

Securaa Solution Architecture

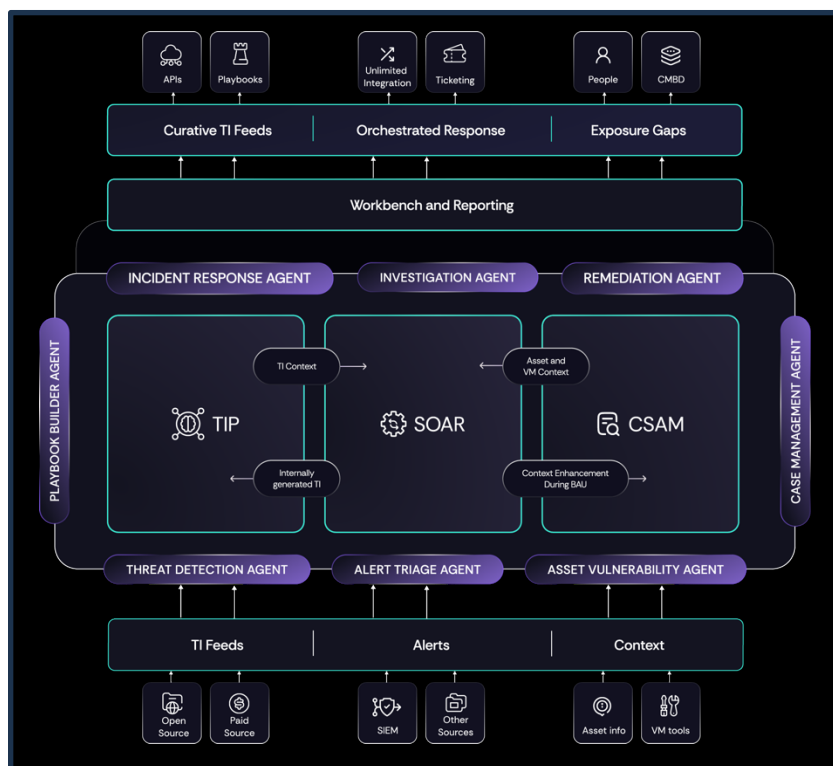
Introduction [🔗](#)

The purpose of this document is to identify and provide details of all the components that are part of Securaa SOAR software along with the communication mechanisms with the third-party tools.

Overview [🔗](#)

Securaa combines the advantages of an established threat intelligence platform (TIP), *Cyber security asset management (CSAM)*, and dependable security orchestration, automation, and response (SOAR).

Securaa collects and analyses security alerts, asset data, and indicators of compromise from a variety of sources, including SIEMs, asset databases, network security tools, threat intelligence feeds, and mailboxes. This information is used to provide visibility of all the cyber assets, and exposures and automate the incident response life cycle. Securaa's playbooks can coordinate across technologies, teams, and users to provide a single pane of glass for complete visibility and to perform triage on the incidents. Pre-configured playbooks and out-of-the-box API interfaces to help SOC's ability to reduce triage and reaction times.



Threat intelligence Platforms can aggregate indicators from commercial and open-source feeds, score them, and add context to the indicators. This enables analysts to utilize the indicators and administer protective and investigative security policies effectively while allowing predictive capabilities.

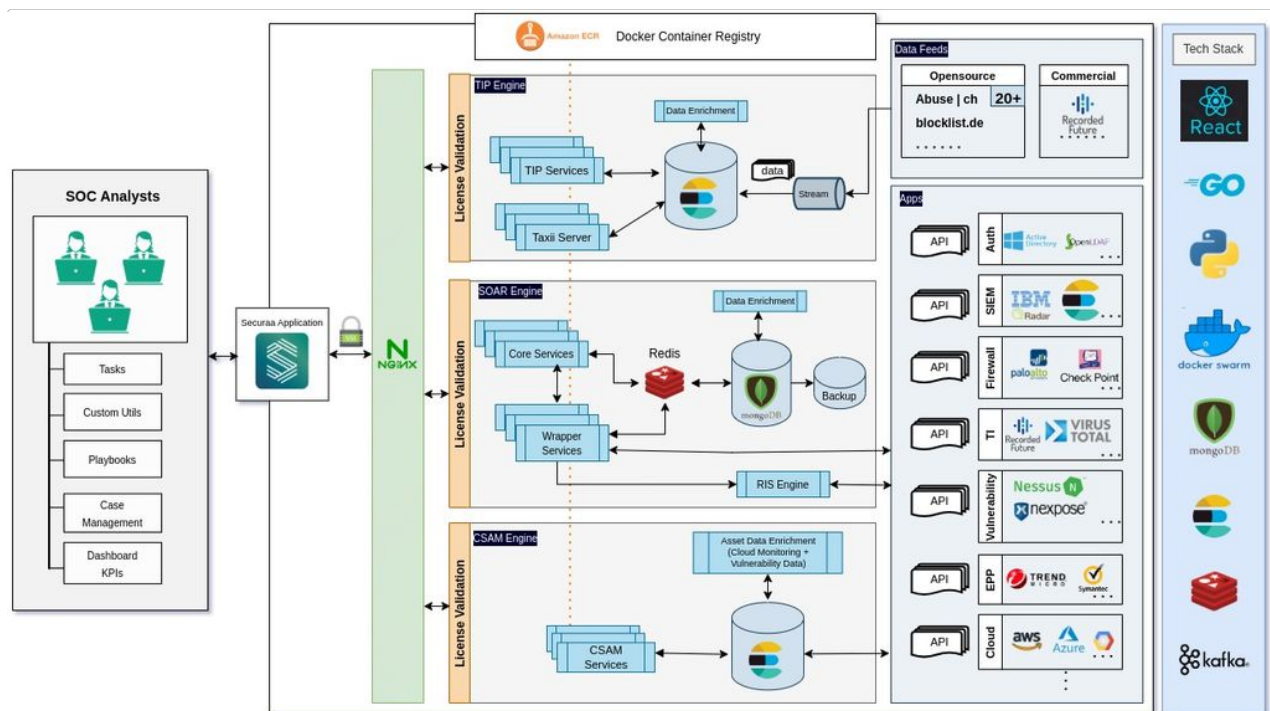
Cyber security asset management (CSAM provides visibility to the assets (on-prem and cloud), existing vulnerabilities, misconfiguration of cloud assets, users, and their associated accounts. This enables SOC teams to uncover security gaps, adds context to the incidents ingested in Securaa, and helps to improve the overall security posture.

Securaa Architecture [🔗](#)

The Securaa SOAR Architecture consists of the following components –

- Securaa UI
- API Gateway
- Wrapper Services
- Core Services
- Database

The following diagram provides a high-level Architecture view for Securaa SOAR.



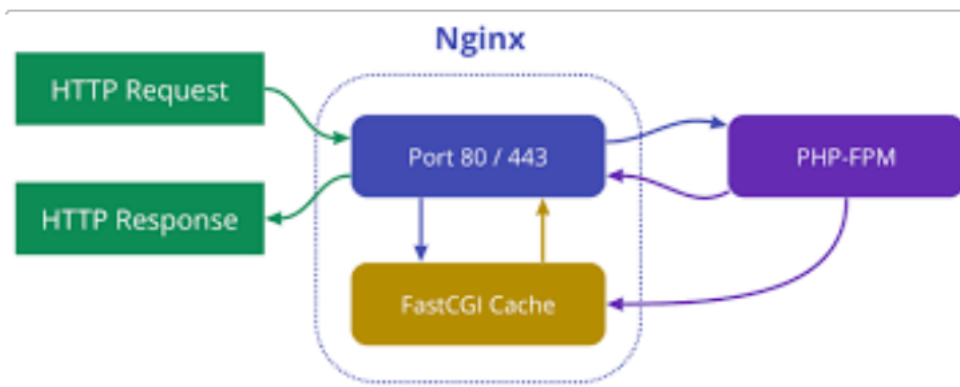
The detailed functions of the SOAR component are as follows:

Securaa UI [🔗](#)

The Securaa UI constitutes a web-centric interface developed using React and deployed on an Nginx server. This interface empowers users to adeptly configure, seamlessly integrate, proficiently manage cases, formulate playbooks, conduct investigations, engage in triage activities, and automate Security Operations Center (SOC) functions. These functionalities are facilitated through the utilization of RESTful APIs that establish communication with the backend APIs.

API Gateway [🔗](#)

The Securaa UI at the backend talks with the PHP gateway and further gateway talks with other core and wrapper services. Here the communication between UI and Nginx is happening through request forwarding over unix socket.



Core Services [🔗](#)

Fundamental to the functioning of Securaa, core services encompass a spectrum of crucial functionalities and serve as the bedrock of the system. Operating on Representational State Transfer (REST), these web services are universally accessible from any facet of the system. Their primary objective is to execute pivotal tasks, including the execution of playbooks, surveillance of Key Performance Indicators (KPIs), generation and distribution of scheduled reports, and the implementation of automated playbooks.

The dockerization of core services facilitates seamless scalability to meet varying demands. Serving as an integral component of Securaa, these core services establish the groundwork for the system's additional services and front end.

Wrapper Services [🔗](#)

Wrapper services within Securaa are REST-based services designed to execute tasks for integrations. Each wrapper service operates as a self-contained entity, distinct from any other system. By virtue of their independence from external components, these services can be swiftly deployed.

The intrinsic advantage of wrapper services lies in their capacity to comprehensively manage individual integrations and their associated tasks. This capability enhances Securaa's ability to exercise precise control over the data flow. Furthermore, the dockerization of these wrapper services streamlines the processes of development, testing, and deployment across diverse environments, thereby simplifying the maintenance of secure integrations.

Database [🔗](#)

Securaa relies upon **MongoDB** as its chosen repository for primary data storage. This NoSQL database distinguishes itself by offering exceptional flexibility, robust performance capabilities, and seamless scalability, making it eminently suitable for various applications. Within MongoDB, all configurations about users, platforms, integrations, playbooks, requests, and responses find their repository.

The advanced indexing functionalities and diverse data structures inherent to MongoDB empower Securaa to maintain strong consistency in data storage, thereby safeguarding against inconsistencies and potential data loss.

Moreover, Securaa leverages MongoDB's inherent features of automatic fail-over and high availability, ensuring the application's continued accessibility even in the event of an outage or a complete system crash. Lastly, MongoDB's capabilities facilitate straightforward backup and data restoration in the face of a disaster, enabling the reconstruction of the entire system with minimal effort and no compromise in data integrity.

Securaa significantly enhances scalability and performance by integrating **Redis** as an in-memory caching layer atop MongoDB using the Cache-Aside Pattern. Redis's lightning-fast data retrieval serves frequently accessed data directly from the cache, reducing the load on MongoDB. This setup minimizes database read operations, decreases latency, and enables the application to seamlessly handle a higher number of concurrent users. The combination of Redis and MongoDB ensures that as demand grows, the application can scale efficiently without compromising speed or user experience.

Solution Components [🔗](#)

Processes [🔗](#)

All components within Securaa are containerized using Docker, epitomizing a modular and efficient architectural approach.

The operational facet of Securaa hinges upon two exclusive binaries:

a) **zona_manager**

b) **zona_process_manager**

Within this paradigm, the zona_manager assumes the pivotal role of orchestrating the initiation of the zona_process_manager and supervising its ongoing functionality.




In turn, the zona_process_manager shoulders the critical responsibility of commencing all containers while diligently ensuring their sustained and stable operation, thereby mitigating the risk of untimely crashes. This binary encapsulates the core functionality of container instantiation and operational continuity within the Securaa ecosystem.




On Securaa's SOAR, the following docker services are running, each serving specific purposes:

Docker Service	Port	Protocol	Exposed	Purpose
securaa_ui_nginx	443	HTTPS	Yes	Web Server
zona_user	8000	HTTPS	Yes	Web Socket Communication
zona_apis_manager	8051	HTTPS	Yes	Core APIs
phpfpm	9000	HTTPS	No	API Gateway, used by only nginx
zona_siem	8223	HTTP	No	Cases related APIs
zona_dashboard	8090	HTTP	No	Dashboard Operations
zona_integrations	8005	HTTP	No	Wrapper Services APIs
zona_playbook	8040	HTTP	No	Playbook and Task Execution
zona_querybuilder	8072	HTTP	No	Query Filter
auto_purge_batch	10007	HTTP	No	Purging of Closed Cases
sla_breach_monitor_batch	10004	HTTP	No	Monitor SLA breach
case_group_batch	10009	HTTP	No	Case Grouping
core_process_batch	10005	HTTP	No	Pre-Process Rules
terminal	8100	HTTPS	No	Python utilites
zona_custom	8063	HTTP	No	No code integrations
zona_custom_utils	8109	HTTP	No	No code integrations
custom_fluentd	24224	HTTP	No	Centralized logging






jira_batch	10003	HTTP	No	JIRA Backend
MongoDB	27017	HTTP	No	Primary Database
python-utils	8110	HTTP	No	Utilities
python-utils-trial	5000-5100	HTTP	No	Utilities in Building/Execution
report_batch	10002	HTTP	No	Report Generation
sbot	10001	HTTP	No	Case Enrichment
servicenow_batch	10006	HTTP	No	Ticketing Backend
tip_localdata	8219	HTTP	No	TIP Controller
zona_sshhclient	8223	HTTP	No	Remote Server Management
zona_ris_server	8057	HTTP	No	Serve RIS Client
connector_health	10008	HTTP	No	Health Monitoring
utils	8100	HTTPS	Yes	Utility
task_handler	8058	HTTPS	Yes	Task handling for RIS
Redis	4380	HTTP	No	In-memory DB for caching

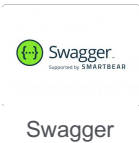


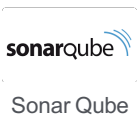

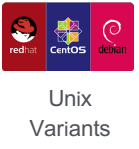
Technical Stack [🔗](#)

Logo	Purpose
 <p>React JS</p>	Web Interface
 <p>Go Language</p>	Backend Services
 <p>MongoDB</p>	Data Storage

 <p>Shell Scripting</p>	Deployment Scripts
 <p>Python</p>	Utilities
 <p>Redis</p>	In-memory DB for Caching

Deployment Stack [🔗](#)

Deployment	Purpose
 <p>Jenkins</p>	Packaging and Deployment
 <p>AWS</p>	Infra Management
 <p>Code Commit</p>	Repository Management(GIT)
 <p>Code Build</p>	Docker images building
 <p>AWS ECR</p>	Docker Images Registry

	Open APIs
	Static Analysis of React Code
	Static Analysis for Golang Code (lint, fmt, vet and gosec)
	Static Analysis using Sonar Qube (With default rules available)
	Secure AWS infrastructure
	Support all Debian and RHEL-based OS for eg. RHEL, Fedora, Centos, Rocky, Alma, Ubuntu etc.

Web Server [🔗](#)

The system is orchestrated through the utilization of the Nginx web server, predominantly interfacing via port 443. Although the web server itself maintains a relatively unrestricted communication environment, meticulous communication constraints have been strategically implemented at the Transport Layer Security (TLS) protocol level.

Specifically, communication has been confined to TLS versions 1.2 and 1.3 purposefully. This deliberate limitation serves as a judicious measure, guaranteeing that all connections facilitated by our Nginx web server strictly adhere to these robust and secure TLS versions. The implementation of such selective restrictions significantly fortifies the security posture of Securaa's communication framework.

Rest Services [🔗](#)

All the services on Securaa have been dockerized and are being executed as Rest Services. REST services operate behind an API gateway that communicates directly with Nginx. We utilize the latest PHP-FPM as our gateway, providing the following benefits:

Performance Enhancement: Segregates PHP processing, leading to improved speed.

Scalability: Efficiently adjusts to different traffic volumes.

Improved Security: Isolates applications, minimizing potential risks.

Resource Management: Fine-tunes settings to optimize performance.

Enhanced Stability: Safeguards the entire server from crashes in individual processes.

Monitoring and Management Tools: Offers real-time tools for effective supervision and control.

Licensing

We manage product licensing via a server deployed within Securaa's AWS infrastructure. Should there be a necessity to update, renew, or modify a license, users are required to request these changes from us. Typically, we respond to these requests within 24 hours by providing the updated license.

Our licensing server operates continuously, 24 hours a day and 7 days a week. Every time a user logs into any development environment, the license validation occurs through this server.

Upgrade

Typically, we release product upgrades every two months. However, in the event of specific custom requirements, we develop patches and deploy them within a 15-day timeframe. Hotfixes are promptly addressed within a 3-day window, followed by the provision of a subsequent patch to address the issue comprehensively.

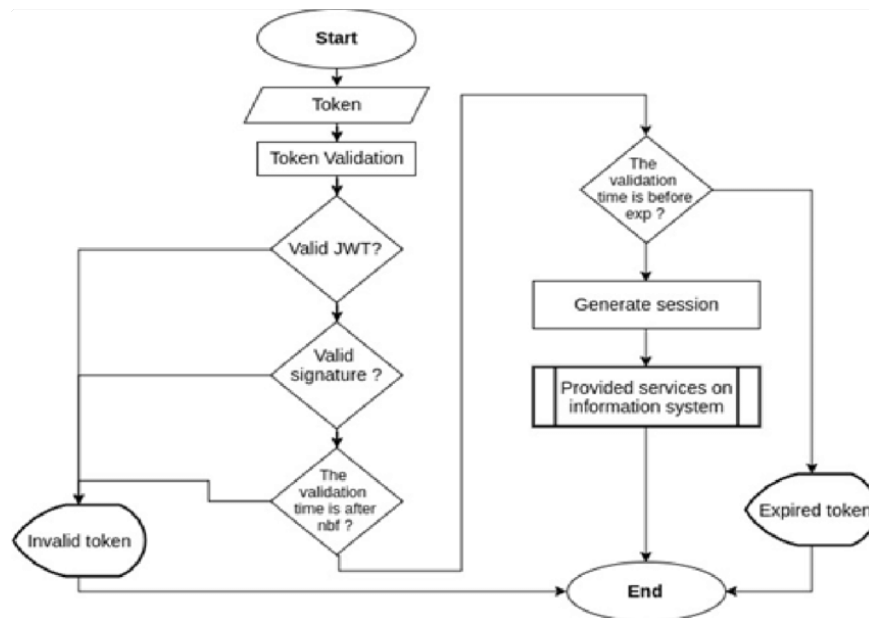
Solution Security

API Security

The Securaa web UI utilizes a PHP-FPM gateway to handle every API consumed through the web interface. This gateway subsequently processes these requests to other services and retrieves the responses.

To authenticate an API request, the initial step involves calling the login API embedded within the `zona_user` module and running over HTTPS. This process requires the user's email and password to obtain a JWT token in response. The hashed (SHA256) credentials are stored within the database, and the validation process involves comparing the provided credentials against those stored for each login.

The JWT token follows standard practices, and the flow diagram detailing this process is below:



Data Security [🔗](#)

The API data and offenses are stored in MongoDB and logical separation is done for each tenant database.

Data at rest is managed in the following way.

Secrets	Storage	Encryption
Email and Password required for login	The email is stored in a clear format as it's considered public information. However, for passwords, we store their hashed versions to maintain the privacy of each user's password. This hashed format (using SHA256) allows for validation against the stored hash without compromising the password's confidentiality.	SHA256
Integrated application credentials	The AES256 key is securely stored and utilized for encrypting/decrypting the application's credentials.	AES256
Playbook execution, task executing, reports, dashboards etc.	The data is kept within MongoDB collections and protected by MongoDB credentials, which are not accessible from the external environment.	Relying on mongoDB authentication.
Downloads, logs, custom utilities, History of playbooks and dashboards	HDD	Application directory in clear format.

Application Security [🔗](#)

The Securaa web application operates within a firewall (firewalld), and the ports accessible to the outside world are as listed below:

Service	Port	Purpose
Nginx (web server)	443	User interface as web application
Licencing APIs	8051	License control APIs
Web Sockets	8000	Serve web socket requests

The exposed ports operate using HTTPS and are exclusively enabled for TLS1.2 and TLS1.3.

Database Security [🔗](#)

The credentials for the primary database, often referred to as the Core DB, are stored in a binary format. This binary storage method involves encoding the credentials in a way that makes it challenging to interpret or recognize their content by anyone

viewing the stored data directly. This adds a layer of security, as the information is not easily identifiable, providing an extra level of protection against unauthorized access.

On the other hand, for tenant databases, which might involve various databases dedicated to different users or entities, a different approach is taken. The credentials related to these tenant databases undergo AES256 encryption before being stored in a database collection. This encryption process involves encoding the credentials using the AES256 algorithm, a widely used and secure encryption method. Once encrypted, these credentials are then safely stored within a designated collection in the database, ensuring that even if someone gains access to the stored information, it remains unreadable without the decryption key. This method aims to enhance the security of the tenant database credentials, safeguarding them from potential breaches or unauthorized access.

Development Process [🔗](#)

Agile development methodology for our product's evolution is followed, conducting two 3-week sprints to deliver software upgrades. CI/CD processes are facilitated by Jenkins. Two server are provisioned daily, one dedicated to development and the other for our testing team. This setup allows for simultaneous issue identification and resolution.

On completion of developing items, developers commit their code using AWS Code Commit. An approval rule mandates that the code undergoes review by two approvers. After a thorough review process, the code is merged into the main trunk.

Developers must craft and execute unit test cases for the specific feature or functionality they are working on before writing the code.

Testing is carried out through various methodologies:

1. Regular Testing: Conducted daily on a dedicated nightly server.
2. Regression Testing: Implemented every two releases, involving a rigorous 15-day cycle for in-depth product testing.
3. Load/Performance Testing: Performed using a machine equipped with 16-core CPUs and 32 GB RAM. Issues identified during load testing are promptly addressed.

Our system operates with 50+ tenants and processes 100,000 cases considering case enrichment, case grouping, and pre-process rules, each ingested case is enabled.

Regarding automation, sanity suites executed are maintained at each release. These suites continually expand with additional cases, executed nightly via Playwright Automation on servers.