Cyber Threat Report,'' SonicWall, Tech. Rep., 2024.

[13] Agence Nationale de la Sécurité des Systèmes d'Information, ``Rapport Annuel sur la Cybersécurité en Algérie 2023,'' ANSI, Tech. Rep., 2023.

[14] IDC, *Worldwide Security Spending Guide, 2025--2028 Forecast*, https://www.idc.com, 2025.

[15] CISA, *Known Exploited Vulnerabilities (KEV) Catalog*, https://www.cisa.gov/known-exploited-vulnerabilities-catalog, Updated continuously; over 1,100 entries as of 2025, 2024.

[16] MITRE Corporation, *MITRE ATT&CK Enterprise Matrix, v14*, https://attack.mitre.org, 2023.

[17] OWASP Foundation, *OWASP Top 10 Web Application Security Risks, 2025 Edition*, https://owasp.org/Top10/, Current normative edition as of May 2026, 2025.

[18] Positive Technologies, ``Web Application Vulnerabilities Statistics 2023," Positive Technologies, Tech. Rep., 2023.

[19] OWASP Foundation, *OWASP API Security Top 10, 2023 Edition*, https://owasp.org/API-Security/editions/2023/en/0x00-header/, 2023.

[20] OWASP Foundation, *OWASP Top 10 for Large Language Model Applications, 2023*, https://owasp.org/www-project-top-10-for-large-language-model-applications/, 2023.

[21] Cobalt, ``State of Pentesting 2024," Cobalt.io, Tech. Rep., 2024.

[22] FIRST.org, ``Common Vulnerability Scoring System v3.1: Specification Document," Forum of Incident Response and Security Teams, Tech. Rep., 2019.

[23] SANS Institute, ``State of Application Security 2023," SANS, Tech. Rep., 2023.

[24] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006, ISBN: 978-0-321-35670-3.

[25] PyCQA, *Bandit: A Security Linter for Python Source Code*, https://github.com/PyCQA/bandit, 2024.

[26] Semgrep Inc., *Semgrep OSS -- Static Analysis at Ludicrous Speed*, https://semgrep.dev, 2024.

[27] Anchore Inc., *Grype -- A Vulnerability Scanner for Container Images and Filesystems*, https://github.com/anchore/grype, 2024.

[28] G. ". Lyon, *Nmap: Network Mapper*, https://nmap.org, Open-source network scanning tool; project homepage, 2024.

[29] Center for Internet Security, *CIS Apache HTTP Server 2.4 Benchmark, v2.3.0*, https://www.cisecurity.org/benchmark/apache_httpd, 2026.

[30] Qualys SSL Labs, *SSL Pulse -- Monthly Survey of TLS Implementation Quality*, https://www.ssllabs.com/ssl-pulse/, 2024.

[31] securityheaders.io, *Security Headers Adoption Report 2023*, https://securityheaders.com, 2023.

[32] DeepStrike, *86 Penetration Testing Statistics 2025: Trends and Takeaways*, https://deepstrike.io, 2025.

[33] TechMagic, ``State of Pentesting 2024 -- Cost and Methodology Analysis," TechMagic, Tech. Rep., 2024.

[34] PortSwigger, *Burp Suite Professional: Documentation and Pricing*, https://portswigger.net/burp/pro, 2024.

[35] République Algérienne Démocratique et Populaire, *Décret présidentiel n° 21-255 du 26 juin 2021 portant création de l'Agence Nationale de la Sécurité des Systèmes d'Information (ANSI)*, Journal Officiel de la République Algérienne, Jun. 2021.

Appendices

Appendix A: Environment and Deployment Prerequisites

The securAX platform is designed for rapid deployment in on-premises, disconnected environments to ensure maximum data sovereignty. To ensure optimal performance across all parallel scanning engines and the embedded AI assistant, the host machine must meet the following minimum specifications:

- **Operating System**: Ubuntu 22.04 LTS (Jammy Jellyfish) or a comparable modern Linux distribution.
- **Processor (CPU)**: 4 Cores minimum; 8 Cores strongly recommended for concurrent SAST and DAST operational modes.
- **Memory (RAM)**: 8 GB minimum; 16 GB recommended when the ARIA AI assistant is processing large repositories locally.
- **Storage**: 50 GB Solid State Drive (SSD) for logs, scan histories, persistent databases, and AI model caching.
- **Core Dependencies**:
  - Docker Engine (v24.0+)
  - Docker Compose (v2.20+)
  - Python 3.10 or higher
  - Node.js (v18 LTS) and npm

Appendix B: Supported Security Standards and Frameworks

securAX implements comprehensive security checks directly aligned with multiple global cybersecurity frameworks to assist organizations in maintaining compliance and facilitating continuous auditing:

- **OWASP Top 10 (2021)**: Full coverage for A01:2021-Broken Access Control, A03:2021-Injection, A05:2021-Security Misconfiguration, and A06:2021-Vulnerable and Outdated Components.

- **CIS Benchmarks**: Rigorous, automated evaluation against the *CIS Apache HTTP Server 2.4 Benchmark v2.3.0* (Level 1 and Level 2) and Nginx security guidelines.
- **NIST SP 800-53 (Rev. 5)**: Supports continuous monitoring (CA-7) and vulnerability scanning (RA-5) control families through automated, repeatable execution.
- **PCI-DSS v4.0**: Facilitates compliance with Requirement 6 (Develop and Maintain Secure Systems and Software) and Requirement 11 (Test Security of Systems and Networks Regularly) through its SAST and DAST integrations.