PROJECT

Multi-tenant cybersecurity operations centre designed
as an open cloud service with machine learning elements

V14 functional infrastructure

KPB5 – Implementation of the solution

type – documentation

month of delivery – M19

Funded by the European Union
NextGenerationEU

RECOVERY AND RESILIENCE PLAN

GOVERNMENT OFFICE OF THE SLOVAK REPUBLIC

Contents

1 ELK Stack 7
1.1 System description 7
1.2 System documentation 7
1.3 Resource requirements 8
1.4 Tool architecture 9
1.4.1 ELK Stack architecture diagram 10
1.5 Installation – Docker variant 10
1.6 Basic configuration 12
1.7 Integrations 15
2 Shuffle 16
2.1 System description 16
2.1.1 Orchestration and automation of responses (Playbooks) 16
2.2 System documentation 17
2.3 Resource requirements 17
2.4 Tool architecture 17
2.4.1 Internal architecture 18
2.5 Installation – Docker variant 19
2.6 Installation – OpenStack variant 21
2.7 Basic configuration 22
2.7.1 Connections to the external applications Wazuh and Greenbone 23
2.7.2 Inputs and outputs 23
2.8 Integrations 24
2.8.1 Position of Shuffle in the SOC 24
3 Wazuh 26
3.1 System description 26
3.2 System documentation 26
3.3 Resource requirements 27
3.4 Tool architecture 27
3.4.1 Diagram 28
3.5 Installation – Quickstart variant 29
3.6 Basic configuration 30
3.7 Integrations 31
4 Greenbone Community edition 32

Funded by the European Union NextGenerationEU
RECOVERY AND RESILIENCE PLAN
GOVERNMENT OFFICE OF THE SLOVAK REPUBLIC

4.1 System description....32
4.2 System documentation....33
4.3 Resource requirements....33
4.4 Tool architecture....33
4.4.1 Diagram....34
4.5 Installation – Docker variant....35
4.6 Basic configuration....40
5 OpenCTI....41
5.1 System description....41
5.2 System documentation....42
5.3 Resource requirements....42
5.4 Tool architecture....42
5.5 Installation – Docker variant....44
5.5.1 Installation steps:....44
5.6 Installation – OpenStack variant....46
5.6.1 Installation steps....46
5.7 Basic configuration....48
5.7.1 MITTRE ATTACK configuration....48
5.7.2 Urlhaus configuration....49
5.7.3 Ransomwarelive configuration....49
5.7.4 Inputs and outputs....50
5.8 Integrations....51
5.8.1 OpenCTI integration – the position of CTI in the SOC and the integration model 51
6 MISP....53
6.1 System description....53
6.2 System documentation....53
6.3 Resource requirements....54
6.4 Tool architecture....54
6.5 Integrations....56
6.5.1 Position of MISP in the SOC....56
6.5.2 Integration of MISP with other SOC components....56
6.5.3 OpenCTI versus MISP....57
7 GLPI....59
7.1 System description....59

7.2 System documentation....59
7.3 Resource requirements....60
7.4 Tool architecture....60
7.4.1 Diagram....61
7.5 Installation – Docker (Compose) variant....62
7.6 Basic configuration....63
7.7 Integrations....64
7.7.1 Position of GLPI in the SOC....65
8 NetBox....66
8.1 System description....66
8.2 Documentation....66
8.3 Resource requirements....67
8.4 Tool architecture....67
8.4.1 Infrastructure modelling....68
8.5 Experiment....70
8.5.1 Diagram....71
8.5.2 Installation – Docker variant....71
8.5.3 Installation – OpenStack variant....88
8.5.4 Basic configuration....89
8.5.5 Integrations....90
8.5.6 Integration of NetBox with SOC components....91
8.5.7 NetBox versus GLPI....92
9 DFIR IRIS – Ticketing....94
9.1 System description....94
9.2 System documentation....94
9.3 Installation – Docker variant....96
9.4 Installation – OpenStack variant....97
9.5 Basic configuration....100
9.5.1 Inputs/Outputs....100
9.6 Integrations....100
10 DFIR IRIS – Incident recording and management....102
10.1 System description....102
10.2 System documentation....102
10.3 Installation – Docker variant....104

10.4 Installation – OpenStack variant 105
10.5 Basic configuration 108
10.5.1 Inputs/Outputs 109
10.6 Integrations 109
11 Kibana (ELK) 110
11.1 Tool description 110
11.2 System documentation 110
11.3 HW requirements 111
11.4 Tool architecture 111
11.4.1 Diagram 111
11.5 Installation – Docker variant 112
11.6 Basic configuration 113
11.7 Integrations 114
12 Wiki.js 115
12.1 System description 115
12.2 System documentation 115
12.3 Resource requirements 116
12.4 Tool architecture 116
12.4.1 Diagram 116
12.5 Installation – Docker variant 117
12.6 Basic configuration 118
12.7 Integrations 119
13 Deployment environment 120
13.1 OpenStack KIS FRI UNIZA 120
13.1.1 Context of the CC solution 120
13.1.2 OpenStack architecture 120
13.1.3 Cloud solution – OpenStack as the platform for INNOSOC 122
13.1.4 Infrastructure layer of the environment 124
13.1.5 Hardware components of the environment 127
13.1.6 Placement and isolation of SOC components in the environment 130
13.2 Kubernetes Brainit 131
13.2.1 Description of the Kubernetes infrastructure 131
13.2.2 Virtual machine infrastructure – IaaC 133
13.2.3 Repository structure 135

13.2.4 Management and deployment of Kubernetes applications....137
13.3 Docker Brainit....137
13.3.1 Description of the docker infrastructure....137
13.3.2 Operating system of the virtual servers....138
13.3.3 Containerization layer....138
13.3.4 Application orchestration....139
13.3.5 Overall architecture of the solution....139
14 Conclusion....141

1 ELK STACK

Tool type: SIEM / Log management / Security analytics

Tool name: ELK Stack (Elasticsearch + Logstash/Beats + Kibana)

Version: ELK Stack 8.x (deployment via official Docker images)

1.1 System description

The ELK Stack provides extensive support for the collection of logs and telemetry through a family of so-called *data shipper* tools (e.g. Filebeat, Winlogbeat, Auditbeat), which enable the collection of events from various types of systems, including Windows, Linux, security audit records and selected application services. These components are designed to offload the source systems and reliably deliver data to the processing layer or directly to Elasticsearch.

At the data processing level, Logstash handles parsing, normalization and enrichment of data (e.g. decomposition of syslog messages, mapping to uniform fields and unification of formats), thereby preparing the data for efficient indexing. Elasticsearch then provides a powerful indexing and query engine that enables fast searching, aggregations and temporal correlations over large volumes of events. On top of this foundation, Kibana provides a user interface for analysis, dashboard creation, filtering and working with time series, which is key in a SOC environment for incident triage and investigation.

From a detection and response perspective, alerting mechanisms (rules and notifications) can be built on top of the data so that suspicious events are escalated to analysts or to downstream processes. The scope of available features depends on the chosen distribution and licensing model: in the Elastic ecosystem, many "core" capabilities are available even without paid editions, whereas more advanced analytical and enterprise functionalities (e.g. selected ML/UEBA scenarios or extended integration options) may be tied to higher plans. In the case of an alternative implementation built on OpenSearch, alerting is typically handled via a separate alerting module/plugin.

From a security point of view, when designing a SOC solution it is important to address the protection of communication between components (encryption), authentication and authorization (access management) and access auditing.

1.2 System documentation

The official ELK Stack documentation is publicly available and provides a detailed description of the architecture, installation, configuration and operation of the individual components. The documentation also covers recommended practices for designing ingest pipelines (parsing, normalization and enrichment), indexing strategy, searching and aggregations, creating visualizations in Kibana, as well as connecting data sources through data-shipper tools (Beats) or a unified approach via Elastic Agent/Fleet.

Official sources (Elastic Docs):

- Elastic documentation portal (overview of all products and versions): https://www.elastic.co/guide/index.html
- Elasticsearch – reference documentation: https://www.elastic.co/docs/reference
- Elasticsearch – integrations: https://www.elastic.co/integrations/data-integrations
- Logstash – reference documentation: https://www.elastic.co/docs/reference/logstash
- Kibana – user and reference documentation: https://www.elastic.co/guide/en/kibana/index.html

Unofficial sources:

- Docker Compose documentation for installation: https://github.com/deviantony/docker-elk

Data collection (logs/events) – Beats and Elastic Agent:

- Fleet & Elastic Agent (central management of agents and integrations): https://www.elastic.co/docs/reference/fleet
- Migration from Beats to Elastic Agent (practical transition guide): https://www.elastic.co/docs/reference/fleet/migrate-from-beats-to-elastic-agent
- Filebeat / Winlogbeat / Auditbeat (versioned manuals are available via the Elastic docs index): https://www.elastic.co/guide/index.html
- Beats GitHub repository (project overview and links to documentation): https://github.com/elastic/beats

Note: When implementing in a SOC environment, it is advisable to work with the documentation in a version-consistent manner (e.g. all components within a single major version 8.x) in order to avoid differences in configuration, ingest pipeline compatibility and the behaviour of the Kibana UI.

1.3 Resource requirements

The hardware requirements of the ELK Stack depend primarily on the volume of data processed, the number of log sources, the retention policy and the complexity of analytical queries and visualizations. In a SOC environment, it is necessary to allow for a continuous flow of data and a requirement for fast response when searching and correlating events.

For the purposes of a Proof of Concept (PoC) and small to medium-sized deployments, the following minimum parameters are recommended:

- CPU: 4 vCPU
- RAM: 8–16 GB (critical especially for Elasticsearch)
- Disk space: min. 100 GB (SSD recommended, depending on data retention settings)
- Disk: local SSD or fast network storage (IOPS are key)

- Network: stable connectivity between log sources and the ingest layer

In a production SOC environment, it is common to:

- separate the ingest layer (Logstash/Beats) from the database layer,
- deploy Elasticsearch in clustered mode,
- use separate nodes for hot/warm data according to the retention policy.

For the purposes of this work, however, the ELK Stack is deployed in a simplified PoC configuration, which enables verification of the detection, analysis and visualization functionality without the need for a fully fledged production cluster.

1.4 Tool architecture

The deployment architecture of the ELK Stack within the SOC is designed as a multi-layer solution that separates data collection, processing, storage and analytical presentation. This approach increases scalability and clarity and enables gradual expansion of the solution.

The input layer contains the data sources, which include operating systems (Linux, Windows), network devices and security sensors (e.g. IDS/IPS). Log collection is carried out by means of data-shipper tools (Filebeat, Winlogbeat, Auditbeat) or by means of syslog mechanisms. Their task is to reliably deliver events to the processing layer without significantly loading the source systems.

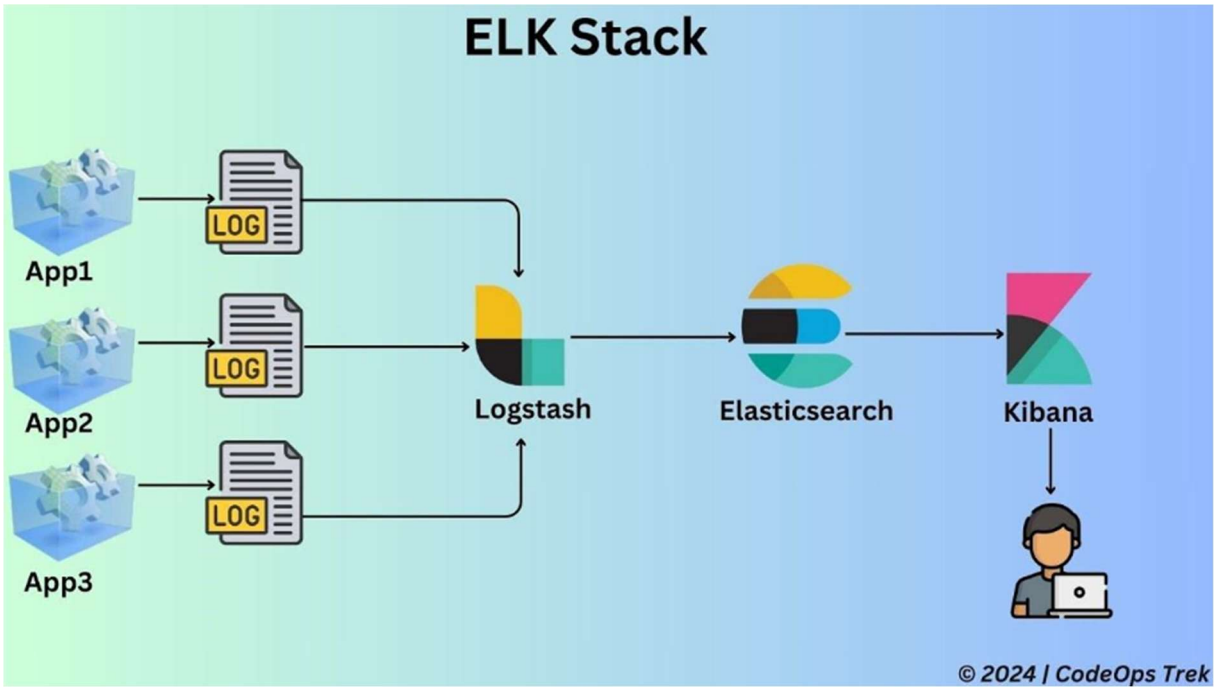
The processing layer consists of Logstash, which handles parsing, normalization and basic enrichment of the data. In this phase, heterogeneous inputs are transformed into a uniform structure, which enables efficient indexing and subsequent event correlation. Logstash is also the place where basic logic for sorting events by their type or severity can be implemented.

The data and analytical layer is represented by Elasticsearch, which stores the processed events in indices optimized for time-based queries and aggregations. Elasticsearch forms the core of the SIEM functionality, as it enables fast searching, correlation of events over time and support for rule-based detection mechanisms.

Above this layer is Kibana, which provides the user interface for SOC analysts. It enables the creation of dashboards, visualization of trends, filtering of events and detailed analysis of incidents. Kibana is the main working tool of the SOC analyst during the triage and investigation of security events.

From the perspective of the SOC architecture, the ELK Stack is complemented by contextual sources, such as the NetBox documentation system. These systems do not provide detection as such, but supply analysts with essential context (information about assets, IP addressing, location), which increases the accuracy of event interpretation and reduces the number of false alarms.

1.4.1 ELK Stack architecture diagram




```
[ Log sources / sensors ]
         |
    (Beats / Syslog)
         |
      [ Logstash ]
         |
   [ Elasticsearch ]
         |
      [ Kibana ]
         |
    [ SOC analyst ]
```

1.5 Installation – Docker variant

```
services:
  elasticsearch:
    build:
      context: elasticsearch/
      args:
        ELASTIC_VERSION: ${ELASTIC_VERSION}
    volumes:
      -
./elasticsearch/config/elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml:
ro,Z
      - elasticsearch:/usr/share/elasticsearch/data:Z
    ports:
      - 9200:9200
      - 9300:9300
    environment:
```

```
      node.name: elasticsearch
      ES_JAVA_OPTS: -Xms512m -Xmx512m
      # Bootstrap password.
      # Used to initialize the keystore during the initial startup of
      # Elasticsearch. Ignored on subsequent runs.
      ELASTIC_PASSWORD: ${ELASTIC_PASSWORD:-}
      # Use single node discovery in order to disable production mode and avoid bootstrap
checks.
       # see:
https://www.elastic.co/guide/en/elasticsearch/reference/current/bootstrap-
checks.html
      discovery.type: single-node
    networks:
      - elk
    restart: unless-stopped
  logstash:
    build:
      context: logstash/
      args:
        ELASTIC_VERSION: ${ELASTIC_VERSION}
    volumes:
      - ./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml:ro,Z
      - ./logstash/pipeline:/usr/share/logstash/pipeline:ro,Z
    ports:
      - 5044:5044
#      - 50000:50000/tcp
      - 514:50000/tcp
#      - 50000:50000/udp
      - 514:50000/udp
      - 9600:9600
    environment:
      LS_JAVA_OPTS: -Xms256m -Xmx256m
      LOGSTASH_INTERNAL_PASSWORD: ${LOGSTASH_INTERNAL_PASSWORD:-}
    networks:
      - elk
    depends_on:
      - elasticsearch
    restart: unless-stopped
  kibana:
    build:
      context: kibana/
      args:
        ELASTIC_VERSION: ${ELASTIC_VERSION}
    volumes:
      - ./kibana/config/kibana.yml:/usr/share/kibana/config/kibana.yml:ro,Z
    ports:
      # - 5601:5601
      - 80:5601
    environment:
      KIBANA_SYSTEM_PASSWORD: ${KIBANA_SYSTEM_PASSWORD:-}
      SERVER_PUBLICBASEURL: http://elk.innosoc.brainit.tech
    networks:
      - elk
    depends_on:
      - elasticsearch
    restart: unless-stopped
```

```
networks:
  elk:
    driver: bridge
volumes:
  elasticsearch:
```

Changes in docker compose: Some network ports have been changed – see the commented-out lines above.

1.6 Basic configuration

Tool configuration:

ELK Stack contains a configuration script that correctly sets the addresses and the necessary authentication between the services. We then left most of the configuration at the default settings. The rest of the configuration then consists only of adapting to the requirements or plugins and setting data retention.

Logstash configuration for syslog from Fortigate 200

`logstash/pipeline/fortigate.conf`

```
input {
  # Listen on the CONTAINER port 50000/udp, mapped from host port 514/udp
  # If you chose TCP on FortiGate, change 'udp' to 'tcp' here.
  udp {
    port => 50000 # <<< This is the CONTAINER port mapped in docker-compose.yml
    codec => plain { charset => "UTF-8" }
    type => "fortigate" # Add a type for easier filtering later
    tags => ["fortigate", "udp"]
  }

  # --- OR if you chose TCP on the FortiGate ---
  # tcp {
  #   port => 50000 # <<< CONTAINER port
  #   codec => line { charset => "UTF-8" } # Use 'line' codec for TCP
  #   type => "fortigate"
  #   tags => ["fortigate", "tcp"]
  # }
}

filter {
  # Only process messages tagged as fortigate AND likely containing key-value pairs
  if "fortigate" in [tags] and [message] =~ /type=/ {

    # Parse the FortiGate key-value format
    kv {
      source => "message"
      field_split => " "
      value_split => "="
      trim_key => "\""
```

```
      trim_value => "\""
      # prefix => "fortigate_" # Optional: Uncomment to prefix all fields
    }

    # Convert known numeric fields
    mutate {
      convert => {
        "rcvdbyte" => "integer"
        "sentbyte" => "integer"
        "duration" => "integer"
        "proto" => "integer" # Protocol number (e.g., 6 for TCP)
        "srcport" => "integer"
        "dstport" => "integer"
        "policyid" => "integer"
        # Add other common numeric fields: devid, vd, sessionid, count, etc.
      }
      # Remove quotes from potentially problematic fields if kv didn't catch them
      gsub => [
        "srcip", '"', '',
        "dstip", '"', '',
        "action", '"', '',
        "service", '"', ''
        # Add others as needed
      ]
    }

    # Parse timestamp from 'date' and 'time' fields
    if [date] and [time] {
      mutate {
        add_field => { "event_timestamp_str" => "%{date} %{time}" }
      }
      date {
        match => [ "event_timestamp_str", "yyyy-MM-dd HH:mm:ss" ] # Verify this format
matches your logs!
        target => "@timestamp"
        # timezone => "Your/Fortigate/Timezone" # IMPORTANT: Set if FortiGate isn't UTC and
doesn't include offset
      }
      mutate {
        remove_field => [ "event_timestamp_str" ]
      }
    }

    # GeoIP Enrichment (Optional, requires plugin install - see below)
    # if [srcip] {
    #   geoip {
    #     source => "srcip"
    #     target => "geoip_src"
    #     # database => "/path/inside/container/to/GeoLite2-City.mmdb" # Needs GeoIP
database mounted or built into image
    #   }
    # }
    # if [dstip] {
    #   geoip {
    #     source => "dstip"
    #     target => "geoip_dst"
```

```
    #     # database => "/path/inside/container/to/GeoLite2-City.mmdb"
    #   }
    # }

    # Optional: Rename fields for ECS compatibility if desired
    # mutate {
    #   rename => { "srcip" => "source.ip" }
    #   rename => { "dstip" => "destination.ip" }
    #   # ... more renames
    # }

  } else {
    # Optionally drop messages that don't look like FortiGate logs if other
    # things might arrive on this port (though dedicated port is better)
    # drop { }
  }
}

output {
  elasticsearch {
    # Use the service name 'elasticsearch' as host because they are on the same 'elk'
docker network
    hosts => ["http://elasticsearch:9200"]
    index => "fortigate-%{+YYYY.MM.dd}" # Daily indices

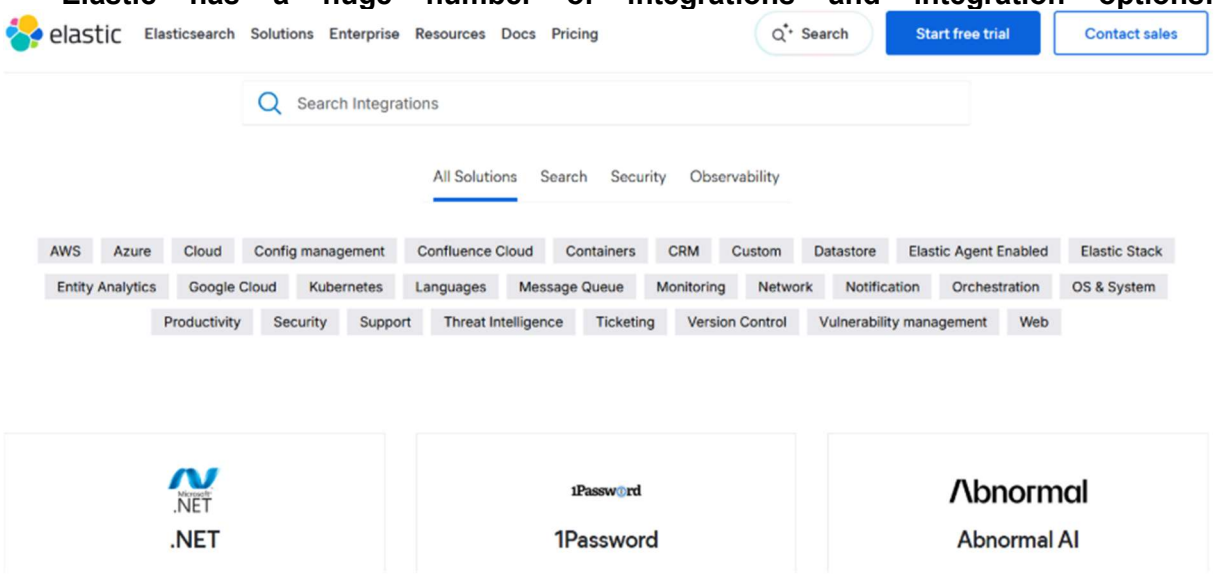
    # === Authentication ===
    # Use the internal user and password defined in your .env file and passed to the
container
    user => "logstash_internal"
    password => "${LOGSTASH_INTERNAL_PASSWORD}" # Logstash reads this from its environment

    # === Index Lifecycle Management (Recommended for Production) ===
    # Setup an ILM policy in Kibana first (e.g., named 'fortigate_policy')
    # ilm_enabled => true
    # ilm_rollover_alias => "fortigate"
    # ilm_policy => "fortigate_policy" # Make sure this policy exists in
Elasticsearch/Kibana
    # ilm_pattern => "000001"
  }

  # Optional: Output to container logs for debugging during setup
  # Remove or comment out for production.
  # stdout {
  #   codec => rubydebug
  # }
}
```

1.7 Integrations

Elastic has a huge number of integrations and integration options:



The stack also includes Logstash. Once Logstash has been correctly configured with how it should process specific syslog logs, it can be integrated with any system that sends syslog. An example of such an integration is the simple integration that we wrote for receiving logs from the Fortigate firewall.

2 SHUFFLE

Tool type: SOAR (Security Orchestration, Automation and Response)

Tool name: Shuffle

Version: Release 2.1.0 (open-source version (containerized deployment))

2.1 System description

Shuffle is an open-source SOAR (Security Orchestration, Automation and Response) tool designed for the orchestration and automation of security processes in a SOC environment. Its main task is to reduce the manual workload of analysts by automating recurring response and enrichment activities that follow on from the detection mechanisms of the SIEM system.

The tool provides a web user interface in which automated workflows, so-called *playbooks*, can be visually designed, managed and run. Shuffle is designed as a lightweight and flexible orchestrator capable of interconnecting security and IT tools via their APIs without the need for extensive infrastructure or complex integration platforms.

Shuffle is neither a fully fledged incident management system nor a replacement for a SIEM solution. Its role is not the detection of events or their long-term analytics, but rapid response, automated event processing and the integration of existing tools. In the SOC architecture, it therefore acts as the response and automation layer, which follows on from detection (ELK Stack) and works with context supplied by other systems (e.g. NetBox).

In the proposed solution, Shuffle is used to automate selected response scenarios, to enrich incidents with external information and to support a consistent procedure in the processing of security events.

2.1.1 Orchestration and automation of responses (Playbooks)

The basic function of the Shuffle tool is the creation and execution of automated playbooks, which represent a sequence of steps responding to a security event. Playbooks are created graphically using a drag-and-drop interface, with the individual steps representing API calls, data processing or decision logic.

Shuffle makes it possible to create:

- simple workflows (e.g. notification, IP address enrichment),
- complex workflows with conditions, branching and feedback loops.

Playbooks can be triggered:

- manually (by a SOC analyst),
- via a webhook,
- automatically in response to an event from an external system (e.g. SIEM).

From the perspective of SOC processes, Shuffle can be used mainly in the Response phase, where it can:

- automatically enrich an incident with external data,

- perform predefined response steps,
- ensure a uniform and repeatable procedure in the processing of events.

2.2 System documentation

The official documentation of the Shuffle tool is publicly available and provides an overview of the architecture, integration options, playbook design and the method of deployment. The documentation focuses mainly on the practical use of the tool in a SOC environment, including working with webhooks, API connectors and the workflow engine.

Links to official sources:

- Official documentation: https://shuffler.io/docs/about
- Project GitHub repository: https://github.com/shuffle/shuffle
- Official Docker image: https://hub.docker.com/r/shuffle/shuffle

2.3 Resource requirements

Shuffle has low hardware requirements, which enables its rapid deployment even in smaller or test environments.

For a test or smaller production deployment, the following is sufficient:

- CPU: 2 cores
- RAM: 4 GB
- Disk space: 20–50 GB

For larger production environments, the following is recommended:

- CPU: 4–8 cores
- RAM: 8–16 GB
- Disk: SSD storage

Higher requirements apply mainly when operating a larger number of workflows, integrations and parallel response processes.

2.4 Tool architecture

The architecture of the Shuffle tool is designed as a modular and lightweight solution that can easily be integrated into an existing SOC infrastructure. The basis of the architecture consists of a web application layer, which provides the user interface for designing and managing playbooks, and a backend API layer, which handles request processing and communication with external systems.

The solution includes a workflow engine, which executes the individual steps of the playbooks and controls their processing logic. Shuffle communicates with the integrated tools through API connectors, thereby enabling automated calls to external services and processing of their responses. An internal database is used to store configurations, workflow states and the results of individual steps.

In the SOC architecture, Shuffle is positioned above the detection layer (SIEM) and serves as an automation component that responds to events identified by other tools. The architecture diagram of the Shuffle tool used in this work is based on the official documentation and illustrates the basic components and their relationships.

2.4.1 Internal architecture

The internal architecture of the Shuffle tool divides its functionalities into separate logical components. The architecture is modular in design, with a clear separation of the user interface, application logic, workflow engine and integration components, which enables flexible deployment in both on-premise and hybrid environments.

2.4.1.1 Frontend and backend layer

The user interface (frontend) is implemented as a web application used for designing, managing and running playbooks. The frontend communicates with the backend exclusively via a REST API, which ensures the separation of the presentation layer from the application logic.

The backend is implemented in the Go language and constitutes the control part of the system. It is responsible for:

- playbook management,
- processing requests from the frontend,
- communication with the workflow engine,
- integration with external systems and (optionally) Shuffle Cloud.

This design allows the frontend to be fully replaceable or automatically controlled by external systems (e.g. SIEM) without intervention in the application core.

2.4.1.2 Workflow engine, Worker and Orborus

The actual execution of playbooks is separated from the backend and carried out by worker components. The worker is responsible for running the individual steps of a workflow and can perform multiple tasks in parallel. This approach increases scalability and prevents longer-running or blocking operations from affecting the availability of the backend.

The Orborus component acts as the scheduler and launcher of workflows. It waits for tasks from the backend and ensures that they are launched in the appropriate worker environment. Thanks to this division, the execution layer can be scaled horizontally without changing the application architecture.

2.4.1.3 Application connectors (Apps)

Integrations with external tools are implemented by means of so-called Apps, which are implemented predominantly in Python and use standardized APIs (e.g. OpenAPI). Apps

represent specific actions that a playbook can perform, for example calling a REST API, processing data or response operations.

Separating the Apps from the system core enables:

- easy extension of functionality,
- isolation of the integration logic,
- safer execution of actions without intervention in the core components.

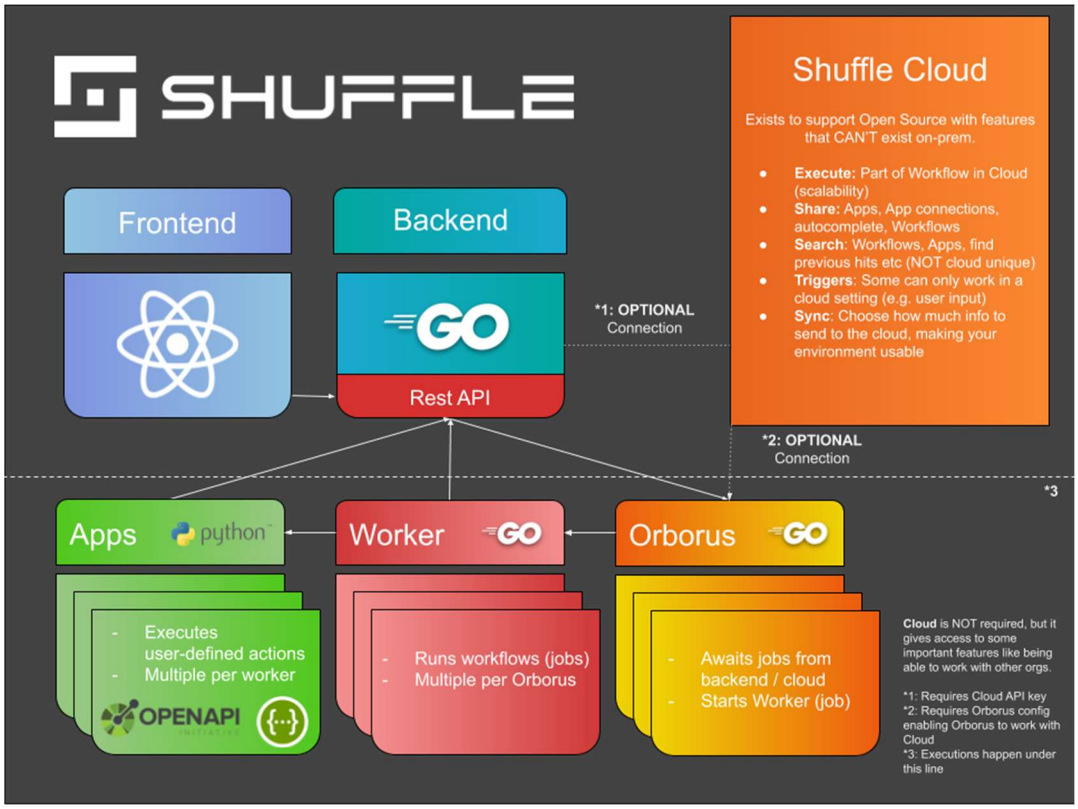


Figure 1 Diagram of the Shuffle tool from the official website

2.5 Installation – Docker variant

```
sudo apt update
sudo apt install -y ca-certificates curl gnupg lsb-release

sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg

echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" \
| sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Funded by the European Union NextGenerationEU
RECOVERY AND RESILIENCE PLAN
GOVERNMENT OFFICE OF THE SLOVAK REPUBLIC

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

docker --version
docker compose version

mkdir /opt/shuffle
cd /opt/shuffle

YAML file

version: "3.8"
services:
  shuffle:
    image: shuffle/shuffle:latest
    container_name: shuffle
    restart: unless-stopped
    ports:
      - "3001:3001"
    environment:
      - SHUFFLE_HOST=0.0.0.0
      - SHUFFLE_PORT=3001
    volumes:
      - shuffle_data:/shuffle
    depends_on:
      - shuffle-db

  shuffle-db:
    image: postgres:14
    container_name: shuffle-db
    restart: unless-stopped
    environment:
      POSTGRES_USER=shuffle
      POSTGRES_PASSWORD=shufflepassword
      POSTGRES_DB=shuffle
    volumes:
      - shuffle_db:/var/lib/postgresql/data

volumes:
  shuffle_data:
  shuffle_db:

docker compose up -d
```

The Shuffle web interface is available at: http://<IP_adresa_servera>:3001

At first login, it is necessary to:

- create an administrator account
- set up the organization
- configure basic API access

System restart – warning:

The command *docker compose down -v* **also removes persistent storage**, which leads to the loss of configuration and workflows. For a normal restart, it is recommended to use only:

```
docker compose down
docker compose up -d
```

2.6 Installation – OpenStack variant

In the OpenStack variant, without the use of containerization, the Shuffle tool will be installed on a virtual machine (VM) running Ubuntu 22.04.

1. System update

```
sudo apt update && sudo apt upgrade -y
```

2. Installing dependencies

```
sudo apt install -y \
git \
python3 \
python3-pip \
python3-venv \
nodejs \
npm \
postgresql \
postgresql-contrib
```

3. Setting up the PosgreSQL database

```
sudo -u postgres psql

CREATE DATABASE shuffle;
CREATE USER shuffle WITH PASSWORD 'shufflepassword';
GRANT ALL PRIVILEGES ON DATABASE shuffle TO shuffle;
\q
```

4. Downloading Shuffle from source code

```
cd /opt
sudo git clone https://github.com/shuffle/shuffle.git
sudo chown -R $USER:$USER shuffle
cd shuffle
```

5. Backend – Python part

```
python3 -m venv venv
source venv/bin/activate
pip install --upgrade pip
```

```
pip install -r requirements.txt
```

Setting up the database connection in the configuration file (according to the Shuffle documentation):

```
export DATABASE_HOST=localhost
export DATABASE_NAME=shuffle
export DATABASE_USER=shuffle
export DATABASE_PASSWORD=shufflepassword
```

Starting the backend:

```
python3 app.py
```

6. Frontend – Web interface

```
cd frontend
npm install
npm run build
npm run start
```

7. Accessing the application

After the backend and frontend have been started successfully, Shuffle is accessible via a web browser on the defined port (3001 by default).

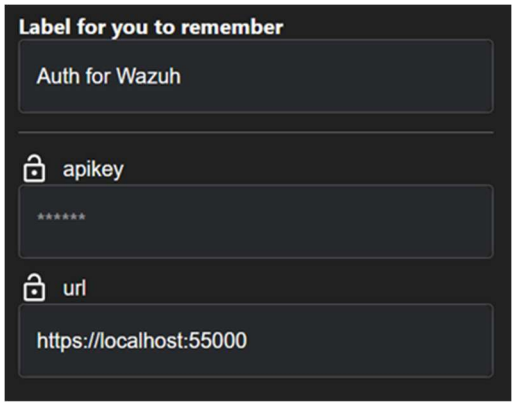
2.7 Basic configuration

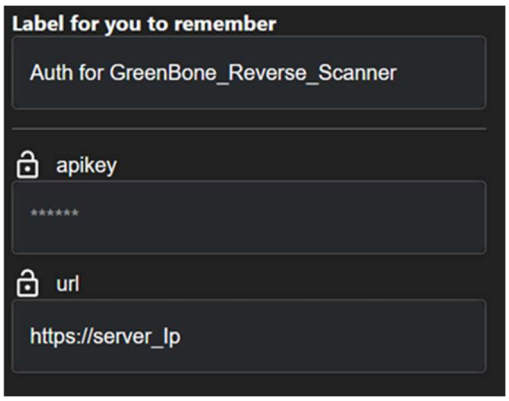
Shuffle is designed as a flexible orchestrator whose configuration is carried out predominantly through the web interface. The basic configuration is performed immediately after installation and includes the settings of users, security mechanisms, integrations and the automation logic itself.

After the first login, it is necessary to create an administrator account and define the organizational structure. User roles and permissions are then set up, which determine who can create and edit workflows, manage integrations or run playbooks. Shuffle provides a basic access control model that is sufficient for small and medium-sized teams.

An important part of the configuration is the management of applications and connectors. Each integrated service is represented in Shuffle as an application for which authentication data are configured, most often in the form of API keys, tokens or OAuth mechanisms. These data are stored centrally and subsequently used in individual workflows, which reduces the need to enter sensitive information repeatedly.

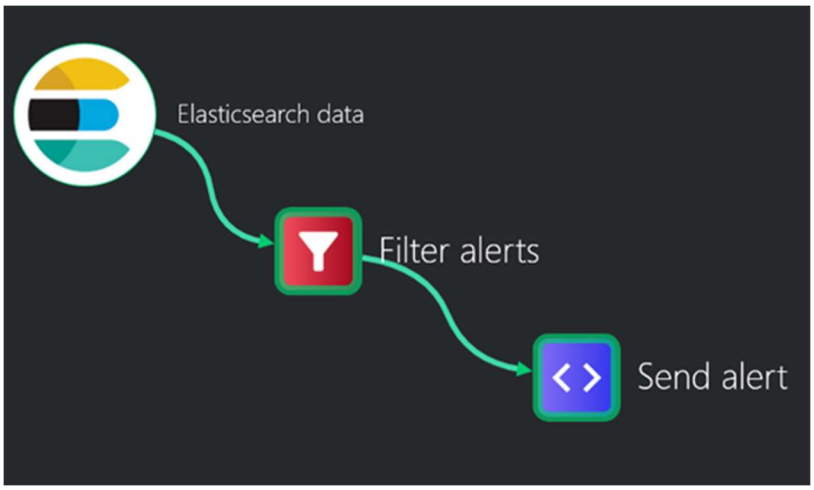
2.7.1 Connections to the external applications Wazuh and Greenbone





Workflows are configured in a visual editor, where the individual automation steps are defined. Each step represents a specific action, for example calling an external API, processing a response, conditional decision-making or sending a notification. Shuffle makes it possible to work with variables that are passed between workflow steps, which enables dynamic data processing and the creation of complex response scenarios.

Workflow for sending notifications to the Discord communication platform.



2.7.2 Inputs and outputs

2.7.2.1 Inputs

The inputs to Shuffle are primarily events and data coming from external systems. The most common input method is the use of webhooks or the REST API, through which security tools can send alerts, logs or incident information. Typical sources of input data are SIEM systems, log management tools, EDR solutions or other security platforms.

In addition to automatic inputs, a workflow can also be triggered manually, for example by an analyst investigating an incident. The input data may contain various types of

information, such as IP addresses, domains, URLs, file hash values, incident identifiers or entire JSON structures with a detailed description of the event.

2.7.2.2 Outputs

The outputs of Shuffle are executed automated actions or generated results that are sent back to external systems. Typical outputs include creating tickets in incident management tools, sending notifications via communication platforms, updating records in security systems or triggering response measures, such as blocking IP addresses or deactivating user accounts.

Outputs may take the form of API calls, messages, logs or structured data, which are subsequently used by other tools in the security ecosystem. Shuffle thus plays the role of a central orchestration point that links input events with automated responses.

2.8 Integrations

Shuffle supports integration with a wide range of security and IT tools through application connectors (Apps) and an open REST API. The integrations are designed so that Shuffle functions as a central orchestration and automation layer that links existing SOC components without the need to modify them directly.

Typical integration scenarios include:

- SIEM tools (e.g. Elastic Stack, Graylog) as a source of events and triggers,
- context systems (e.g. NetBox, OpenCTI) for incident enrichment,
- external services (VirusTotal, WHOIS, DNS services) for enrichment,
- communication and ticketing tools (Slack, Jira, e-mail),
- repositories and storage (GitHub, Git) for retaining outputs and audit artefacts.

Thanks to the open API and the modularity of the connectors, it is also possible to create custom integrations, which makes it possible to adapt Shuffle to the specific needs of an organization or of an experimental PoC environment.

2.8.1 Position of Shuffle in the SOC

From the perspective of the SOC architecture, this internal structure allows Shuffle to function as a separate response and automation layer that:

- receives events from detection systems (e.g. SIEM),
- performs enrichment and decision logic,
- triggers response steps through external tools.

Separating control (backend) from execution (workers) reduces the risk that automation will negatively affect the stability of the SOC infrastructure, and supports a secure and controlled deployment in which the ELK Stack detects, NetBox provides context and Shuffle provides the automated response.

```
[ Event sources ]
   | (syslog / beats / IDS)
   v
[ ELK Stack ]
  - Elasticsearch (indices)
  - Kibana (analysis/alert)
        |
        | (webhook / alert action / manual trigger)
        v
[ Shuffle (SOAR) ]
  - Web UI (playbooks)
  - Workflow engine (step execution)
  - Internal DB (state, results)
        |
        |----> [ Context / Enrichment ]
        |        - NetBox (asset/IP/topology) via API
        |        - OSINT/WHOIS/DNS (optional)
        |
        |----> [ Response / Actions ]
                 - Notification (email/chat)
                 - Ticketing (Jira/GLPI/…)
                 - Network action (FW/ACL) (optional)
                 - Git/artefacts (output log) (optional)
```

Where the intended event processing flow is:

1. Detection (ELK): Events (e.g. syslog, IDS alert, Windows event) are collected into ELK. An analyst or a rule identifies an event of interest.
2. Trigger (Shuffle): Shuffle is triggered either:
   - manually (the analyst selects an incident and runs a playbook),
   - or automatically by a webhook/alert action (PoC mode).
3. Enrichment (context): The playbook supplements the incident with context:
   - NetBox: "who the IP belongs to", "what device", "location", "segment/VLAN",
   - optionally DNS/WHOIS/GeoIP, reputation.
4. Response (action): Based on the results (conditions in the playbook), it:
   - sends a notification,
   - opens a ticket,
   - or performs a simple response (e.g. blocking an IP on the FW).
5. Audit/Output: Shuffle stores the workflow state and the results of the steps (the records are traceable for audit and retrospective analysis).

3 WAZUH

Tool type: Endpoint security / SIEM / EDR / FIM

Tool name: Wazuh

Version: Wazuh 4.14 (open-source, containerized deployment)

3.1 System description

Wazuh is an open-source security platform focused on endpoint monitoring, security event detection and basic response, and in a SOC environment it plays the role of a combined EDR, SIEM and FIM solution. Its main purpose is the collection, correlation and evaluation of security-relevant data from operating systems, applications and infrastructure components, and their subsequent provision to the central SIEM environment.

Data collection is carried out primarily through Wazuh agents, which are installed on endpoint systems (Linux, Windows, macOS). These agents enable:

- collection of system and application logs,
- file integrity monitoring (File Integrity Monitoring – FIM),
- auditing of system events,
- basic detection of anomalies and threats at endpoint level.

In addition to agent-based collection, Wazuh can also receive data via Syslog, which enables centralized log collection from network devices, security appliances or other systems where it is not possible or appropriate to deploy an agent. This approach extends the applicability of Wazuh to passive or embedded devices as well.

Wazuh contains its own correlation and detection engine, which uses an extensive set of rules based on indicators of compromise, behavioural patterns and mapping to MITRE ATT&CK techniques. The rules are regularly updated by the community and provide immediate usability without the need to create detection rules manually.

The Wazuh web user interface supports role-based access control (RBAC), agent management, an overview of events and integration with the visualization layer (Kibana). In the proposed SOC architecture, Wazuh acts as a source of endpoint-level detection events, which are subsequently sent to the central SIEM (ELK Stack) for further correlation and analysis.

Its main strength is the combination of EDR capabilities with a clear SIEM approach, which makes it a suitable solution especially for small and medium-sized SOC teams that need to cover both endpoint security and basic correlation without deploying several separate tools.

3.2 System documentation

The official documentation of the Wazuh platform is publicly available and provides a comprehensive overview of the system architecture, data collection options, detection mechanisms and integration scenarios. The documentation includes guides for installing

agents, configuring server components, managing rules and integrating with external SIEM tools.

The documentation is practically oriented and also covers areas such as:

- configuration of FIM and audit modules,
- management of agents and their life cycle,
- working with MITRE ATT&CK mapping,
- integration with Elasticsearch and Kibana.

Links to official sources:

- Official Wazuh documentation: https://documentation.wazuh.com/
- Official project website: https://wazuh.com/
- GitHub repository: https://github.com/wazuh/wazuh
- Official Docker images: https://hub.docker.com/u/wazuh

3.3 Resource requirements

The hardware requirements of Wazuh depend on the number of monitored endpoints, the volume of generated events and the extent to which correlation rules are used. Since Wazuh combines several security functions, its requirements are higher than those of simple log collectors.

Minimum requirements (PoC / smaller environment):

- CPU: 2–4 cores
- RAM: 4–8 GB
- Disk: 50 GB (logs and indices)
- Number of agents: tens

Recommended requirements (production environment):

- CPU: 8+ cores
- RAM: 16–32 GB
- Disk: SSD (depending on data retention)
- Number of agents: hundreds or more

Increased requirements arise mainly in the case of:

- intensive FIM monitoring,
- a high volume of audit events,
- long-term data retention in the SIEM.

3.4 Tool architecture

The Wazuh architecture is designed as a multi-layer distributed solution that separates data collection, detection and the presentation of results. The basis of the architecture consists of three main components: agents, the server part and the visualization layer.

Agents are deployed on endpoint systems and are responsible for collecting security data. The Wazuh server part processes the received events, applies correlation rules and generates security alerts. The resulting events are then sent to Elasticsearch, where they are indexed and made available for further analysis.

The visualization layer is implemented by means of a web interface and integration with Kibana dashboards, which allow analysts to monitor the state of endpoints, detections and trends over time.

In the SOC architecture, Wazuh is positioned at the interface between the endpoints and the SIEM layer:

- on the one hand, it provides detailed detection at endpoint level,
- on the other hand, it provides normalized and correlated events to the central ELK Stack.

In this way, Wazuh significantly extends the detection capabilities of the SOC without the need to deploy separate EDR and SIEM solutions for basic use.

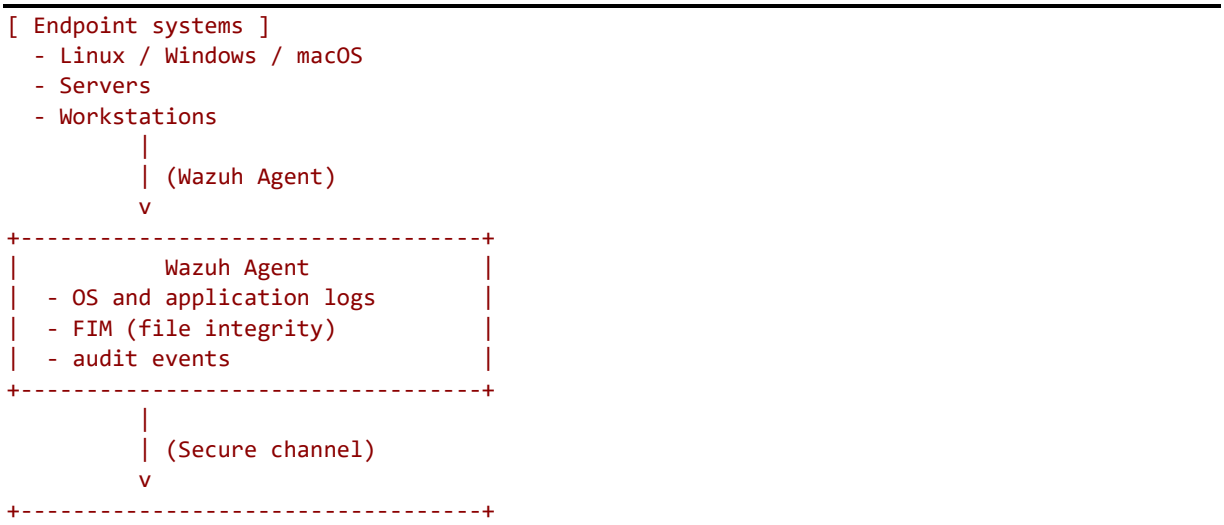
3.4.1 Diagram

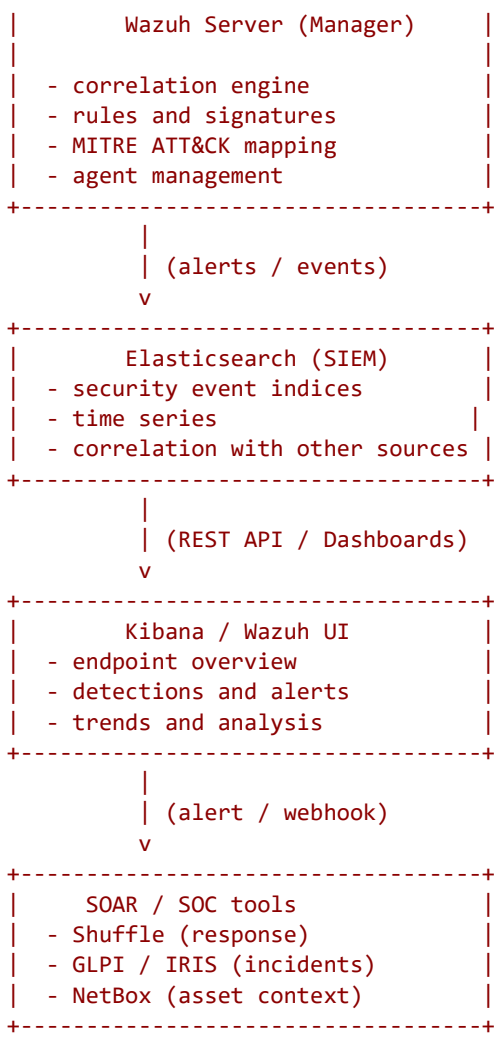
The Wazuh architecture is based on a distributed model in which the data collection, detection, correlation and visualization components are clearly separated. This approach makes it possible to scale the solution according to the number of endpoints and the volume of security events, while maintaining a clear integration link to the SIEM layer.

The basis of the architecture consists of:

- Wazuh agents deployed on endpoint systems,
- the Wazuh server (manager), which performs correlation and detection,
- the indexing and analytics layer (Elasticsearch),
- the visualization layer (Kibana / Wazuh UI).

Agents collect security-relevant data (logs, audit, FIM events) and send them to the Wazuh server. The server applies correlation rules and mapping to MITRE ATT&CK and sends the resulting alerts to the SIEM layer. The visualization part allows analysts to monitor the state of endpoints and detections in real time.





The diagram illustrates that Wazuh is the **primary source of endpoint detections**, which are:

- processed and correlated at the level of the Wazuh server,
- sent to the central SIEM (ELK Stack),
- visualized in Kibana,
- used for response (SOAR) and incident management.

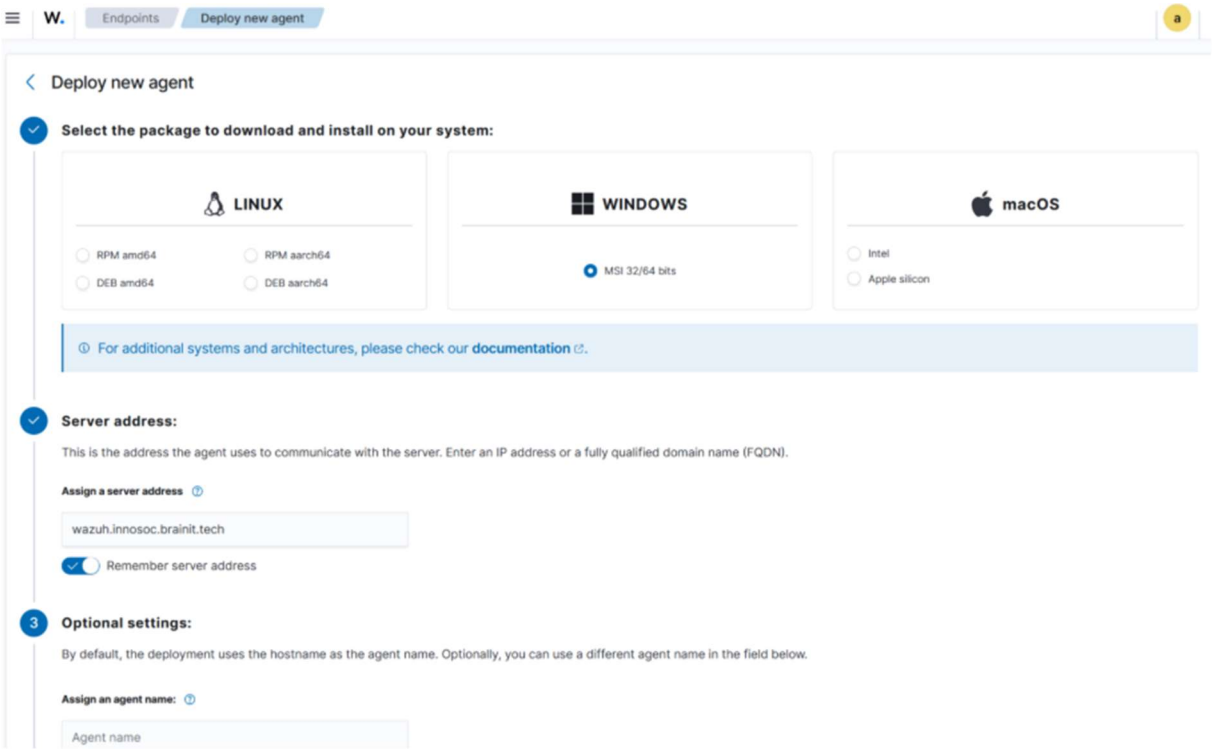
Wazuh thus fills the **gap between raw log collection and higher-level correlation** in the SOC architecture, while providing a detailed view of the state and behaviour of endpoints.

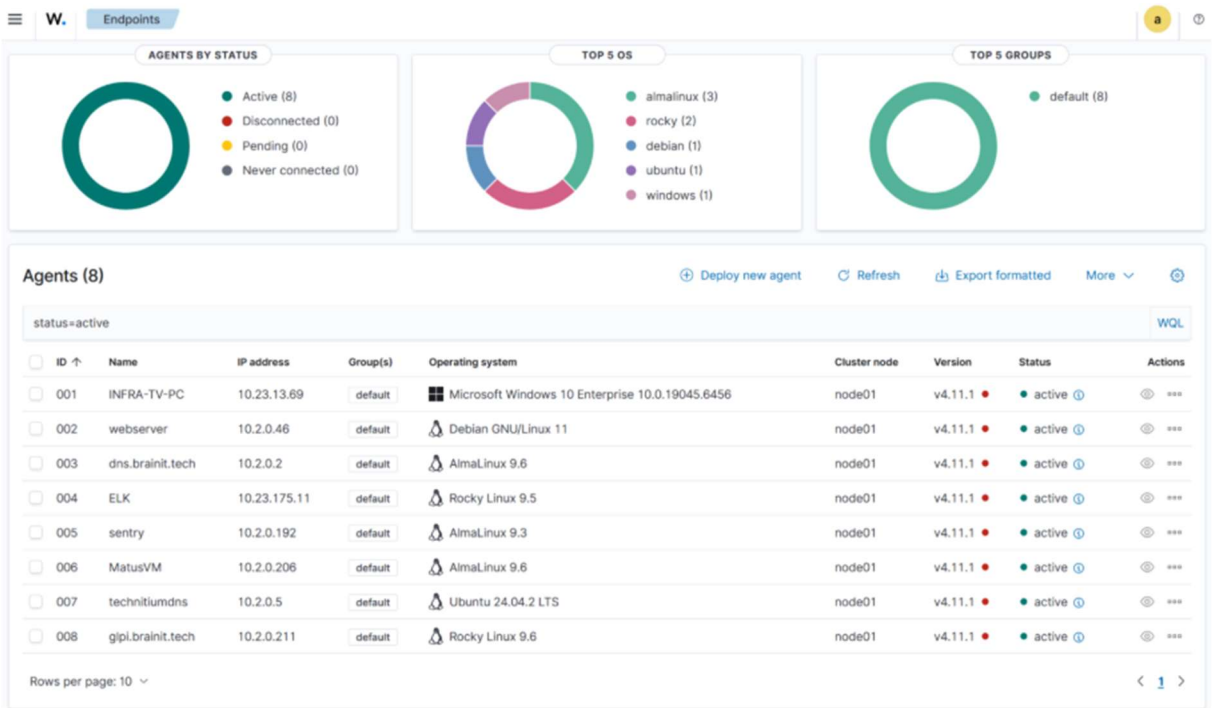
3.5 Installation – Quickstart variant

Installed on a Rocky Linux 9 VM according to the official quickstart documentation https://documentation.wazuh.com/current/quickstart.html

3.6 Basic configuration

Wazuh has a built-in system for providing the Wazuh agent installer and onboarding client workstations:





3.7 Integrations

Wazuh can be integrated with various SIEM systems, including Elasticsearch; it was therefore integrated into the ELK Stack:

- https://documentation.wazuh.com/current/integrations-guide/index.html
- https://documentation.wazuh.com/current/integrations-guide/elastic-stack/index.html

4 GREENBONE COMMUNITY EDITION

Tool type: Vulnerability Management / Vulnerability Assessment

Tool name: Greenbone Community Edition (GCE)

Version: Version 24.10 Open-source edition (OpenVAS / Greenbone Vulnerability Management)

4.1 System description

Greenbone Community Edition (GCE) is an open-source tool intended for regular scanning of infrastructure and services and for vulnerability reporting, which in a SOC environment plays the role of a basic component for vulnerability management. It serves to identify technical weaknesses of systems before they can potentially be exploited and provides a basis for preventive security measures.

GCE is the free open-source version of the solution from Greenbone Networks, based on the OpenVAS (Open Vulnerability Assessment System) tool. It provides high-capacity scanning using an extensive vulnerability database available through the community GVMD feed, which contains regularly updated vulnerability tests (NVTs). These tests cover known vulnerabilities, misconfigurations, weak cryptographic settings and outdated services.

The system supports scheduled and recurring scans, thereby enabling long-term monitoring of the security state of the infrastructure and the identification of vulnerability trends. The scan results are processed in the modern Greenbone Security Assistant (GSA) web application, which enables findings to be filtered, sorted and evaluated by severity (CVSS), asset or vulnerability type. Outputs can be generated in several formats, in particular CSV, XML and PDF, which enables their further processing, archiving or presentation to management.

Greenbone Community Edition supports standards such as CVE and CPE, thereby ensuring compatibility with other security tools and risk assessment methodologies. In terms of integration, it offers several mechanisms, including GMP (Greenbone Management Protocol), as well as support for SSH, LDAP, RADIUS and other authentication and integration interfaces. These options enable linking with existing identity management systems, automation tools or ticketing systems.

For large or distributed environments, GCE also supports a master–sensor architecture, in which the central control instance is supplemented by several scanning sensors. This approach enables scanning to be scaled, network load to be reduced and separate infrastructure segments to be covered without the need for central scanning across security boundaries.

From the SOC perspective, Greenbone Community Edition does not perform real-time detection or incident response. Its role is to provide a systematic and repeatable view of the vulnerability level of assets, which supplements the detection tools (SIEM/EDR) with preventive security context. The scan results serve as input for risk analysis, the planning of corrective measures and decision support in the area of the organization's security policy.

4.2 System documentation

Greenbone Community Edition has official documentation available that describes the solution architecture, the deployment method, scan configuration and results management. The documentation focuses primarily on the technical side of the tool and its components.

Official sources:

- Greenbone Community Edition documentation: https://greenbone.github.io/docs/
- Project GitHub repository: https://github.com/greenbone
- OpenVAS / GVM documentation: https://www.openvas.org/
- DockerHub repositories: https://hub.docker.com/u/greenbone

4.3 Resource requirements

The hardware requirements of Greenbone depend on:

- the number of scanned assets,
- the scope of the scans,
- the frequency with which they are run.

Minimum requirements (PoC / smaller environment):

- CPU: 2–4 cores
- RAM: 4–8 GB
- Disk: 40–80 GB (databases, NVT feeds, reports)

Recommended for a larger environment:

- CPU: 4–8 cores
- RAM: 8–16 GB
- Disk: SSD storage (100 GB or more)

Higher requirements apply mainly when scanning several ranges in parallel and when retaining historical results.

4.4 Tool architecture

The architecture of Greenbone Community Edition is based on a modular model in which the components for scanning, results management and the user interface are separated. This model enables scaling and integration into the existing SOC infrastructure.

Basic architecture components:

- Scanner (OpenVAS Scanner) – performs the vulnerability tests themselves,
- Manager (gvmd) – controls scans, processes the results and stores them in the database,
- Database (PostgreSQL) – stores scan results and metadata,
- Web UI (Greenbone Security Assistant – GSA) – interface for management and analysis.

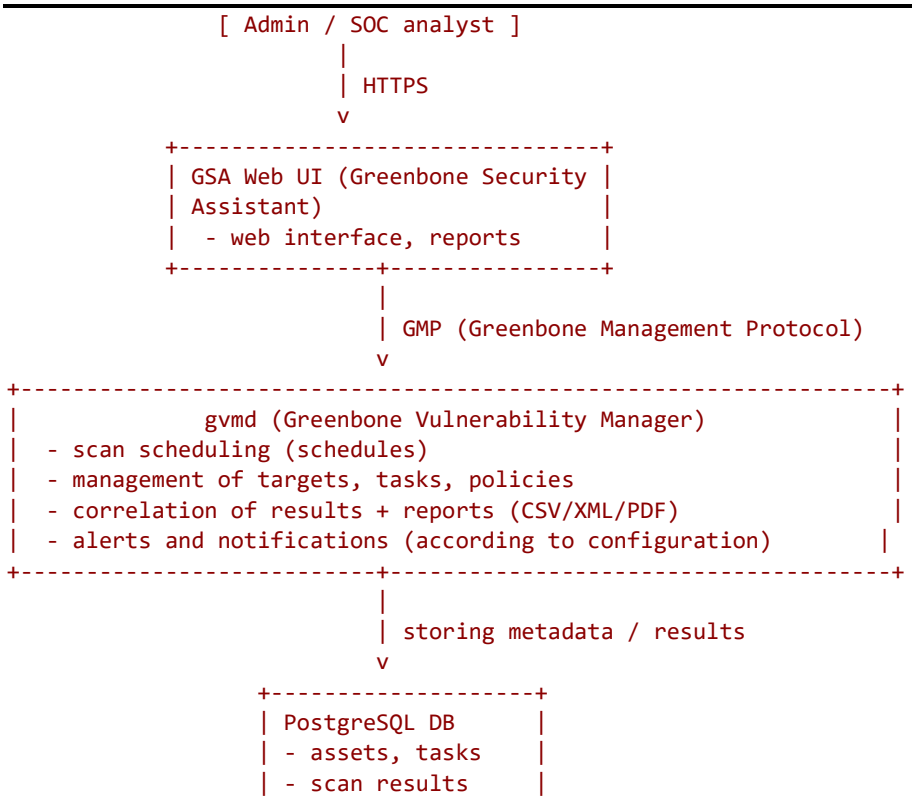
Greenbone communicates actively with target systems over the network (TCP/UDP), and scans may be:

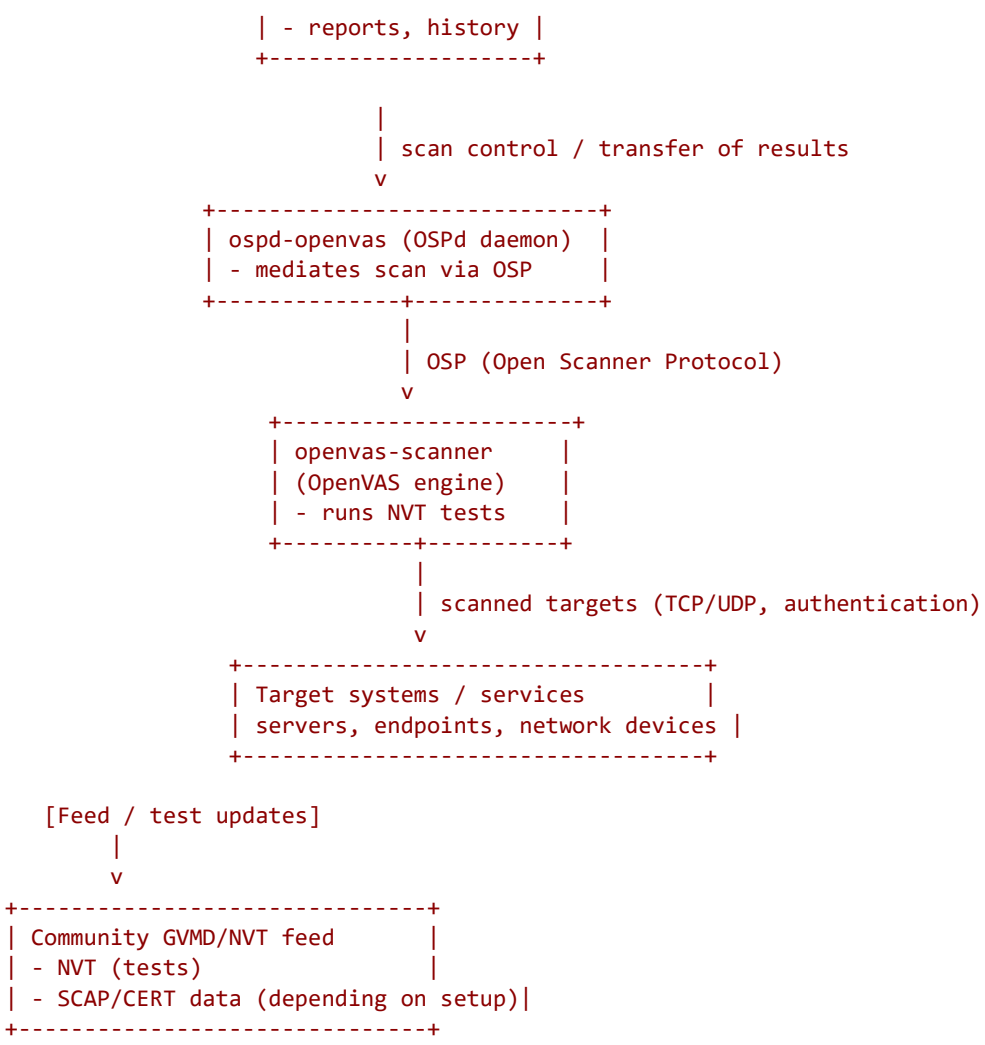
- unauthenticated (black-box),
- authenticated (white-box, higher accuracy).

In the SOC architecture, Greenbone is positioned outside the production data flow so as not to affect service availability, and its outputs are used as supplementary security context.

4.4.1 Diagram

The diagram shows the reference architecture of Greenbone Community Edition (GVM/OpenVAS), which is based on the separation of the control, scanning and presentation layers. The GSA web interface serves as the entry point for SOC analysts and communicates with the gvmd component, which handles scan scheduling, target management and the evaluation of results. The scanning itself is performed by the openvas-scanner component, which is controlled via ospd-openvas and runs vulnerability tests against the target systems. The scan results are stored in a PostgreSQL database and presented in the form of reports and alerts. The architecture supports regular and repeatable scanning of the infrastructure, thereby providing the SOC with a preventive view of the security risks of assets.





4.5 Installation – Docker variant

Docker Compose file used:

```
services:
  vulnerability-tests:
    image: registry.community.greenbone.net/community/vulnerability-tests
    environment:
      FEED_RELEASE: "24.10"
      KEEP_ALIVE: 1
    volumes:
      - vt_data_vol:/mnt

  notus-data:
    image: registry.community.greenbone.net/community/notus-data
    environment:
      KEEP_ALIVE: 1
    volumes:
      - notus_data_vol:/mnt
```

```
scap-data:
  image: registry.community.greenbone.net/community/scap-data
  environment:
    KEEP_ALIVE: 1
  volumes:
    - scap_data_vol:/mnt

cert-bund-data:
  image: registry.community.greenbone.net/community/cert-bund-data
  environment:
    KEEP_ALIVE: 1
  volumes:
    - cert_data_vol:/mnt

dfn-cert-data:
  image: registry.community.greenbone.net/community/dfn-cert-data
  environment:
    KEEP_ALIVE: 1
  volumes:
    - cert_data_vol:/mnt
  depends_on:
    cert-bund-data:
      condition: service_healthy

data-objects:
  image: registry.community.greenbone.net/community/data-objects
  environment:
    FEED_RELEASE: "24.10"
    KEEP_ALIVE: 1
  volumes:
    - data_objects_vol:/mnt

report-formats:
  image: registry.community.greenbone.net/community/report-formats
  environment:
    FEED_RELEASE: "24.10"
    KEEP_ALIVE: 1
  volumes:
    - data_objects_vol:/mnt
  depends_on:
    data-objects:
      condition: service_healthy

gpg-data:
  image: registry.community.greenbone.net/community/gpg-data
  volumes:
    - gpg_data_vol:/mnt

redis-server:
  image: registry.community.greenbone.net/community/redis-server
  restart: on-failure
  volumes:
    - redis_socket_vol:/run/redis/

pg-gvm:
  image: registry.community.greenbone.net/community/pg-gvm:stable
```

```
    restart: on-failure
    volumes:
      - psql_data_vol:/var/lib/postgresql
      - psql_socket_vol:/var/run/postgresql

  gvmd:
    image: registry.community.greenbone.net/community/gvmd:stable
    restart: on-failure
    volumes:
      - gvmd_data_vol:/var/lib/gvm
      - scap_data_vol:/var/lib/gvm/scap-data/
      - cert_data_vol:/var/lib/gvm/cert-data
      - data_objects_vol:/var/lib/gvm/data-objects/gvmd
      - vt_data_vol:/var/lib/openvas/plugins
      - psql_data_vol:/var/lib/postgresql
      - gvmd_socket_vol:/run/gvmd
      - ospd_openvas_socket_vol:/run/ospd
      - psql_socket_vol:/var/run/postgresql
    depends_on:
      pg-gvm:
        condition: service_started
      scap-data:
        condition: service_healthy
      cert-bund-data:
        condition: service_healthy
      dfn-cert-data:
        condition: service_healthy
      data-objects:
        condition: service_healthy
      report-formats:
        condition: service_healthy

  gsa:
    image: registry.community.greenbone.net/community/gsa:stable
    restart: on-failure
    ports:
      - 127.0.0.1:9392:80
    volumes:
      - gvmd_socket_vol:/run/gvmd
    depends_on:
      gvmd:
        condition: service_started
  # Sets log level of openvas to the set LOG_LEVEL within the env
  # and changes log output to /var/log/openvas instead /var/log/gvm
  # to reduce likelyhood of unwanted log interferences
  configure-openvas:
    image: registry.community.greenbone.net/community/openvas-scanner:stable
    volumes:
      - openvas_data_vol:/mnt
      - openvas_log_data_vol:/var/log/openvas
    command:
      - /bin/sh
      - -c
      - |
        printf "table_driven_lsc = yes\nopenvasd_server = http://openvasd:80\n" >
/mnt/openvas.conf
```

```
        sed "s/127/128/" /etc/openvas/openvas_log.conf | sed 's/gvm/openvas/' >
/mnt/openvas_log.conf
        chmod 644 /mnt/openvas.conf
        chmod 644 /mnt/openvas_log.conf
        touch /var/log/openvas/openvas.log
        chmod 666 /var/log/openvas/openvas.log

  # shows logs of openvas
  openvas:
    image: registry.community.greenbone.net/community/openvas-scanner:stable
    restart: on-failure
    volumes:
      - openvas_data_vol:/etc/openvas
      - openvas_log_data_vol:/var/log/openvas
    command:
      - /bin/sh
      - -c
      - |
        cat /etc/openvas/openvas.conf
        tail -f /var/log/openvas/openvas.log
    depends_on:
      configure-openvas:
        condition: service_completed_successfully

  openvasd:
    image: registry.community.greenbone.net/community/openvas-scanner:stable
    restart: on-failure
    environment:
      # `service_notus` is set to disable everything but notus,
      # if you want to utilize openvasd directly, remove `OPENVASD_MODE`
      OPENVASD_MODE: service_notus
      GNUPGHOME: /etc/openvas/gnupg
      LISTENING: 0.0.0.0:80
    volumes:
      - openvas_data_vol:/etc/openvas
      - openvas_log_data_vol:/var/log/openvas
      - gpg_data_vol:/etc/openvas/gnupg
      - notus_data_vol:/var/lib/notus
    # enable port forwarding when you want to use the http api from your host machine
    # ports:
    #   - 127.0.0.1:3000:80
    depends_on:
      vulnerability-tests:
        condition: service_healthy
      notus-data:
        condition: service_healthy
      configure-openvas:
        condition: service_completed_successfully
      gpg-data:
        condition: service_completed_successfully
    networks:
      default:
        aliases:
          - openvasd

  ospd-openvas:
```

```
    image: registry.community.greenbone.net/community/ospd-openvas:stable
    restart: on-failure
    hostname: ospd-openvas.local
    cap_add:
      - NET_ADMIN # for capturing packages in promiscuous mode
      - NET_RAW # for raw sockets e.g. used for the boreas alive detection
    security_opt:
      - seccomp=unconfined
      - apparmor=unconfined
    command:
      [
        "ospd-openvas",
        "-f",
        "--config",
        "/etc/gvm/ospd-openvas.conf",
        "--notus-feed-dir",
        "/var/lib/notus/advisories",
        "-m",
        "666",
      ]
    volumes:
      - gpg_data_vol:/etc/openvas/gnupg
      - vt_data_vol:/var/lib/openvas/plugins
      - notus_data_vol:/var/lib/notus
      - ospd_openvas_socket_vol:/run/ospd
      - redis_socket_vol:/run/redis/
      - openvas_data_vol:/etc/openvas/
      - openvas_log_data_vol:/var/log/openvas
    depends_on:
      redis-server:
        condition: service_started
      gpg-data:
        condition: service_completed_successfully
      configure-openvas:
        condition: service_completed_successfully
      vulnerability-tests:
        condition: service_healthy
      notus-data:
        condition: service_healthy

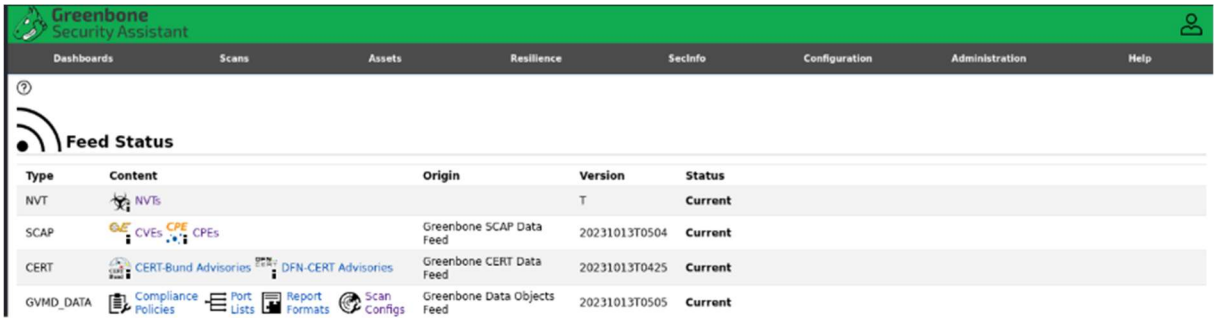
  gvm-tools:
    image: registry.community.greenbone.net/community/gvm-tools
    volumes:
      - gvmd_socket_vol:/run/gvmd
      - ospd_openvas_socket_vol:/run/ospd
    depends_on:
      - gvmd
      - ospd-openvas

volumes:
  gpg_data_vol:
  scap_data_vol:
  cert_data_vol:
  data_objects_vol:
  gvmd_data_vol:
  psql_data_vol:
```

```
vt_data_vol:
notus_data_vol:
psql_socket_vol:
gvmd_socket_vol:
ospd_openvas_socket_vol:
redis_socket_vol:
openvas_data_vol:
openvas_log_data_vol:
```

4.6 Basic configuration

Updating the CVE databases, adding hosts.



5 OPENCTI

Tool type: Threat Intelligence Platform (TIP)

Tool name: OpenCTI

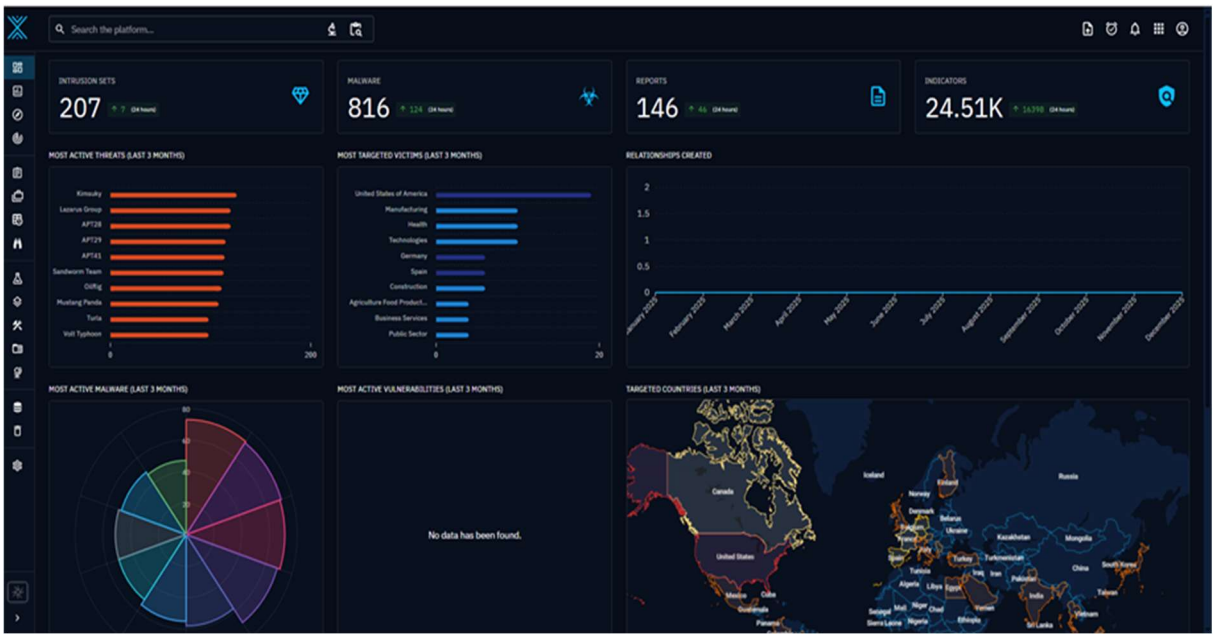
Version: Version 6.9.0 - Open-source version (containerized deployment)

5.1 System description

OpenCTI is an open-source platform designed for the centralized management of Cyber Threat Intelligence. It serves as a central place where organizations gather information about cyberattacks, attackers, malware, vulnerabilities and other technical indicators that may be related to security incidents. The main idea of OpenCTI is not only to collect data but, above all, to link them into meaningful relationships so that security analysts can see the broader context of threats and are better able to understand who is attacking, what tools they are using and what objectives they are pursuing.

The platform supports the STIX/TAXII standards and integration with numerous external databases and security tools. This means that threat information is structured in a uniform and shareable way. OpenCTI displays these data in the form of a graph in which the relationships between attacker groups, malware, IP addresses, domains and CVE-type vulnerabilities are clearly visible.

From a technical point of view, OpenCTI is built mainly on Python and uses ElasticSearch for data storage and search. As it is an open-source solution, it is available free of charge and organizations can adapt it to their own needs, whether in terms of extensions, connectors or the deployment method.



5.2 System documentation

The official documentation of the OpenCTI platform is publicly available and continuously updated by the development team and the project community. The documentation provides a detailed overview of the system architecture, the object and data model based on the STIX standard, the integration options via connectors, as well as recommended practices for deployment and operation in a SOC environment.

The documentation includes detailed guides for containerized deployment using Docker, configuration of the individual components, management of users and permissions, as well as a description of the internal mechanisms for processing threat intelligence data. The documentation also includes reference examples of integrations with external threat information sources and security tools.

Links to official sources:

- Official OpenCTI documentation: https://docs.opencti.io/latest/
- Project GitHub repository: https://github.com/OpenCTI-Platform/opencti
- Official Docker images (OpenCTI and connectors): https://hub.docker.com/u/opencti

5.3 Resource requirements

The hardware requirements of the OpenCTI platform depend on the volume of threat intelligence processed, the number of active connectors and the intensity of work with the graph data model. Since OpenCTI works with structured data and relationships between objects, its computing resource requirements are higher than those of simple database applications, but still reasonable for deployment in a SOC environment.

For a test or PoC deployment, the following is sufficient:

- CPU: 2–4 cores
- RAM: 8 GB
- Disk space: 50–100 GB (SSD recommended)

For a production or larger SOC environment, the following is recommended:

- CPU: 4–8 cores
- RAM: 16–32 GB
- Disk: SSD storage (especially for the database and indexing layer)

Higher requirements are driven mainly by the number of ingested TI feeds, the size of the data model (STIX objects and relationships), the update frequency and the number of connectors running in parallel. In practice, OpenCTI is often deployed in containers, which enables flexible scaling of the individual components according to the current load.

5.4 Tool architecture

The architecture of the OpenCTI platform is designed as a modular and distributed system that enables efficient processing, enrichment and correlation of threat intelligence data from various sources. The basis of the architecture consists of the web application layer, the

application API and a set of separate connectors that handle the import, export and processing of data.

The central component is the OpenCTI platform (backend + frontend), which provides the web user interface and the REST/GraphQL API. Through this interface, analysts can browse, analyse and manage threat intelligence objects such as threat actors, campaigns, malware, attack techniques and indicators of compromise. The backend also handles the application logic, access control and the management of relationships between individual objects according to the STIX data model.

The persistent data layer is implemented using database and indexing mechanisms that enable the storage of large amounts of structured threat intelligence data and efficient searching in them. The architecture is optimized for working with graph relationships between objects, which is key to understanding the context of threats.

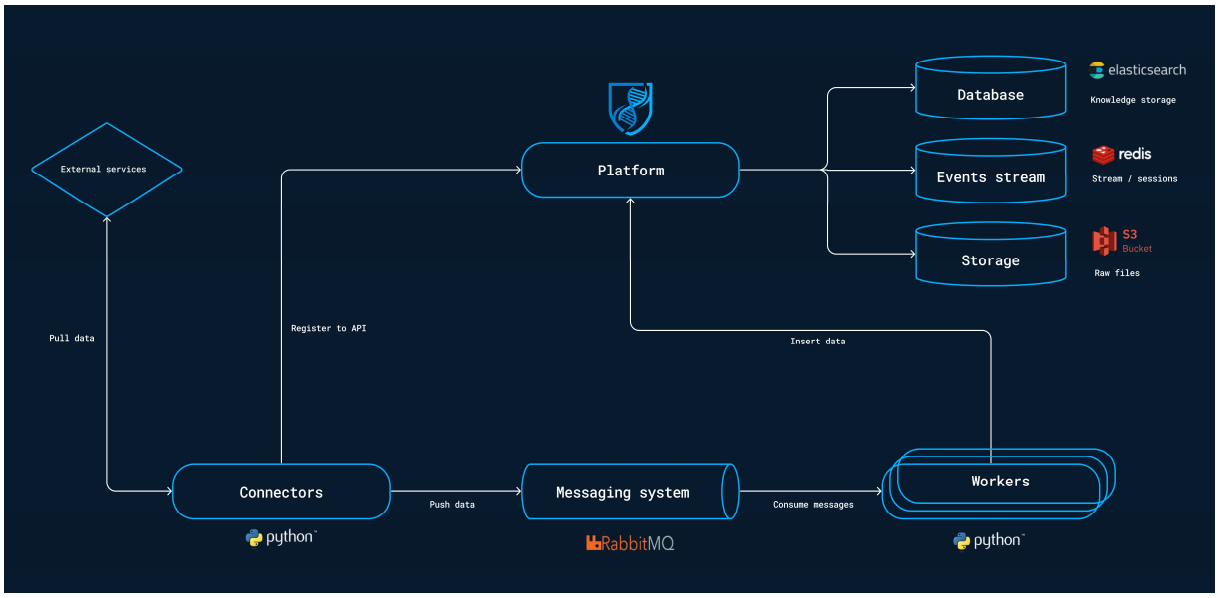
An important part of OpenCTI are the connectors, which function as separate processes or containers. The connectors handle:

- import of data from external sources (OSINT, commercial TI feeds),
- export of data to external systems (SIEM, SOAR),
- enrichment of existing objects with additional information.

This approach makes it possible to extend the functionality of OpenCTI without interfering with the application core and supports secure separation of the integration logic from the main platform.

From the perspective of the SOC architecture, OpenCTI is positioned as an intelligence layer between detection and response. It provides context for detection tools (e.g. SIEM) and inputs for automated response mechanisms (SOAR). OpenCTI itself performs neither detection nor response, but improves the quality of decision-making by linking technical indicators with a tactical and strategic view of threats.

The OpenCTI architecture diagram (used in the document) is based on the official project documentation and illustrates the flow of threat intelligence data between the platform, the connectors and external systems within the SOC infrastructure.



5.5 Installation – Docker variant

The installation was performed according to the official OpenCTI documentation using containerization, deployed on a device running Linux Ubuntu 22.04.

5.5.1 Installation steps:

1. Installing docker-conpose on the device

```
sudo apt install docker-compose
```

2. Creating a new folder for the installation package and moving into this folder

```
mkdir -p /path/to/your/app && cd /path/to/your/ap
```

3. Downloading the installation package from the GitHub platform

```
git clone https://github.com/OpenCTI-Platform/docker.git
```

4. Editing the sample .env file (.env sample), from which docker-compose takes the variables during installation, according to the user's preferences

root@opencti-s: /opt/docker

```
  GNU nano 6.2                                        .env.sample
############################
# DEPENDENCIES             #
############################

MINIO_ROOT_USER=opencti
MINIO_ROOT_PASSWORD=changeme
RABBITMQ_DEFAULT_USER=opencti
RABBITMQ_DEFAULT_PASS=changeme
SMTP_HOSTNAME=localhost
OPENSEARCH_ADMIN_PASSWORD=changeme
ELASTIC_MEMORY_SIZE=4G

############################
# COMMON                   #
############################

XTM_COMPOSER_ID=8215614c-7139-422e-b825-b20fd2a13a23
COMPOSE_PROJECT_NAME=xtm

############################
# OPENCTI                  #
############################

OPENCTI_HOST=localhost
OPENCTI_PORT=8080
OPENCTI_EXTERNAL_SCHEME=http
OPENCTI_ADMIN_EMAIL=admin@opencti.io
OPENCTI_ADMIN_PASSWORD=changeme
OPENCTI_ADMIN_TOKEN=ChangeMe_UUIDv4
OPENCTI_HEALTHCHECK_ACCESS_KEY=changeme

############################
# OPENCTI CONNECTORS       #
############################

CONNECTOR_EXPORT_FILE_STIX_ID=dd817c8b-abae-460a-9ebc-97b1551e70e6
CONNECTOR_EXPORT_FILE_CSV_ID=7ba187fb-fde8-4063-92b5-c3da34060dd7
CONNECTOR_EXPORT_FILE_TXT_ID=ca715d9c-bd64-4351-91db-33a8d728a58b
CONNECTOR_IMPORT_FILE_STIX_ID=72327164-0b35-482b-b5d6-a5a3f76b845f
CONNECTOR_IMPORT_DOCUMENT_ID=c3970f8a-ce4b-4497-a381-20b7256f56f0
CONNECTOR_IMPORT_FILE_YARA_ID=7eb45b60-069b-4f7f-83a2-df4d6891d5ec
```

5. After the .env file has been edited, the docker service is started and the individual containers with the OpenCTI software and its components are created

```
sudo systemctl start docker.service
docker-compose up -d
```

6. Opening the application and logging in to the account

```
http://ip_adresa:8080
```

5.6 Installation – OpenStack variant

The manual installation of OpenCti on the OpenStack platform likewise followed the official web documentation. A virtual machine (VM) running Linux Ubuntu 22.04 was created in OpenStack, as was the case with the installation using containerization.

5.6.1 Installation steps

Prerequisites:

1. **Installing dependencies**
   Before installing the OpenCTI software packages themselves, it is "good practice" to install the dependencies needed to run the main application and the individual related components.

```
sudo apt-get update
sudo apt-get install build-essential nodejs npm python3 python3-pip python3-dev
```

2. **Downloading the application files**
   First, it is necessary to download the latest released version of the application (or application package). For Linux systems (Ubuntu, Debian,...), it is necessary to install the current version of the application using the command:

```
opencti-release_{RELEASE_VERSION}.tar.gz
```

Installing the main platform:

3. **Configuring the application**
   The main application contains one JSON configuration file, which needs to be set according to the user's preferences (i.e. the file must be edited so that it corresponds to the ElasticSearch configuration), and several Python modules to be installed.

```
cd opencti
cp config/default.json config/production.json
```

4. **Installing Python modules**

```
cd src/python
pip3 install -r requirements.txt
cd ../..
```

5. **Starting the application**
   The application is just a NodeJS process; the database schema creation and migration are performed at start-up.

```
npm install -g corepack
```

After checking the installed NodeJS versions, the application can be started inside the **opencti** folder.

```
yarn install
yarn build
yarn serv
```

The default user name and password for the application are defined by the user in

```
config/production.json
```

Installing workers

6. Configuring the worker

```
cd worker
pip3 install -r requirements.txt
cp config.yml.sample config.yml
```

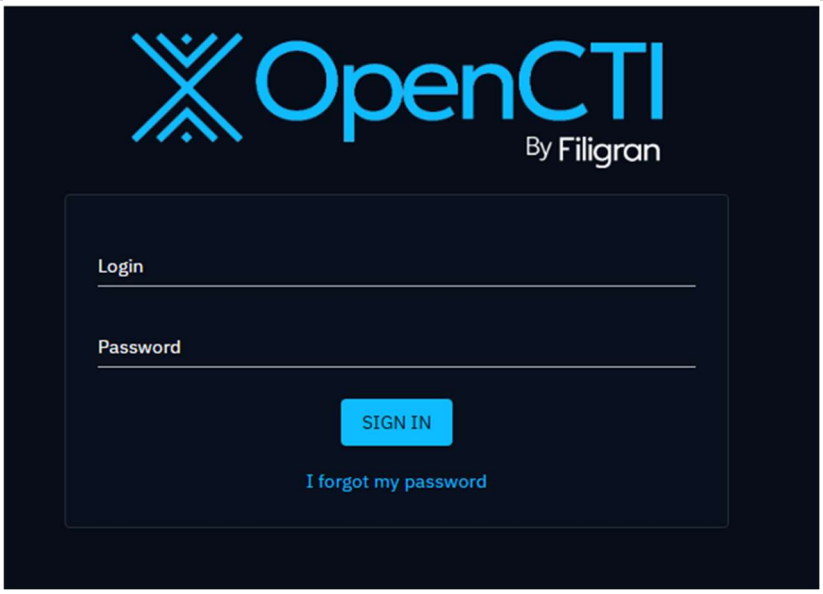
7. Starting the worker

In this step, it is possible to start as many workers as needed

```
python3 worker.py &
python3 worker.py &
```

8. **Opening** the application and logging in to the account

```
http://ip_adresa:8080
```



5.7 Basic configuration

The basic configuration of OpenCTI is most often carried out using Docker Compose and consists of several key components that together form a functional ecosystem for cyber threat management. The core is the OpenCTI application part itself, which provides the web interface and the API. This application communicates with the database layer, which consists of ElasticSearch, intended for storing and quickly searching STIX objects, and with Redis, which is used to manage task queues and for asynchronous data processing. The basic configuration also includes MinIO, an object storage used to store file attachments, reports and binary data.

After the initial installation, it is necessary to set the basic operating parameters, such as the administrator account, organization, user roles and access levels. Data retention policies, performance limits and the authentication method, which may be local or connected to an external identity system, are also configured.

An important part of the configuration is also the setting up of connectors, which ensure the supply of data into OpenCTI or, conversely, the export of information to other security tools. In the currently deployed version of the application, 3 connectors are active in OpenCTI, for the MITTRE ATT&CK, Urlhuas (abuse.ch) and Ransomwarelive databases.

We added all new connectors to docker-compose.yml, adding the following configuration for the individual connectors:

5.7.1 MITTRE ATTACK configuration

```
connector-mitre:
  image: opencti/connector-mitre:6.9.4
  environment:
    - OPENCTI_URL=http://opencti:8080
    - OPENCTI_TOKEN=${OPENCTI_ADMIN_TOKEN}
    - CONNECTOR_ID=${CONNECTOR_MITRE_ID}
    - CONNECTOR_AUTO_CREATE_SERVICE_ACCOUNT=true
    - CONNECTOR_AUTO_CREATE_SERVICE_ACCOUNT_CONFIDENCE_LEVEL=75
  restart: always
  depends_on:
    opencti:
      condition: service_healthy
```

5.7.2 Urlhaus configuration

```
connector-urlhaus:
  image: opencti/connector-urlhaus:latest
  environment:
    - OPENCTI_URL=http://opencti:8080
    - OPENCTI_TOKEN=${OPENCTI_ADMIN_TOKEN}
    - CONNECTOR_ID=df05d65f-b4d9-4438-9988-231e7c99f994
    - CONNECTOR_NAME=Abuse.ch URLhaus
    - CONNECTOR_SCOPE=urlhaus
    - CONNECTOR_LOG_LEVEL=error
    - URLHAUS_INTERVAL=3
    - URLHAUS_CSV_URL=https://urlhaus.abuse.ch/downloads/csv_recent/
    - URLHAUS_DEFAULT_X_OPENCTI_SCORE=80
    - URLHAUS_IMPORT_OFFLINE=true
    - URLHAUS_THREATS_FROM_LABELS=true
  restart: always
  depends_on:
    opencti:
      condition: service_healthy
```

5.7.3 Ransomwarelive configuration

```
connector-ransomware:
  image: opencti/connector-ransomwarelive:latest
  container_name: ransomware-connector
  environment:
    - OPENCTI_URL=http://opencti:8080
    - OPENCTI_TOKEN=${OPENCTI_ADMIN_TOKEN}
  restart: always
  depends_on:
    opencti:
      condition: service_healthy
```

All connectors had to be assigned the administrator token of the OpenCTI platform. In the case of the MITRE ATT&CK and Urlhaus connectors, it was also necessary to generate a unique UUID4-type identifier, which was used as the value of the *connector_id* parameter. Finally, a condition was added to the configuration ensuring that the individual connectors are initialized only after the OpenCTI service has started successfully.

After the connectors had been configured, the Docker container stack was stopped using the *docker-compose down* command and then restarted with the *docker-compose up -d* command so that the new settings were applied correctly. It should be emphasized that the *docker-compose down -v* command should not be used, as it would remove the persistent storage and thus also cause the loss of the CTI data already downloaded.

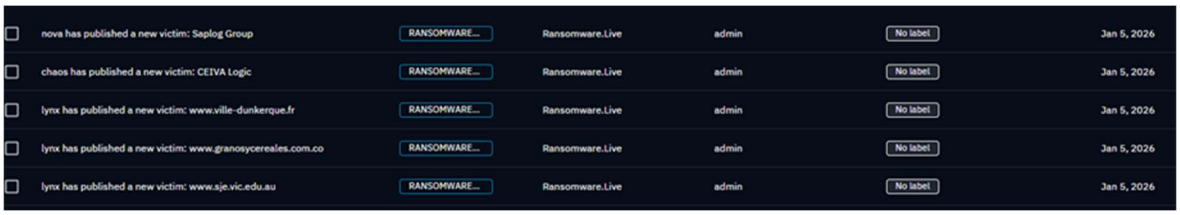
After the connectors had been started successfully, the OpenCTI platform database gradually began to be populated with cyber threat data.

5.7.4 Inputs and outputs

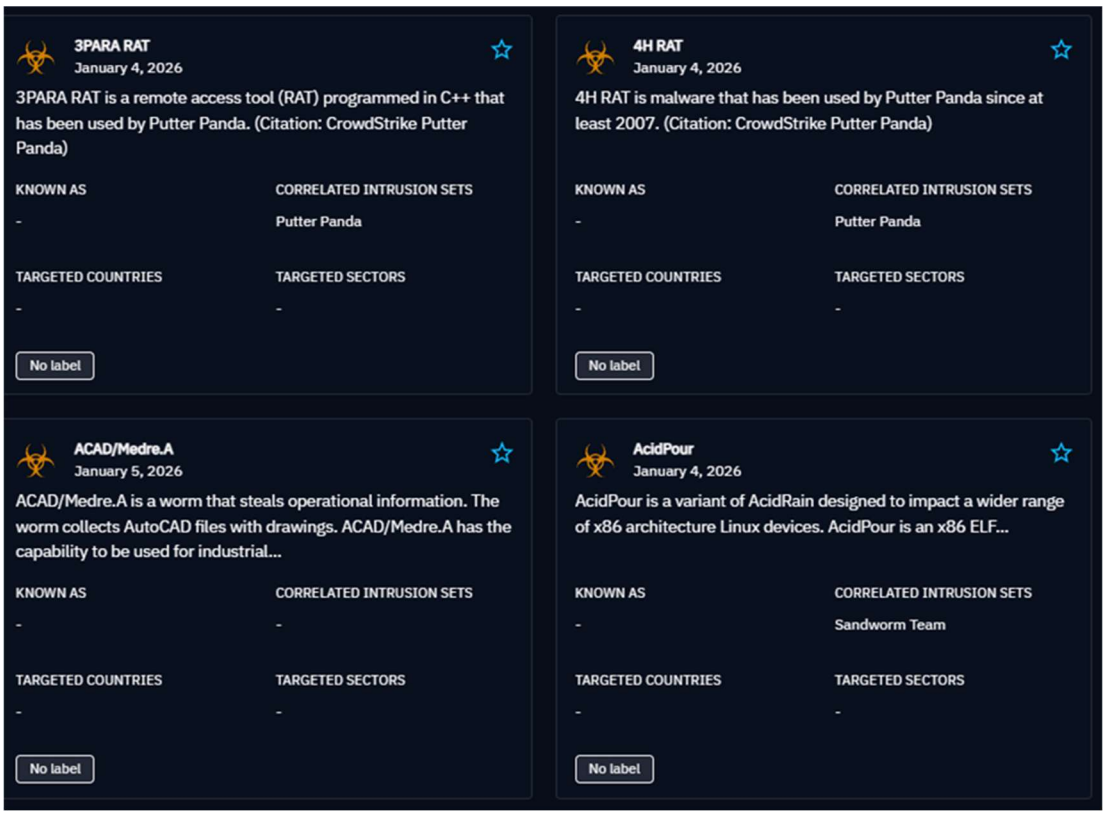
The inputs to OpenCTI are information on cyber threats that come from external sources through connectors or manual entry. These are mainly data on malware, attacker groups, vulnerabilities, IP addresses, domains and other indicators of compromise, for example from systems such as MISP or VirusTotal.

The outputs of OpenCTI are processed and contextualized threat intelligence data, which are available in the web interface, via the API or through export connectors. These outputs are used in other security tools, such as SIEM, SOAR, EDR or firewalls, and serve for the detection and prevention of, and response to, cyber threats.

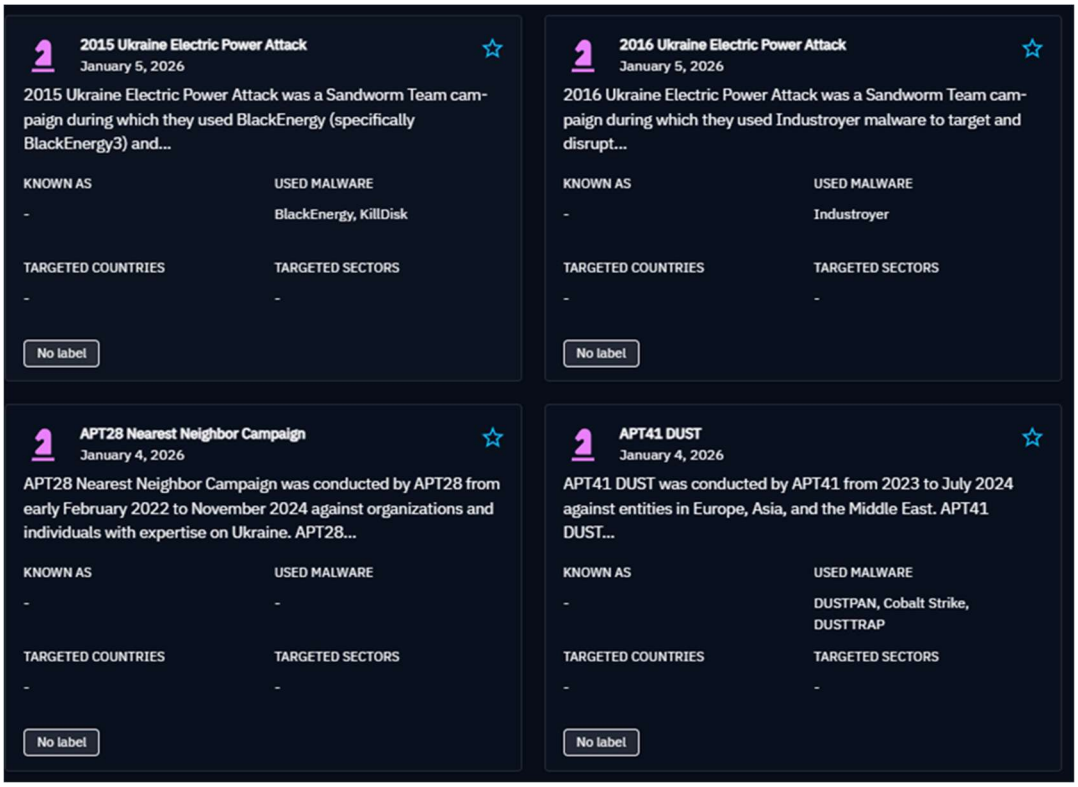
Information on new victims in OpenCTI



Information on known malware in OpenCTI



Information on ongoing campaigns



5.8 Integrations

The platform is designed to cooperate with SIEM and SOAR tools, to which it can provide verified indicators or contextual threat information. In the opposite direction, it can receive data from incidents and enrich them with threat intelligence context. Integrations are also available for EDR or firewall-type systems, where OpenCTI serves as a central threat database from which blocking or detection rules are subsequently generated.

Thanks to its open API and support for the STIX/TAXII standard, OpenCTI is very flexible and makes it possible to create custom integrations according to the organization's needs. This means that, in addition to ready-made connectors, it is easy to program a custom data source or export mechanism, which makes OpenCTI a powerful central tool for cyber threat management in a modern security environment.

5.8.1 OpenCTI integration – position of CTI in the SOC and integration model

5.8.1.1 Position of OpenCTI in the SOC

In the SOC architecture, OpenCTI plays the role of a central threat intelligence (CTI) platform that links strategic, tactical-operational and technical threat information. Unlike SIEM or SOAR tools, OpenCTI does not process logs in real time and does not perform response actions, but provides context that improves the quality of detection and response.

From the perspective of SOC processes, OpenCTI is placed:

- before detection – as a source of indicators and knowledge about threats,
- during analysis – as a contextualization tool in incident investigation,
- before response – as input for decision-making on appropriate measures.

OpenCTI makes it possible to link individual technical indicators (IP addresses, domains, hashes) with a higher-level context, such as campaigns, threat actors or MITRE ATT&CK techniques, thereby supporting analytical decision-making in the SOC.

5.8.1.2 Integration of OpenCTI with other SOC components

OpenCTI is designed as an integration platform, with separate connectors and API interfaces as the basic integration mechanism. Integration is bidirectional – OpenCTI can both receive and provide data.

Integration with SIEM (ELK Stack):

- OpenCTI provides indicators of compromise and context (e.g. IP reputation, link to a campaign),
- the SIEM can use this information in event correlation and in reducing false alarms,
- during incident investigation, the analyst uses OpenCTI to better understand the origin and significance of a detection.

Integration with SOAR (Shuffle):

- OpenCTI serves as a source of threat intelligence for automated playbooks,
- in the enrichment phase, Shuffle can query OpenCTI for the context of an indicator,
- the query results influence the decision logic of the playbook (e.g. escalation, blocking).

Integration with external TI sources:

- OpenCTI ingests data from OSINT or commercial TI feeds through connectors,
- these data are normalized into a uniform STIX model,
- a uniform and auditable view of threat information is ensured.

5.8.1.3 Summary of the integration logic

In the proposed SOC architecture, the following division of roles applies:

- ELK Stack – event detection and analysis,
- NetBox – asset and infrastructure context,
- OpenCTI – threat context and knowledge layer,
- Shuffle – automated response and orchestration.

OpenCTI thus acts as the knowledge node of the SOC, which links technical indicators with a higher-level context and increases both the accuracy and the efficiency of detection and response processes without directly interfering with the infrastructure.

6 MISP

Tool type: Threat Intelligence Platform (TIP) / Information Sharing

Tool name: MISP (Malware Information Sharing Platform)

Version: Open-source version (containerized deployment)

6.1 System description

MISP (Malware Information Sharing Platform) is an open-source platform intended for the collection, management, correlation and sharing of threat information among security teams and organizations. The primary goal of MISP is to enable the efficient exchange of indicators of compromise (IoC), incident information and threat knowledge in a standardized and structured format.

MISP is designed as an operational CTI platform that focuses mainly on the technical level of threat intelligence. It works with objects such as IP addresses, domains, URLs, hashes, e-mail artefacts and other indicators, and enables their correlation across events and sources. Compared with OpenCTI, which places emphasis on modelling threats and their relationships, MISP provides a fast and practical mechanism for exchanging IoCs between systems and organizations.

In the SOC architecture, MISP acts as a source and distribution node of threat indicators, which are used in detection and automated responses. MISP itself performs neither detection nor response, but provides up-to-date and verified inputs for SIEM and SOAR tools, thereby increasing the accuracy of detection rules and supporting a rapid response to known threats.

In the proposed solution, MISP is used as a practical TI feed backend that supplements internal and external sources of threat information and enables their integration into SOC workflows.

6.2 System documentation

The official documentation of the MISP platform is publicly available and provides a detailed description of the architecture, data model, API interfaces and integration options. The documentation focuses mainly on the practical aspects of using MISP in security teams, including the management of events, attributes, correlations and the sharing of data between instances.

Links to official sources:

- Official MISP documentation: https://www.misp-project.org/documentation/
- Project GitHub repository: https://github.com/MISP/MISP
- Official Docker images: https://hub.docker.com/u/misp

6.3 Resource requirements

The hardware requirements of MISP are relatively modest, as the platform is optimized for working with structured TI data and correlations.

For a test or PoC deployment, the following is sufficient:

- CPU: 2 cores
- RAM: 4–8 GB
- Disk space: 50 GB

For production or more intensive use, the following is recommended:

- CPU: 4–8 cores
- RAM: 8–16 GB
- Disk: SSD storage

Higher requirements may arise with a large number of correlations, longer data retention and synchronization with several external MISP instances.

6.4 Tool architecture

The architecture of the MISP platform is designed as a centralized web system intended for the management, correlation and sharing of threat information. The basis of the solution consists of the web application layer, which provides the user interface for managing events, attributes and sharing policies, and the backend application logic, which handles data processing and the correlation mechanisms.

The persistent data layer is implemented by means of a relational database in which events, indicators of compromise (attributes), their metadata and mutual correlation links are stored. MISP automatically correlates new indicators with existing data, thereby enabling the rapid identification of recurring threats or known attack patterns across multiple incidents.

An important element of the architecture is the synchronization mechanism between MISP instances, which enables the secure sharing of events and indicators between organizations. Synchronization takes place on the basis of defined trust rules, filters and sharing policies, which enables the selective provision of data without losing control over sensitive information.

MISP provides a REST API, which is the key integration interface for external systems. Through the API, it is possible to:

- import new indicators and events,
- export IoCs in various formats,
- synchronize data with other platforms (SIEM, SOAR, TIP).

The MISP architecture also supports extensions in the form of modules that enable automated data enrichment or the export of data to external tools. These modules are separated from the system core, which increases the flexibility of the solution and allows it to be adapted to the specific needs of the SOC.

From the perspective of the SOC architecture, MISP is positioned as an operational CTI node that sits between external sources of threat information and the detection tools. It provides normalization, correlation and distribution of indicators of compromise, thereby creating a reliable and auditable data source for detection and response processes.



6.5 Integrations

6.5.1 Position of MISP in the SOC

In the SOC architecture, MISP plays the role of an operational CTI platform focused on practical work with threat indicators (IoC) and their rapid distribution. Unlike SIEM or SOAR tools, MISP does not process logs in real time and does not perform response actions, but provides verified indicators and metadata that increase the accuracy of detection and support rapid decision-making during an incident.

From the perspective of SOC processes, MISP is placed:

- before detection – as a source of IoCs and feeds for detection rules (correlation),
- during analysis – as a reference register of known indicators (tags, scoring, links to events),
- before response – as input for automated decision-making (e.g. escalation, blocking, notification).

MISP is therefore suitable especially where the SOC needs to work quickly with IoCs (IPs/domains/URLs/hashes) and to have control over what is shared, with whom and at what quality.

6.5.2 Integration of MISP with other SOC components

MISP is designed as an integration node for threat intelligence, with the REST API, export formats and synchronization mechanisms between MISP instances as the basic integration mechanism. Integration is bidirectional – MISP can both provide and receive data (e.g. IoCs from incidents, external feeds).

Integration with SIEM (ELK Stack):

- MISP provides IoCs and their metadata (tags, type, severity, time validity),
- the SIEM uses these data in event correlation and alert enrichment (e.g. "the indicator matches TI"),
- investigation outputs (new IoCs from an incident) can be fed back into MISP as internal TI.

Integration with SOAR (Shuffle):

- MISP serves as a TI backend for automated playbooks (lookup / enrichment),
- Shuffle queries MISP for the presence of an indicator, its tags or scoring,
- the result influences the decision logic (escalation, ticket, blocking, notification).

Integration with security systems (EDR / IDS / firewall):

- MISP can serve as a source of indicators for generating detection or blocking rules,
- typically, this involves exporting IoCs into formats suitable for specific tools (e.g. blacklists, signatures),
- the goal is TI-driven detection/response while maintaining auditability and control over distribution.

Integration with external TI sources and the community:

- MISP supports the ingestion of indicators from external feeds and sharing between organizations,
- synchronization between MISP instances enables selective sharing based on trust and filters,
- this ensures up-to-date information and the operational exchange of IoCs in the SOC ecosystem.

6.5.3 OpenCTI versus MISP

MISP and OpenCTI are two different, but to a considerable extent overlapping, platforms for threat intelligence (CTI). Both solutions are capable of fulfilling the role of a central tool for working with threat information in a SOC environment, and in certain scenarios they can be used as alternatives to each other.

From a functional point of view, MISP can be regarded as an opposite (alternative) approach to OpenCTI, since:

- both platforms enable the recording and management of indicators of compromise (IoC),
- they support integration with SIEM and SOAR tools,
- they provide API interfaces for bidirectional data exchange,
- they serve as a source of context for detection and response processes in the SOC.

Within the project, both platforms – MISP and OpenCTI – were tested in order to verify their suitability for the role of the CTI component in the proposed SOC architecture. The testing confirmed that both solutions are capable of performing the basic CTI functions on their own, especially in providing context for detections and supporting analytical decision-making.

At the same time, however, it turned out that MISP and OpenCTI naturally complement each other, with each solution being stronger in a different area:

- MISP is more suitable for ***operational threat intelligence***, rapid work with IoCs, their sharing, synchronization between organizations and direct connection to detection and blocking mechanisms.
- OpenCTI is more suitable for the ***knowledge and analytical layer of CTI***, where it provides richer threat context, relationships between actors, campaigns and techniques (MITRE ATT&CK) and supports strategic and tactical analysis.

In a combined deployment, MISP can play the role of a fast operational source of indicators, while OpenCTI functions as a central threat knowledge base that enriches these indicators with higher-level context. Such a model makes it possible to divide responsibilities between the tools without unnecessary duplication and increases the flexibility of the SOC architecture.

From the perspective of SOC infrastructure design, it is therefore possible to:

- use only MISP or only OpenCTI (depending on the focus of the SOC),
- or combine them, with each tool addressing a different layer of threat intelligence.

This approach makes it possible to scale the CTI capabilities of the SOC according to the needs of the organization without having to commit to a single solution.

7 GLPI

Tool type: Asset Management / IT Service Management (ITSM) / Ticketing

Tool name: GLPI

Version: GLPI 10.0.16 (open-source version, containerized deployment)

7.1 System description

GLPI (Gestionnaire Libre de Parc Informatique) is an open-source tool intended for IT asset inventory, configuration management (CMDB) and IT service management (ITSM). In a SOC environment, it plays the role of a central record-keeping and process system that links technical security events with the organizational context, asset owners and the services provided.

The main purpose of deploying GLPI is:

- asset inventory (hardware, software, virtual systems, network devices),
- CMDB/ITSM records, including relationships between assets, services and users,
- linking assets to owners, responsible persons and organizational units,
- management of incidents, requests and changes through a ticketing system.

GLPI makes it possible to model IT infrastructure and services from a process perspective – that is, to answer the questions *"who the asset belongs to", "what service it supports" and "who is responsible for its operation"*. This view complements the technical documentation provided by tools such as NetBox and creates a bridge between the infrastructure, security and organizational levels.

In the proposed SOC architecture, GLPI is used as:

- a system for the formal recording of security incidents,
- a tool for tracking the incident life cycle (from reporting to closure),
- a component ensuring auditability, accountability and process consistency of incident handling.

GLPI thus does not act as a detection or response tool, but as the process and record-keeping layer of the SOC, which ensures that technical events identified by the SIEM and processed by SOAR tools are properly documented, assigned to assets and handled in accordance with internal procedures and responsibilities.

7.2 System documentation

The official documentation of the GLPI tool is publicly available and covers all the main areas of the system – from installation and configuration to asset management, ticketing, workflows and integrations through the API and plugins. The documentation reflects the community development of the project and supports various deployment models, including a containerized environment.

Links to official sources:

- Official documentation: https://glpi-project.org/documentation/

- Official project website: https://glpi-project.org/
- GitHub repository: https://github.com/glpi-project/glpi
- Official Docker image: https://hub.docker.com/_/glpi
- Official add-ons and plugins: https://glpi-plugins.readthedocs.io/en/latest/

7.3 Resource requirements

The hardware requirements of GLPI depend on the number of recorded assets, users and the volume of tickets processed. For a test or smaller production deployment, the following parameters are sufficient:

Minimum requirements (PoC / smaller environment):

- CPU: 2 cores
- RAM: 4 GB
- Disk space: 20–40 GB
- Database: MySQL / MariaDB

Recommended requirements (production SOC environment):

- CPU: 4–8 cores
- RAM: 8–16 GB
- Disk: SSD storage
- Database: separate DB instance

Higher requirements apply mainly with intensive use of the ticketing system, a large number of assets and extensions through plugins.

7.4 Tool architecture

The architecture of the GLPI tool is designed as a multi-tier web application that separates the presentation, application and data layers. This architectural approach enables easy deployment, scaling and integration into the existing IT and SOC infrastructure without the need for specialized components.

The basis of the solution is the web application layer, which provides the user interface for managing assets, incidents, changes and services. GLPI is implemented as a PHP application running on a web server (Apache or Nginx) and is accessible through a standard web browser. This layer is intended for SOC analysts, IT administrators and other users involved in incident handling.

The application logic of the system provides:

- management of assets and CMDB relationships,
- processing of tickets (incidents, requests, changes),
- workflows and escalation rules,
- notifications and a history of interventions,

- integration mechanisms through the REST API and plugins.

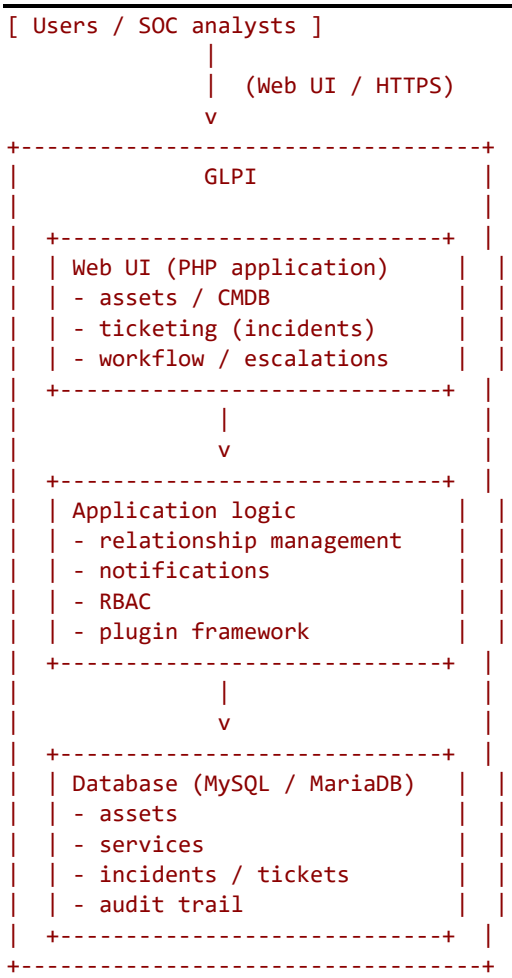
The data layer is represented by a relational database (MySQL or MariaDB), which stores all the operational and historical data of the system, including:

- records of assets and their relationships,
- records of incidents and tickets,
- user accounts and roles,
- the audit trail of changes and interventions.

GLPI supports a plugin mechanism that makes it possible to extend functionality without interfering with the application core. Plugins are used mainly for integration with external tools (monitoring, inventory, security systems) or for extending CMDB and ITSM functions.

In the context of the SOC architecture, GLPI is positioned behind the detection and response layer. It performs neither active detection nor automated responses, but serves as a process and record-keeping system that formally captures security incidents identified by other tools and ensures their controlled handling.

7.4.1 Diagram



```
              |
              | (REST API / Plugins)
              v
+----------------------------------+
| External systems                 |
| - ELK Stack (detection)          |
| - Shuffle (SOAR)                 |
| - NetBox (asset SSOT)            |
| - E-mail / notifications         |
+----------------------------------+
```

7.5 Installation – Docker (Compose) variant

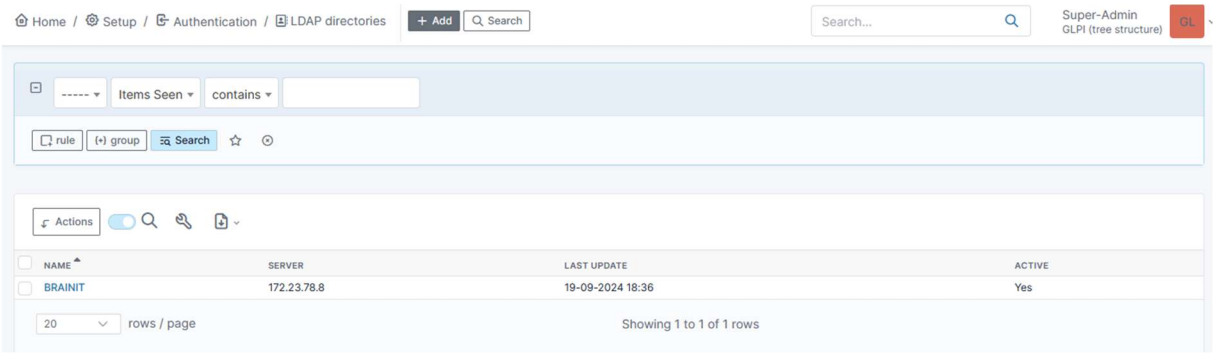
Docker-compose.yml (identical to the official documentation, with NGINX added as an HTTPS reverse proxy for SSL/TLS)

```
services:
  glpi:
    image: "glpi/glpi:latest"
    restart: "unless-stopped"
    volumes:
      - "./storage/glpi:/var/glpi:rw"
    env_file: .env # Pass environment variables from .env file to the container
    depends_on:
      db:
        condition: service_healthy
    ports:
      - "80:80"
  db:
    image: "mysql"
    restart: "unless-stopped"
    volumes:
       - "./storage/mysql:/var/lib/mysql"
    environment:
      MYSQL_RANDOM_ROOT_PASSWORD: "yes"
      MYSQL_DATABASE: ${GLPI_DB_NAME}
      MYSQL_USER: ${GLPI_DB_USER}
      MYSQL_PASSWORD: ${GLPI_DB_PASSWORD}
    healthcheck:
      test: mysqladmin ping -h 127.0.0.1 -u $$MYSQL_USER --password=$$MYSQL_PASSWORD
      start_period: 5s
      interval: 5s
      timeout: 5s
      retries: 10
    expose:
      - "3306"
  nginx-https-proxy:
    image: nginx:latest
    restart: always
    ports:
      - "443:443"
    volumes:
      - ./storage/nginx/conf.d/:/etc/nginx/conf.d/:ro
      - ./storage/nginx/certs/:/etc/nginx/certs/:ro
    environment:
```

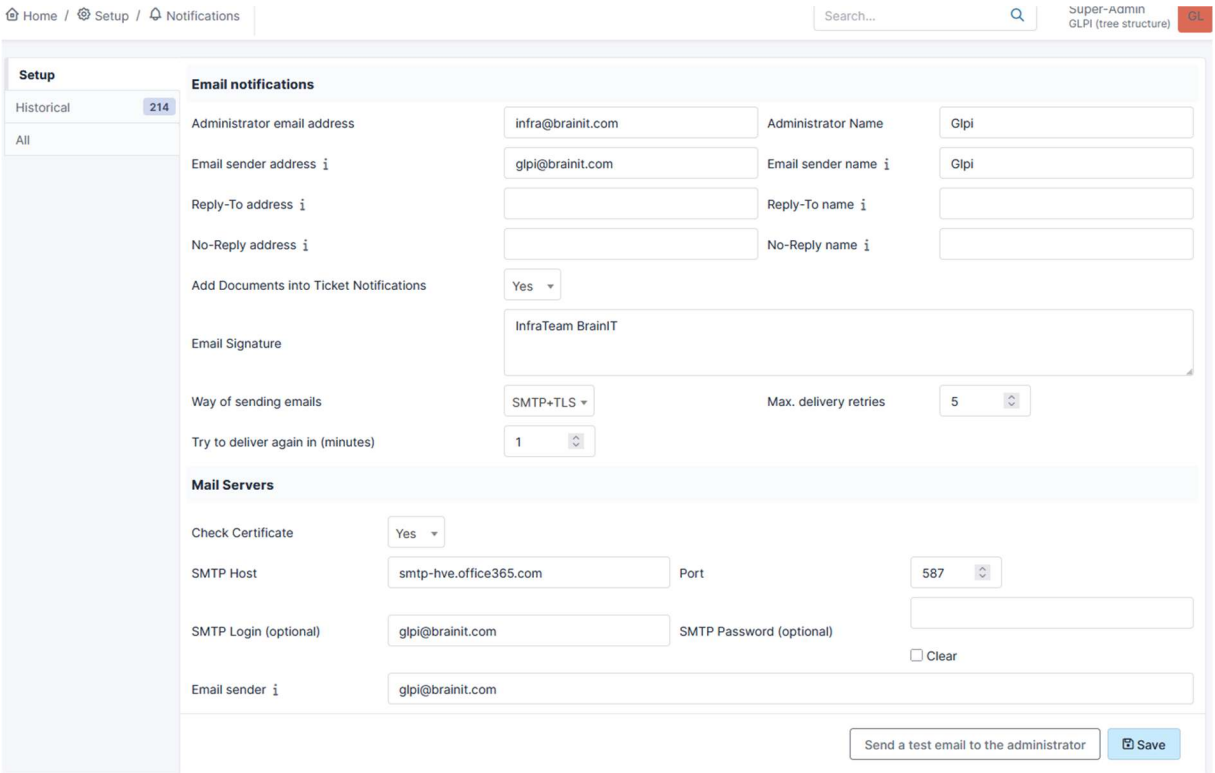
```
    - TIMEZONE=Europe/Bratislava
    - NGINX_HOST=glpi.brainit.tech
    - NGINX_PORT=443
```

```
The .env file
GLPI_DB_HOST=db
GLPI_DB_PORT=3306
GLPI_DB_NAME=glpi
GLPI_DB_USER=glpi
GLPI_DB_PASSWORD=glpi
```

7.6 Basic configuration

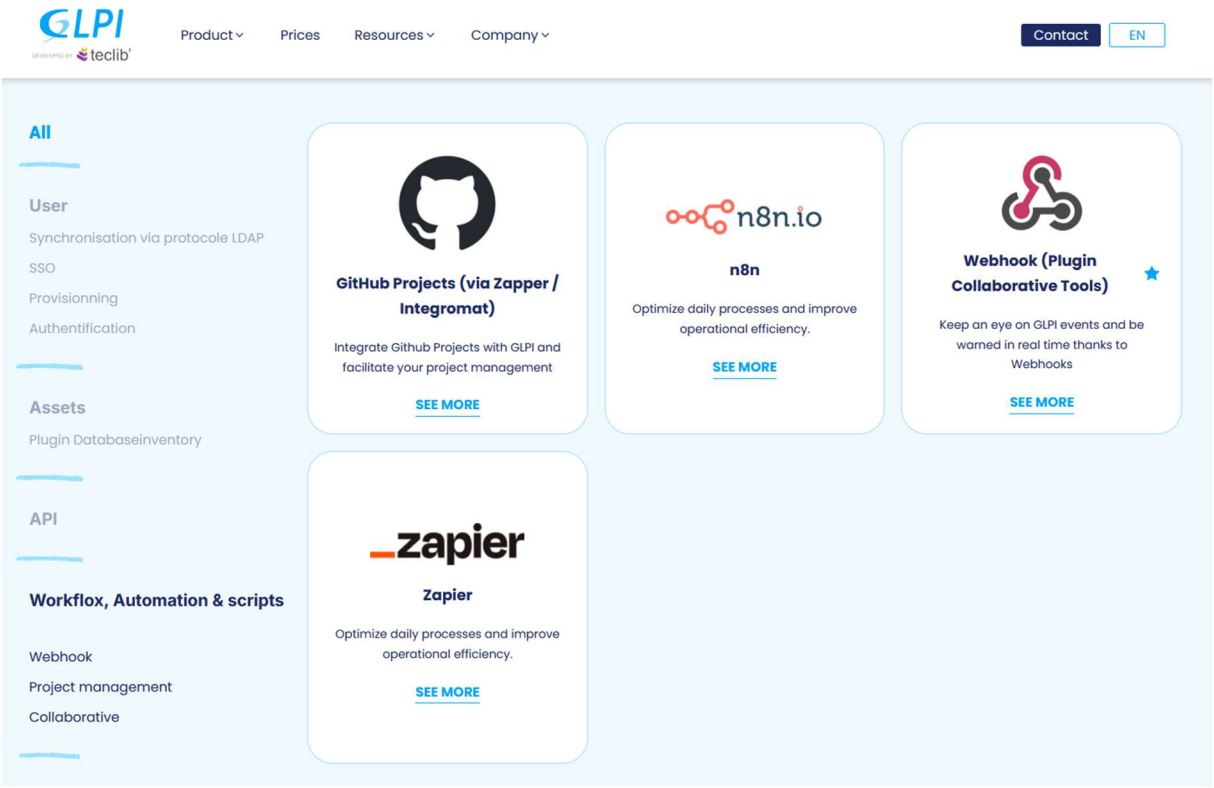


Authentication via AD/LDAP for the brainit.tech domain

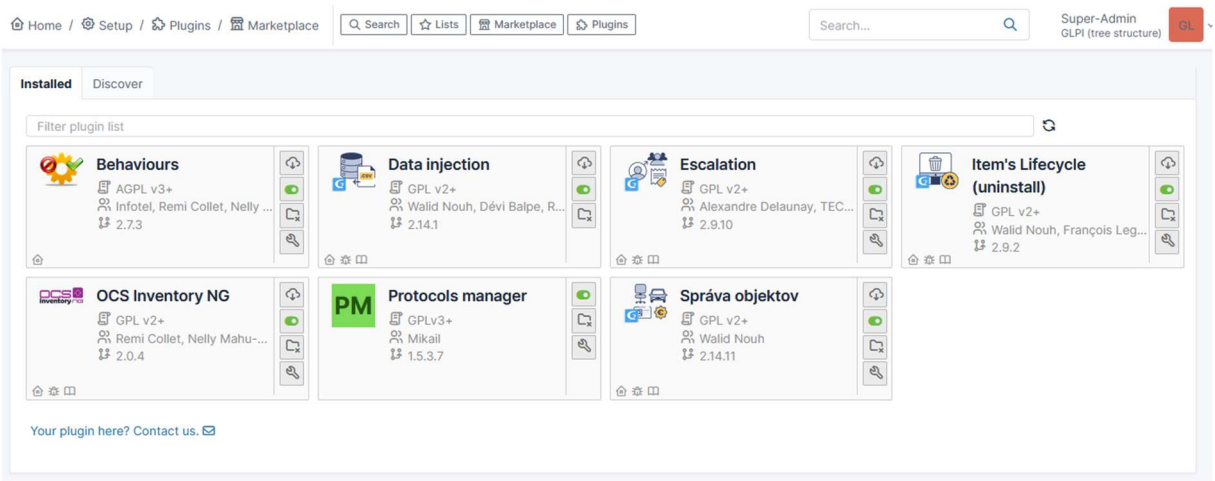


Setting up e-mail notifications as glpi@brainit.com

7.7 Integrations



More on integration options at: https://www.glpi-project.org/en/integrations/



List of plugins in our instance:

- adding mandatory fields, enforcing data formatting, https://github.com/InfotelGLPI/behaviors
- ticket management and escalation https://github.com/pluginsGLPI/escalade
- adding inventory states for decommissioning https://github.com/pluginsglpi/uninstall
- adding custom object types https://github.com/pluginsGLPI/genericobject
- for synchronization with "OCS Inventory NG" https://github.com/pluginsGLPI/ocsinventoryng
- data import in CSV format https://plugins.glpi-project.org/#/plugin/datainjection

7.7.1 Position of GLPI in the SOC

From an architectural point of view, GLPI is the process core of incident handling, which:

- receives technical outputs from detection and response tools,
- maps them to assets, owners and services,
- ensures auditable and controlled incident handling.

This approach makes it possible to separate the technical security logic (SIEM/SOAR) from the process management of incidents (ITSM/CMDB), which increases the clarity, scalability and long-term sustainability of the SOC architecture.

8 NETBOX

Tool type: Asset management / IPAM/DCIM

Tool name: NetBOX

Version: NetBox 4.x (deployment using the official Docker images)

8.1 System description

NetBox is an open-source tool intended for the central management and recording of network and infrastructure documentation, which in a SOC environment plays the role of the central source of truth (Single Source of Truth – SSOT). It is used for the uniform recording of IP address spaces, network devices, physical and logical connections, sites, racks and other infrastructure objects that are essential for proper operation, monitoring and security analyses.

Within the proposed infrastructure, NetBox represents the basic reference layer on which other security and operational tools build. It enables precise recording of IP ranges, VLANs, prefixes and individual IP addresses, thereby creating a consistent basis for network scans, vulnerability scanning and the correlation of security events in SIEM tools or for automation. At the same time, it provides a detailed overview of the physical network topology, devices and their interconnections, which is key in incident analysis, change planning and assessing the impact of outages.

NetBox supports infrastructure modelling at several levels – from the physical layer (racks, devices, ports, cables, power) to the logical layer (IPAM, VLAN, VPN, virtual machines). This makes it possible to link technical, operational and administrative information about the infrastructure in a single system. The solution also includes a REST API, which enables integration with external tools, automation scripts and security platforms.

In the context of the multi-tenant SOC solution, NetBox is used as a stable and consistent data source that reduces ambiguity in the interpretation of infrastructure data and supports process automation. Its role is not active detection of or response to incidents, but the provision of accurate and up-to-date data on which the other components of the SOC infrastructure reliably rely.

8.2 Documentation

The official documentation of the NetBox tool is publicly available and continuously updated by the community and the project developers. The documentation covers all the main areas of the system, including the architecture, installation procedures, configuration options, working with the object model and the REST API interface, as well as the options for extension through plugins and custom scripts. The documentation also reflects the rapid development of the project and contains recommendations for production deployment and operation in larger infrastructures.

- Official documentation: https://github.com/netbox-community/netbox
- Official NetBox project website: https://netboxlabs.com/
- Project GitHub repository: https://github.com/netbox-community/netbox

- Official Docker image (NetBox and related components): https://hub.docker.com/r/netboxcommunity/netbox

8.3 Resource requirements

The hardware requirements of NetBox depend on the scope of the documented infrastructure and the intensity of API and automation use. For a basic to medium-sized deployment, the following minimum parameters are recommended:

- **CPU:** 2 vCPU
- **RAM:** 4 GB
- **Disk space:** 20 GB (SSD recommended)
- **Database:** PostgreSQL (separate container or external instance)
- **Cache / Queue:** Redis

For larger deployments or intensive use of automation and plugins, it is recommended to increase memory and computing resources.

8.4 Tool architecture

The architecture of the NetBox tool is based on a multi-tier web application model in which the individual functional components are separated into distinct logical and operational layers. This approach enables scalability, easy maintenance and a clear separation of the application logic from persistent data and auxiliary services.

In a deployment using containerization technologies, NetBox is operated as a set of interconnected Docker containers, each of which performs a specific role within the overall solution. The application layer consists of the NetBox application itself, which provides the web user interface and the REST API. This layer processes requests from users and external systems and executes the application logic on the stored data.

The data layer is represented by the PostgreSQL database system, which stores all the configuration and documentation data of the infrastructure. To optimize performance and process background tasks, the Redis system is used, which serves both as a cache and as a queue for asynchronous tasks. These tasks are processed by separate worker processes, which minimizes the load on the main application layer.

Communication between the individual components takes place through defined network interfaces within the Docker environment, and the architecture allows all services to be deployed on a single host or distributed across several nodes. This model is in line with the official Docker-based architecture of NetBox and represents the recommended deployment method for production environments.

The textual architecture diagram below illustrates the basic relationships between the individual NetBox components in a containerized environment and serves as a reference overview of the deployment architecture.

```
[ User / API client ]
          |
      [ NetBox ]
       /        \
[ PostgreSQL ] [ Redis ]
                    |
               [ RQ Workers ]
```

8.4.1 Infrastructure modelling

8.4.1.1 Modelling relationships and objects in NetBox – principles

NetBox is built on an object model of the infrastructure in which the individual network elements are represented as separate objects with defined properties and relationships. The basic principle of this approach is the separation of the definition of objects (what an object is) from their mutual links (how they relate to each other). This makes it possible to model the infrastructure in a way that faithfully reflects its real physical and logical structure.

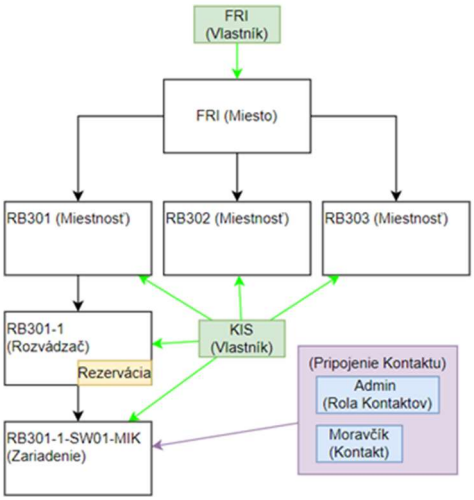
The relationships between objects in NetBox are not arbitrary, but are based on predefined rules and a hierarchy that determine which objects can exist independently and which depend on others. A typical example is the link of a device to a site or room, or the link of IP addresses to prefixes and VLANs. These links ensure data consistency and prevent ambiguous or incomplete records from arising in the documentation.

Modelling is based on the principle of gradual refinement – first, general context objects (organization, sites, owners) are defined, to which the technical infrastructure objects (devices, interfaces, IP addresses) are subsequently linked. Such an approach enables uniform documentation of the infrastructure across different levels of detail and supports the scalability of the model as the network grows.

An important aspect of modelling in NetBox is also the emphasis on a single source of truth. Each object and its relationships are to be recorded in only one place, and other systems refer to these data via the API or exports. In the context of the SOC infrastructure, this means that NetBox does not provide active monitoring or security analytics, but creates an accurate and consistent reference framework that is essential for the correct interpretation of data in other tools.

Based on these principles, the following chapters elaborate on specific categories of objects and their relationships, focusing on organizational objects and device objects, which form the basis of network infrastructure documentation.

8.4.1.2 Modelling organization objects



2 Relationships – Organization

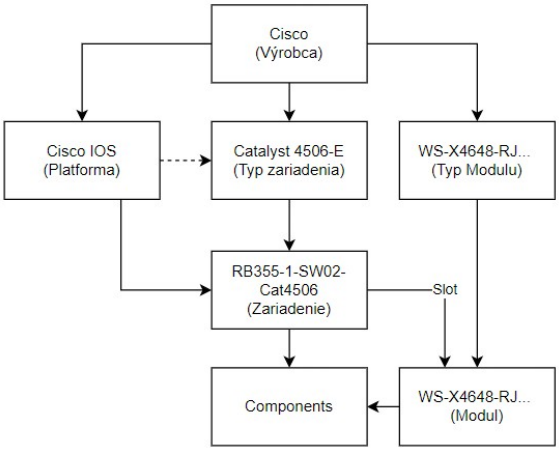
In NetBox, the **Organization** category is used to model the non-technical context of the infrastructure – in particular **ownership**, **location** and **contact persons**. In practice, this model relies on the basic objects **Sites** and **Locations (rooms)**, which ensure that each device has a defined *physical location*. In the KIS environment, the model is simplified to a single branch (Site) representing a building, e.g. **FRI**, with individual classrooms, offices and technical rooms recorded as **Locations** linked to the given Site. Racks or devices (Devices) are then placed directly in the rooms.

An important part of the organizational model are **Tenants (owners)**, which determine to whom objects such as devices, sites or IP prefixes belong. In this approach, each object is assigned at most one owner, and ownership is used mainly for administrative purposes and clarity of management (e.g. distinguishing FRI/KIS/another owner).

The model is complemented by **Contacts**, **Contact Groups** and **Contact Roles**, which make it possible to record responsible persons and their roles in infrastructure management (e.g. device administrator). Linking a contact, its role and a specific object creates a "contact assignment" – a practical link that supports rapid escalation and operational orientation in asset management.

In terms of the recommended order of object creation (according to the internal relationship diagram), it is key first to define the owners, then the branches (Sites), then the rooms (Locations), and only then the dependent infrastructure objects. This logic is also emphasized in the diagram "Relationships, Organization category", which describes the recommended sequence and mandatory links between objects.

8.4.1.3 Modelling infrastructure objects



Relationships – Devices

The Devices category in the NetBox tool is used to model the physical network infrastructure and its components. Devices represent both active and passive network elements, such as switches, routers, firewalls and wireless access points, but also patch panels or distribution cables. Each device is linked to a specific site (Site) and room (Location), which ensures precise physical localization within the infrastructure. When creating a device, it is mandatory to define at least its role, device type, location and status.

The basis of device modelling are Device Types, which function as templates. These templates contain information about the device model, its dimensions (U units), interfaces, modules, and power and console ports. The use of device types ensures the consistency of documentation and the automatic creation of all the necessary components when a new device is set up. Device types are managed centrally and can also be imported from public vendor or community repositories.

The device category also includes Device Roles, which express the function of a device in the network (e.g. switch, router, firewall). Roles enable devices to be sorted logically and used further in automation, filtering or when generating configurations. Platforms define the type of operating system of a device and are used for simplified differentiation of technology platforms without the need to record exact software versions.

NetBox also enables the modelling of more advanced constructs, such as Virtual Chassis, which represent several physical devices sharing a single control plane (e.g. StackWise or vPC technologies). Devices may contain modules, interfaces, console ports, power ports and other components that enable a detailed and realistic representation of the physical network topology. These components are subsequently used for documenting connections and visualizing the topology.

Device modelling in NetBox creates an accurate and uniform picture of the physical infrastructure, which can be used as a basis for network maps, monitoring, automation and security analyses within the SOC environment.

8.5 Experiment

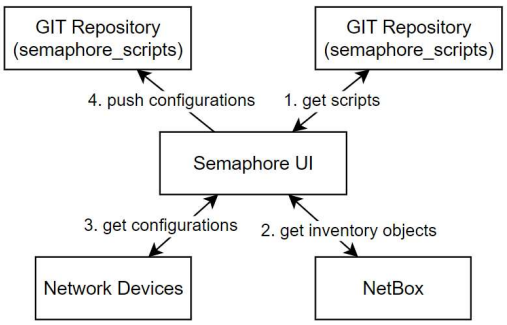
The aim of the experimental part was to verify the deployment and practical use of the NetBox tool as a central source of truth for the documentation of assets relevant to the SOC. The Proof of Concept (PoC) was carried out primarily for the purpose of introducing consistent asset records (devices, sites, connections, IP addressing and selected services) so that these data could be used further in security monitoring, scanning, event correlation and incident handling. The goal was therefore not to create a complete production solution, but a controlled verification of the basic principles of documentation and of the readiness of the data model for the needs of the SOC.

The PoC builds on internal experience with network and infrastructure documentation and on published results demonstrating that NetBox is suitable for replacing static documentation (wikis, spreadsheets) with a database approach and provides a stable basis for further operational and automation processes. In the experiment, emphasis was placed on ensuring that the documentation was not created as an isolated "inventory", but as a reference layer for follow-up SOC activities – in particular for the purposes of asset identification and classification, topology mapping and the unambiguous assignment of IP addresses and network segments to specific devices and sites.

The PoC included the verification of several NetBox deployment variants, with the main focus on containerized deployment using Docker. The basic configuration of the system, extensibility through plugins and the options for integration with automation tools (in particular for repeatable operational tasks such as inventory management and configuration backup) were also verified. The outputs of the experiment constitute the technical basis for further

extension of the asset documentation and its linking with the other components of the SOC infrastructure.

8.5.1 Diagram



Experiment diagram

The experiment diagram shows the PoC workflow in which NetBox is used as the central asset register (asset inventory / SSOT) for the needs of the SOC. The primary goal is to ensure that SOC processes work with consistent data on which assets exist, where they are located, what IP addresses/segments they have and what relationships exist between them. This layer of "truth" is critical for the correct interpretation of events (logs/alerts), for mapping incidents to specific devices and for precise orientation in segmentation during an investigation. In the diagram, Semaphore UI acts as the orchestrator of automation tasks (Ansible), with the information sources divided as follows:

1. the Git repository provides scripts and task definitions (runbook/automation-as-code),
2. NetBox provides inventory data on assets (devices, roles, IP addressing, relationships),
3. network devices are the target of operations (retrieval of configurations).

From the SOC perspective, the logic is as follows:

- Asset (inventory): NetBox defines the authoritative view of assets and their context (where a device is, what interfaces, IP addresses and segments it has).
- Detection (correct interpretation): consistent asset data reduce ambiguity when mapping events to a specific object (e.g. "this IP address belongs to this device at this site"). This supports triage and prioritization of incidents, because the asset can be quickly identified and placed in the right context.
- Response (repeatable operations): Semaphore/Ansible performs standardized tasks (in this PoC mainly configuration backup), thereby creating an operational artefact usable during an incident: a reference configuration state for change analysis, recovery and forensic comparison. The output (configurations) is versioned in the Git repository, which supports auditability and retrospective traceability of changes.

The experiment thus demonstrates that NetBox as an asset SSOT is not a "detection tool", but provides a key foundation for the SOC: the correct asset context for detection and a repeatable mechanism for supporting response (through automated operational tasks and versioned configurations). It also confirms that NetBox is suitable as a reference database for automated activities relevant to the SOC, without itself actively intervening in the infrastructure.

8.5.2 Installation – Docker variant

In the PoC, NetBox is deployed in containers using **Docker Compose**. The following Docker containers were running within the experimental PoC deployment:

1. netbox – main application (Web UI + REST API)
2. netbox-worker – background worker (RQ)

3. netbox-housekeeping – scheduled housekeeping tasks
4. postgres – database (PostgreSQL)
5. redis – queue/cache layer (in your case it is the Valkey image)
6. redis-cache – separate cache/redis endpoint (also Valkey)

The configuration is split into several files so that it is possible to separate

- the service definitions,
- the runtime environment variables and
- the NetBox application configuration, including plugins.

The deployment is carried out via a combination of docker-compose.yml and docker-compose.override.yml (compose automatically loads the override file if it is in the same folder).

Configuration of the application services (NetBox)

The basic configuration is built on a combination of:

- netbox.env – runtime variables,
- configuration.py – NetBox/Django configuration, which maps these variables into the system.

Key settings in the PoC (selection):

- **DB connection:** DB_HOST=postgres, DB_NAME=netbox, DB_USER=netbox (the password is in env).
- **Redis separated into "tasks" and "cache":**
  - tasks: REDIS_HOST=redis, REDIS_DATABASE=0
  - cache: REDIS_CACHE_HOST=redis-cache, REDIS_CACHE_DATABASE=1
    This is also directly reflected in configuration.py, where there are two separate sections, REDIS['tasks'] and REDIS['caching'], with the recommendation to use different DB IDs.
- **API and integration extensions:** GRAPHQL_ENABLED=true, WEBHOOKS_ENABLED=true (readiness for integrations and automation).
- **CORS:** CORS_ORIGIN_ALLOW_ALL=True (open CORS in PoC mode).
- **Housekeeping:** HOUSEKEEPING_INTERVAL=86400 (periodicity 24h).

Plugins (installation + activation)

Plugins are handled in two steps:

- **(a) Installing the packages into the image**
  The plugins are defined in plugin_requirements.txt:
  - netbox-topology-views
  - netbox-reorder-rack
  - netbox-attachments

- **(b) Activation and configuration in NetBox**
  In plugins.py, the plugins are activated via PLUGINS and configured via PLUGINS_CONFIG. Specifically, netbox_topology_views has the saving of coordinates enabled (topology visualization).

8.5.2.1 Prerequisites for running

- Host system: Linux (or a VM in OpenStack IaaS), network access as required by the PoC.
- Installed tools:
  - **Docker Engine**
  - **Docker Compose v2** (the docker compose … command)
- Prepared directory structure according to the PoC (i.e. netbox/ contains docker-compose.yml, docker-compose.override.yml, Dockerfile, configuration/ and env/) **as shown below**.

8.5.2.2 List of files (composite compose)

- **docker-compose.yml:** The basic orchestration file for Docker Compose – defines the services of the NetBox stack (NetBox application, worker processes, housekeeping, PostgreSQL database and Redis layers), their dependencies, networking and persistent volumes.
- **docker-compose.override.yml:** An override file used to supplement or overwrite parameters from docker-compose.yml (typically local modifications, mounts of configuration files, build settings). This makes it possible to preserve the basic compose structure and change only the necessary parts for a specific environment.
- **Dockerfile:** A custom build of the application image on top of the official NetBox image, used to add plugins/packages. In the PoC, it is used together with plugin_requirements.txt (installation of selected NetBox plugins).
- **plugin_requirements.txt:** A list of plugin packages to be additionally installed into the NetBox image. In this deployment, the following plugins are defined:
  - netbox-topology-views
  - netbox-reorder-rack
  - netbox-attachments
- **netbox.env:** Central runtime environment variables for the NetBox stack. In the PoC, the following key parameters are set:
  - DB connection (DB_HOST=postgres, DB_NAME=netbox, DB_USER=netbox)
  - Redis for tasks (REDIS_HOST=redis, REDIS_DATABASE=0)
  - Redis cache (REDIS_CACHE_HOST=redis-cache, REDIS_CACHE_DATABASE=1)
  - functional modules: GRAPHQL_ENABLED=true, WEBHOOKS_ENABLED=true
  - housekeeping interval: HOUSEKEEPING_INTERVAL=86400
- **redis-cache.env:** Variables for the Redis/Valkey cache layer (in the PoC it contains, for example, the password for the cache backend).
- **Configuration.py:** The main application configuration of NetBox, which reads values from netbox.env and maps them into Django/NetBox settings. Crucially, the config sets:

  - the database connection via the DB_* variables (PostgreSQL)
  - two separate Redis configurations: tasks and caching (separate database IDs and endpoints)
  - CORS and other optional settings via env variables
- **plugins.py:** Plugin configuration – explicitly defines which plugins are active and their settings. In the PoC, the following are activated:
  - netbox_topology_views (with saving of coordinates and a static image directory)
  - netbox_reorder_rack
  - netbox_attachments

8.5.2.2.1 File structure

```
+---netbox
|   |   docker-compose.yml
|   |   docker-compose.override.yml
|   |   Dockerfile
|   |   plugin_requirements.txt
|   |
|   +---configuration
|   |       configuration.py
|   |       plugins.py
|   |
|   \---env
|           netbox.env
|           postgres.env
|           redis-cache.env
|           redis.env
|
\---semaphore
        docker-compose.yml
        requirements.txt
```

8.5.2.3 Functional principle of the deployment (what ensures "full functionality")

- Compose ensures that all the necessary services are started (application + database + queue/cache layers + background processes).
- netbox.env holds the runtime parameters (DB/Redis, enabled API modules and the housekeeping interval), which are passed into the containers.
- configuration.py is the "source of truth" for the application settings – it maps the env variables into the NetBox configuration and ensures the correct connection to PostgreSQL and both Redis backends (tasks + caching).
- Plugins are handled in two steps: (a) installation of the packages (plugin_requirements.txt) and (b) activation + configuration (plugins.py).

8.5.2.3.1 Netbox\docker-compose.yml

```
services:
  netbox: &netbox
    image: docker.io/netboxcommunity/netbox:${VERSION-v4.2-3.2.0}
    depends_on:
```

```
    - postgres
    - redis
    - redis-cache
  env_file: env/netbox.env
  user: "unit:root"
  healthcheck:
    test: curl -f http://localhost:8080/login/ || exit 1
    start_period: 90s
    timeout: 3s
    interval: 15s
  volumes:
    - ./configuration:/etc/netbox/config:z,ro
    - netbox-media-files:/opt/netbox/netbox/media:rw
    - netbox-reports-files:/opt/netbox/netbox/reports:rw
    - netbox-scripts-files:/opt/netbox/netbox/scripts:rw
netbox-worker:
  <<: *netbox
  depends_on:
    netbox:
      condition: service_healthy
  command:
    - /opt/netbox/venv/bin/python
    - /opt/netbox/netbox/manage.py
    - rqworker
  healthcheck:
    test: ps -aux | grep -v grep | grep -q rqworker || exit 1
    start_period: 20s
    timeout: 3s
    interval: 15s
netbox-housekeeping:
  <<: *netbox
  depends_on:
    netbox:
      condition: service_healthy
  command:
    - /opt/netbox/housekeeping.sh
  healthcheck:
    test: ps -aux | grep -v grep | grep -q housekeeping || exit 1
    start_period: 20s
    timeout: 3s
    interval: 15s

# postgres
postgres:
  image: docker.io/postgres:17-alpine
  healthcheck:
    test: pg_isready -q -t 2 -d $$POSTGRES_DB -U $$POSTGRES_USER
    start_period: 20s
    timeout: 30s
    interval: 10s
    retries: 5
  env_file: env/postgres.env
  volumes:
    - netbox-postgres-data:/var/lib/postgresql/data

# redis
```

```
  redis:
    image: docker.io/valkey/valkey:8.0-alpine
    command:
      - sh
      - -c # this is to evaluate the $REDIS_PASSWORD from the env
      - valkey-server --appendonly yes --requirepass $$REDIS_PASSWORD ## $$ because of
docker-compose
    healthcheck: &redis-healthcheck
      test: '[ $$(valkey-cli --pass "$${REDIS_PASSWORD}" ping) = ''PONG'' ]'
      start_period: 5s
      timeout: 3s
      interval: 1s
      retries: 5
    env_file: env/redis.env
    volumes:
      - netbox-redis-data:/data
  redis-cache:
    image: docker.io/valkey/valkey:8.0-alpine
    command:
      - sh
      - -c # this is to evaluate the $REDIS_PASSWORD from the env
      - valkey-server --requirepass $$REDIS_PASSWORD ## $$ because of docker-compose
    healthcheck: *redis-healthcheck
    env_file: env/redis-cache.env
    volumes:
      - netbox-redis-cache-data:/data

volumes:
  netbox-media-files:
    driver: local
  netbox-postgres-data:
    driver: local
  netbox-redis-cache-data:
    driver: local
  netbox-redis-data:
    driver: local
  netbox-reports-files:
    driver: local
  netbox-scripts-files:
    driver: local
```

8.5.2.3.2 Netbox\docker-compose.override.yml

```
services:
  netbox:
    image: netbox:plugins
    ports:
      - "8000:8080"  # Change the external port
    build:
      context: .
      dockerfile: Dockerfile
  netbox-worker:
    image: netbox:plugins
    pull_policy: never
  netbox-housekeeping:
    image: netbox:plugins
```

```
    pull_policy: never
```

8.5.2.3.3 Netbox\Dockerfile

```
FROM netboxcommunity/netbox:latest

COPY ./plugin_requirements.txt /opt/netbox/
RUN /usr/local/bin/uv pip install -r /opt/netbox/plugin_requirements.txt

# These lines are only required if your plugin has its own static files.
COPY configuration/configuration.py /etc/netbox/config/configuration.py
COPY configuration/plugins.py /etc/netbox/config/plugins.py
RUN DEBUG="true" SECRET_KEY="dummydummydummydummydummydummydummydummydummydummy" \
    /opt/netbox/venv/bin/python /opt/netbox/netbox/manage.py collectstatic --no-input
```

8.5.2.3.4 Netbox\plugin_requirements.txt

```
netbox-topology-views
netbox-reorder-rack
netbox-attachments
```

8.5.2.3.5 Netbox\configuration\configuration.py

```
####
## We recommend to not edit this file.
## Create separate files to overwrite the settings.
## See `extra.py` as an example.
####

import re
from os import environ
from os.path import abspath, dirname, join
from typing import Any, Callable, Tuple

# For reference see https://docs.netbox.dev/en/stable/configuration/
# Based on https://github.com/netbox-
community/netbox/blob/develop/netbox/netbox/configuration_example.py

###
# NetBox-Docker Helper functions
###

# Read secret from file
def _read_secret(secret_name: str, default: str | None = None) -> str | None:
    try:
        f = open('/run/secrets/' + secret_name, 'r', encoding='utf-8')
    except EnvironmentError:
        return default
    else:
        with f:
            return f.readline().strip()
```

```
# If the `map_fn` isn't defined, then the value that is read from the environment (or the
default value if not found) is returned.
# If the `map_fn` is defined, then `map_fn` is invoked and the value (that was read from
the environment or the default value if not found)
# is passed to it as a parameter. The value returned from `map_fn` is then the return value
of this function.
# The `map_fn` is not invoked, if the value (that was read from the environment or the
default value if not found) is None.
def _environ_get_and_map(variable_name: str, default: str | None = None, map_fn:
Callable[[str], Any | None] = None) -> Any | None:
    env_value = environ.get(variable_name, default)

    if env_value == None:
        return env_value

    if not map_fn:
        return env_value

    return map_fn(env_value)

_AS_BOOL = lambda value : value.lower() == 'true'
_AS_INT = lambda value : int(value)
_AS_LIST = lambda value : list(filter(None, value.split(' ')))

_BASE_DIR = dirname(dirname(abspath(__file__)))

#########################
#                       #
#   Required settings   #
#                       #
#########################

# This is a list of valid fully-qualified domain names (FQDNs) for the NetBox server.
NetBox will not permit write
# access to the server via any other hostnames. The first FQDN in the list will be treated
as the preferred name.
#
# Example: ALLOWED_HOSTS = ['netbox.example.com', 'netbox.internal.local']
ALLOWED_HOSTS = environ.get('ALLOWED_HOSTS', '*').split(' ')
# ensure that '*' or 'localhost' is always in ALLOWED_HOSTS (needed for health checks)
if '*' not in ALLOWED_HOSTS and 'localhost' not in ALLOWED_HOSTS:
    ALLOWED_HOSTS.append('localhost')

# PostgreSQL database configuration. See the Django documentation for a complete list of
available parameters:
#   https://docs.djangoproject.com/en/stable/ref/settings/#databases
DATABASE = {
    'NAME': environ.get('DB_NAME', 'netbox'),       # Database name
    'USER': environ.get('DB_USER', ''),             # PostgreSQL username
    'PASSWORD': _read_secret('db_password', environ.get('DB_PASSWORD', '')),
                                                    # PostgreSQL password
    'HOST': environ.get('DB_HOST', 'localhost'),    # Database server
```

```
    'PORT': environ.get('DB_PORT', ''),             # Database port (leave blank for
default)
    'OPTIONS': {'sslmode': environ.get('DB_SSLMODE', 'prefer')},
                                                    # Database connection SSLMODE
    'CONN_MAX_AGE': _environ_get_and_map('DB_CONN_MAX_AGE', '300', _AS_INT),
                                                    # Max database connection age
    'DISABLE_SERVER_SIDE_CURSORS': _environ_get_and_map('DB_DISABLE_SERVER_SIDE_CURSORS',
'False', _AS_BOOL),
                                                    # Disable the use of server-side
cursors transaction pooling
}

# Redis database settings. Redis is used for caching and for queuing background tasks such
as webhook events. A separate
# configuration exists for each. Full connection details are required in both sections, and
it is strongly recommended
# to use two separate database IDs.
REDIS = {
    'tasks': {
        'HOST': environ.get('REDIS_HOST', 'localhost'),
        'PORT': _environ_get_and_map('REDIS_PORT', 6379, _AS_INT),
        'SENTINELS': [tuple(uri.split(':')) for uri in
_environ_get_and_map('REDIS_SENTINELS', '', _AS_LIST) if uri != ''],
        'SENTINEL_SERVICE': environ.get('REDIS_SENTINEL_SERVICE', 'default'),
        'SENTINEL_TIMEOUT': _environ_get_and_map('REDIS_SENTINEL_TIMEOUT', 10, _AS_INT),
        'USERNAME': environ.get('REDIS_USERNAME', ''),
        'PASSWORD': _read_secret('redis_password', environ.get('REDIS_PASSWORD', '')),
        'DATABASE': _environ_get_and_map('REDIS_DATABASE', 0, _AS_INT),
        'SSL': _environ_get_and_map('REDIS_SSL', 'False', _AS_BOOL),
        'INSECURE_SKIP_TLS_VERIFY': _environ_get_and_map('REDIS_INSECURE_SKIP_TLS_VERIFY',
'False', _AS_BOOL),
    },
    'caching': {
        'HOST': environ.get('REDIS_CACHE_HOST', environ.get('REDIS_HOST', 'localhost')),
        'PORT': _environ_get_and_map('REDIS_CACHE_PORT', environ.get('REDIS_PORT', '6379'),
_AS_INT),
        'SENTINELS': [tuple(uri.split(':')) for uri in
_environ_get_and_map('REDIS_CACHE_SENTINELS', '', _AS_LIST) if uri != ''],
        'SENTINEL_SERVICE': environ.get('REDIS_CACHE_SENTINEL_SERVICE',
environ.get('REDIS_SENTINEL_SERVICE', 'default')),
        'USERNAME': environ.get('REDIS_CACHE_USERNAME', environ.get('REDIS_USERNAME', '')),
        'PASSWORD': _read_secret('redis_cache_password',
environ.get('REDIS_CACHE_PASSWORD', environ.get('REDIS_PASSWORD', ''))),
        'DATABASE': _environ_get_and_map('REDIS_CACHE_DATABASE', '1', _AS_INT),
        'SSL': _environ_get_and_map('REDIS_CACHE_SSL', environ.get('REDIS_SSL', 'False'),
_AS_BOOL),
        'INSECURE_SKIP_TLS_VERIFY':
_environ_get_and_map('REDIS_CACHE_INSECURE_SKIP_TLS_VERIFY',
environ.get('REDIS_INSECURE_SKIP_TLS_VERIFY', 'False'), _AS_BOOL),
    },
}
```

```
# This key is used for secure generation of random numbers and strings. It must never be
exposed outside of this file.
# For optimal security, SECRET_KEY should be at least 50 characters in length and contain a
mix of letters, numbers, and
# symbols. NetBox will not run without this defined. For more information, see
# https://docs.djangoproject.com/en/stable/ref/settings/#std:setting-SECRET_KEY
SECRET_KEY = _read_secret('secret_key', environ.get('SECRET_KEY', ''))


#########################
#                       #
#   Optional settings   #
#                       #
#########################

# # Specify one or more name and email address tuples representing NetBox administrators.
These people will be notified of
# # application errors (assuming correct email settings are provided).
# ADMINS = [
#    # ['John Doe', 'jdoe@example.com'],
# ]

if 'ALLOWED_URL_SCHEMES' in environ:
    ALLOWED_URL_SCHEMES = _environ_get_and_map('ALLOWED_URL_SCHEMES', None, _AS_LIST)

# Optionally display a persistent banner at the top and/or bottom of every page. HTML is
allowed. To display the same
# content in both banners, define BANNER_TOP and set BANNER_BOTTOM = BANNER_TOP.
if 'BANNER_TOP' in environ:
    BANNER_TOP = environ.get('BANNER_TOP', None)
if 'BANNER_BOTTOM' in environ:
    BANNER_BOTTOM = environ.get('BANNER_BOTTOM', None)

# Text to include on the login page above the login form. HTML is allowed.
if 'BANNER_LOGIN' in environ:
    BANNER_LOGIN = environ.get('BANNER_LOGIN', None)

# Maximum number of days to retain logged changes. Set to 0 to retain changes indefinitely.
(Default: 90)
if 'CHANGELOG_RETENTION' in environ:
    CHANGELOG_RETENTION = _environ_get_and_map('CHANGELOG_RETENTION', None, _AS_INT)

# Maximum number of days to retain job results (scripts and reports). Set to 0 to retain
job results in the database indefinitely. (Default: 90)
if 'JOB_RETENTION' in environ:
    JOB_RETENTION = _environ_get_and_map('JOB_RETENTION', None, _AS_INT)
# JOBRESULT_RETENTION was renamed to JOB_RETENTION in the v3.5.0 release of NetBox. For
backwards compatibility, map JOBRESULT_RETENTION to JOB_RETENTION
elif 'JOBRESULT_RETENTION' in environ:
    JOB_RETENTION = _environ_get_and_map('JOBRESULT_RETENTION', None, _AS_INT)

# API Cross-Origin Resource Sharing (CORS) settings. If CORS_ORIGIN_ALLOW_ALL is set to
True, all origins will be
```

```
# allowed. Otherwise, define a list of allowed origins using either CORS_ORIGIN_WHITELIST
or
# CORS_ORIGIN_REGEX_WHITELIST. For more information, see https://github.com/ottoyiu/django-
cors-headers
CORS_ORIGIN_ALLOW_ALL = _environ_get_and_map('CORS_ORIGIN_ALLOW_ALL', 'False', _AS_BOOL)
CORS_ORIGIN_WHITELIST = _environ_get_and_map('CORS_ORIGIN_WHITELIST', 'https://localhost',
_AS_LIST)
CORS_ORIGIN_REGEX_WHITELIST = [re.compile(r) for r in
_environ_get_and_map('CORS_ORIGIN_REGEX_WHITELIST', '', _AS_LIST)]

# Set to True to enable server debugging. WARNING: Debugging introduces a substantial
performance penalty and may reveal
# sensitive information about your installation. Only enable debugging while performing
testing.
# Never enable debugging on a production system.
DEBUG = _environ_get_and_map('DEBUG', 'False', _AS_BOOL)

# This parameter serves as a safeguard to prevent some potentially dangerous behavior,
# such as generating new database schema migrations.
# Set this to True only if you are actively developing the NetBox code base.
DEVELOPER = _environ_get_and_map('DEVELOPER', 'False', _AS_BOOL)

# Email settings
EMAIL = {
    'SERVER': environ.get('EMAIL_SERVER', 'localhost'),
    'PORT': _environ_get_and_map('EMAIL_PORT', 25, _AS_INT),
    'USERNAME': environ.get('EMAIL_USERNAME', ''),
    'PASSWORD': _read_secret('email_password', environ.get('EMAIL_PASSWORD', '')),
    'USE_SSL': _environ_get_and_map('EMAIL_USE_SSL', 'False', _AS_BOOL),
    'USE_TLS': _environ_get_and_map('EMAIL_USE_TLS', 'False', _AS_BOOL),
    'SSL_CERTFILE': environ.get('EMAIL_SSL_CERTFILE', ''),
    'SSL_KEYFILE': environ.get('EMAIL_SSL_KEYFILE', ''),
    'TIMEOUT': _environ_get_and_map('EMAIL_TIMEOUT', 10, _AS_INT),  # seconds
    'FROM_EMAIL': environ.get('EMAIL_FROM', ''),
}

# Enforcement of unique IP space can be toggled on a per-VRF basis. To enforce unique IP
space within the global table
# (all prefixes and IP addresses not assigned to a VRF), set ENFORCE_GLOBAL_UNIQUE to True.
if 'ENFORCE_GLOBAL_UNIQUE' in environ:
    ENFORCE_GLOBAL_UNIQUE = _environ_get_and_map('ENFORCE_GLOBAL_UNIQUE', None, _AS_BOOL)

# By default, netbox sends census reporting data using a single HTTP request each time a
worker starts.
# This data enables the project maintainers to estimate how many NetBox deployments exist
and track the adoption of new versions over time.
# The only data reported by this function are the NetBox version, Python version, and a
pseudorandom unique identifier.
# To opt out of census reporting, set CENSUS_REPORTING_ENABLED to False.
if 'CENSUS_REPORTING_ENABLED' in environ:
    CENSUS_REPORTING_ENABLED = _environ_get_and_map('CENSUS_REPORTING_ENABLED', None,
_AS_BOOL)
```

```
# Exempt certain models from the enforcement of view permissions. Models listed here will
be viewable by all users and
# by anonymous users. List models in the form `<app>.<model>`. Add '*' to this list to
exempt all models.
EXEMPT_VIEW_PERMISSIONS = _environ_get_and_map('EXEMPT_VIEW_PERMISSIONS', '', _AS_LIST)

# HTTP proxies NetBox should use when sending outbound HTTP requests (e.g. for webhooks).
HTTP_PROXIES = {
    'http': environ.get('HTTP_PROXY', None),
    'https': environ.get('HTTPS_PROXY', None),
}

# IP addresses recognized as internal to the system. The debugging toolbar will be
available only to clients accessing
# NetBox from an internal IP.
INTERNAL_IPS = _environ_get_and_map('INTERNAL_IPS', '127.0.0.1 ::1', _AS_LIST)

# Enable GraphQL API.
if 'GRAPHQL_ENABLED' in environ:
    GRAPHQL_ENABLED = _environ_get_and_map('GRAPHQL_ENABLED', None, _AS_BOOL)

# # Enable custom logging. Please see the Django documentation for detailed guidance on
configuring custom logs:
# #   https://docs.djangoproject.com/en/stable/topics/logging/
# LOGGING = {}

# Automatically reset the lifetime of a valid session upon each authenticated request.
Enables users to remain
# authenticated to NetBox indefinitely.
LOGIN_PERSISTENCE = _environ_get_and_map('LOGIN_PERSISTENCE', 'False', _AS_BOOL)

# When enabled, only authenticated users are permitted to access any part of NetBox.
# Disabling this will allow unauthenticated users to access most areas of NetBox (but not
make any changes).
LOGIN_REQUIRED = _environ_get_and_map('LOGIN_REQUIRED', 'True', _AS_BOOL)

# The length of time (in seconds) for which a user will remain logged into the web UI
before being prompted to
# re-authenticate. (Default: 1209600 [14 days])
LOGIN_TIMEOUT = _environ_get_and_map('LOGIN_TIMEOUT', 1209600, _AS_INT)

# Setting this to True will display a "maintenance mode" banner at the top of every page.
if 'MAINTENANCE_MODE' in environ:
    MAINTENANCE_MODE = _environ_get_and_map('MAINTENANCE_MODE', None, _AS_BOOL)

# Maps provider
if 'MAPS_URL' in environ:
    MAPS_URL = environ.get('MAPS_URL', None)

# An API consumer can request an arbitrary number of objects =by appending the "limit"
parameter to the URL (e.g.
# "?limit=1000"). This setting defines the maximum limit. Setting it to 0 or None will
allow an API consumer to request
```

```
# all objects by specifying "?limit=0".
if 'MAX_PAGE_SIZE' in environ:
    MAX_PAGE_SIZE = _environ_get_and_map('MAX_PAGE_SIZE', None, _AS_INT)

# The file path where uploaded media such as image attachments are stored. A trailing slash
is not needed. Note that
# the default value of this setting is derived from the installed location.
MEDIA_ROOT = environ.get('MEDIA_ROOT', join(_BASE_DIR, 'media'))

# Expose Prometheus monitoring metrics at the HTTP endpoint '/metrics'
METRICS_ENABLED = _environ_get_and_map('METRICS_ENABLED', 'False', _AS_BOOL)

# Determine how many objects to display per page within a list. (Default: 50)
if 'PAGINATE_COUNT' in environ:
    PAGINATE_COUNT = _environ_get_and_map('PAGINATE_COUNT', None, _AS_INT)

# # Enable installed plugins. Add the name of each plugin to the list.
# PLUGINS = []

# # Plugins configuration settings. These settings are used by various plugins that the
user may have installed.
# # Each key in the dictionary is the name of an installed plugin and its value is a
dictionary of settings.
# PLUGINS_CONFIG = {
# }

# When determining the primary IP address for a device, IPv6 is preferred over IPv4 by
default. Set this to True to
# prefer IPv4 instead.
if 'PREFER_IPV4' in environ:
    PREFER_IPV4 = _environ_get_and_map('PREFER_IPV4', None, _AS_BOOL)

# The default value for the amperage field when creating new power feeds.
if 'POWERFEED_DEFAULT_AMPERAGE' in environ:
    POWERFEED_DEFAULT_AMPERAGE = _environ_get_and_map('POWERFEED_DEFAULT_AMPERAGE', None,
_AS_INT)

# The default value (percentage) for the max_utilization field when creating new power
feeds.
if 'POWERFEED_DEFAULT_MAX_UTILIZATION' in environ:
    POWERFEED_DEFAULT_MAX_UTILIZATION =
_environ_get_and_map('POWERFEED_DEFAULT_MAX_UTILIZATION', None, _AS_INT)

# The default value for the voltage field when creating new power feeds.
if 'POWERFEED_DEFAULT_VOLTAGE' in environ:
    POWERFEED_DEFAULT_VOLTAGE = _environ_get_and_map('POWERFEED_DEFAULT_VOLTAGE', None,
_AS_INT)

# Rack elevation size defaults, in pixels. For best results, the ratio of width to height
should be roughly 10:1.
if 'RACK_ELEVATION_DEFAULT_UNIT_HEIGHT' in environ:
    RACK_ELEVATION_DEFAULT_UNIT_HEIGHT =
_environ_get_and_map('RACK_ELEVATION_DEFAULT_UNIT_HEIGHT', None, _AS_INT)
```

```
if 'RACK_ELEVATION_DEFAULT_UNIT_WIDTH' in environ:
    RACK_ELEVATION_DEFAULT_UNIT_WIDTH =
_environ_get_and_map('RACK_ELEVATION_DEFAULT_UNIT_WIDTH', None, _AS_INT)

# Remote authentication support
REMOTE_AUTH_AUTO_CREATE_GROUPS = _environ_get_and_map('REMOTE_AUTH_AUTO_CREATE_GROUPS',
'False', _AS_BOOL)
REMOTE_AUTH_AUTO_CREATE_USER = _environ_get_and_map('REMOTE_AUTH_AUTO_CREATE_USER',
'False', _AS_BOOL)
REMOTE_AUTH_BACKEND = _environ_get_and_map('REMOTE_AUTH_BACKEND',
'netbox.authentication.RemoteUserBackend', _AS_LIST)
REMOTE_AUTH_DEFAULT_GROUPS = _environ_get_and_map('REMOTE_AUTH_DEFAULT_GROUPS', '',
_AS_LIST)
# REMOTE_AUTH_DEFAULT_PERMISSIONS = {} # dicts can't be configured via environment
variables. See extra.py instead.
REMOTE_AUTH_ENABLED = _environ_get_and_map('REMOTE_AUTH_ENABLED', 'False', _AS_BOOL)
REMOTE_AUTH_GROUP_HEADER = _environ_get_and_map('REMOTE_AUTH_GROUP_HEADER',
'HTTP_REMOTE_USER_GROUP')
REMOTE_AUTH_GROUP_SEPARATOR = _environ_get_and_map('REMOTE_AUTH_GROUP_SEPARATOR', '|')
REMOTE_AUTH_GROUP_SYNC_ENABLED = _environ_get_and_map('REMOTE_AUTH_GROUP_SYNC_ENABLED',
'False', _AS_BOOL)
REMOTE_AUTH_HEADER = environ.get('REMOTE_AUTH_HEADER', 'HTTP_REMOTE_USER')
REMOTE_AUTH_USER_EMAIL = environ.get('REMOTE_AUTH_USER_EMAIL', 'HTTP_REMOTE_USER_EMAIL')
REMOTE_AUTH_USER_FIRST_NAME = environ.get('REMOTE_AUTH_USER_FIRST_NAME',
'HTTP_REMOTE_USER_FIRST_NAME')
REMOTE_AUTH_USER_LAST_NAME = environ.get('REMOTE_AUTH_USER_LAST_NAME',
'HTTP_REMOTE_USER_LAST_NAME')
REMOTE_AUTH_SUPERUSER_GROUPS = _environ_get_and_map('REMOTE_AUTH_SUPERUSER_GROUPS', '',
_AS_LIST)
REMOTE_AUTH_SUPERUSERS = _environ_get_and_map('REMOTE_AUTH_SUPERUSERS', '', _AS_LIST)
REMOTE_AUTH_STAFF_GROUPS = _environ_get_and_map('REMOTE_AUTH_STAFF_GROUPS', '', _AS_LIST)
REMOTE_AUTH_STAFF_USERS = _environ_get_and_map('REMOTE_AUTH_STAFF_USERS', '', _AS_LIST)

# This repository is used to check whether there is a new release of NetBox available. Set
to None to disable the
# version check or use the URL below to check for release in the official NetBox
repository.
RELEASE_CHECK_URL = environ.get('RELEASE_CHECK_URL', None)
# RELEASE_CHECK_URL = 'https://api.github.com/repos/netbox-community/netbox/releases'

# Maximum execution time for background tasks, in seconds.
RQ_DEFAULT_TIMEOUT = _environ_get_and_map('RQ_DEFAULT_TIMEOUT', 300, _AS_INT)

# The name to use for the csrf token cookie.
CSRF_COOKIE_NAME = environ.get('CSRF_COOKIE_NAME', 'csrftoken')

# Cross-Site-Request-Forgery-Attack settings. If Netbox is sitting behind a reverse proxy,
you might need to set the CSRF_TRUSTED_ORIGINS flag.
# Django 4.0 requires to specify the URL Scheme in this setting. An example environment
variable could be specified like:
# CSRF_TRUSTED_ORIGINS=https://demo.netbox.dev http://demo.netbox.dev
CSRF_TRUSTED_ORIGINS = _environ_get_and_map('CSRF_TRUSTED_ORIGINS', '', _AS_LIST)
```

```
# The name to use for the session cookie.
SESSION_COOKIE_NAME = environ.get('SESSION_COOKIE_NAME', 'sessionid')

# If true, the `includeSubDomains` directive will be included in the HTTP Strict Transport
Security (HSTS) header.
# This directive instructs the browser to apply the HSTS policy to all subdomains of the
current domain.
SECURE_HSTS_INCLUDE_SUBDOMAINS = _environ_get_and_map('SECURE_HSTS_INCLUDE_SUBDOMAINS',
'False', _AS_BOOL)

# If true, the `preload` directive will be included in the HTTP Strict Transport Security
(HSTS) header.
# This directive instructs the browser to preload the site in HTTPS. Browsers that use the
HSTS preload list will force the
# site to be accessed via HTTPS even if the user types HTTP in the address bar.
SECURE_HSTS_PRELOAD = _environ_get_and_map('SECURE_HSTS_PRELOAD', 'False', _AS_BOOL)

# If set to a non-zero integer value, the SecurityMiddleware sets the HTTP Strict Transport
Security (HSTS) header on all
# responses that do not already have it. This will instruct the browser that the website
must be accessed via HTTPS,
# blocking any HTTP request.
SECURE_HSTS_SECONDS = _environ_get_and_map('SECURE_HSTS_SECONDS', 0, _AS_INT)

# If true, all non-HTTPS requests will be automatically redirected to use HTTPS.
SECURE_SSL_REDIRECT = _environ_get_and_map('SECURE_SSL_REDIRECT', 'False', _AS_BOOL)

# By default, NetBox will store session data in the database. Alternatively, a file path
can be specified here to use
# local file storage instead. (This can be useful for enabling authentication on a standby
instance with read-only
# database access.) Note that the user as which NetBox runs must have read and write
permissions to this path.
SESSION_FILE_PATH = environ.get('SESSION_FILE_PATH', environ.get('SESSIONS_ROOT', None))

# Time zone (default: UTC)
TIME_ZONE = environ.get('TIME_ZONE', 'UTC')

# If true disables miscellaneous functionality which depends on access to the Internet.
ISOLATED_DEPLOYMENT = _environ_get_and_map('ISOLATED_DEPLOYMENT', 'False', _AS_BOOL)
```

8.5.2.3.6 Netbox\configuration\plugins.py

```
# Add your plugins and plugin settings here.
# Of course uncomment this file out.

# To learn how to build images with your required plugins
# See https://github.com/netbox-community/netbox-docker/wiki/Using-Netbox-Plugins

# PLUGINS = ["netbox_bgp"]

# PLUGINS_CONFIG = {
```

```
#   "netbox_bgp": {
#     ADD YOUR SETTINGS HERE
#   }
# }
PLUGINS = [
    "netbox_topology_views",
    "netbox_reorder_rack",
    "netbox_attachments",
]

PLUGINS_CONFIG = {
    'netbox_topology_views': {
        'static_image_directory': 'netbox_topology_views/img',
        'allow_coordinates_saving': True,
        'always_save_coordinates': True
    },
    "netbox_reorder_rack": {},
    "netbox_attachments": {},
}
```

8.5.2.3.7 Netbox\env\netbox.env

```
CORS_ORIGIN_ALLOW_ALL=True
DB_HOST=postgres
DB_NAME=netbox
DB_PASSWORD=PASS
DB_USER=netbox
EMAIL_FROM=netbox@bar.com
EMAIL_PASSWORD=
EMAIL_PORT=25
EMAIL_SERVER=localhost
EMAIL_SSL_CERTFILE=
EMAIL_SSL_KEYFILE=
EMAIL_TIMEOUT=5
EMAIL_USERNAME=netbox
# EMAIL_USE_SSL and EMAIL_USE_TLS are mutually exclusive, i.e. they can't both be `true`!
EMAIL_USE_SSL=false
EMAIL_USE_TLS=false
GRAPHQL_ENABLED=true
HOUSEKEEPING_INTERVAL=86400
MEDIA_ROOT=/opt/netbox/netbox/media
METRICS_ENABLED=false
REDIS_CACHE_DATABASE=1
REDIS_CACHE_HOST=redis-cache
REDIS_CACHE_INSECURE_SKIP_TLS_VERIFY=false
REDIS_CACHE_PASSWORD=PASS
REDIS_CACHE_SSL=false
REDIS_DATABASE=0
REDIS_HOST=redis
REDIS_INSECURE_SKIP_TLS_VERIFY=false
REDIS_PASSWORD=PASS
REDIS_SSL=false
```

```
RELEASE_CHECK_URL=https://api.github.com/repos/netbox-community/netbox/releases
SECRET_KEY=KEY'
SKIP_SUPERUSER=true
WEBHOOKS_ENABLED=true
```

8.5.2.3.8 Netbox\env\postgres.env

```
POSTGRES_DB=netbox
POSTGRES_PASSWORD=PASS
POSTGRES_USER=netbox
```

8.5.2.3.9 Netbox\env\redis-cache.env

```
REDIS_PASSWORD=PASS
```

8.5.2.3.10 Netbox\env\redis.env

```
REDIS_PASSWORD=PASS
```

8.5.2.3.11 Semaphore\docker-compose.yml

```yaml
services:
  postgres:
    restart: unless-stopped
    image: postgres:17
    hostname: postgres
    volumes:
     - ./data/semaphore-postgres:/var/lib/postgresql/data
    environment:
     POSTGRES_USER: semaphore
     POSTGRES_PASSWORD: semaphore
     POSTGRES_DB: semaphore
  semaphore:
    restart: unless-stopped
    ports:
      - 3000:3000
    image: semaphoreui/semaphore:latest
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      SEMAPHORE_DB_USER: semaphore
      SEMAPHORE_DB_PASS: semaphore
      SEMAPHORE_DB_HOST: postgres # for postgres, change to: postgres
      SEMAPHORE_DB_PORT: 5432 # change to 5432 for postgres
      SEMAPHORE_DB_DIALECT: postgres # for postgres, change to: postgres
      SEMAPHORE_DB: semaphore
      SEMAPHORE_PLAYBOOK_PATH: /tmp/semaphore/
      SEMAPHORE_ADMIN_PASSWORD: 'admin'
      SEMAPHORE_ADMIN_NAME: admin
      SEMAPHORE_ADMIN_EMAIL: admin@localhost
      SEMAPHORE_ADMIN: admin
      SEMAPHORE_ACCESS_KEY_ENCRYPTION: gs72mPntFATGJs9qK0pQ0rKtfidlexiMjYCH9gWKhTU=
      SEMAPHORE_LDAP_ACTIVATED: 'no' # if you wish to use ldap, set to: 'yes'
      SEMAPHORE_LDAP_HOST: dc01.local.example.com
      SEMAPHORE_LDAP_PORT: '636'
```

```
      SEMAPHORE_LDAP_NEEDTLS: 'yes'
      SEMAPHORE_LDAP_DN_BIND:
'uid=bind_user,cn=users,cn=accounts,dc=local,dc=shiftsystems,dc=net'
      SEMAPHORE_LDAP_PASSWORD: 'ldap_bind_account_password'
      SEMAPHORE_LDAP_DN_SEARCH: 'dc=local,dc=example,dc=com'
      SEMAPHORE_LDAP_SEARCH_FILTER:
"(\u0026(uid=%s)(memberOf=cn=ipausers,cn=groups,cn=accounts,dc=local,dc=example,dc=com))"
      TZ: UTC
    depends_on:
      - postgres # for postgres, change to: postgres
    volumes:
      - ./requirements.txt:/etc/semaphore/requirements.txt
      - ./data/playbooks:/playbooks
```

8.5.2.3.12 Semaphore\requirements.txt

```
paramiko
pytz
```

8.5.2.4 Start-up

In the root folder **netbox/**:

```
docker compose build
docker compose up -d
docker compose ps
docker compose logs -f netbox
```

After start-up, the web interface is available at the URL (according to the override port):

```
http://<host_IP_or_hostname>:8000/
```

Since SKIP_SUPERUSER=true is set in the PoC, the first administrator account is created explicitly:

```
docker compose exec netbox /opt/netbox/venv/bin/python /opt/netbox/netbox/manage.py
createsuperuser
```

Functionality check:

- the login page loads in the browser,
- docker compose ps shows the services as healthy (especially netbox).

8.5.3 Installation – OpenStack variant

In this variant, NetBox is deployed as a virtual server (VM) in the IaaS environment of the OpenStack cloud infrastructure. OpenStack provides the virtualization layer, including compute resources, storage and network connectivity, while the NetBox application logic itself is run in containers using Docker directly inside the virtual machine.

The virtual machine serves as an isolated application environment in which Docker Engine and Docker Compose are installed. All components of the NetBox stack (the application service, the PostgreSQL database, the Redis layers and auxiliary processes) are then deployed identically to the Docker variant, which ensures consistency of the configuration across environments. This approach enables the solution to be moved easily between the test and production environments without the need for substantial changes to the application configuration.

Deploying NetBox as a VM in OpenStack is particularly suitable in environments where cloud infrastructure is the standard and where integration with existing mechanisms for virtual machine management, network segmentation and security policies at the IaaS level is required.

8.5.4 Basic configuration

8.5.4.1 Tool configuration

The basic configuration of NetBox within the PoC focused on creating a stable and consistent environment for documenting assets relevant to the SOC and on preparing the system for further integration and automation scenarios. The configuration is implemented through a combination of environment variables and application settings defined in configuration files, which ensures that the configuration is separable from the application itself.

At the application level, NetBox is configured to use the PostgreSQL database system as persistent data storage and two separate Redis backends – one for processing background tasks (tasks) and the other for the application cache. This separation corresponds to the recommended NetBox architecture and increases the stability and predictability of the system's behaviour even under higher load.

Within the PoC, interfaces and functions supporting integrations are enabled, in particular:

- REST API as the primary interface for data access,
- GraphQL API for more flexible querying of the object model,
- Webhooks as a mechanism for notifications about changes in the documented assets.

From the configuration perspective, emphasis is placed on NetBox acting as the authoritative single source of truth (SSOT), with all changes to the infrastructure documentation being made centrally and subsequently made available to other systems through API interfaces.

8.5.4.2 Outputs / Outputs

The main output of the basic NetBox configuration is a structured asset database, which contains:

- records of network devices and their roles,
- physical location (sites, rooms, racks),
- IP address spaces, prefixes and IP address assignments,
- basic topological relationships between devices.

These outputs serve as a reference context for SOC activities, in particular for mapping security events to specific assets, rapid identification of affected segments and support for

incident analysis. In this model, NetBox provides neither detection nor response, but creates the data foundation without which these processes would be ambiguous or ineffective.

8.5.4.3 Extending NetBox functionality with plugins (why and for what)

In the PoC deployment, NetBox is extended with selected plugins so that the system is more usable for practical documentation of assets and their operational context in the SOC. The intention is not to “add extra features”, but to supplement missing elements that increase the quality of the inventory and its usability for analysis and operations (topology, physical arrangement, attachments to assets).

The plugins are included in the deployment in two steps:

1. Installation of the packages into a custom image via plugin_requirements.txt (build according to the Dockerfile).
2. Activation and configuration in plugins.py (PLUGINS and PLUGINS_CONFIG).

The following plugins are used in the PoC, with their benefits:

- **Topology Views (netbox-topology-views):** Used to visualize the network topology and interconnections. In the PoC, coordinate saving is enabled (allow_coordinates_saving and always_save_coordinates), which makes it possible to maintain a consistent topology layout. The benefit for the SOC is mainly faster orientation during an incident (visual mapping of the affected segment and of the links between devices).
- **Reorder Rack (netbox-reorder-rack):** Enables practical management and visual arrangement of devices in a rack (drag&drop). The benefit is more accurate physical documentation, which facilitates interventions and escalation in physical incidents (e.g. identifying a device in a distribution cabinet).
- **Attachments (netbox-attachments)**: Enables files to be attached to objects (e.g. diagrams, configurations, operational documents, runbooks). For the SOC, this is beneficial because the “asset context” (instructions, exceptions, diagrams) is stored directly with the object and is available during triage and response.

8.5.5 Integrations

8.5.5.1 API and integration mechanisms

Within the PoC, NetBox is integrated into the SOC environment through standardized interfaces that enable it to be used as a backend system for other tools and processes.

The basic integration mechanism is the REST API, which exposes all key NetBox objects (devices, interfaces, IP addresses, sites, relationships). This interface is intended for:

- automation tools,
- scripts and integration components,
- external systems requiring up-to-date asset context.

A complementary interface is the GraphQL API, which enables more efficient and more targeted queries over the data model, which is particularly suitable for more complex integration scenarios or analytical tasks.

For event-driven integrations, Webhooks are enabled, which make it possible to generate notifications upon changes in the documented objects (e.g. a change in IP addressing, addition of a device). This mechanism creates the foundation for future interconnection of NetBox with automated response or synchronization processes.

From the access control perspective, NetBox is prepared for role-based access control (RBAC), which makes it possible to restrict access to objects and functions according to user roles. In the PoC, RBAC is regarded as an essential prerequisite for the secure use of the documentation system in the SOC environment, especially when multiple teams or users with different permissions are involved.

The result of this phase of the PoC is a functional deployment of NetBox as the documentation and integration core for the SOC, ready to be connected to detection, monitoring and automation tools. In this phase, the integrations are used mainly for reading data and synchronizing context, which corresponds to the role of NetBox as a supporting, not an active, security tool.

8.5.5.2 Semaphore integration (PoC) – token → Semaphore → playbook → Git

1. Generating an API token in NetBox
   In the NetBox UI, create a token for the user (API Token) and copy it.
2. Inserting the token into Semaphore
   In the Semaphore UI, create:
3. an Inventory/Environment or a variable (e.g. NETBOX_TOKEN) and insert the token there,
4. optionally also set the NetBox URL (e.g. NETBOX_URL=http://netbox:8080 or an external URL depending on the network).
5. Running an automation task (playbook)
   In Semaphore, set up the project so that it uses the playbooks stored in the attached repository/volume (./data/playbooks:/playbooks in compose). Then run a job (template) that:
6. loads the inventory from NetBox (via the API),
7. performs the required use case (e.g. backup of configurations).
8. Output to Git (versioning)
   The output of the experiment is stored in a Git repository (configurations/artefacts), thereby creating an auditable history of changes. In the PoC, Git thus serves as the repository and versioning mechanism for the results of the automation task.

8.5.6 Integration of NetBox with SOC components

8.5.6.1 Integration with SIEM (ELK Stack)

NetBox provides SIEM tools with context for detected events:

- mapping an IP address to a specific device,
- identification of the site, segment or VLAN,
- determination of the device role (core, access, firewall, server).

In practice, this means that a security event from the ELK Stack can be correctly interpreted in the context of the infrastructure (*"which device, where, what significance"*), which significantly reduces ambiguity during incident triage.

8.5.6.2 Integration with SOAR (Shuffle)

NetBox is used in the enrichment phase of automated playbooks:

- Shuffle queries the NetBox API for information about an IP address or a device,
- the context obtained influences the decision logic of the playbook (priority, escalation, response).

NetBox thus supports automation without itself performing any active interventions.

8.5.6.3 Integration with CTI platforms (OpenCTI / MISP)

In combination with CTI tools, NetBox provides the **infrastructure context of a threat**:

- CTI answers the question *"what kind of threat is it"*,
- NetBox answers the question *"where is it located in the infrastructure"*.

This combination is key to correctly assessing the impact of an incident.

8.5.7 NetBox versus GLPI

NetBox and GLPI are two different, yet partially overlapping tools intended for asset records and support of operational processes in an IT environment. Both solutions are capable of fulfilling the role of a central asset register in the SOC environment; however, each of them approaches this role from a different perspective and with a different primary focus.

From a functional point of view, GLPI can be regarded as a process-oriented alternative to NetBox, since:

- both tools enable IT asset records to be kept,
- they support links of assets to owners and the organizational context,
- they provide API interfaces for integration with external systems,
- they can serve as a source of context for handling security incidents in the SOC.

Within the project, both tools – NetBox and GLPI – were tested in order to verify their suitability for the role of asset records in the proposed SOC architecture. The testing confirmed that both solutions are usable as a supporting system for the SOC, but with a differing degree of accuracy and technical detail of the data provided.

At the same time, it became apparent that NetBox and GLPI naturally complement each other, with each tool being stronger in a different area:

- NetBox is more suitable for technical and infrastructure documentation, detailed modelling of the network, IP addressing, topology and relationships between devices. It provides precise technical context, which is key to the correct interpretation of security events and to supporting automation.
- GLPI is more suitable for process and organizational records, incident management, ticketing, CMDB and links of assets to services, users and responsibilities. It supports

auditability and controlled incident handling, but with less emphasis on detailed network topology.

In a combined deployment, NetBox can fulfil the role of the technical single source of truth (SSOT) for the infrastructure, while GLPI functions as the process and record-keeping layer for incident and IT service management. Such a model makes it possible to divide responsibilities between the tools without unnecessary data duplication and increases the clarity of the SOC architecture.

From the perspective of SOC infrastructure design, it is therefore possible to:

- use only NetBox as the primary source of technical context (in a technically oriented SOC),
- use only GLPI in environments with an emphasis on ITSM processes,
- or combine them, with NetBox ensuring technical accuracy and GLPI the process-based management of incidents.

This approach makes it possible to scale asset records and the process capabilities of the SOC according to the needs of the organization, with NetBox providing a detailed infrastructure view and GLPI ensuring formal management and auditability of incident handling.

9 DFIR IRIS – TICKETING

Tool type: Ticketing

Tool name: DFIR IRIS

Version: v2.4.20

9.1 System description

DFIR-IRIS (Digital Forensics and Incident Response – Incident Response Investigation System) is an open-source platform designed for recording, managing and documenting cybersecurity incidents. Although it is not a classic helpdesk tool, from the SOC perspective it fulfils the role of a ticketing system by enabling centralized management of incidents, tasks and decisions throughout the entire incident life cycle.

DFIR-IRIS is designed primarily for the needs of Incident Response and digital forensic analysis. The basic object of the system is not a ticket, but an incident case (Case), which brings together all information related to the handling of the incident. This approach eliminates the data fragmentation typical of traditional ticketing systems and provides a comprehensive view of the incident in a single interface.

The purpose of DFIR-IRIS in the area of ticketing is to provide the SOC team with a tool for the systematic recording and management of security incidents, with an emphasis on technical context and documentation of the investigation. The tool is particularly suitable for SOC and CSIRT teams that need to link ticketing processes with technical analysis, work with IOCs, evidence and incident reporting.

Management of SOC tickets and tasks (triage, escalations, SLA)

The ticketing functionality of DFIR-IRIS is implemented through incident cases and tasks (Tasks). Each incident case fulfils the role of the main “ticket”, within which triage, classification, escalation and resolution of the incident take place. The system makes it possible to assign an incident to specific analysts and to define its severity, priority and resolution status.

Tasks within a case serve to divide work among the members of the SOC team and replace classic sub-ticket mechanisms. Each task has its own status and responsible person and is recorded in the case timeline. Although DFIR-IRIS does not implement a native SLA module typical of ITSM tools, it supports tracking of time data and an audit trail, which makes it possible to evaluate compliance with internal processes and response times.

9.2 System documentation

Official documentation:

https://docs.dfir-iris.org/latest/

Official GitHub project: https://github.com/dfir-iris/iris-web

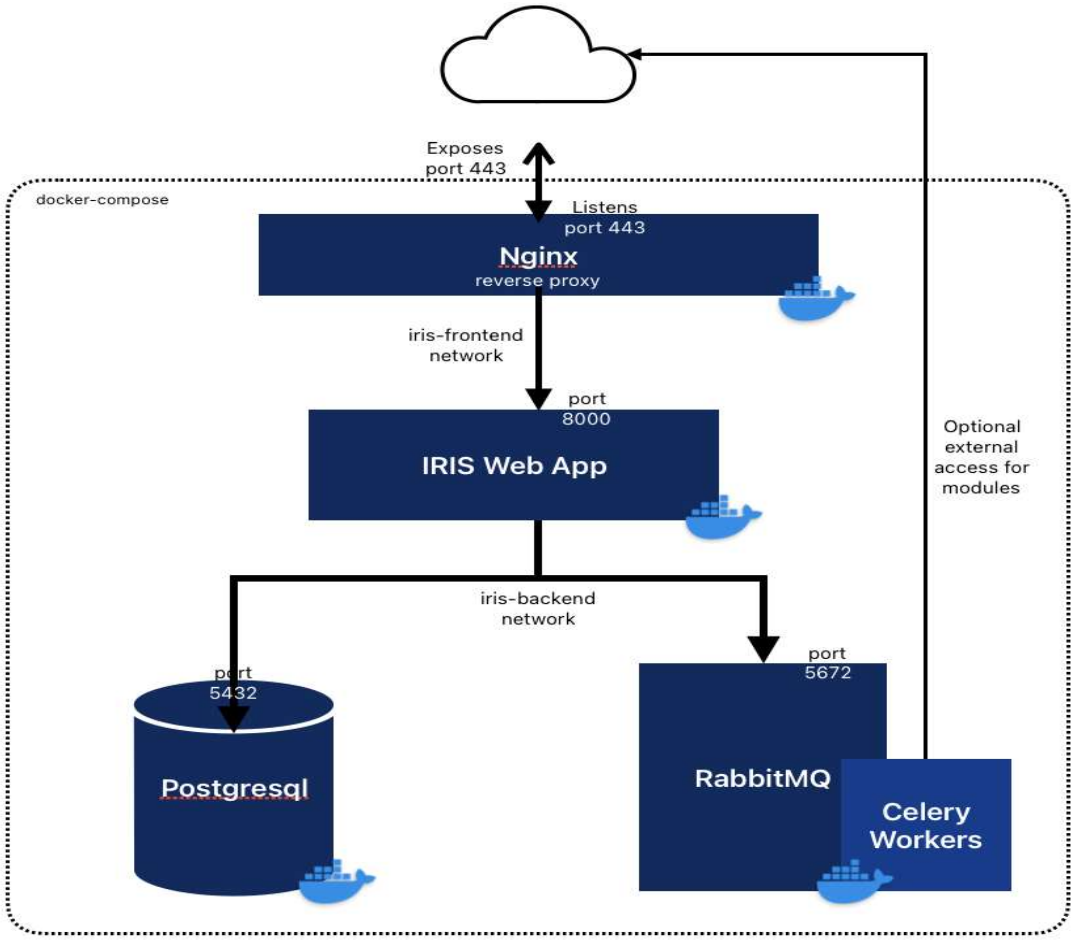
Hardware requirements

DFIR-IRIS has moderate hardware requirements. For a test or smaller production deployment, a virtual server with at least 2–4 CPU cores, 8 GB of RAM and approximately 40 GB of disk space is sufficient. For larger SOC environments with multiple analysts and long-term archiving of incidents, 4–8 CPU cores, 16 GB of RAM and fast SSD storage are recommended. The requirements grow mainly when working with attachments and evidence and when generating reports.

Architecture and diagrams

The DFIR-IRIS architecture is modular and consists of a web application, a database layer and asynchronous processing components. The web interface serves as the main point of user interaction with the system. The backend communicates with the PostgreSQL database, while time-consuming operations are processed asynchronously through a message queue (RabbitMQ) and workers. The architecture includes a REST API, which enables integration with other SOC components.

DFIR IRIS architecture diagram from the official website



9.3 Installation – Docker variant

1. System update

```
sudo apt update && sudo apt upgrade -y
```

2. Installing the required packages

```
sudo apt install -y ca-certificates curl gnupg lsb-release git
```

3. Adding the Docker GPG key

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

4. Adding the Docker repository

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
```

Funded by the European Union
NextGenerationEU
RECOVERY AND RESILIENCE PLAN
GOVERNMENT OFFICE OF THE SLOVAK REPUBLIC

```
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

5. Installing Docker Engine and Docker Compose

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

6. Enabling the Docker service

```
sudo systemctl enable docker
sudo systemctl start docker
```

7. Verifying Docker

```
docker --version
docker compose version
```

8. Downloading DFIR-IRIS

```
cd /opt
git clone https://github.com/dfir-iris/iris-web.git
cd iris-web
```

9. Starting DFIR-IRIS

```
docker compose up -d
```

10. Verifying the running containers

```
docker compose ps
```

11. Finding the admin password (first login)

```
docker compose logs app | grep create_safe_admin
```

12. Access to the web interface

```
https://<server_IP_address>
```

9.4 Installation – OpenStack variant

1. System update

```
sudo apt update && sudo apt upgrade -y
```

2. Installing system dependencies

```
sudo apt install -y \
       git curl build-essential \
```

```
        python3 python3-venv python3-pip \
        postgresql postgresql-contrib \
        rabbitmq-server \
        nodejs npm \
        nginx
```

Verifying versions:

```
python3 --version
        psql --version
        node --version
```

3. Setting up the PostgreSQL database

Starting the database

```
sudo systemctl enable postgresql
sudo systemctl start postgresql
```

Creating the database and user

```
sudo -u postgres psql

CREATE DATABASE iris;
CREATE USER irisuser WITH PASSWORD 'StrongPassword123';
GRANT ALL PRIVILEGES ON DATABASE iris TO irisuser;
ALTER DATABASE iris OWNER TO irisuser;
\q
```

4. Setting up RabbitMQ

```
sudo systemctl enable rabbitmq-server
sudo systemctl start rabbitmq-server
```

Creating a user:

```
sudo rabbitmqctl add_user iris irispassword
sudo rabbitmqctl add_vhost iris_vhost
sudo rabbitmqctl set_permissions -p iris_vhost iris ".*" ".*" ".*"
```

5. Downloading the DFIR-IRIS source code

```
cd /opt
        sudo git clone https://github.com/dfir-iris/iris-web.git
        sudo chown -R $USER:$USER iris-web
        cd iris-web
```

6. Backend – Python application

Creating a virtual environment

```
python3 -m venv venv
source venv/bin/activate
```

Installing Python packages

```
pip install --upgrade pip
pip install -r requirements.txt
```

7. Setting up the backend configuration

Creating configuration variables:

```
export IRIS_ENV=production
export DATABASE_URL=postgresql://irisuser:StrongPassword123@localhost/iris
export RABBITMQ_URL=amqp://iris:irispassword@localhost:5672/iris_vhost
export SECRET_KEY=$(openssl rand -hex 32)
```

Database initialization:

```
python manage.py db upgrade
python manage.py create_admin
```

8. Starting Celery workers

```
celery -A app.celery worker --loglevel=info &
```

9. Frontend – web interface

```
cd frontend
npm install
npm run build
```

10. Nginx – reverse proxy

Configuration

```
sudo nano /etc/nginx/sites-available/iris

server {
    listen 80;
    server_name _;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

Activating the configuration:

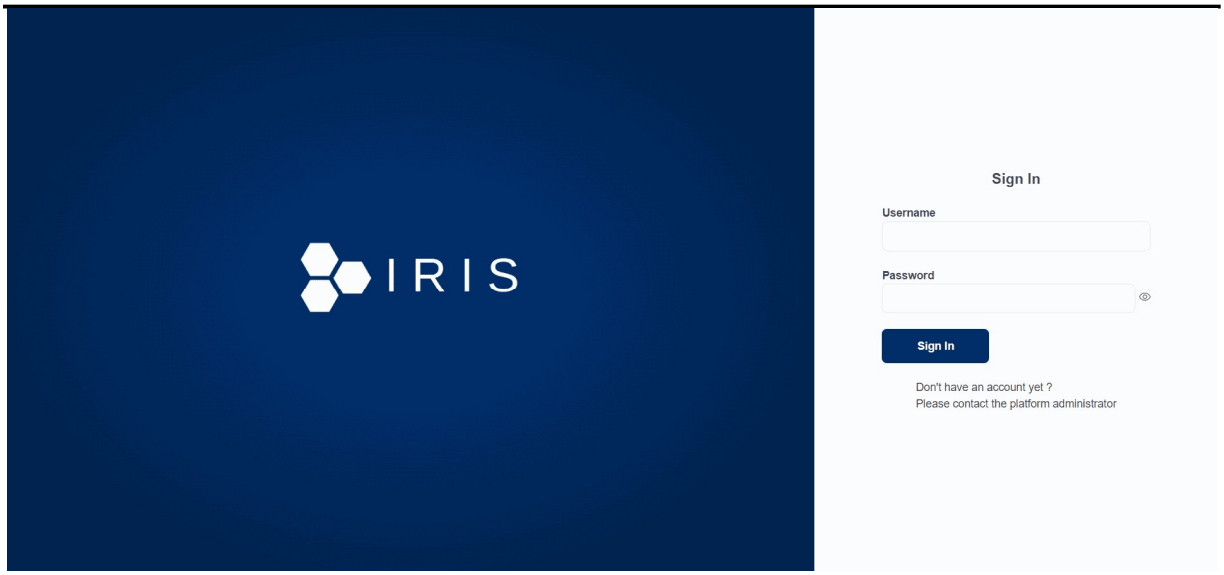
```
sudo ln -s /etc/nginx/sites-available/iris /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

11. Starting the DFIR-IRIS backend

```
cd /opt/iris-web
source venv/bin/activate
gunicorn -w 4 -b 127.0.0.1:8000 app:app
```

12. Accessing the application

```
http://<IP_address_VM>
```



9.5 Basic configuration

Tool configuration.

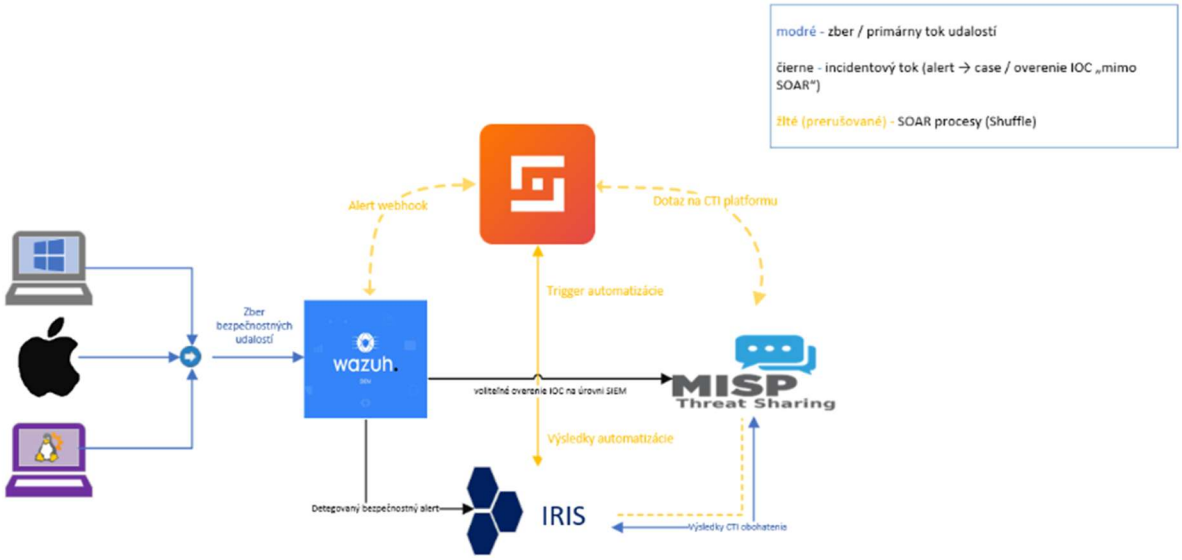
9.5.1 Inputs/Outputs

The inputs to DFIR-IRIS are security alerts, incident reports or manually created cases. The data may originate from SIEM systems, CTI platforms or from analysts. The outputs are processed incident cases, audit records, reports and structured data that can be exported or shared with other systems.

9.6 Integrations

DFIR-IRIS supports integration with tools used in the SOC environment, in particular with SIEM systems, CTI platforms and SOAR solutions. The integrations enable automated intake of alerts, enrichment of incidents with threat context and write-back of investigation results.

Proposed interconnection of components within the SOC environment



10 DFIR IRIS – INCIDENT RECORDS AND MANAGEMENT

Tool type: Incident records and management

Tool name: DFIR IRIS

Version: v2.4.20

10.1 System description

The DFIR-IRIS tool is designed as a centralized platform for recording and managing cybersecurity incidents throughout their entire life cycle. The system provides a unified environment in which all information related to the handling of an incident is concentrated, including its classification, the course of the investigation, the measures taken and the final evaluation.

The basic element of incident records in DFIR-IRIS is the incident case (Case), which represents a comprehensive record of a specific security incident. Each case contains basic identification data, such as the incident name, its description, category, severity and the assigned organization. In addition to this metadata, the system enables analysts to be assigned, tasks to be recorded, notes to be kept and the incident handling timeline to be created automatically.

DFIR-IRIS supports a systematic approach to incident management by enabling technical information, such as indicators of compromise, artefacts, evidence files and analysis results, to be added continuously. All steps performed are recorded in an auditable manner, which ensures the traceability of decisions and supports the needs of internal audits or subsequent reporting.

Compared with traditional ticketing systems, DFIR-IRIS provides a broader context of incident management, since it not only records the event itself but also supports the entire Incident Response process. This approach enables analysts to work with an incident as a single comprehensive object, thereby reducing information fragmentation and increasing the clarity of incident handling in the security operations centre environment.

10.2 System documentation

Official documentation:

https://docs.dfir-iris.org/latest/

Official GitHub project: https://github.com/dfir-iris/iris-web

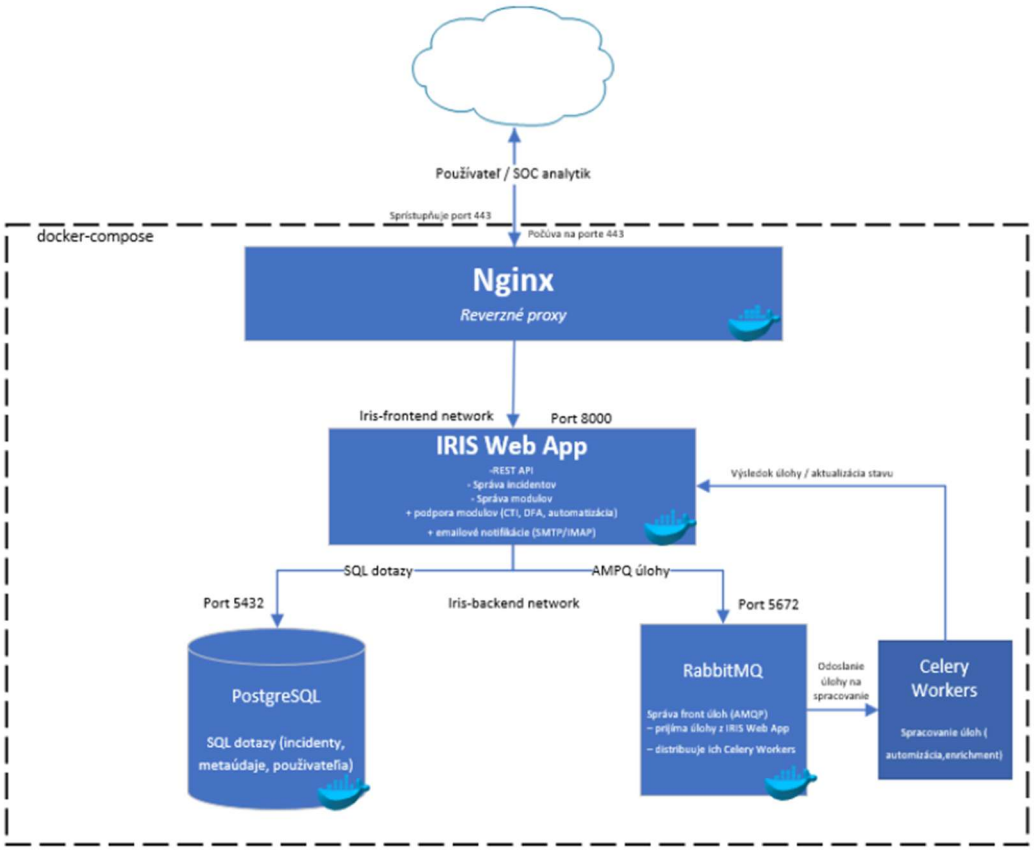
Hardware requirements

DFIR-IRIS has moderate hardware requirements. For a test or smaller production deployment, a virtual server with at least 2–4 CPU cores, 8 GB of RAM and approximately 40 GB of disk space is sufficient. For larger SOC environments with multiple analysts and long-term archiving of incidents, 4–8 CPU cores, 16 GB of RAM and fast SSD storage are recommended. The requirements grow mainly when working with attachments and evidence and when generating reports.

Architecture and diagrams

The DFIR-IRIS architecture is modular and consists of a web application, a database layer and asynchronous processing components. The web interface serves as the main point of user interaction with the system. The backend communicates with the PostgreSQL database, while time-consuming operations are processed asynchronously through a message queue (RabbitMQ) and workers. The architecture includes a REST API, which enables integration with other SOC components.

DFIR IRIS architecture diagram



10.3 Installation – Docker variant

1. System update

```
sudo apt update && sudo apt upgrade -y
```

2. Installing the required packages

```
sudo apt install -y ca-certificates curl gnupg lsb-release git
```

3. Adding the Docker GPG key

```
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

4. Adding the Docker repository

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

5. Installing Docker Engine and Docker Compose

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

6. Enabling the Docker service

```
sudo systemctl enable docker
sudo systemctl start docker
```

7. Verifying Docker

```
docker --version
docker compose version
```

8. Downloading DFIR-IRIS

```
cd /opt
git clone https://github.com/dfir-iris/iris-web.git
cd iris-web
```

9. Starting DFIR-IRIS

```
docker compose up -d
```

10. Verifying the running containers

```
docker compose ps
```

11. Finding the admin password (first login)

```
docker compose logs app | grep create_safe_admin
```

12. Access to the web interface

```
https://<server_IP_address>
```

10.4 Installation – OpenStack variant

1. System update

```
sudo apt update && sudo apt upgrade -y
```

2. Installing system dependencies

```
sudo apt install -y \
       git curl build-essential \
       python3 python3-venv python3-pip \
       postgresql postgresql-contrib \
       rabbitmq-server \
       nodejs npm \
       nginx
```

Verifying versions:

```
python3 --version
       psql --version
       node --version
```

3. Setting up the PostgreSQL database

Starting the database

```
sudo systemctl enable postgresql
sudo systemctl start postgresql
```

Creating the database and user

```
sudo -u postgres psql

CREATE DATABASE iris;
CREATE USER irisuser WITH PASSWORD 'StrongPassword123';
GRANT ALL PRIVILEGES ON DATABASE iris TO irisuser;
```

```
ALTER DATABASE iris OWNER TO irisuser;
\q
```

4. Setting up RabbitMQ

```
sudo systemctl enable rabbitmq-server
sudo systemctl start rabbitmq-server
```

Creating a user:

```
sudo rabbitmqctl add_user iris irispassword
sudo rabbitmqctl add_vhost iris_vhost
sudo rabbitmqctl set_permissions -p iris_vhost iris ".*" ".*" ".*"
```

5. Downloading the DFIR-IRIS source code

```
cd /opt
       sudo git clone https://github.com/dfir-iris/iris-web.git
       sudo chown -R $USER:$USER iris-web
       cd iris-web
```

6. Backend – Python application

Creating a virtual environment

```
python3 -m venv venv
source venv/bin/activate
```

Installing Python packages

```
pip install --upgrade pip
pip install -r requirements.txt
```

7. Setting up the backend configuration

Creating configuration variables:

```
export IRIS_ENV=production
export DATABASE_URL=postgresql://irisuser:StrongPassword123@localhost/iris
export RABBITMQ_URL=amqp://iris:irispassword@localhost:5672/iris_vhost
export SECRET_KEY=$(openssl rand -hex 32)
```

Database initialization:

```
python manage.py db upgrade
python manage.py create_admin
```

8. Starting Celery workers

```
celery -A app.celery worker --loglevel=info &
```

9. Frontend – web interface

```
cd frontend
npm install
npm run build
```

10. Nginx – reverse proxy

Configuration

```
sudo nano /etc/nginx/sites-available/iris

server {
    listen 80;
    server_name _;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

Activating the configuration:

```
sudo ln -s /etc/nginx/sites-available/iris /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

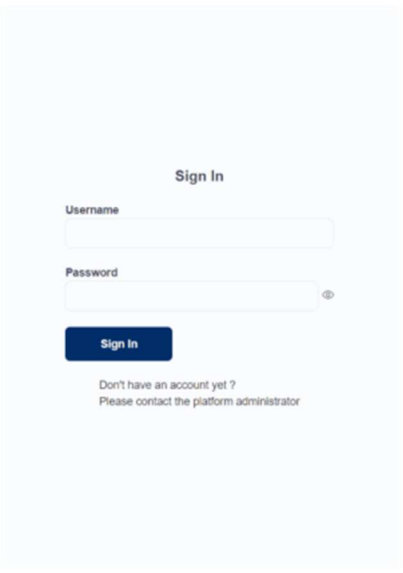
1. Starting the DFIR-IRIS backend

```
cd /opt/iris-web
source venv/bin/activate
gunicorn -w 4 -b 127.0.0.1:8000 app:app
```

1. Accessing the application

```
http://<IP address VM>
```





10.5 Basic configuration

A test of manually creating a new incident case was carried out in the web interface of the DFIR-IRIS tool. The system enables the entry of basic data about the incident, such as the incident name, its description, the assigned organization and the identifier of the external SOC ticket. The process of creating the incident case proceeded without technical problems, thereby verifying the basic functionality of the tool in the area of incident records.

During further work with the system, it was identified that the default configuration of the tool lacks incident templates for specific types of cybersecurity events. There were no pre-prepared templates, for example, for phishing attacks, malware incidents, compromise of user accounts or ransomware attacks. As a result, each incident would have to be created manually, without a uniform structure and without predefined handling steps.

In a real operational environment, such a way of working would lead to reduced efficiency of incident handling and to the absence of a consistent, standardized procedure within the SOC team. For this reason, it was necessary to verify the existence of available incident templates that could be added to the DFIR-IRIS tool.

During this verification, several community incident templates intended for older versions of the DFIR-IRIS tool were identified. However, subsequent tests of their import demonstrated incompatibility with the current version of the system, as DFIR-IRIS no longer supports the original template format based on a JSON structure.

Based on these findings, it was decided to modify the existing templates and adapt them to the current format of the tool so that they could be successfully imported and subsequently used for incident records and management.

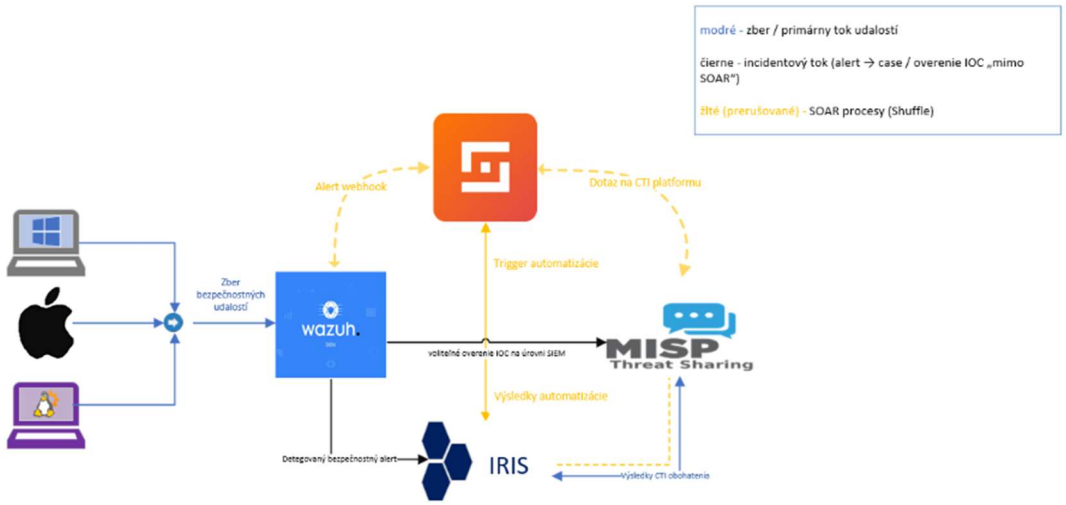
10.5.1 Inputs/Outputs

The inputs to DFIR-IRIS are security alerts, incident reports or manually created cases. The data may originate from SIEM systems, CTI platforms or from analysts. The outputs are processed incident cases, audit records, reports and structured data that can be exported or shared with other systems.

10.6 Integrations

DFIR-IRIS supports integration with tools used in the SOC environment, in particular with SIEM systems, CTI platforms and SOAR solutions. The integrations enable automated intake of alerts, enrichment of incidents with threat context and write-back of investigation results.

Proposed interconnection of components within the SOC environment



11 KIBANA (ELK)

Tool type: Visualization and analytical tool (SIEM front-end / Data Visualization)

Tool name: Kibana

Version: Kibana 8.x (open-source / Elastic Stack)

11.1 Tool description

Kibana is a web-based visualization and analytical tool that forms the presentation and analytical layer of the Elastic Stack. In the SOC environment, its main purpose is to make security and operational data accessible in the form of clear visualizations, dashboards and reports that support both the operational decision-making of analysts and management oversight.

The basic function of Kibana is to enable interactive work with data stored in Elasticsearch without the need for direct manipulation of the database. It provides users with tools for:

- visualization of security events and logs,
- analysis of trends and anomalies,
- correlation of events across sources,
- creation of custom dashboards for SOC analysts,
- generation of reports for management and audit.

In the proposed SOC architecture, Kibana fulfils the role of the central graphical interface for working with detection data, enabling rapid orientation in the security situation, monitoring of the infrastructure status and support for incident investigation.

From the management perspective, Kibana serves as a reporting and presentation tool that provides an aggregated view of security indicators (numbers of incidents, trends, threat types) without the need for technical knowledge of infrastructure details.

11.2 System documentation

The official documentation of the Kibana tool is part of the Elastic Stack documentation and provides a detailed description of the architecture, visualization options, work with data and integration with the other components of the stack. The documentation also covers security mechanisms, reporting and management of user rights.

Links to official sources:

- Official Kibana documentation: https://www.elastic.co/guide/en/kibana/current/index.html
- Official Elastic Stack website: https://www.elastic.co/elastic-stack
- GitHub repository: https://github.com/elastic/kibana
- Official Docker image: https://hub.docker.com/_/kibana
- Unofficial docker compose documentation: https://github.com/deviantony/docker-elk

11.3 HW requirements

The hardware requirements of Kibana depend on the volume of data processed, the number of users and the complexity of the dashboards. Since Kibana works exclusively on data stored in Elasticsearch, its requirements are lower than those of the indexing layer, but they are significantly affected by interactive use.

Minimum requirements (PoC / smaller SOC):

- CPU: 2 cores
- RAM: 2–4 GB
- Disk: minimal (no data persistence)

Recommended requirements (production SOC):

- CPU: 4–8 cores
- RAM: 8 GB or more
- Disk: SSD (logs and cache)

Higher requirements apply in the case of:

- a large number of simultaneously connected users,
- complex visualizations and correlations,
- generation of reports and long-term dashboards.

11.4 Tool architecture

The Kibana architecture is designed as a client-server web application that operates exclusively in close connection with Elasticsearch. Kibana does not store data on its own, but works with indices stored in Elasticsearch, over which it performs queries and visualizations.

The basis of the architecture consists of:

- the web application layer, which provides the user interface,
- the application logic, which processes queries, filtering and visualizations,
- the communication layer, which ensures interaction with the Elasticsearch API.

In the context of the SOC architecture, Kibana is positioned above the detection layer (Elasticsearch) and serves as the main interface for analysts and management. It makes it possible to monitor the current security status, analyse historical data and support decision-making without direct intervention in the data layer.

11.4.1 Diagram

The diagram shows the position of Kibana as the presentation and analytical layer above Elasticsearch within the SOC architecture. Kibana does not process data directly, but enables their visualization, analysis and reporting for the needs of SOC analysts and management on the basis of data indexed in Elasticsearch.

```
[ Event sources ]
  - servers (Linux / Windows)
```

```
  - network devices
  - applications
  - security systems
          |
          | (Beats / Logstash / Ingest pipelines)
          v
+-----------------------------------+
|          Elasticsearch            |
|                                   |
|  - log indices                    |
|  - security events                |
|  - metadata / time series         |
+-----------------------------------+
          |
          | (REST API / Query DSL)
          v
+-----------------------------------+
|              Kibana               |
|                                   |
|  +-----------------------------+  |
|  | Web UI                      |  |
|  | - dashboards (SOC)          |  |
|  | - visualizations            |  |
|  | - time-series analysis      |  |
|  | - reporting                 |  |
|  +-----------------------------+  |
|                                   |
|  +-----------------------------+  |
|  | Analytical logic            |  |
|  | - filters / correlations    |  |
|  | - alerting (rules)          |  |
|  | - saved searches            |  |
|  +-----------------------------+  |
+-----------------------------------+
          |
          | (visualizations / reports)
          v
+-----------------------------------+
| Users                             |
| - SOC analysts                    |
| - security operators              |
| - management                      |
+-----------------------------------+
```

11.5 Installation – Docker variant

```
services:
  kibana:
    build:
      context: kibana/
      args:
        ELASTIC_VERSION: ${ELASTIC_VERSION}
    volumes:
      - ./kibana/config/kibana.yml:/usr/share/kibana/config/kibana.yml:ro,Z
```

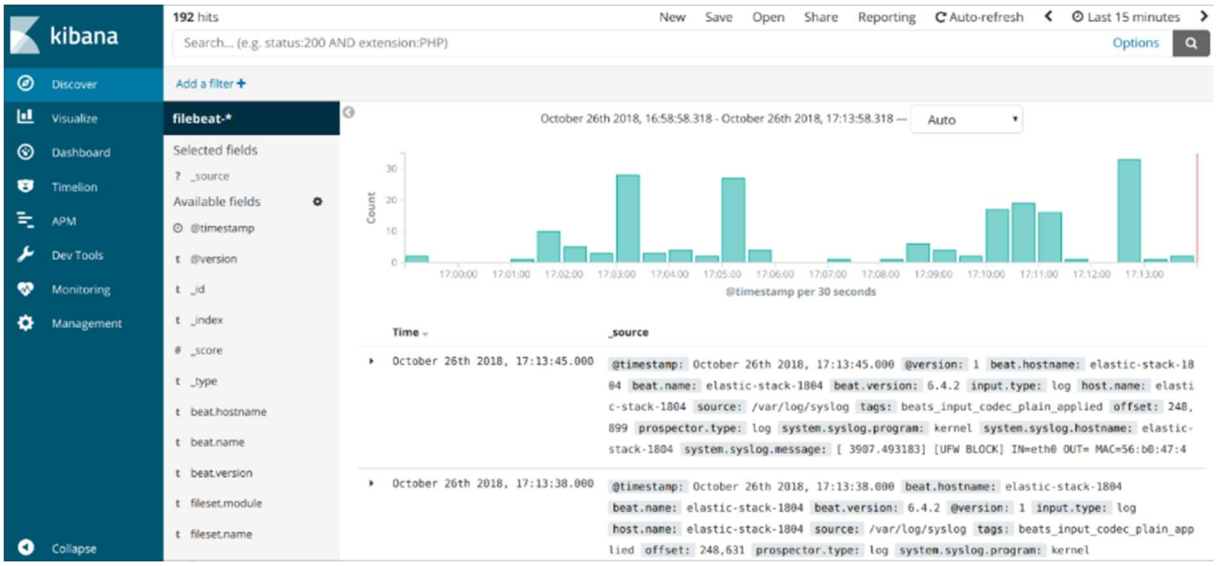
```
    ports:
      # - 5601:5601
      - 80:5601
    environment:
      KIBANA_SYSTEM_PASSWORD: ${KIBANA_SYSTEM_PASSWORD:-}
      SERVER_PUBLICBASEURL: http://elk.innosoc.brainit.tech
    networks:
      - elk
    depends_on:
      - elasticsearch
    restart: unless-stopped
networks:
  elk:
    driver: bridge
volumes:
  elasticsearch:
```

Changes in docker compose:

- This is Kibana itself, extracted from the complete ELK stack (1. ELK Stack)
- Some network ports changed – see the commented-out lines above.

11.6 Basic configuration

If it is part of the ELK Stack, it contains a configuration script that correctly sets the addresses and the necessary authentication between the services. Otherwise, it is only necessary to set the addresses and authentication manually according to the official documentation.



11.7 Integrations

Kibana is only a frontend, which as a rule is integrated mainly only with Elasticsearch. Elasticsearch then has further extensive options for further integration.

12 WIKI.JS

Tool type: Knowledge Management / Internal documentation

Tool name: Wiki.js

Version: Wiki.js 2.5 (open-source, containerized deployment)

12.1 System description

Wiki.js is an open-source tool designed for managing a knowledge base and internal documentation. In the SOC environment, it serves as a central knowledge repository that gathers operational procedures, runbooks, playbooks, internal guidelines and documentation of the systems used in securing the infrastructure.

The main purpose of deploying Wiki.js is:

- creation and management of SOC runbooks and operational procedures,
- documentation of security processes and playbooks,
- support for the onboarding of new analysts and operators,
- central records of internal systems, architectures and configurations.

Wiki.js makes it possible to store and organize knowledge in a manner that is sustainable in the long term and independent of specific individuals. The documentation is available through a web interface and supports versioning, structuring of content and access control.

In the proposed SOC architecture, Wiki.js does not act as a technical security tool, but as a knowledge and methodological layer that supports a consistent approach by analysts in incident handling and ensures that the organization's know-how is systematically recorded and available.

12.2 System documentation

The official documentation of the Wiki.js tool is publicly available and provides detailed guidance on installation, configuration, content management and integration with external systems. The documentation focuses mainly on the practical use of the tool as an internal knowledge base.

Links to official sources:

- Official documentation: https://docs.requarks.io/
- Official project website: https://js.wiki/
- GitHub repository: https://github.com/requarks/wiki
- Official Docker image: https://hub.docker.com/r/requarks/wiki
- Official integration options https://js.wiki/modules

12.3 Resource requirements

The hardware requirements of Wiki.js are low and allow it to be deployed even in smaller environments or as a supplementary component of the SOC infrastructure.

Minimum requirements (PoC / smaller environment):

- CPU: 1–2 cores
- RAM: 2–4 GB
- Disk space: 10–20 GB
- Database: PostgreSQL / MySQL / MariaDB

Recommended requirements (production environment):

- CPU: 2–4 cores
- RAM: 4–8 GB
- Disk: SSD (depending on the volume of documentation)

Higher requirements may arise with a large number of users or an extensive version history of the documentation.

12.4 Tool architecture

The Wiki.js architecture is designed as a lightweight multi-tier web application consisting of a presentation layer, an application layer and a data layer. This model enables simple deployment, maintenance and integration with other systems in the SOC environment.

The presentation layer consists of a web user interface that enables:

- browsing and editing of documentation,
- management of the knowledge base structure,
- searching the content,
- management of access rights.

The application layer ensures content processing, document versioning, the publishing workflow and user authentication. Wiki.js supports integration with external authentication systems (e.g. LDAP, OAuth, SSO), which is advantageous in an organized SOC environment.

The data layer consists of a relational database, in which the following are stored:

- the documents themselves,
- the versioned history of changes,
- user accounts and permissions,
- documentation metadata.

In the context of the SOC architecture, Wiki.js is positioned outside the detection and response layer and serves as a supporting system that ensures that procedures, decisions and operational knowledge are uniformly documented and available to all members of the SOC team.

12.4.1 Diagram

The diagram shows the position of Wiki.js as the knowledge and documentation layer in the SOC architecture. Wiki.js centralizes runbooks, operational procedures and playbooks,

which are used by analysts during incident handling. Through links and references, it interconnects the process, technical and analytical layers of the SOC without direct intervention in the detection or response systems.

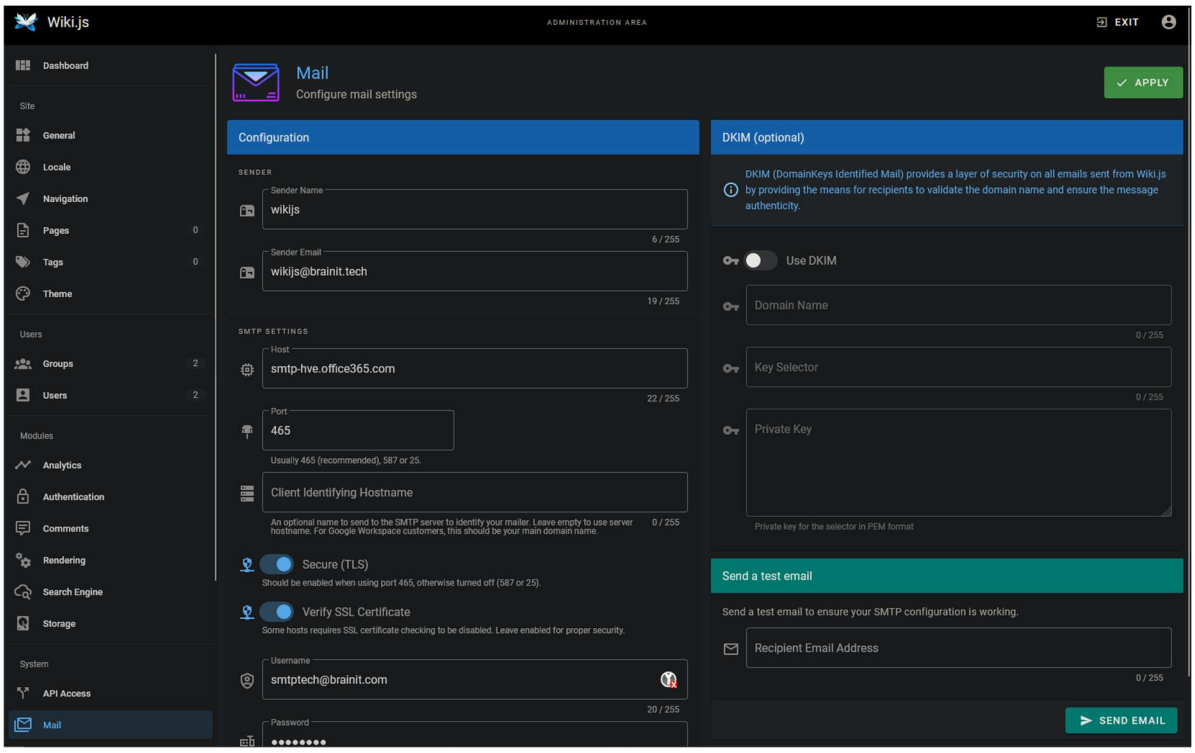
```
[ SOC USERS ]
 (Analysts, Management, Guests)
             |
             | HTTPS (TLS) + SSO / MFA
             v
+-------------------------------------------------------------+
|                          Wiki.js                            |
|                                                             |
|  +-------------------------------------------------------+  |
|  |            WEB UI (Rendering Engine)                  |  |
|  |  - Markdown / Visual Editor (runbooks, playbooks)     |  |
|  |  - Diagrams and charts (Mermaid.js)                     |  |
|  +---------------------------+---------------------------+  |
|                              |                              |
|  +---------------------------v---------------------------+  |
|  |              APPLICATION LOGIC (Node.js)              |  |
|  |  [AUTH]   - RBAC, LDAP / OIDC, MFA                    |  |
|  |  [SEARCH] - Internal DB search                        |  |
|  |             (optionally external search backend)      |  |
|  |  [API]    - REST / GraphQL API                        |  |
|  +---------------------------+---------------------------+  |
|               |               |               |              |
|  +------------v-----------+   |   +-----------v-----------+  |
|  |  DATABASE (Metadata)   |   |   |  STORAGE (Content)    |  |
|  |  - PostgreSQL          |   |   |  - Local file system  |  |
|  |  - Users, roles        |   |   |  - Git sync (versions)|  |
|  |  - Tags, revisions     |   |   |  - Attachments, images|  |
|  +------------------------+   |   +-----------------------+  |
+-------------------------------+-------------------------------+
                                |
        +-----------------------v-----------------------------+
        |            EXTERNAL CONNECTIVITY / INTEGRATIONS     |
        |  - Git repository (audit, DR, versioning)           |
        |  - Webhooks (Slack / Teams / notifications)         |
        |  - References to SOC systems (ELK, NetBox, GLPI)    |
        +-----------------------------------------------------+
```

12.5 Installation – Docker variant

```
Docker-compose.yml
services:
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: wiki
      POSTGRES_PASSWORD: wikijsrocks
      POSTGRES_USER: wikijs
    logging:
      driver: none
    restart: unless-stopped
    volumes:
```

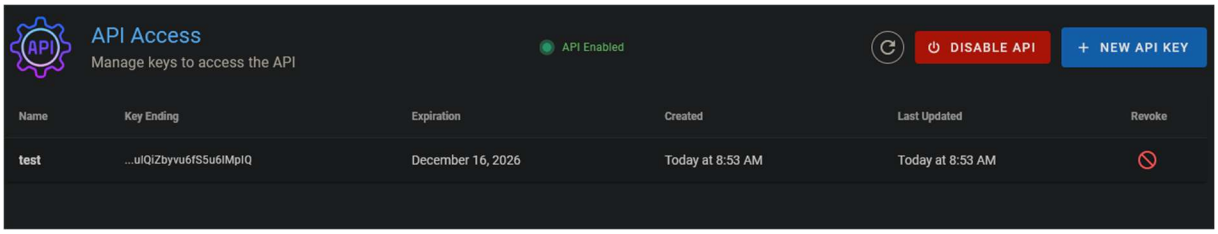
```
      - db-data:/var/lib/postgresql/data
  wiki:
    image: ghcr.io/requarks/wiki:2
    depends_on:
      - db
    environment:
      DB_TYPE: postgres
      DB_HOST: db
      DB_PORT: 5432
      DB_USER: wikijs
      DB_PASS: wikijsrocks
      DB_NAME: wiki
    restart: unless-stopped
    ports:
      - "80:3000"
volumes:
  db-data:
```

12.6 Basic configuration

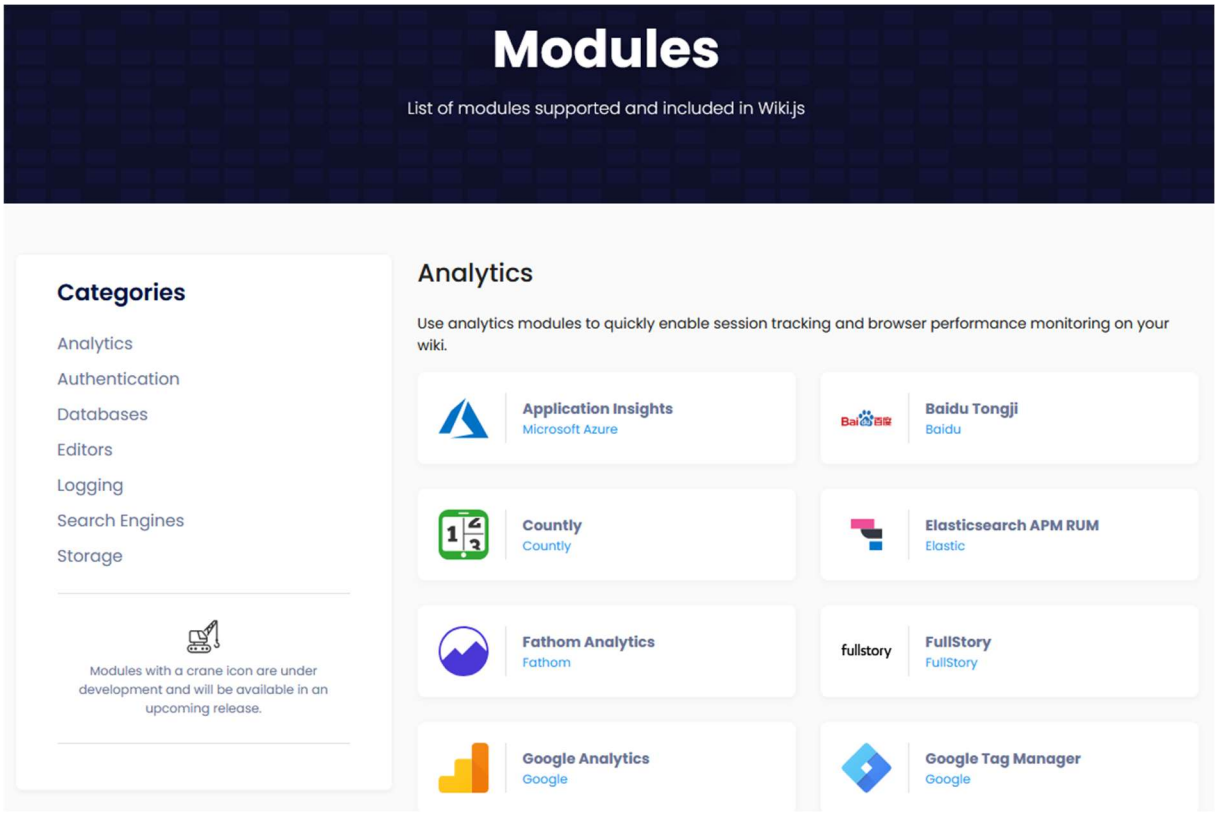


SMTP configuration

12.7 Integrations



Integration option via API keys



Other integration options available on the website: https://js.wiki/modules

13 DEPLOYMENT ENVIRONMENT

13.1 OpenStack KIS FRI UNIZA

13.1.1 Context of the CC solution

The deployment environment of the proposed solution is based on the private OpenStack cloud environment operated at the Department of Information Networks (KIS), Faculty of Management Science and Informatics, University of Žilina (FRI UNIZA). This environment has long been used to support teaching, research and the experimental deployment of information and network technologies. The KIS FRI UNIZA cloud is conceived as an IaaS (Infrastructure as a Service) platform that provides virtualized compute resources, network services and storage. The use of OpenStack enables the creation of isolated virtual environments that are suitable for testing complex architectures without affecting the production services of the faculty or the university.

From the project's perspective, the KIS environment is characterized as a controlled academic cloud environment that combines the properties of a production cloud with the flexibility of experimental deployment. This character enables the implementation of Proof of Concept (PoC) solutions, the verification of architectural designs and the testing of the integration of security tools in a realistic operational context. The KIS OpenStack cloud supports the creation of multiple virtual networks, security groups and separate tenants, which enables the logical separation of individual experiments and services. This property is key for the deployment of the SOC architecture, where it is necessary to separate the detection, analytical and response components while preserving the possibility of their mutual integration.

The deployment of the proposed solution in the KIS FRI UNIZA environment does not aim to create a fully fledged production security operations centre, but serves to verify the functionality, integration links and operational properties of the individual components of the SOC architecture. The environment is therefore regarded as a reference test and research environment that enables the replicability of experiments and an objective evaluation of the proposed solution.

13.1.2 OpenStack architecture

13.1.2.1 Description of the OpenStack architecture

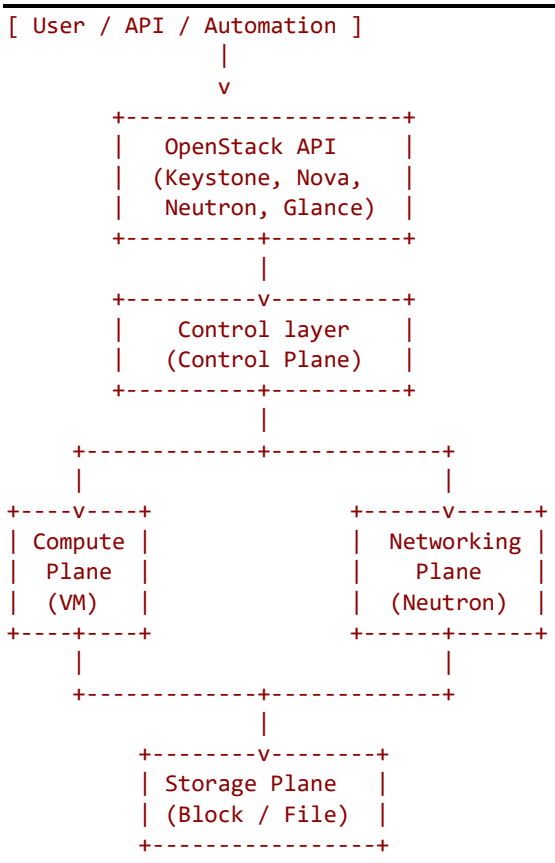
OpenStack is a modular open-source cloud platform of the Infrastructure as a Service (IaaS) type, designed for the management and orchestration of compute, network and storage resources in data centres and private cloud environments. The OpenStack architecture is based on a set of separate yet interconnected services that together form a unified cloud interface.

From an architectural point of view, OpenStack functions as a control and abstraction layer between the physical infrastructure and the application services. The user or higher-level systems (e.g. automation tools) do not work directly with the hardware, but with logical objects – virtual machines, networks, images and storage – which OpenStack dynamically maps to physical resources.

13.1.2.2 Basic architectural layers of OpenStack

From a logical point of view, the OpenStack architecture can be divided into the following layers:

- **Control layer (Control Plane):** Responsible for authentication, resource scheduling, API management and coordination of the individual services. This layer consists mainly of the Keystone, Nova (control), Neutron (control) and Glance services.
- **Compute layer (Compute Plane):** Comprises the compute nodes on which the virtual machines run. Nova ensures the scheduling of VMs and their mapping to physical resources.
- **Network layer (Networking Plane):** Implemented through the Neutron service, which enables the creation of virtual networks, subnets, routing and security rules. The network layer is key for tenant isolation and the secure interconnection of SOC components.
- **Storage layer (Storage Plane):** Provides both persistent and ephemeral storage for virtual machines and applications. OpenStack enables the integration of various types of storage backends according to the needs of the environment.



This diagram depicts OpenStack as an intermediary between the infrastructure layer and the SOC application services, which are run in virtual machines.

13.1.2.3 OpenStack documentation

OpenStack has extensive and detailed official documentation covering the system architecture, the individual services, deployment options, security aspects and recommended practices for operating cloud environments. The documentation is maintained by the community and reflects real production deployments in both academic and commercial environments.

Official documentation sources:

- Official OpenStack documentation: https://docs.openstack.org/
- OpenStack Architecture Design Guide: https://docs.openstack.org/arch-design/
- OpenStack Project Documentation (Git): https://opendev.org/openstack

These sources serve as a reference basis for the design of the cloud environment architecture used in this work and for the positioning of OpenStack as the infrastructure platform for the experimental SOC solution.

13.1.3 Cloud solution – OpenStack as a platform for INNOSOC

The architecture of the KIS FRI UNIZA cloud environment is based precisely on the OpenStack platform, which provides a modular set of services intended for building a private IaaS cloud solution. This chapter presents only those OpenStack components that are **functionally essential** for the deployment and operation of the experimental SOC solution and that are, at the same time, commonly used in OpenStack environments.

13.1.3.1 OpenStack architecture components used for CC INNOSOC

OpenStack consists of several independent services, each of which performs a precisely defined task. For the purposes of this solution, the following components are particularly relevant:

- **Nova (Compute):** Manages the life cycle of virtual machines. It enables the creation, starting, stopping and removal of VMs, which form the basic underlying environment for the individual SOC components (SIEM, SOAR, CTI, etc.).
- **Neutron (Networking):** Provides network connectivity between virtual machines and enables the creation of virtual networks, subnets and security rules. Neutron is key to the logical isolation of SOC components and the separation of test and experimental segments.
- **Glance (Image Service):** Is used to manage the operating system images from which virtual machines are created. The use of Glance enables repeatable deployment of identical VM templates, which is important for the replicability of PoC experiments.

These three services form a minimal and generally accepted basis of an OpenStack IaaS solution and are sufficient to carry out an experimental deployment of the SOC architecture. In addition to the core components, OpenStack also uses services that provide management, authentication and user access:

- **Keystone (Identity Service):** Is responsible for the authentication and authorization of users and services. It enables the management of tenants and access rights, which is

important for separating individual projects and experiments in an academic environment.

- **Horizon (Web Dashboard):** Provides a web interface for managing cloud resources. It serves as an administration interface for creating and managing virtual machines and networks without the need to work directly with the API.

These additional components enable secure and clear management of the cloud environment without interfering with the infrastructure itself.

13.1.3.2 The role of OpenStack in the INNOSOC SOC architecture

In the proposed solution, OpenStack does not act as a security tool and is not directly involved in detection, correlation or incident response. Its role is to provide an IaaS infrastructure platform that enables the individual SOC components to be deployed and operated as isolated services in virtual machines. In the INNOSOC environment, OpenStack thus plays the role of an abstracted infrastructure layer on top of which experimental architectures can be flexibly built and modified without interfering with physical resources and without affecting the faculty's other services.

From a practical point of view, OpenStack enables the creation of separate virtual environments (projects/tenants) in which SOC components are separated into their own instances. This supports the logical separation of the detection, analytical and response parts of the solution and at the same time allows their life cycle to be managed independently. In PoC mode, what matters most is the ability to quickly create and tear down test environments, adjust virtual machine parameters according to current needs (CPU, RAM, disk, network connections) and repeatedly deploy the same system templates, which increases the replicability of experiments.

Separating the infrastructure logic from the security architecture also has a methodological benefit: it allows the proposed SOC solution to be evaluated primarily through the integration links and operational properties of the individual tools (SIEM, SOAR, CTI, asset context, etc.), rather than through the specifics of particular hardware. OpenStack thus forms a stable operational base that provides controlled and repeatable conditions for the experimental deployment of the SOC architecture.

13.1.3.3 Automated deployment and management of the OpenStack platform

The KIS FRI UNIZA OpenStack cloud environment is built and managed with an emphasis on deployment automation and consistent infrastructure management. The fundamental aim of this approach is to minimize manual interventions in the platform, simplify updates of the individual layers and ensure repeatable deployment in line with the principles of Infrastructure as Code.

Automated deployment of the platform is carried out using Canonical tools, which are commonly used in the operation of OpenStack environments. The physical infrastructure is managed using MAAS (Metal as a Service), which enables automated deployment of operating systems on bare-metal servers via PXE booting. MAAS keeps an inventory of hardware resources and handles their initialization and preparation for the further layers of the cloud platform.

Above the physical layer, the Juju tool is used, which serves as the main orchestration mechanism of the OpenStack platform. Juju makes it possible to model individual services as

separate application entities and to manage their life cycle by means of so-called charms. Charms define how individual components are installed, configured, interconnected and updated, and are distributed via the Charmhub repository, which in this context plays a role similar to Docker Hub for containerized applications.

Thanks to this approach, OpenStack services can be deployed consistently across different types of environments, whether virtual machines, containers (LXD) or bare-metal servers directly. Automation also enables efficient management of platform updates and reduces the risk of inconsistent configurations between individual nodes.

In addition to the core services used in the PoC scenarios, further OpenStack components are deployed in the environment, which extend the functionality of the cloud platform and support more advanced operational scenarios.

The key deployed services include:

- **Nova** for managing compute resources,
- **Neutron** in combination with **OVN** for a software-defined network architecture,
- **Glance** for managing virtual machine images,
- **Horizon** as the web administration interface,
- **Keystone** for authentication and identity management,
- **Cinder** for block storage,
- **Heat** for orchestration and declarative deployment of resources,
- **Vault** for secure management of secrets and cryptographic material.

The storage layer is implemented by means of a **Ceph cluster**, which provides distributed and highly available storage for the needs of OpenStack services as well as application workloads. The integration of Ceph enables separation of the compute and storage layers, horizontal scaling of capacity and increased resilience to failures of individual nodes, which is particularly advantageous for data-intensive SOC components.

An automated deployment model designed in this way makes it possible to keep the cloud platform in a consistent state, simplifies the management of its life cycle and creates a stable basis for both experimental and longer-term deployment of the SOC architecture in the INNOSOC environment.

13.1.4 Infrastructure layer of the environment

The infrastructure layer of the KIS FRI UNIZA OpenStack environment represents the technical foundation of the university cloud on which teaching, research and experimental services are operated. In this work, the infrastructure is understood as a supporting platform for deploying and testing the components of the proposed SOC solution. It is therefore not a detailed description of the operational topology, but an explanation of the principles that are important for the stability, isolation and replicability of the PoC deployment.

The infrastructure is based on physical compute nodes, network devices and storage systems. On top of them run the virtualization layer and the OpenStack control layer, which handles resource allocation, management of virtual networks and access control. Within this layer, a multi-user (multi-tenant) environment with logical separation of projects can be created, which is suitable for parallel experiments and also for separating services with different security and availability requirements. From an architectural point of view, the infrastructure is designed to:

- enable the separation of compute, network and storage resources,
- support a multi-user (multi-tenant) environment,
- provide isolation of individual projects and experiments,
- enable flexible allocation of resources according to the needs of services.

For the purposes of this work, the infrastructure is used to run virtual machines in which the individual SOC components (SIEM, SOAR, CTI, supporting tools) are deployed. These components are divided into separate instances, which enables:

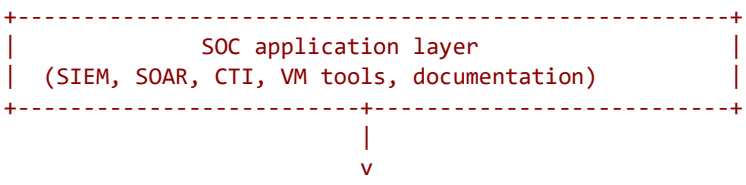
- isolation of individual functional blocks,
- separate management and recovery of services,
- experimentation with configuration without affecting other systems.

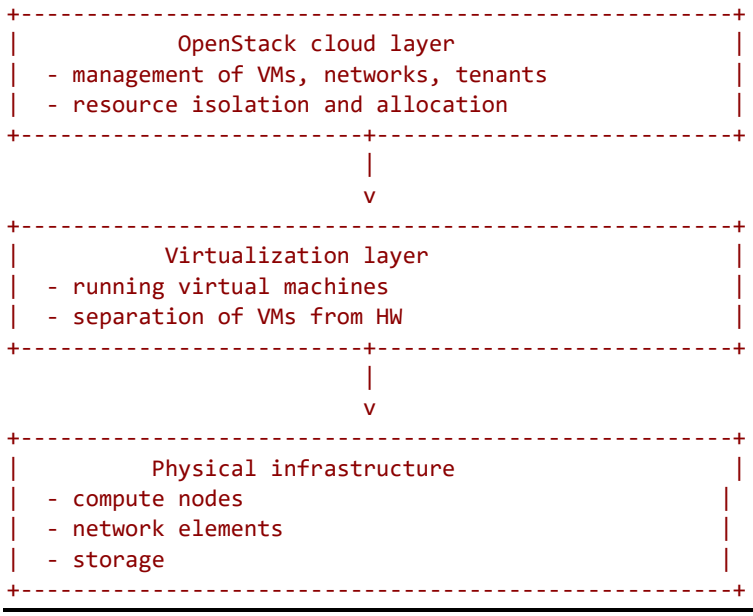
From an operational perspective, the infrastructure is designed as a long-term available and stable platform that enables continuous operation of cloud services while supporting configuration changes, capacity expansion and the testing of new solutions. The implementation details are chosen to be consistent with the common architectural principles of OpenStack environments used in academic institutions.

13.1.4.1 Architecture of the infrastructure layer

The architecture of the infrastructure layer is based on the standard reference model of OpenStack environments, in which the individual infrastructure layers and their responsibilities are clearly separated. As a multi-level model, the architecture of the infrastructure layer enables independent development of the infrastructure and of the services built on top of it. The basic layers of the architecture are:

- Physical layer – compute nodes, network switches and storage systems,
- Virtualization layer – runs virtual machines and ensures resource isolation,
- Cloud control layer (OpenStack) – manages resource allocation, network segments and access rights,
- Application layer – SOC components operated on top of OpenStack.





In terms of design, the infrastructure is dimensioned to support:

- separation of the control and data planes,
- scaling of compute nodes without interfering with the application layer,
- separate network segments for management, services and data traffic.

13.1.4.2 Network architecture and recommendations

Although the specific network topology and addressing plan are not the subject of this work, it is advisable to follow the basic principles of segmentation for the secure and stable functioning of both the cloud environment and the SOC components. From an architectural point of view, it is recommended to separate at least three types of traffic: management, tenant (virtual machine traffic) and storage communication. Such a division reduces the risk of lateral movement if one service is compromised, simplifies monitoring and allows security policies to be applied more precisely.

From an architectural point of view, the following is recommended:

- a dedicated VLAN for the management of OpenStack components,
- a separate VLAN for tenant traffic (VM networks),
- a separate VLAN for storage traffic, especially if distributed or network storage is used,
- restriction of access between VLANs by means of firewall rules.

Such a division increases security, simplifies monitoring and reduces the risk of lateral movement by an attacker if a service is compromised.

13.1.4.3 Storage architecture and recommendations

The storage layer of the infrastructure is particularly important for SOC components that process large volumes of data and require stable persistence, such as log stores, indexing databases or the outputs of vulnerability scanning. OpenStack makes it possible to abstract different types of storage into a unified interface, which simplifies management and allows the storage to be adapted to the type of service.

From an architectural point of view, it is advisable to distinguish between system storage for the operating systems of virtual machines, data storage for the persistent data of SOC tools, and temporary storage for processing and cache. Fast storage (ideally SSD) is preferred for database and indexing services, since the performance of the disk subsystem has a major impact on the speed of searching, correlation and report generation. At the same time, it is necessary to provide for regular backup of critical data and to restrict direct access to the storage network to only those components that actually need it

In OpenStack-based environments, the distributed storage system **Ceph** is often used as a suitable storage layer technology; it provides object, block and file storage within a single platform. Ceph is designed as a scalable and highly available solution that allows data to be distributed across multiple nodes, thereby increasing resilience to failures of individual components. In combination with OpenStack, Ceph is commonly used as a backend for virtual machine block storage or as data storage for applications with high capacity and availability requirements.

For SOC workloads such as SIEM systems or vulnerability scanners, the main advantage of Ceph is its capability for horizontal scaling and for separating compute and storage capacity. At the same time, however, performance characteristics must be taken into account – database and indexing services can benefit significantly from fast storage (e.g. SSD), whereas less critical or archival data can be placed on capacity-oriented storage.

In terms of recommendations, it is advisable to:

- separate the system and data disks of virtual machines,
- prefer fast storage for database and indexing services,
- ensure regular backup of critical data,
- restrict direct access to the storage network to authorized infrastructure components only.

A storage architecture designed in this way provides a balanced trade-off between performance, availability and flexibility, which is suitable for both experimental and longer-term PoC deployment of the SOC solution in an academic cloud environment.

13.1.5 Hardware components of the environment

The hardware components of the environment represent the physical foundation of the KIS FRI UNIZA OpenStack cloud infrastructure on which the experimental SOC solution is operated. This layer provides the computing power, network connectivity and storage capacity needed to run the virtualization and cloud control layers.

The infrastructure consists of a combination of controller and compute servers, which together form a flexible and scalable platform for cloud services. The controller servers run the control components of the virtualization and cloud layers, while the compute nodes are intended primarily for running virtual machines and application workloads, including SOC components.

The compute nodes form the main source of computing capacity of the cloud environment. They are equipped with multi-core processors, a large amount of main memory and local disk space, which is used for the system disks of virtual machines or as part of the storage architecture. Such a configuration enables multiple virtual machines with different performance requirements to run in parallel, which is essential for operating heterogeneous SOC components.

From the perspective of the experimental SOC solution, it is important that the computing capacity makes it possible to separate the individual functional blocks into separate virtual machines, for example SIEM, SOAR, CTI platforms, vulnerability scanners and supporting services. This approach increases the stability of the environment, facilitates management and allows individual components to be scaled or restarted independently without affecting the other parts of the solution.

The current functional HW equipment comprises:

Controller servers:

- 2 pcs Cisco UCS C220M3: 2x Xeon E5-2667v2 8c/16t, 128GB RAM, 500GB SSD, 600GB HDD, 2x 10GbE, VMWare ESXi, XCP-ng

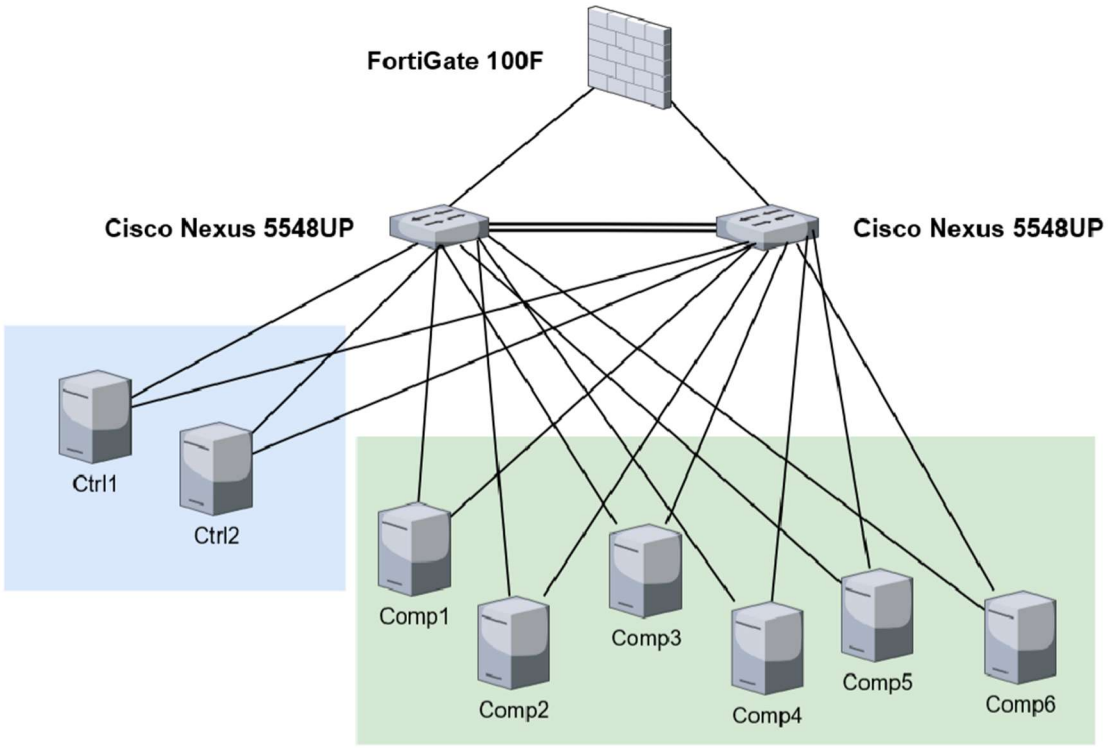
Compute servers:

- 6 pcs Lenovo x3650M5: 2x Xeon E5-2699v3 18c/36t, (2 servers 2x Xeon E5-2680v414c/28t), 512GB RAM, System: 300GB HDD, Storage: 4x 1.2TB HDD, 2x 10GbE

13.1.5.1 Network devices and infrastructure

The network infrastructure interconnects the compute nodes, controller servers and storage systems into a single functional whole. It is designed to support high throughput, low latency and traffic segmentation, which is essential for cloud and security workloads.

Support for technologies such as VLANs and high-speed network interfaces enables the logical separation of individual types of traffic, for example management, tenant traffic and storage communication. From an architectural point of view, this separation is key to security and operational stability, particularly in an environment where teaching, research and experimental services are operated in parallel.



13.1.5.2 Storage systems

The storage systems in the CC infrastructure provide persistent space for the operating systems of virtual machines as well as for application data. For SOC components, the storage layer is particularly important, since they process large volumes of data and often require long-term archiving.

The storage architecture is designed to enable the separation of system and data disks and to support the deployment of distributed storage solutions. Such an approach is compatible with technologies such as Ceph, which are commonly used in OpenStack environments and enable capacity scaling and increased resilience to failures. For performance-sensitive services, such as databases and the indexing mechanisms of SIEM tools, fast storage is preferred, whereas less critical data can be placed on capacity-oriented disks.

13.1.5.3 Summary of capacity for the PoC

From the Proof of Concept perspective, the KIS FRI UNIZA hardware infrastructure provides sufficient compute, network and storage capacity to verify the functionality of the proposed SOC architecture. The environment enables multiple SOC components to run in parallel, experimentation with their configuration and testing of integration links under conditions close to real operation.

Importantly, the infrastructure is not optimized for the maximum performance of any one particular tool, but for the flexibility and stability of the environment as a whole. This corresponds to the aims of the work, which are to verify the architectural design, the integration scenarios and the operational properties of the SOC solution in an academic cloud environment.

Funded by the European Union NextGenerationEU
RECOVERY AND RESILIENCE PLAN
GOVERNMENT OFFICE OF THE SLOVAK REPUBLIC

13.1.6 Placement and isolation of SOC components in the environment

The placement and isolation of the individual components of the SOC architecture represent a key aspect of the design of the experimental solution. In the KIS FRI UNIZA environment, the aim is to ensure that the individual tools can be deployed, configured and tested independently of one another, without adversely affecting each other and without affecting other cloud services. This approach also enables realistic verification of the SOC architecture under conditions close to real operation.

The fundamental design principle is the use of the **virtual machine as the primary unit of deployment**. Each main component of the SOC solution is operated in its own virtual instance, which achieves clear isolation at the level of the operating system, network interfaces and allocated resources. This model corresponds to common practice in production SOC environments, where critical security services are separated for reasons of stability, security and ease of management.

Virtual machines are created and managed via the OpenStack platform, which ensures tenant isolation, control of access rights and flexible allocation of compute and network resources. In PoC mode, this enables rapid creation of new instances, configuration changes or complete removal of the experimental environment without interfering with the physical infrastructure

13.1.6.1 Deployment model of SOC components (VMs as basic units)

In the proposed solution, the individual SOC components are divided into separate virtual machines according to their functional role. Typically, these are separate instances for detection tools (e.g. SIEM), response and automation tools (SOAR), threat intelligence (CTI) platforms, tools for managing assets and infrastructure context, as well as supporting services, for example knowledge bases or vulnerability scanners.

Such a division enables not only logical separation of roles but also independent dimensioning of resources according to the nature of the workload. For example, SIEM components may require more memory and disk performance, whereas SOAR or CTI platforms have lower computing power requirements but higher requirements for availability and integration with external systems.

13.1.6.2 Containerization within VMs (Docker/Compose) – where it makes sense and why

Inside individual virtual machines, containerization by means of Docker and Docker Compose is used in justified cases. This approach enables simpler deployment of complex applications, simpler updates and better control over the dependencies of individual components.

However, containerization is not used as a replacement for virtualization but as a complement to it. The virtual machine provides a security and isolation boundary between the individual SOC tools, while containers serve for efficient management of application components within a single service. This two-layer model (VM + Docker) is particularly suitable for experimental and PoC environments, where configurations need to be changed quickly and different deployment variants tested.

13.1.6.3 Logical separation of roles (detection / response / CTI / knowledge / vuln scanning)

Logical separation of roles is achieved through a combination of the virtualization layer, network segmentation and the application architecture. Detection tools, response mechanisms, threat intelligence platforms, asset management systems and knowledge databases are separated not only at the level of virtual machines but also by means of separate network segments and controlled integration interfaces.

Such an approach reduces the risk of lateral movement if one component is compromised and at the same time supports a clear delineation of the responsibilities of the individual tools. Each component acts in the SOC architecture as a separate service with precisely defined inputs and outputs, which simplifies integration, auditing and long-term maintenance of the solution.

13.1.6.4 PoC mode and experiments (tenant/network isolation, deployment replicability)

The deployment of the SOC architecture in the KIS FRI UNIZA environment has the character of a Proof of Concept, the aim of which is not to ensure continuous production operation, but to verify the functionality of the architectural design, the integration links and the operational properties of the individual tools. The way the environment is isolated and managed is adapted accordingly.

PoC mode enables parallel experiments, testing of different configuration scenarios and repeated deployment of the same architectural patterns. Thanks to the use of OpenStack, it is possible to create separate projects, separate virtual networks and isolated service instances, which supports the replicability of experiments and objective comparison of results.

The placement and isolation of SOC components designed in this way create a stable basis for the experimental verification of the proposed solution and at the same time provide an architectural framework that is also transferable to production cloud or on-premises environments.

13.2 Kubernetes Brainit

13.2.1 Description of the Kubernetes infrastructure

Kubernetes is an open-source platform designed for the automated deployment, scaling and management of containerized applications – which is also why in practice it is most often referred to as an application orchestrator. Unlike more traditional ways of deploying containers within a single compute server – most often using Docker – Kubernetes technology groups several such compute nodes under a single abstract plane.

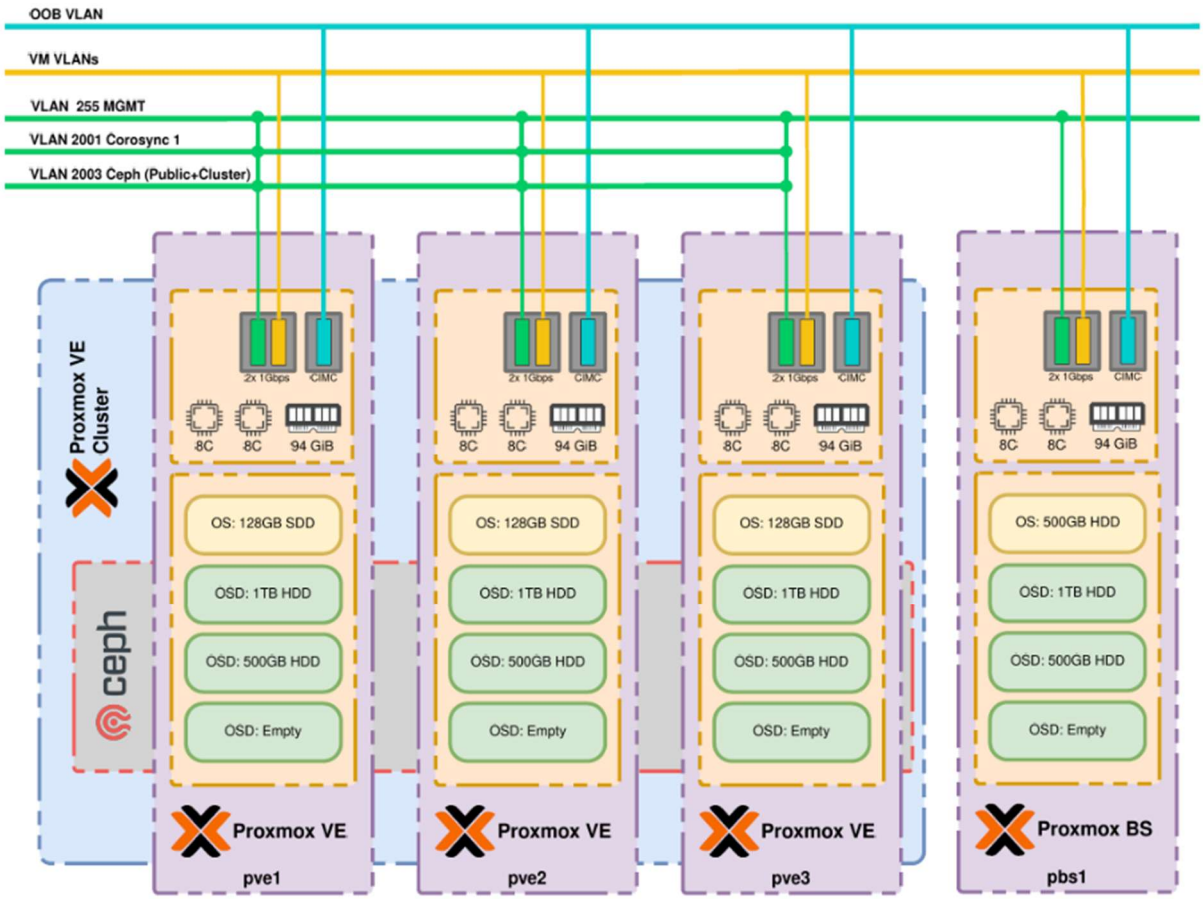
These nodes form a cluster that runs applications not only with high availability but also, for example, with dynamic scaling based on the current load of the given application by creating multiple replicas. Even if the user decides not to use the functionality of multiple replicas of the same application, Kubernetes enables rapid recovery of the application even if any node of the cluster fails – in that case, the Kubernetes node controllers responsible for the individual parts of the deployment ensure that the individual applications are restarted on the cluster nodes that are still available. As a result, Kubernetes extends the capabilities of containerization daemons with more comprehensive management of the container infrastructure.

Kubernetes uses a declarative model to define and abstract running components – it provides this functionality by means of objects of various types (in Kubernetes terminology also \code{kind}), where each type has a clearly predefined structure. Typically, these objects are defined in files in \ac{YAML} format, which the user can then apply to the cluster using the command \code{kubectl apply --filename file.yaml}.

Proxmox Virtual Environment was chosen as the hypervisor. This required several tasks to be completed:

- updating the firmware and drivers on all physical servers and configuring Out-of-band (OOB) management for remote server administration.
- installing the PVE platform on each of the selected servers
- basic configuration of the network connectivity of the nodes and preparation of the individual subnets for the required services:
    - A subnet intended for the management of the PVE servers.
    - Corosync subnet (the technology used for clustering the PVE servers)
    - Ceph subnet (the technology responsible for storage for the Kubernetes cluster
    - Subnets intended for virtual machines (e.g. Kubernetes nodes)
- Creating a PVE cluster from the created and configured servers.
- Creating a Ceph cluster within the PVE nodes and deploying the required daemons. These include monitoring services (Monitors), management services (Managers) and the individual storage daemons (Object Storage Daemon (OSD)), where each OSD instance is typically responsible for storing data on a specific assigned disk.
- Deploying a backup system in the form of Proxmox Backup Server (PBS) on a dedicated server. PBS is a software solution designed for efficient and incremental backup of virtual machines and containers running in the PVE environment and provides native integration with the PVE system, which is why it was the natural choice when selecting the backup system.

Completing all these tasks led to the creation of the following fully functional hypervisor infrastructure, which could be used in the subsequent steps to implement the Kubernetes cluster itself:



13.2.2 Virtual machine infrastructure – IaaC

For the deployment of the virtual machine infrastructure responsible for running the cluster, HashiCorp's Terraform was chosen; it allows the desired state of the infrastructure to be defined declaratively using configuration files in the HashiCorp Configuration Language (HCL) format.

Terraform supports a wide range of providers (terminologically and in configuration available as provider), on which it is able to deploy the desired objects (in this case primarily virtual machines).

This gives the created solution easy extensibility in the future in the event of a decision to migrate from Proxmox to another hypervisor, or to extend the cluster with additional virtual machines that need not necessarily be part of the same hypervisor cluster.

This also opens up possibilities for creating test clusters outside such a default solution. A repository was created for working with Terraform, in which a custom module (in Terraform terminology, module) was created as part of the work for easily creating multiple replicas in the form of different environments.

To create virtual machines, it is then only necessary to fill in the created variable virtual_machines of the given module, which represents a list of specially defined objects, where each object represents one virtual machine. When this module is used, the deployment configuration therefore looks as follows:

```
module "infrastructure" {
source = "../../modules/k8s-proxmox"
virtual_machines = [
{
name = "server1.k8s.kis.fri.uniza.sk"
vm_id = 51151
description = "K8S - PROD"
tags = local.tags
node_name = "pve1"
pool_id = local.pool_id
cores = local.server_cpus
memory = local.server_memory
network_bridge = local.network_bridge
boot_order = local.boot_order
cdrom = {
file_id = local.iso
}
disk = [
{
size = local.disk_size
interface = "scsi0"
}
]
datastore_id = "vd"
ipv4_address = "10.23.153.151/24"
ipv4_gateway = local.ipv4_gateway
dns_domain = local.dns_domain
dns_servers = local.dns_servers
},
{
name = "server2.kis.fri.uniza.sk"
```

Code: Example of using the created Terraform module

It can be seen that under the exposed variable virtual_machines, any number of virtual machines can be defined, and it also allows their properties to be defined, such as the name (name), a unique identifier in the Kubernetes cluster (vm_id), a dynamic number of disks and their corresponding sizes, network settings and several other settings – a detailed description is part of the documentation of the given repository.

As for the local.* variables, they were defined in order to reduce duplicated information when creating multiple similar virtual machines (as is usually the case with Kubernetes clusters). They include information common to all virtual machines, or also differing information based on the role of the virtual machine within the Kubernetes cluster (control/worker node).

An example is the amount of allocated memory or virtual processor cores (e.g. server_cpus and agent_cpus); however, this concept is applicable to anything:

```
locals {
tags = ["terraform", "taloslinux"]
pool_id = "K8S"
server_cpus = 4
server_memory = 8192
agent_cpus = 4
agent_memory = 8192
network_bridge = "vlan51"
ipv4_gateway = "10.23.153.1"
dns_domain = "test.k8s.fri.uniza.sk"
dns_servers = [ "8.8.8.8" ]
iso = "iso:iso/talos-v1.9.5-nocloud-amd64-qemuagent.iso"
disk_size = 50
boot_order = ["scsi0","ide0"]
}
```

Code: Example of using the created Terraform module – local variables

13.2.3 Repository structure

The designed and created repository structure looks as follows:

```
terraform/
├─ k8s/
│  ├─ prod/
│  │  ├─ (backend.tf)
│  │  ├─ main.tf
│  │  ├─ provider.tf
│  │  ├─ (terraform.tfvars)
│  │  └─ variables.tf
│  └─ test/
│     └─ ...
├─ modules/k8s-proxmox/
│  ├─ main.tf
│  └─ variables.tf
├─ Makefile
├─ Makefile.vars
└─ README.md
```

Note: Files placed in parentheses – "()" – are not part of the Git repository life cycle, as they contain sensitive variables that must be defined when the local repository is initialized.

• modules/ – folder containing modules. Within the Terraform tool, modules are reusable groupings of several Terraform configurations, often representing a certain part of the infrastructure. This makes it possible to define a configuration once (also with the option of parameterization) and then reuse it in different parts of the project/projects. It is precisely for this purpose that the module (folder) modules/k8s-proxmox/ was created, which, through this approach, makes it possible to create various replicas of Kubernetes environments for testing purposes.

• k8s/ – contains subfolders that define a specific deployment (environment) of the Kubernetes cluster (e.g. k8s/prod/ for the production deployment or k8s-test/ for the test deployment).

• k8s/<env>/ – environment folder (e.g. k8s/prod/).

– backend.tf – defines where the current state of the infrastructure is stored (typically a *.tfstate file) 34

– provider.tf – defines the provider, in this particular case a specific Proxmox provider

– variables.tf – declares the variables that the user will have to define (using the terraform.tfvars file) after initializing the local repository. These variables hold the login credentials for the Proxmox cluster on which the virtual machines of the module used are to be deployed.

– main.tf – the main Terraform file, which uses the module modules/k8s-proxmox defined above

ArgoCD provides a clear web interface (as well as a CLI) for displaying and managing deployed applications and supports managing multiple clusters at once, which is also why it is often the preferred tool in application development. It also has a number of add-on modules providing additional functionality and connection to external systems (such as integration with HashiCorp Vault for directly adding sensitive data).

FluxCD/**ArgoCD** integration – In addition to the operators mentioned above, it is also possible to use various integrations built into the GitOps tools FluxCD and ArgoCD. ArgoCD offers, for example, its own HashiCorp Vault plugin for direct integration without an additional operator, but additional support for KSOPS can also be enabled by changing the ArgoCD configuration (kustomize.buildOptions must be set with the additional switches "–enable-alpha-plugins and –enable-exec FluxCD, by contrast, given also its architecture, opts for a more minimalist design and mostly leaves this responsibility to external systems – with one exception, as it has native and robust support for SOPS.

13.2.4 Management and deployment of Kubernetes applications

FluxCD was chosen as a suitable tool for deploying additional applications. For it, an additional repository had to be created that would contain all the YAML manifests needed to deploy the individual applications. It was then necessary to create a suitable structure that would be easily extensible with further, new applications and would at the same time allow an application not only to be deployed but also to be deleted from a specific cluster in a simple and clear way. It is equally important that the repository allows easy extension with new clusters without unnecessary duplication, for which the Kustomize tool was primarily used. Another aim in using this tool was to separate the versioning of the individual installed applications at cluster level – that is, so that updates of individual applications could be tested outside the production deployment and, once tested, safely applied to the production deployment as well.

13.3 Docker Brainit

13.3.1 Description of the Docker infrastructure

The infrastructure is based on the Proxmox VE hypervisor deployed directly on a physical server (bare-metal). Proxmox uses a combination of KVM (Kernel-based Virtual

Machine) technologies for full virtualization and LXC for operating-system-level containerization. In the given design, KVM virtualization is used exclusively, which ensures full isolation of the individual virtual servers.

The hypervisor provides:

- management of compute resources (CPU, RAM),
- management of disk storage (LVM-thin, ZFS or a local RAID array),
- network virtualization by means of Linux bridge interfaces,
- centralized management and monitoring of virtual machines.

Each virtual server (VM) is allocated precisely defined hardware resources, including the number of virtual CPU cores, the size of the main memory and the disk space. The configuration is designed so that resources can be scaled dynamically without the need to reinstall the operating system.

13.3.2 Operating system of the virtual servers

The individual virtual servers run the AlmaLinux distribution, which is binary-compatible with Red Hat Enterprise Linux. This operating system was chosen because of its:

- high stability and long-term support,
- suitability for server and production deployments,
- compatibility with Docker Engine and modern DevOps tools.

The basic configuration of the AlmaLinux servers includes:

- minimal installation (without a graphical interface),
- package management using DNF,
- centralized updates of security patches,
- firewall configuration (firewalld or nftables),
- secured access via SSH using key-based authentication.

Each virtual server is conceived as an application node running Docker Engine and the related application containers.

13.3.3 Containerization layer

Docker Engine is installed on the AlmaLinux virtual servers and ensures isolated running of application components in the form of containers. Docker provides a uniform environment for deploying applications that is independent of the host system.

Docker Engine provides:

- management of the container life cycle,
- networking between containers using virtual Docker networks,
- data persistence by means of Docker volumes,
- isolation of processes, the file system and network interfaces.

The containers are designed so that each provides one specific function (e.g. database, backend service, frontend application), thereby adhering to the principle of microservice architecture.

13.3.4 Application orchestration

Containers are started and configured using Docker Compose files (docker-compose.yml). Docker Compose makes it possible to define declaratively:

- the individual services (services),
- the Docker images used,
- network connections between services,
- port mapping,
- persistent storage,
- configuration environment variables.

The Docker Compose files are stored directly on the virtual servers and enable:

- simple deployment of the entire application stack with a single command,
- reproducibility of the environment (test, staging, production),
- simple maintenance and updating of services.

Each application unit is separated into its own Compose project, which minimizes the risk of conflicts between the individual tools and services.

13.3.5 Overall architecture of the solution

The entire infrastructure is designed as a multi-layer architecture:

- physical layer (hardware),
- virtualization layer (Proxmox VE),
- operating system (AlmaLinux VM),
- containerization layer (Docker Engine),
- application layer (services managed by Docker Compose).

Such an approach ensures a high degree of modularity, scalability and ease of infrastructure management. It also enables effective separation of infrastructure, system and application responsibilities, which is suitable for modern cloud-ready and DevOps-oriented environments.

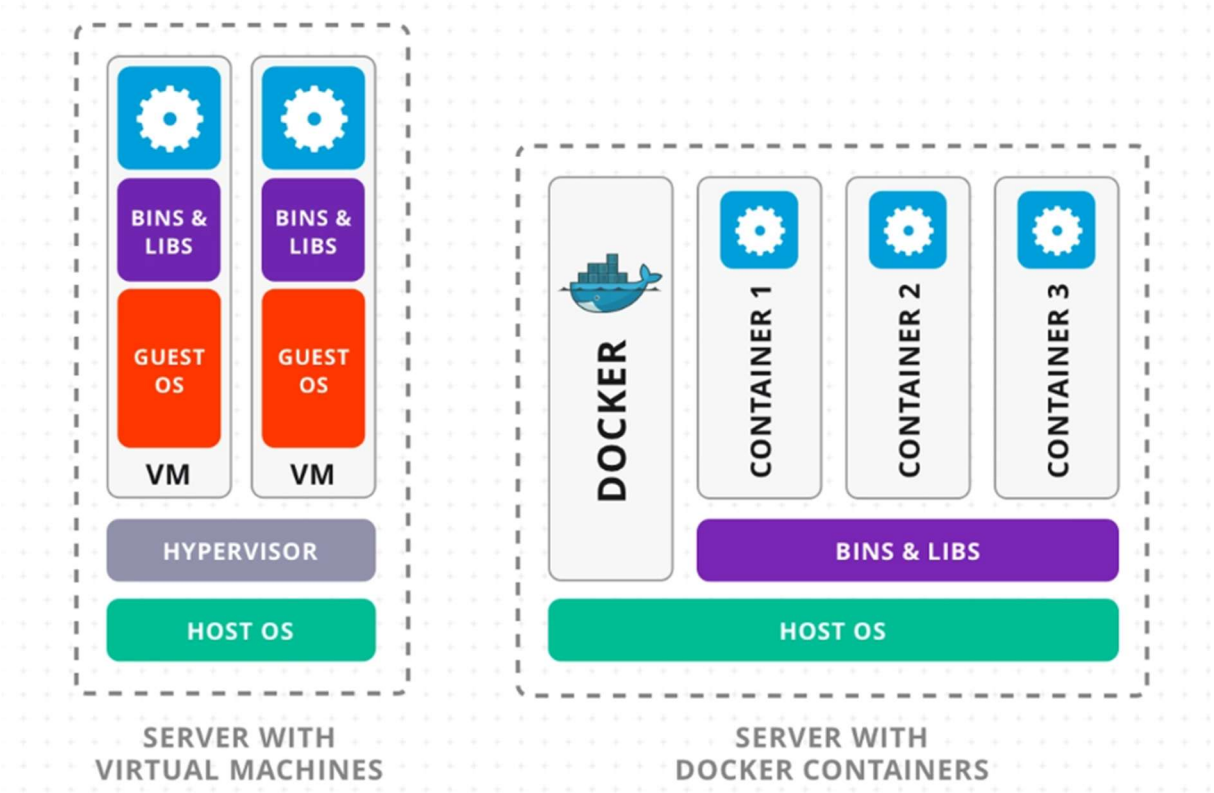


Diagram showing the topology of the Docker infrastructure

14 CONCLUSION

This document constitutes the technical documentation of the functional PoC deployment of a multi-tenant SOC ecosystem in which the individual security and supporting components are deployed, configured and described in a way that enables repeatable reproduction of the environment. The structure of the document is designed so that the key operational aspects are covered consistently for each tool: a description of the component's purpose in the SOC, references to documentation, hardware and system requirements, architecture with a diagram, the deployment procedure (typically Docker, for selected tools also OpenStack), basic configuration and integration links to the rest of the architecture. This approach ensures that the result is not merely a "list of tools", but a coherent infrastructure description of how an operable SOC layer in PoC mode is assembled from the selected open-source elements.

The technical line of the document begins with a chapter devoted to the ELK Stack, which in the design serves as the central point for the collection, storage and analytical processing of events. The chapter is written in the form of a complete deployment "runbook" – from the description and requirements through to Docker installation, configuration and integrations – thereby defining the SIEM/log-management foundation on which further layers (detection, response, record-keeping) build. This layer is followed by the chapter on Shuffle, where the SOC is extended with SOAR capabilities, i.e. orchestration and automation of responses by means of playbooks. The text explicitly takes into account that Shuffle is deployed in the PoC in both Docker and OpenStack variants, and it also contains a part devoted to connections to external applications (e.g. Wazuh and Greenbone), the definition of inputs/outputs and an explanation of its position in the SOC.

Next, Wazuh is elaborated as an endpoint source of detections (EDR/SIEM/endpoint security), with the document describing the architecture with a diagram and the way in which Wazuh detections are integrated into the central analytical layer and subsequently used for response and incident management. Technically, it is important that Wazuh is presented as a component filling the gap between raw log collection and higher-level correlation – that is, as a systematic source of endpoint alerts with subsequent linkage to ELK/Kibana and to other SOC tools. In parallel with the detection layer, the document also covers prevention through Greenbone Community Edition, which is treated as a component for vulnerability management – with an emphasis on deployment in Docker, requirements, architecture and basic configuration for vulnerability scanning and reporting.

A separate part of the document is devoted to the CTI layer, where two complementary approaches are addressed. OpenCTI is described as a knowledge and context platform (including configuration steps such as MITRE ATT&CK and selected feeds) with deployment in both Docker and OpenStack, and with a separately named part that explains the CTI integration model in the SOC.

The subsequent chapter on MISP elaborates the more operational sharing of indicators and the integration links to the other SOC components, and the document also contains an explicit comparison of OpenCTI vs. MISP and an explanation of how a combined deployment can divide tasks between a "fast source of indicators" and a "central threat knowledge base".

At the level of process support and infrastructure context, the document then moves on to GLPI (asset management/ITSM/ticketing) and NetBox (IPAM/DCIM/asset modelling). GLPI is described as a process-oriented layer that maps technical security outputs to assets, owners and services and ensures auditable and controlled incident handling, and integration options and configuration examples are also given (e.g. authentication/notification, the plugin ecosystem). NetBox is covered in detail, including an experimental part (diagrams, Docker/OpenStack deployment), with an emphasis on the fact that NetBox provides a technical source of truth about the infrastructure and is usable mainly in the enrichment phase – that is, as a source of precise context (IP → device mapping, location, segment/VLAN, device role), which reduces ambiguity during triage and supports automation without NetBox performing any active interventions. Also included is a technically formulated comparison of NetBox vs. GLPI, which documents that both tools cover asset records from different perspectives (technical topology vs. a process/organizational CMDB/ITSM layer) and naturally complement each other.

The incident and record-keeping part is covered by two chapters devoted to DFIR-IRIS – first in ticketing mode and then in the mode of recording and managing incidents throughout their entire life cycle. For DFIR-IRIS, the document emphasizes the case-based approach (the incident case as the main object), work with tasks, auditability and the possibility of linking technical artefacts, IOCs, notes and the investigation timeline to an incident; at the same time, the inputs/outputs and the integration character with respect to SIEM/CTI/SOAR tools are described (automated receipt of alerts, enrichment of the incident with context, export and sharing of investigation results).

The document is also complemented by a separate chapter on Kibana (ELK), which technically separates the visualization and analytical layer from the "stack" itself and describes its requirements, architecture, Docker installation and integration links.

The knowledge and documentation level is concluded by a chapter on Wiki.js, which is treated as an internal knowledge base, with the docker-compose deployment, basic configuration (e.g. SMTP) and integration options (e.g. API keys) given.

A significant concluding element is the chapter Deployment environment, which gives the entire implementation its infrastructure context. Here the document technically frames the deployment in the KIS FRI UNIZA OpenStack environment as having the character of a Proof of Concept: the aim is to verify the architectural design, the integration links and the operational properties of the tools, not continuous production operation. At the same time, the logical separation of roles (detection/response/CTI/knowledge/vuln scanning) through a combination of the virtualization layer, network segmentation and controlled integration interfaces is described, with OpenStack enabling tenant isolation, separate networks and replicability of experiments. The chapter also broadens the view with alternative approaches through Kubernetes Brainit (orchestration of containerized applications, infrastructure description) and Docker Brainit (containerization layer and compose-oriented deployment), so that the document covers not only the specific PoC deployment but also the infrastructure variants under consideration, which share the same aim: to operate SOC components as separate services in an isolated and repeatable manner.

As a whole, the document technically demonstrates that the designed SOC ecosystem is feasible as a modular architecture in which the roles of the individual components and their integration boundaries are clearly defined. From an operational perspective, it is key that the

text also describes the intended flow of event processing across the layers (detection in ELK, triggering of the response in Shuffle, enrichment via NetBox and other sources, subsequent action and an auditable output), which links the isolated "installation guides" into a single operationally readable SOC scenario.