Politecnico di Torino

DET - Department of Electronics and Telecommunications

MSc Mechatronic Engineering

A.a. 2024/2025

Graduation Session July 2025

Auto-Improving NIDS

Self-regulating Network Instrusion Detection System

Tutors:

Prof. Rizzo Alessandro
Silvio Massimino
Saverio Milo

Candidate:

Davi Antony
s321940

November 2024 - January 2025

Abstract

This project aims to develop a **self-regulating Network Intrusion Detection System (NIDS)** using **Wazuh**, integrating automation and adaptive security responses. The work is structured over a **300-hour internship**, balancing research, development, and testing phases.

Initial Phase:

- **Problem definition** and **literature review**, identifying key challenges in traditional NIDS.
- Exploring self-improving mechanisms.
- Establishing a **project roadmap** with milestones from **simulation setup** to **prototype validation**.

Implementation Phase:

- Setting up a **virtualized test environment**.
- Defining detection rules.
- Developing an adaptive response mechanism.

Evaluation Phase:

- Testing system efficiency through controlled attack simulations.
- Leading to the **first prototype demonstration** and analysis of results.

This project relies on a multidisciplinary approach:

- Cybersecurity notions
- High-skilled programming
- Multi-agent systems
- Game theory concepts
- Generative AI

Tokens are used throughout the project. You can identify them using a TODOs extension with appropriate metadata :

- 🐞 **BUG** — Indicates a bug or malfunction that needs fixing.
- 🔧 **HACK** — Marks a workaround or temporary solution that may need revisiting.
- ✅ **TODO** — A task or improvement to be completed.
- ✓ **TOHAVE** — A desired feature or requirement to be implemented.
- ✗ **TOREMOVE** — A deprecated or unneeded section to be removed.
- 📝 **DRAFT** — Indicates work-in-progress or notes not yet finalized.

> Adaptiveness to Wazuh is only a cover; what matters is the adaptable structure to any other field, having a self-modifying code that would reprogram itself to tackle encountered issues.
>
> Project is available on GitHub at the following link[a].
>
> [a] `https://github.com/k0raty/master_thesis.git`

Table of Contents

1 Global Presentation of the Project 4

1.1 Analysis of the problem . . . 4

1.1.1 Thesis Proposal . . . 4

1.2 Main Tools Used in the Project . . . 5

1.2.1 Wazuh . . . 5

1.2.2 Example: Web-Attack Scenario . . . 7

1.2.3 Setting Up the Virtual Environment using Vagrant . . . 9

1.2.4 Simplifying and Automating the Process . . . 10

2 Vagrant Environment 13

2.1 Structure of the Environment . . . 13

2.1.1 Tree Structure . . . 13

2.1.2 Top-Level Files . . . 13

2.1.3 Top-Level Directories . . . 14

2.2 Vagrantfile . . . 16

2.2.1 SSH Keys . . . 16

2.2.2 Setting Up the Machines . . . 17

2.2.3 Tests, scenarios and Cleaning . . . 19

3 Core of the Software 22

3.1 Structure of the Project . . . 22

3.2 Top-Level Files . . . 22

3.3 VM Agents . . . 24

3.3.1 VM Agents Structure . . . 24

3.3.2 Vmagents . . . 24

3.3.3 Onion View of a VM Agent . . . 25

3.3.4 Agent Types . . . 26

3.4 Usage . . . 27

3.4.1 Initializing Machines . . . 27

3.4.2 Remote.py . . . 27

3.4.3 XML Handler . . . 28

3.4.4 Purpose of XML Files . . . 28

3.4.5 Local Rules . . . 29

3.4.6 Decoder . . . 30

3.4.7 Ossec.py . . . 30

3.4.8 Organization of Ossec.conf File . . . 31

3.4.9 Methods and Structure of the Class . . . 32

3.4.10 File Monitoring Class . . . 33

3.4.11 Yara.py . . . 34

4 Provisioning of the vagrant environment 38

4.1 Cleaner . . . 38

4.2 Setup . . . 39

4.3 Wazuh Dashboard . . . 40

4.4 SSH Keys . . . 40

4.5 Tests & Scenario . . . 41

4.5.1 Testing suricata and ssh connections . . . 41

4.5.2 Wazuh scenario . . . 43

4.5.3 Wazuh dashboard identification . . . 44

4.5.4 Logs 45

5 Execution of the prototype 47
5.1 Unit Tests 47
5.1.1 Logs 47
5.1.2 Ossec.conf 48
5.1.3 Rules verification 51
5.1.4 File integrity monitoring 52
5.1.5 Xml modification 54
5.1.6 YARA 56
5.2 Prototype 60
5.2.1 Content provided by the AI 60
5.2.2 Main.py 61

6 Conclusion 63
6.1 Project 63
6.2 Greetings 64
6.3 Glossary 64
6.3.1 A Rule 64
6.3.2 Active Response 64

7 Annexes 65

A Overview 65
A.1 Timeline and project management 65
A.2 LLM enrichment tutorial 66

B SSH Key Generation Procedure 66

C Vagrant Provisionning 67
C.1 Cron jobs 68
C.2 cleaner.sh 68
C.3 Execution 69
C.3.1 Ssh connection via Virtual Studio Code 69
C.3.2 SSH Tests 69

D Src folder 70
D.1 Overall Methods 70
D.2 Non-Developed Folders 71

E Prototype execution 71
E.1 Yara installation logs 71
E.2 FIM detail modification 73
E.3 FIM logs 74

1 Global Presentation of the Project

1.1 Analysis of the problem

The aim of this part is to assess the problematic and explain how it was tackled via explaining the project overall and detailing the thinking process to end up to this solution.

1.1.1 Thesis Proposal

The objective of this thesis is to design a control software that allows for the monitoring of network traffic.
After studying its behavior, the software should be simplified by optimizing the firmware and enabling the following actions:

- Traffic monitoring.
- Reading signatures to identify any malicious events.
- Saving the communication header.

Thus, we can reformulate the question.
The objective is to create a Network-Based Intrusion Detection System (NIDS) capable of:

- Remembering headers and signatures of malicious packets.
- Being versatile and easy to set up within a company.
- Utilizing innovative and up-to-date tools.

Key Terms

- **NIDS (Network Intrusion Detection System)**: Solutions that analyze traffic to detect unusual activities such as scanning, intrusion attempts, lateral movements, exfiltration, backdoors, command and control, etc.
- **Signature**: A recognizable pattern associated with an attack, such as a binary string in a virus or a set of keystrokes used to gain unauthorized access to a system.
- **Headers**: Metadata of an instance, which can be an email, HTTP request, etc.

Approach to Tackle the Issue

The NIDS is the foundation of the entire thesis, and to counter an attack using it, you need to:

- Spot malicious activity through continuous monitoring and detection rules.
- Set up rules that accurately identify threats based on known patterns or behaviors.
- Configure active response commands to automatically mitigate threats and capture relevant information for further analysis.

To achieve the objective, we rely on an innovative tool called *Wazuh*.

Steps	**How**	**Why**
a. Detection of Unknown Patterns	1. Use an ML or trained detection tool to analyze data and identify anomalies or suspicious activities. 2. Use adapted tools leveraged via Wazuh.	**Wazuh is tailored to detect known patterns** involving known headers and signatures. **Without initial detection, a threat cannot be monitored by an NIDS**. We must spot it at least once to include it in the database. This subject was part of another master's thesis, aiming to coordinate both theses to create a sustainable system capable of identifying unknown threats and adapting correct measures without human intervention.
b. Monitoring and Detection	1. Wazuh continuously monitors system logs, network traffic, and other data sources for suspicious activities. 2. Predefined or custom rules in `local_rules.xml` are used to detect known threats and anomalies.	Once the pattern is known, it can be integrated into Wazuh as local rules. This allows Wazuh to trigger an appropriate response to tackle the issue.
c. Implementing an Active Response	Upon detecting a threat, Wazuh can trigger active response commands configured in `ossec.conf`.	This serves as a countermeasure against the attack.

Table 1: Steps to Tackle the Objective

1.2 Main Tools Used in the Project

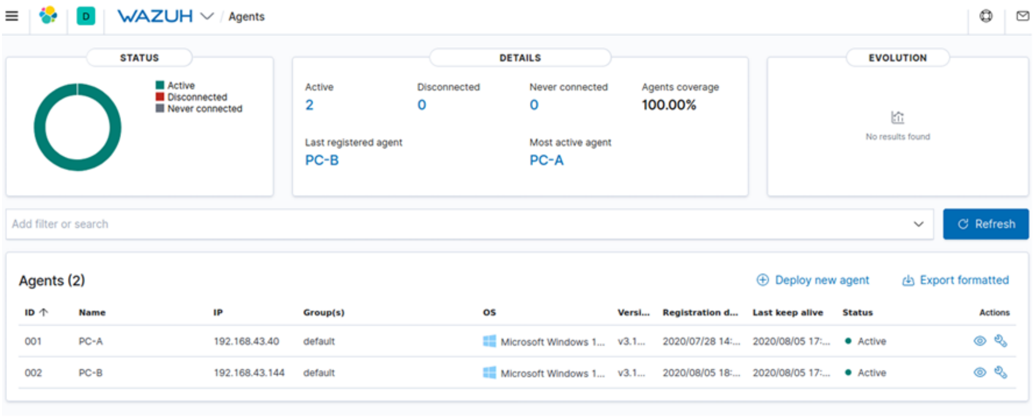
1.2.1 Wazuh

Braintech requested the use of *Wazuh* as a Network Intrusion Detection System (NIDS). Wazuh can be divided into two main entities:

- **The Manager**: Collects data from monitored endpoints to analyze and identify potential issues.
- **The Monitored Agents**: Endpoints from which data is collected.

The NIDS is installed on a separate entity called the manager, not directly on the victim computer.

On the manager, we use the Kibana dashboard for ergonomic and visual purposes to monitor traffic flow on our agents. It allows us to quickly spot any detected threats and the corresponding active responses initiated.



Active 2
Disconnected 0
Never connected 0
Agents coverage 100.00%
Last registered agent PC-B
Most active agent PC-A

Agents (2)

ID ↑	Name	IP	Group(s)	OS	Versi...	Registration d...	Last keep alive	Status	Actions
001	PC-A	192.168.43.40	default	Microsoft Windows 1...	v3.1...	2020/07/28 14:...	2020/08/05 17:...	● Active	
002	PC-B	192.168.43.144	default	Microsoft Windows 1...	v3.1...	2020/08/05 18:...	2020/08/05 17:...	● Active	

Figure 1: Kibana Dashboard Interface

Wazuh relies mainly on two files:

- `ossec.conf`: The main configuration file defining rules, active responses, and operational settings.
- `local_rules.xml`: Defines custom rules for detecting specific patterns or anomalies.

local_rules.xml

- **Purpose**: Defines custom rules for detecting specific patterns or anomalies.
- **Rules**: Specifies conditions and actions based on logs, FIM, and other data sources.

These rules are downloaded onto the manager, the XML file orchestrates them.

ossec.conf

- **Purpose**: Main Wazuh configuration file, defining rules, active responses, and operational settings.
- **Rules**: Specifies applicable rules (e.g., `local_rules.xml`) and actions for matches.
- **Active Responses**: Configures automated threat mitigation commands.

Wazuh can handle various types of attacks , but for this thesis, the focus is on:

- **Intrusion Detection**: Detects rootkits and malware.
- **Log Data Analysis**: Collects and organizes logs from monitored systems, including applications like Docker (e.g., from websites).
- **File Integrity Monitoring**: Monitors changes in predefined directories to detect unauthorized changes.

There exist various kinds of attacks and these considerations constrain us to focus only on specific (althrough mainly common) attack cases. Howerver, handling an attack using wazuh rely always on the same process.

Workflow Summary to Handle a Threat Using Wazuh

1. Identify the signature of the threat.

2. Define rules in `local_rules.xml` and download necessary files (rules, other enrichment software).

3. Call associated active response in `ossec.conf`.

1.2.2 Example: Web-Attack Scenario

To illustrate the concept, let's detail a basic cyber-defense procedure to counter a web attack* using Wazuh and DVWA (Damn Vulnerable Web Application).
Below is a scheme of the several interactions between entities in case of a web – intrusion scenario.

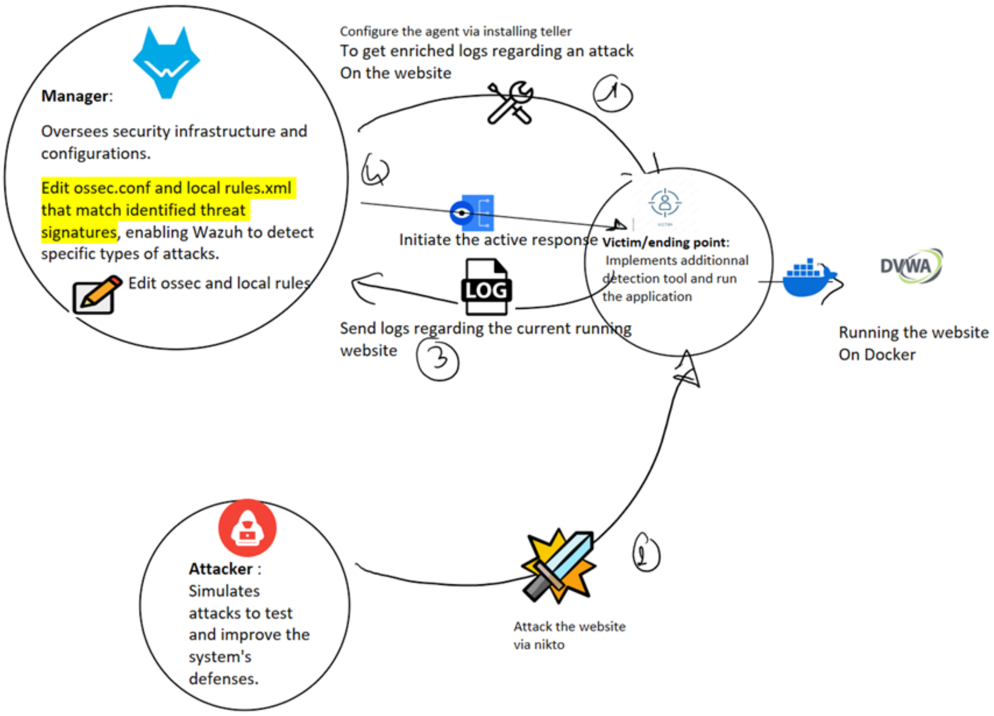

Manager:
Oversees security infrastructure and configurations.
Edit ossec.conf and local rules.xml that match identified threat signatures, enabling Wazuh to detect specific types of attacks.
Edit ossec and local rules
Configure the agent via installing teller
To get enriched logs regarding an attack
On the website
1
4
Initiate the active response
Victim/ending point:
Implements additionnal detection tool and run the application
LOG
Send logs regarding the current running website
3
DVWA
Running the website
On Docker
Attacker :
Simulates attacks to test and improve the system's defenses.
2
Attack the website
via nikto

Figure 2: Web-Intrusion Scenario with Wazuh

Comment:

- To implement a scenario, we need an **orchestration** between each unit.

- Logs are essential to understand the overall process.

- **Without editing** the `ossec.conf` and the local rules, there can't be an active response.

This setup ensures that threats are effectively identified, appropriate rules are defined for detection, and active responses are configured to mitigate threats using Wazuh.

*`https://wazuh.com/blog/detecting-web-attacks-using-wazuh-and-teler/`

Beside the orchestration between elements, we can have a chronological perspective

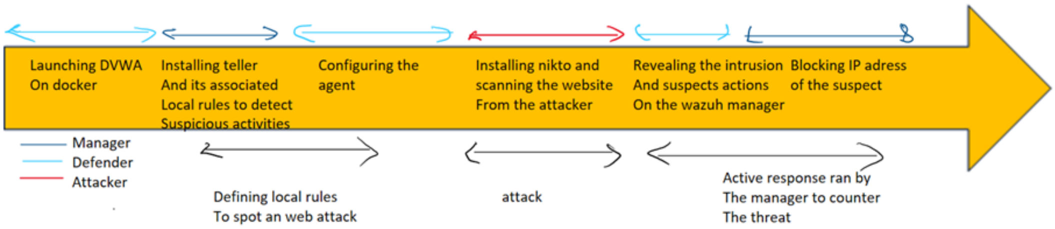

Launching DVWA On docker
Installing teller And its associated Local rules to detect Suspicious activities
Configuring the agent
Installing nikto and scanning the website From the attacker
Revealing the intrusion And suspects actions On the wazuh manager
Blocking IP adress of the suspect
Manager
Defender
Attacker
Defining local rules To spot an web attack
attack
Active response ran by The manager to counter The threat

Figure 3: Web-Intrusion chronological perspective

We retrieve the several steps highlighted during the theoretical analysis, in particular these two steps that consist in defining the two XML files.
In local rules, we need to modify the file system so that the manager can decode the logs sent by Teler from the agent consisting in adding an XML block of the form:

```
<group name="teler,">
    <rule id="100012" level="10">
        <mitre>
            ...
        </mitre>
        <description>teler detected $(category) against resource $(request_uri) from $(remote_addr)</description>
    </rule>
</group>
```

Listing 1: Local rules for Teller

In an identical way, we have to add a block for the `ossec.conf` file that will trigger the active response once Teler detects a suspicious activity:

```
<localfile>
    <log_format>syslog</log_format>
    <location><PATH_TO_LOGFILE>/output.log</location>
</localfile>
```

Listing 2: Local file for Teller

The issue is that filling those two files manually each time a new threat is spotted can be tedious and lead to mistakes.
Indeed, simply adding the block at the end of the file can lead to redundancies (having most of the time the same active response for several anomalies, alias blocking the IP) or any other syntactical mistakes.
This is why the main purpose of the thesis is to:

- Find a simple and efficient way to fill these two files.
- Permit a versatile approach to have an auto-adaptive system each time a new threat is found.
- Create a suitable environment to test the concept.

We finally retrieve the thesis problematic definition introduced above 1.1.1.

1.2.3 Setting Up the Virtual Environment using Vagrant

To test our program, a virtualized environment is necessary. This involves creating virtual machines that can communicate efficiently with each other. The virtual environment must be suitably configured to meet the constraints of an agency.

Why Vagrant? Vagrant was chosen for its benefits. Vagrant is an open-source tool for building and managing virtualized development environments, primarily using simple configuration files.

Vagrantfile A Vagrantfile provisions virtual machines via *VirtualBox*. It orchestrates their security, memory allocation, and downloads essential packets for each entity. Once the machines are provisioned, they can be launched via the command line using `vagrant up`.

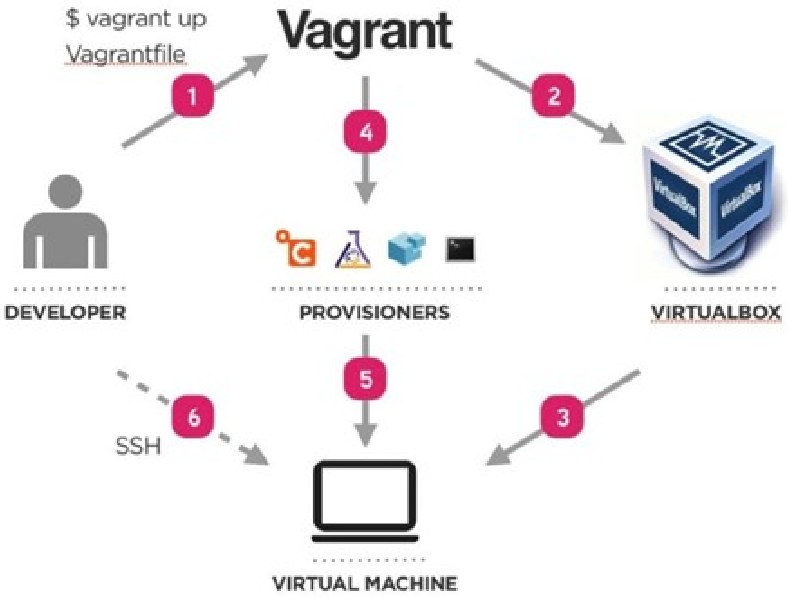


Figure 4: Vagrant Configuration for Virtual Machines

The environment must assume several features:

- Scalability
- Portability
- Automated
- Suitable for testing purposes (communication between Python & Bash programs)
- Clear and structured

Arguments to Pass

- IP address of the server
- Number of agents

To meet these requirements, the following features were implemented:

- Secure communication
- Shared folder & file synchronization between VMs
- Centralized configuration on the host machine

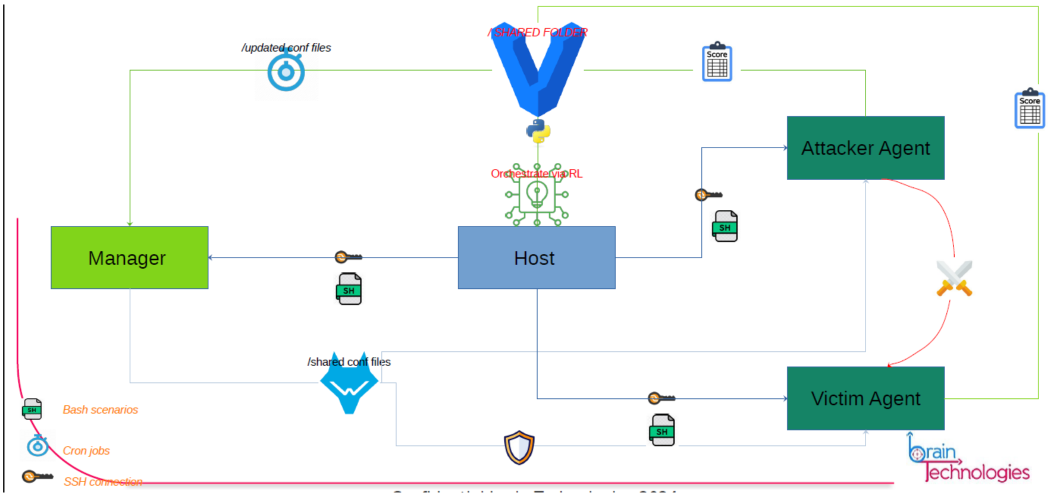


Figure 5: Vagrant Environment of the project

From this setup, we conclude:

- The host can connect to each machine, acting as a conductor to manage attack and defense scenarios.
- Complex tasks (such as generating `ossec.conf` and rules, data analysis) are handled in Python, while provisioning the machines is done in Bash using Vagrant (e.g., downloading new packets, logs).
- A shared folder `/vagrant` facilitates file exchanges between the machines.

> **To Go Further**
>
> The last point is crucial as, when transitioning from a virtual environment to a test/real environment, an SFTP server will be required to ensure file exchanges between endpoints and the manager.

1.2.4 Simplifying and Automating the Process

The primary challenge lies in the tedious process of setting up the agent and configuring the two essential files.

Why is it Tedious?

- A preliminary analysis is required to identify the measures to be taken regarding the threat after receiving the logs.
- The two main files to configure must adhere to a precise typography (`ossec.conf` and `local_rules.xml`).
- Manual intervention between each machine is necessary as the manager, victim, and attacker need to be set separately (during the testing phase).

To address this issue, a class-based Python program was developed on top of the Vagrant provisioning scripts.
This approach offers several benefits:

- Allows the definition of unit tests and the use of Python to verify the application's functionality.
- Facilitates the implementation of a multi-agent system coordinated by an evaluator.
- Generally simplifies the procedures.

Integration with Large Language Models (LLMs)
This approach is designed to accommodate the use of a Large Language Model (LLM) defined on the host machine. With simple commands, the LLM can interact with the class-based functions to:

- Identify the threat via logs and determine the corresponding measures.
- Complete the two essential files.
- Set up the victim with the required packets.

This method was inspired by the following tutorial: Leveraging LLMs for Alert Enrichment A.2[†]

Based on this concept, the current project can parse new malware logs to a language model, allow the LLM to make modifications via the class-based script, and prompt the owner to apply the modifications based on test results.
This innovative approach may become prevalent in the industry, where:

- An AI modifies code using defined rules within a set environment.
- Modifications are rated and applied or rejected.
- The system targets self-improvement without human intervention.

The project relies on three main actors:

1. A virtual environment setting up a manager, victim, and attacker.
2. A host running this virtual environment, interacting with it using a class-based Python script.
3. An LLM providing the host with the required actions to execute. [‡]

The project is tested in the context of file modification but can be adapted to various other types of attacks. It will simply require defining an appropriate script to parse logs from the targeted field to the LLM.

[†] `https://documentation.wazuh.com/current/proof-of-concept-guide/leveraging-llms-for-alert-enrichment.html`

[‡] The last point wasn't implemented, but is only theorical (yet feasible) as the two main points were already a large work to implement in the case of master thesis using rigorous methods (300 hours).

A timeline of the project is available in Appendix A.1 and can be resumed as follows :

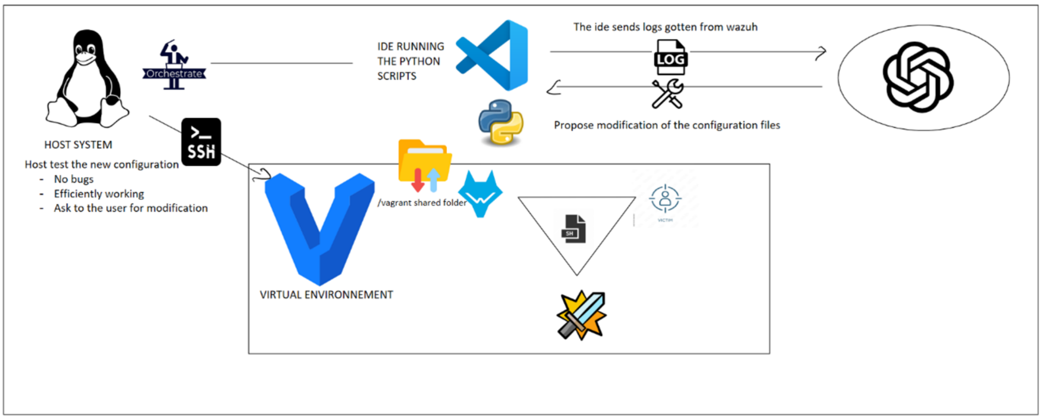

The ide sends logs gotten from wazuh
IDE RUNNING THE PYTHON SCRIPTS
Orchestrate
LOG
SSH
HOST SYSTEM
Host test the new configuration
- No bugs
- Efficiently working
- Ask to the user for modification
Propose modification of the configuration files
/vagrant shared folder
SH
VICTIM
VIRTUAL ENVIRONNEMENT

Figure 6: Simple overview of the project

In summary, the project addresses the original problem by creating an environment that allows a virtual manager to correct its configuration files and set up monitoring endpoints (agents) using artificial intelligence.
We wanted a software able via Wazuh, to identify a threat and correct its system to adopt corrective measures in case of a new encountering.
This AI analyzes logs sent by Wazuh, adapts to situations, and proposes modifications that are tested for efficiency.
This versatile approach allows for:

- Choosing the number of agents and designing the configuration for any company.
- Being class-based, adding as many classes as needed to define agent types and handle various attack types.

This represents the foundation of a self-sufficient system that, if well-designed, will not become outdated as it relies on up-to-date tools.

2 Vagrant Environment

Since the project is divided into two parts, we first delve into the initial phase, which focuses on designing the simulation environment and exploring its complexities.

2.1 Structure of the Environment

The structure of the environment is crucial for understanding the overall process that defines the setup and operation of the virtualized system. This section provides a detailed overview of the folder structure, top-level files, and directories involved in the Vagrant setup.

2.1.1 Tree Structure

The tree structure of the environment is illustrated in Figure 3. This structure helps in visualizing the organization of various folders and files necessary for building and managing the Vagrant environment.

```
[DIR] .vagrant
[DIR] backup
[DIR] common
[DIR] ephemere
[DIR] host
[DIR] src
[DIR] VM
[FILE] cleaner.sh
[FILE] debug.sh
[FILE] logging.sh
[FILE] make_executable.sh
[FILE] requirements_attacker.txt
[FILE] requirements_defender.txt
[FILE] requirements.txt
[FILE] setup_host.sh
[FILE] sourcer.sh
[FILE] utils.sh
[FILE] Vagrantfile
[FILE] variables.sh
```

Listing 3: Structure of the overall project

2.1.2 Top-Level Files

The top-level files are essential for the initial setup and debugging of the environment. These files serve as headers in other bash scripts, ensuring consistency and simplifying the debugging process.

- **logging.sh**: Facilitates the display of logs with different verbosity levels (INFO, DEBUG, WARNING, ERROR). This is crucial for debugging all bash scripts.
- **make_executable.sh**: Ensures that all bash scripts are executable at the start.
- **utils.sh**: Contains utility functions for various operations.
- **variables.sh**: Defines useful variables such as paths, IPs, and keys.
- **sourcer.sh**: Automatically imports essential scripts like `logging.sh`, `utils.sh`, and `variables.sh` into lower-level scripts.
- **Requirements.txt**: Lists dependencies and packages to be installed on virtual machines or the host.

We basically for every bash script :

- Source the sourcer
- Log information with the logger
- Use redundant functions via utils
- Eventually create a backup in the backup folder

To do so , two templates are available in the code and can be used regarding the situation.

2.1.3 Top-Level Directories

The top-level directories are organized to manage different aspects of the environment, ensuring a structured approach to handling backups, common provisioning scripts, and machine-specific configurations. Below is a detailed description of each directory along with its tree architecture:

- **.vagrant**: This directory is automatically created by Vagrant and contains the virtual machines. It is not meant to be modified manually.

```
.vagrant/
|- bundler/
|- machines/
|    |- wazagent1/
|    |    \- virtualbox/
|    |- wazagent2/
|    |    \- virtualbox/
|    \- wazidx1/
|         \- virtualbox/
\- rgloader/
```

- **backup**: Contains backups of important content, with timestamps for version control.

```
backup/
|- bin/
|- csv/
|- keys/
\- log/
```

- **common**: Includes common provisioning scripts, especially for testing connections between the host and virtual machines.

```
common/
\- tests/
```

- **ephemere**: Stores session-specific content such as SSH keys, passwords, and IP addresses, which are regenerated with each session.

```
ephemere/
|- bin/
|- csv/
|- keys/
\- log/
    |- bin.txt
    \- credentials.txt
```

- **Host**: Contains scripts and configurations specific to the host machine.

```
Host/
|- configs/
|- localrules/
|- scenarios/
|   |- alienvault/
|   |   |- attacker/
|   |   |- defender/
|   |   \- manager/
|   |- brute_force/
|   |   |- attacker/
|   |   |- defender/
|   |   \- manager/
|   \- dvwa/
|       |- attacker/
|       |- defender/
|       \- manager/
\- setups/
```

- **VM**: Includes installers and setup scripts for virtual machines, focusing on Wazuh agent and server configurations.

```
VM/
|- configs/
|- installers/
|   |- install_braintech.sh
|   |- install_node_exporter.sh
|   |- install_wazuh_agent.sh
|   \- install_wazuh_server.sh
|- setups/
|   |- logrotation/
|   |   |- setup_cron_job.sh
|   |   |- setup_log_rotation.sh
|   |   \- setup_ssh_keys.sh
\- tests/
    \- snort-scan.sh
```

This structured organization ensures that each component of the environment is easily accessible and modifiable, facilitating efficient management and troubleshooting.

We can structure the setup into three main components:

- HOST : Contains all Bash scripts intended to run on the host machine during the virtual machine creation process.
- VM : Includes scripts specifically designed to execute within each virtual machine.
- COMMON : Serves as a shared space for backups, credits, and scripts that facilitate communication between the host and virtual machines.

2.2 Vagrantfile

The `Vagrantfile` is the core configuration file for setting up the Vagrant environment. It defines the initial setup and provisioning of virtual machines.
It is inspired form Xavki gitlab tutorials[§] where key aspects include:

- **Initial Setup vs. Usual Start**: Distinguishes between the first-time setup and subsequent starts. The initial setup involves creating and configuring virtual machines, while subsequent starts activate already built VMs.
- **Provisioning Scripts**: Executed only once during the initial setup to configure the environment.
- **SSH Keys**: Generated and distributed to allow secure access to virtual machines.

The pseudocode for the `Vagrantfile` is illustrated in Appendix C.

2.2.1 SSH Keys

SSH keys are essential for secure communication between the host and virtual machines.
The setup involves generating key pairs and configuring them for user and root access, for more info refer to Appendix B

- **setup_host.py**: Generates SSH keys for user access to virtual machines.
- **setup_ssh_keys.sh**: Generates SSH keys for root access to virtual machines.
- **setup_host_finish.py**: Configures VSCode for connection to virtual machines.

The process of generating SSH keys is illustrated in Figure 17.
By default, **Vagrant** permits connection from the home user to its virtual machines. By executing `sudo vagrant ssh <nameVM>` in the command line, we can access each of them individually.
The terminal screen used for accessing virtual machines is shown in Figure 8.
From this terminal, we are able to configure each virtual machine via command line.
However, this approach is not ideal, which is why we decided to configure Visual Studio Code for remote access.

[§] `https://gitlab.com/xavki/tutoriels-wazuh`

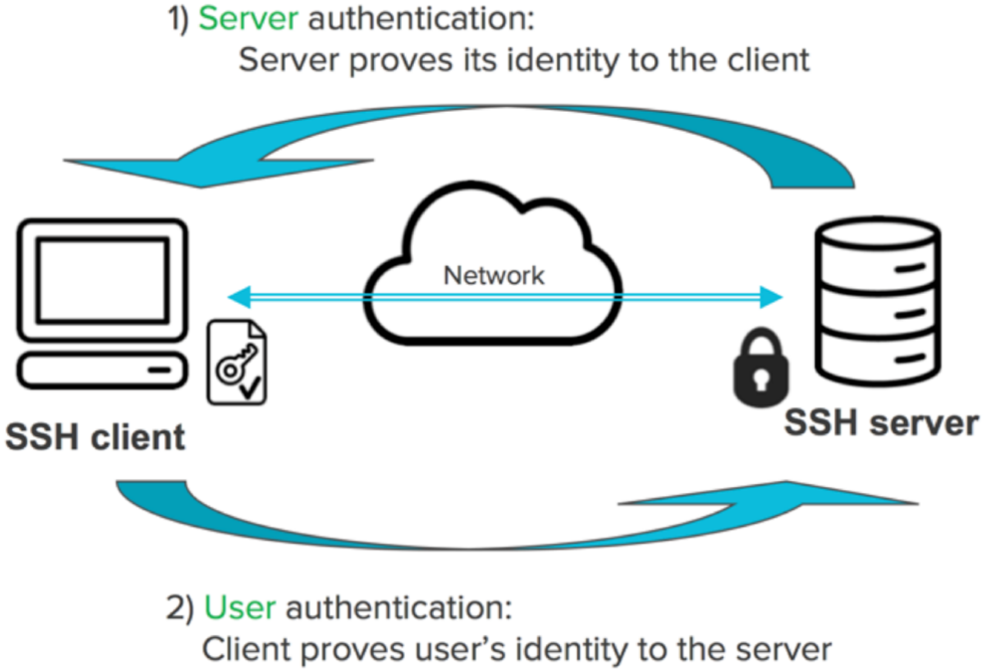

1) Server authentication:
Server proves its identity to the client
Network
SSH client
SSH server
2) User authentication:
Client proves user's identity to the server

Figure 7: Generating SSH keys for secure access.

2.2.2 Setting Up the Machines

The setup of virtual machines involves configuring the server and agents within the Vagrant environment. This includes setting up the Wazuh server and agents, as well as configuring SSH keys for secure access.

- **Server Setup**: Configures the Wazuh server on the manager virtual machine.
- **Agent Setup**: Configures Wazuh agents on the monitored virtual machines.

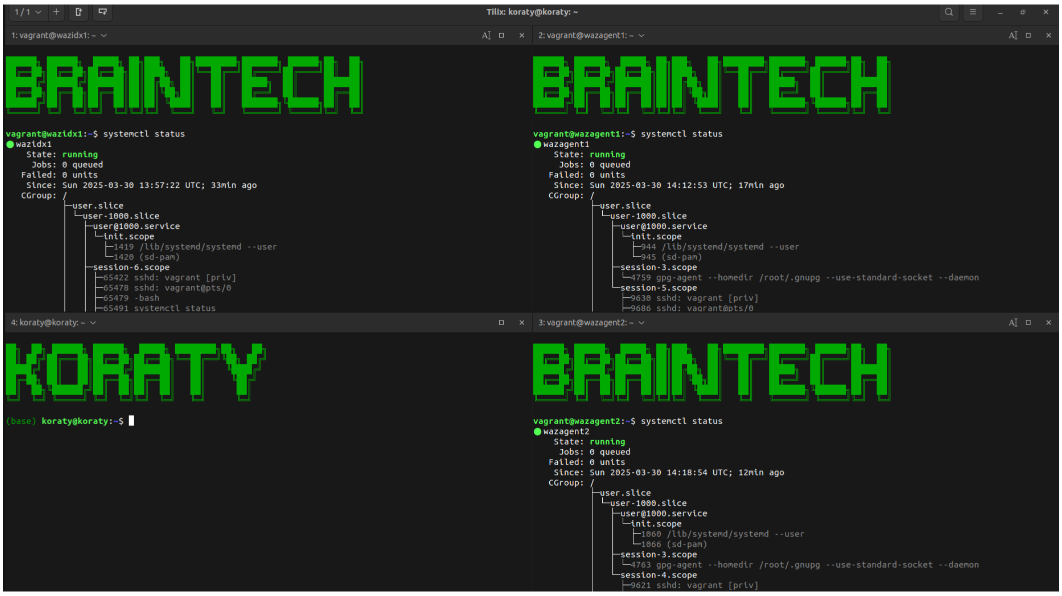
The configuration process is designed to be modular, allowing for easy management and scalability. Below is an explanation of the key scripts and configurations used to set up the machines:

1. **Setting Up the Server**

   As a reminder, it concerns the following script:

```
if node[:type] == "server"
    # Installation #
    cfg.vm.provision :shell, :path => "VM/installers/install_wazuh_server.sh"
    ## HERE Pushing configuration files to server dedicated folder's ##
    cfg.vm.provision :shell, :path => "VM/configs/manager/config_manager.sh"
    ## HERE Cron Jobs ##
    #cfg.vm.provision :shell, :path => "VM/setups/setup_cron_job.sh"
    ## HERE Cron Jobs ##
    cfg.vm.provision :shell, :path => "VM/setups/setup_log_rotation.sh"
end

```

   We indeed need to configure the manager. This is done via `config_manager.sh` where also we define `agent.conf`.



```
vagrant@wazidx1:~$ systemctl status
● wazidx1
    State: running
     Jobs: 0 queued
   Failed: 0 units
    Since: Sun 2025-03-30 13:57:22 UTC; 33min ago
   CGroup: /
           ├─user.slice
           │ └─user-1000.slice
           │   ├─user@1000.service
           │   │ └─init.scope
           │   │   ├─1419 /lib/systemd/systemd --user
           │   │   └─1420 (sd-pam)
           │   ├─session-6.scope
           │   │ ├─65422 sshd: vagrant [priv]
           │   │ ├─65478 sshd: vagrant@pts/0
           │   │ ├─65479 -bash
           │   │ ├─65491 systemctl status
```

```
vagrant@wazagent1:~$ systemctl status
● wazagent1
    State: running
     Jobs: 0 queued
   Failed: 0 units
    Since: Sun 2025-03-30 14:12:53 UTC; 17min ago
   CGroup: /
           ├─user.slice
           │ └─user-1000.slice
           │   ├─user@1000.service
           │   │ └─init.scope
           │   │   ├─944 /lib/systemd/systemd --user
           │   │   └─945 (sd-pam)
           │   ├─session-3.scope
           │   │ └─4759 gpg-agent --homedir /root/.gnupg --use-standard-socket --daemon
           │   └─session-5.scope
           │     ├─9630 sshd: vagrant [priv]
           │     ├─9686 sshd: vagrant@pts/0
```

```
(base) koraty@koraty:~$ █
```

```
vagrant@wazagent2:~$ systemctl status
● wazagent2
    State: running
     Jobs: 0 queued
   Failed: 0 units
    Since: Sun 2025-03-30 14:18:54 UTC; 12min ago
   CGroup: /
           ├─user.slice
           │ └─user-1000.slice
           │   ├─user@1000.service
           │   │ └─init.scope
           │   │   ├─1060 /lib/systemd/systemd --user
           │   │   └─1066 (sd-pam)
           │   ├─session-3.scope
           │   │ └─4763 gpg-agent --homedir /root/.gnupg --use-standard-socket --daemon
           │   └─session-4.scope
           │     ├─9621 sshd: vagrant [priv]
```

Figure 8: Terminal screen for accessing virtual machines via command-line tool.

¶ `agent.conf` will set the machine to use the following tools:

- **Suricata**: A network threat detection engine.
- **Teller**: A tool for monitoring and alerting.

The aim is to test the system at the end of the provisioning to ensure that the environment is working properly without using the `/src` code but simply by configuring the machine manually.
We also set up cron jobs in setup_cron_jobs.sh that may be useful
This file is modular and other cron jobs can be added. (Appendix C.1)

We for instance ensure that the system is able to synchronize ossec.conf between the host and the manager. Synchronization of files is crucial for the sake of the thesis, as later modifications will be made on the host and synchronized on the VMs.

2. **Setting Up the Agents**

```
if node[:type] == "agent"
    # Installation #
    cfg.vm.provision :shell, :path => "VM/installers/install_wazuh_agent.sh"
    ## HERE Configuration of the agent ##
    cfg.vm.provision :shell, :path => "VM/configs/agents/config_agent.sh"
    ## HERE Agents tests ##
    cfg.vm.provision "shell", inline: <<-SHELL
      sudo /vagrant/VM/tests/snort-scan.sh #{node[:ip]}
    SHELL
end

```

¶**agent.conf**: The agent configuration file is identical to `ossec.conf` but only concerns the specified virtual machine.

We first configure the endpoints as agents via `install_wazuh_agent.sh`.
As for the manager, we configure the agents both for *Wazuh* and *Suricata*[‖].

2.2.3 Tests, scenarios and Cleaning

Once both the manager and agents are configured, we can run final tests and scenarios.

1. **Setup Host Finish (setup_host_finish.sh)**

   This script performs the final setup tasks, including copying SSH keys, running final tests, and executing scenarios to ensure everything is functioning correctly.

```
# HERE Running the final tests & scenarios for first configurations &
    ensuring everything works fine #
sudo -su $(hostname) $PATH_TO_TESTS_FILE
sudo -su $(hostname) $PATH_TO_SCENARIOS_FILE

```

These tests and scenarios are modular and can be enriched without issues

- **Final Tests**

  The test folder contains scripts for testing SSH connections:

```
.
|- test_ssh_VM_to_host.sh
|- test_ssh_host_to_VM.sh
\- tests.sh
```

  - `tests.sh` coordinates the tests run by the other two scripts:
    - A connection attempt from VMs to host (not effective).
    - A connection attempt from host to VMs.
  - In case of failure, an error is logged, but provisioning does not stop.

- **Scenarios**

  The main purpose of these scenarios is to demonstrate the orchestration by the host system to coordinate each virtual machine using SSH commands. This serves as the final phase of the Vagrant provisioning, acting as a prototype of the thesis but written in Bash.

  The scenario folder is structured as follows:

```
.
|- alienvault/
|- brute_force/
|- dvwa/
\- scenarios.sh
```

  - `scenarios.sh` coordinates the execution of each scenario.
  - Each scenario follows a structured approach:

[‖] `https://documentation.wazuh.com/current/proof-of-concept-guide/integrate-network-ids-suricata.html`

```
.
|- attacker/
|- defender/
|- manager/
\- scenario_x.sh
```

- `scenario_x.sh` follows a pseudo-code structure:
  (a) **Initialization**: Change to the working directory, check existence, define logs, etc.
  (b) **Node Processing**: Read the CSV file for node information, determine node type, and execute corresponding scripts via SSH. Handle errors by logging and incrementing an error counter.
  (c) **Scenario-Specific Actions**: Perform additional checks or actions specific to the scenario.
  (d) **Completion**: Log the completion status, indicating success or errors.

Directly related to the following GitLab repository: Wazuh Tutorials[**] where the key difference here is that instead of manually executing each action, the host system orchestrates all simulation attack and defense procedures.

2. **Cleaning (cleaner.sh)**

If the machines need to be destroyed, this script is executed. It performs cleanup operations to prepare the system environment, see Appendix C.2

This script ensures that:

- SSH configurations are backed up.
- Specific `.txt` and `.pub` files are cleared, with their contents appended to a central file.
- `.tar` files and log files are removed.
- The cleanup is marked as complete to prevent redundant executions.

[**] https://gitlab.com/xavki/tutoriels-wazuh

In conclusion this Vagrant environment is:

- **Structured**
  - Harmonized templates of scripts.
  - Rigorous coding practices: backups, logs, cleans, sourcing, etc.
  - Logical and ergonomic folder structure.
- **Consistent**
  - SSH connection between host and VMs.
  - Tests ensuring proper functioning of VMs.
  - Scenarios essential for the prototype described in the next steps (`/src` folder).
- **Modular**
  - Configurable Wazuh settings (version, number of VMs, IP).
  - Configurable tests.
  - Configurable scenarios.

This setup ensures a robust, modular, and consistent environment for testing and deploying the NIDS.

3 Core of the Software

As the environment has been studied in detail, we will focus on the core of the software, this /src folder is written in Python.
As presented in subsubsection 1.2.4, the aim of this software is to (in priority order):

- Simplify procedures to configure the Wazuh system
- Simulate scenarios
- Permit the integration of an LLM

A **class-based system** was implemented for this purpose. We will examine its architecture and the reasoning behind this design, using a similar analysis as for the Vagrant environment.

3.1 Structure of the Project

The project has the following structure:

```
[DIR] agents_elements
[DIR] conf_files
[DIR] dependencies
[DIR] integrations
[DIR] vagrant_programs
[FILE] README
[FILE] agents.py
[FILE] command.py
[FILE] main.py
[FILE] remote.py
[FILE] requirements_python_VM.txt
[FILE] run_ssh_function.sh
[FILE] typography.py
[FILE] utils.py
[FILE] variables.py
```

Listing 4: Structure of the src folder

Some similarities exist such as requirements, utils, and variables, which serve the same purpose as their Vagrant counterparts (see listing 3).
However, note that many Python files among them define a class, let's define them hierarchically.

3.2 Top-Level Files

As we can see, these top-files permit setting up a rigorous bond between VMs and the host using Python. We then have 4 classes required before reaching our machines:
`Typography -> Command -> Agents -> Remote -> [VM's and their tools].`

Class	Description	Level
Typography.py	Base class for logging and harmonization. Automatically applies the `base_function` decorator to all methods and provides dynamic logging setup.	Top class
Command.py	A utility class for executing and managing shell commands. Automatically created via Agents and applies logging via Typography.	2nd
Agents.py	Superclass for managing different types of agents with logging and configuration capabilities. An agent corresponds to a VM.	3rd
Remote.py	Utility class for managing remote operations.	For VM's agents.
Run_ssh_function.sh	To run utils functions (.py or .sh) on a VM using Python.	Depends on Command class

Table 2: Top-Level Files and Their Descriptions

Without having a strict differentiation of each category, we can't expect great performance as we need:

- Clarity for debugging
- Low complexity to avoid latency
- Versatility to permit further improvement

Below is a summary on the workflow followed each time an Agent makes a move related to its associated virtual machine.

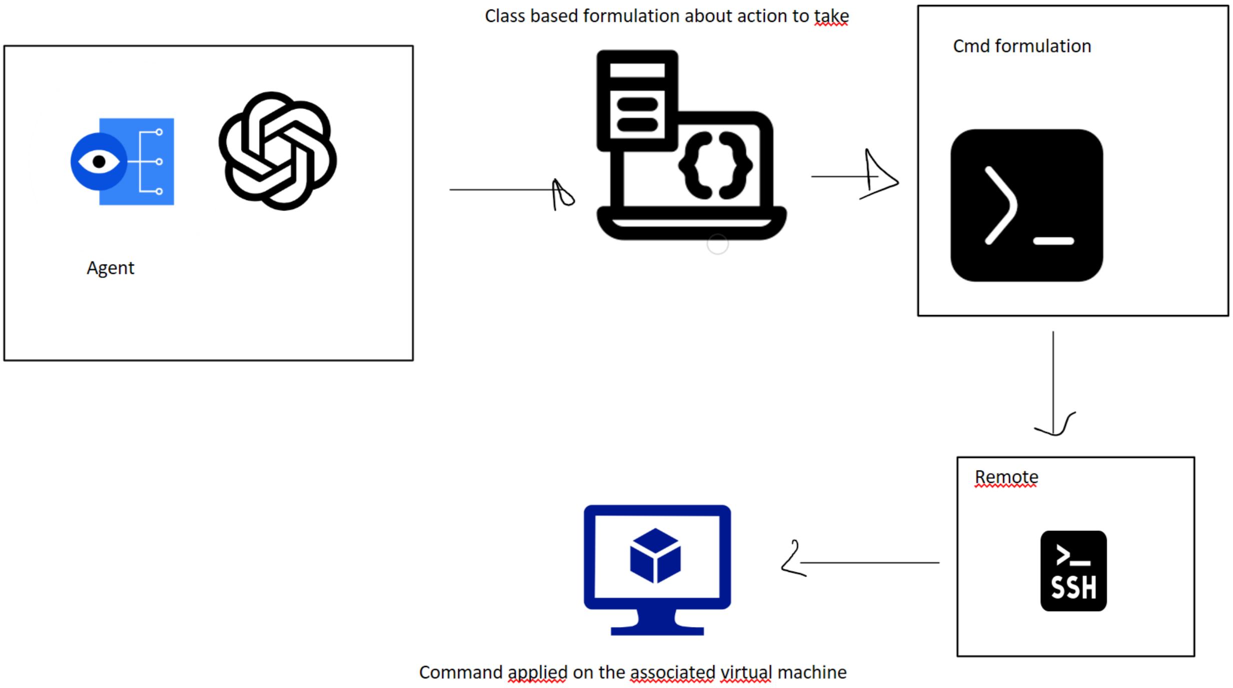

Class based formulation about action to take
Agent
Cmd formulation
Remote
SSH
Command applied on the associated virtual machine

Figure 9: Workflow between the class-based agent and its VM's homologue

3.3 VM Agents

We first had a look on how a VM agent may cooperate with the rest of its environment. We take a deeper insight on the structure of the agent itself.

3.3.1 VM Agents Structure

This directory contains the main elements to define our virtual machines. Not developing the subfolders, we get:

```
|--agent_types
|   |--attacker
|   |--defender
|   |--evaluator
|   \--manager
\--vmagents
    \--vmagent
```

3.3.2 Vmagents

This class will define main characteristics related to our virtual machines.

```
\--vmagents
    | log.py
    | ossec_conf.py
    | xml_handler.py
    | __init__.py
    \--vmagent
         fim.py
         vmagent.py
         yara.py
         __init__.py
```

This part is important and defines the core of the system. On the same level than `vm_agents` folder, the main motivations of these classes are:

- `Log.py`: To analyze logs sent by Wazuh for detecting and verification purposes.
- `Ossec_conf.py`: A utility class for managing `ossec.conf` configurations.
- `Xml_handler.py`: A utility class for managing XML configurations. This class provides methods to add group and decoder sections to an XML file, ensuring that duplicates are handled appropriately with warnings.

These classes are made in reference to the first part 1.2.1, as we sought to define a convenient way to build the two main files required by Wazuh, which are:

- `Localrules.xml`
- `Ossec.conf`

In the `vmagent` folder, we have:

- `Vmagent` class that by default imports every other class in this folder (modulable aspect).
- Latests correspond to a potential cyberfield as defined in here 1.2.1 and address adapted tools.

We detail the pseudo-code for the `vmagent` class:

Configuration of the VM agents attributes

```
Class Vmagent (inherits Fim, Yara):
    # Initializes the Vmagent instance
    Function __init__(kwargs):
        Extract agent type and name from kwargs
        Retrieve remote attributes (VM config path, remote name)
        Retrieve OSSEC config attributes (OSSEC paths)
```

Calling classes for every handled cyberfield

```
        # Call parent class constructor with computed attributes
        Call Fim.__init__(self)
        Call Yara.__init__(self)
        [... Any other cyberfield classes]
```

Those called classes are mainly inspired by these two links FIM[††] and Yara[‡‡].
`Yara` is the class that will simulate the main scenario, and it needs `FIM` to work.

- `FIM`: Allow file monitoring and helps the manager to identify file changes.
- `Yara`: An advanced detecting tool for malware that works in pair with Wazuh by analyzing file changes.

The objectives of these tools were covered in the first part A.2.

3.3.3 Onion View of a VM Agent

We can see this VM agent as the cell component of a broader complex system.
It has the possibility to:

- Get Wazuh logs
- Generate logs
- Use specific tools in a cyber-security context
- Generate configuration files

Indeed, it has the keys to:

- Apply remote commands on its associated virtual machine
- Synchronize files of its VM to the host

This conception put into practice every component adopted earlier and is the bedrock of any simulation or multi-role scenario.

[††] https://documentation.wazuh.com/current/user-manual/capabilities/file-integrity/index.html

[‡‡] https://documentation.wazuh.com/current/proof-of-concept-guide/detect-malware-yara-integration.html

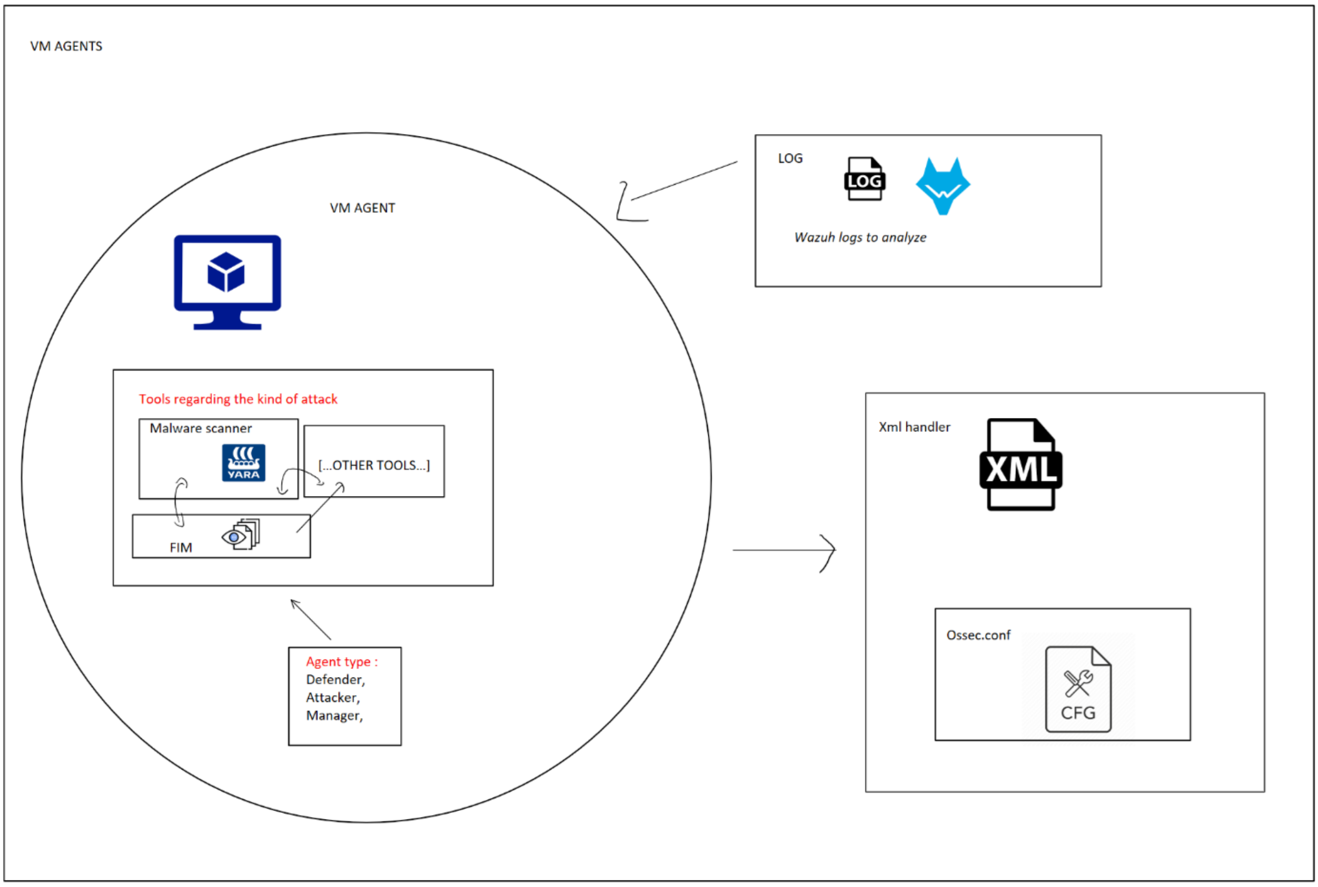


Figure 10: Onion View of a VM Agent to generate configuration files

3.3.4 Agent Types

```
|--attacker
|    attacker.py
|
|--defender
|    defender.py
|
|--evaluator
|    evaluator.py
|
\--manager
     manager.py
```

Agent types can be attacker, defender, evaluator, and manager.

- Evaluator isn't a virtual machine but a neutral entity that can be used for further purposes (ie : implementing a cost function regarding a proposed modification ...).
- Attacker, defender, and manager correspond to the roles of each virtual machines above and depend on `vmagents` classes.

These classes are the lowest developed classes in the hierarchy; they simply define the type of the entity with associated methods so that for an attacker, we would have:

```
from ...vmagents.vmagent.vmagent import Vmagent

class Attacker(Vmagent):
    def __init__(self, **kwargs):
        super().__init__(**kwargs, category=’attacker’)

    def perform_attack(self):
        pass

    def list_attacks(self):
        pass
```

Listing 5: Attacker class

A summary of the class architecture is available at the end of the thesis in figure 39.

3.4 Usage

3.4.1 Initializing Machines

Everything starts from the initialization code, this is where we define our elements with their features. We also define their actions and defense mechanisms.

```
# Manager Instance
manager = Manager(name=’wazidx1’, ip_address=’192.168.56.13’)

# Defender Instance
defender = Defender(name=’wazagent1’, ip_address=’192.168.56.14’)
```

Listing 6: Agents definitions

In this way, it becomes simple to run scripts or commands for each instance via a simple line command.

3.4.2 Remote.py

Attribute	**Description**
self.remote_name	Name of the Vagrant VM used for SSH and remote actions
self.VM_conf_path	Path on the VM to the main configuration file (default: /var/ossec/etc/ossec.conf)
self.VM_local_rules_path	Path to the local rules file inside the VM
self.VM_local_decoder_path	Path to the local decoder file inside the VM

Table 3: Attributes of Remote.py

These attributes are essential for the SSH connection to run without issues on the associated VM.

A particular attention is allowed to the following methods:

Method	**Description**
`run_remote_command`	Runs a remote command
`run_bash_script_on_remote_host`	Runs a bash script on a remote host

Table 4: Methods of Remote.py

They allow executing bash scripts on our machines.

For instance:

```
# Step 1: Defender downloads the fake malware script
print("Defender downloading the fake malware script...")

# Install YARA on the remote host
bash_script_name = 'copy_malware.sh'
defender.run_bash_script_on_remote_host(bash_script_name)
\end{verbatim}

In this way, the bash script will be executed on the remote machine (defender).
Those remote functions were already tested during the provisioning \ref{provisionning_test}
```

Listing 7: Main remote's class methods

3.4.3 XML Handler

Main attributes are:

Attribute	**Description**
self.xml_path	Path to the XML file
self.vm_local_rules_path	Path to the local rules file inside the VM
self.vm_local_decoder_path	Path to the local decoder file inside the VM

Table 5: Attributes of XML Handler

Main methods to consider are:

Method	**Description**
`add_group_to_xml`	Adds a group to the XML file
`add_decoder_to_xml`	Adds a decoder to the XML file
`synchronize_xml_with_vm`	Synchronizes the XML file with the VM

Table 6: Methods of XML Handler

In particular, the last method ensures that the XML file is syntactically correct. If errors occur, no changes are made.

3.4.4 Purpose of XML Files

The purpose of these XML files, files that we discussed here, is to define custom rules and decoders for Wazuh.

- `Localrules.xml`: Contains custom rules that are specific to the environment.
- `Ossec.conf`: Main configuration file for Wazuh.

3.4.5 Local Rules

Based on Wazuh's documentation:
"Custom rules in Wazuh allow users to define specific conditions or patterns in log data that are relevant to their unique environment, applications, or security requirements."
While Wazuh comes with a set of default rules, these are custom and added by group tags. Groups help you label and organize rules by functionality (sshd, syslog, authentication_failed, etc.) and makes rules easier to manage and understand.

To add a new set of rules, we:

- Add a group tag in the local_rules file that links rule IDs to their description.
- Download the associated rules as a folder.
- Define a decoder if necessary (used for specific rules).
- Test the overall system.

Taking the following example:

```
<group name="local,syslog,sshd,">
    <rule id="100001" level="5">
        <if_sid>5716</if_sid>
        <srcip>1.1.1.1</srcip>
        <description>sshd: authentication failed from IP 1.1.1.1.</description
    >
        <group>authentication_failed,pci_dss_10.2.4,pci_dss_10.2.5,</group>
    </rule>
</group>
```

Listing 8: Group name example

Tag	**Purpose**
`<group name="...">`	Groups a set of rules into logical categories
`<rule id="..." level="...">`	Defines a rule with a unique ID and severity level
`<if_sid>`	Links this rule to a previously triggered rule
`<srcip>`	Limits rule to a specific source IP
`<description>`	Provides a clear explanation of the alert
`<group> (inside rule)`	Classifies the rule itself (e.g., type, compliance tags)

Table 7: Tags and Their Purposes in Local Rules

In that way, `add_group_to_xml` as discussed with table 6 will add a group tag to the XML file ensuring that this group doesn't already exist. If it does, it adds only the new rules (ID). This is a versatile approach that allows an agent to add rules content without having the fear of conflicts due to redundancy.
We retrieve the rules in the dashboard once they are triggered by their associated action.

https://documentation.wazuh.com/current/user-manual/ruleset/rules/custom.html

3.4.6 Decoder

As mentioned, we can also have to handle the decoder. based on this documentation.
A **decoder** is a component in Wazuh that:

- Parses raw log lines.
- Breaks them into **structured fields** (like srcip, dstip, user, etc.).
- Makes those fields usable by rules.

Without a decoder, rules can't match specific parts of a log line because the log would be treated as one big string.
Example:

```
<decoder name="example">
    <program_name>^example</program_name>
</decoder>

<decoder name="example">
    <parent>example</parent>
    <regex>User '(\w+)' logged from '(\d+\.\d+\.\d+\.\d+)'</regex>
    <order>user, srcip</order>
</decoder>
```

Listing 9: Decoder example

We're building a **decoder chain**:

1. The first decoder detects logs that come from a program called example.
2. The second decoder extracts specific fields (user and srcip) from the message content using a regex.

Example log line this decoder would match: `User 'admin' logged from '192.168.1.100'`.
We use `/var/ossec/bin/wazuh-logtest` to verify that the decoder is working.

3.4.7 Ossec.py

This class is directly linked to the XML one. It uses its tools to allow configuration of the `ossec.conf` and main attributes are:

Attribute	**Description**
self.ossec_conf_path	Path of the ossec.conf file on the VM
self.base_conf_path	Path to a “basis” for this ossec.conf with scripts that never change
self.excel_conf_path	Path to an Excel conversion of the configuration file

Table 8: Attributes of Ossec.py

https://documentation.wazuh.com/current/user-manual/ruleset/decoders/custom.html

3.4.8 Organization of Ossec.conf File

From Wazuh's documentation:
"The ossec.conf file is the main configuration file on the Wazuh manager, and it also plays an important role on the agents. It is located at /var/ossec/etc/ossec.conf both in the manager and agent on Linux machines."
The main tags to consider in case of a modification are:

- `Commands Settings`
  - `<command>`: Contains command configuration settings.
  - `<name>`: Specifies the name of the command.
  - `<executable>`: Specifies the executable for the command.
- `Ruleset Settings`
  - `<ruleset>`: Contains ruleset configuration settings.
  - `<group>`: Specifies the group for the ruleset.
  - `<decoder>`: Specifies the decoder for the ruleset.
- `Active Response Settings`
  - `<active-response>`: Contains active response configuration settings.
  - `<command>`: Specifies the command for the active response.
  - `<rules_id>`: Specifies the rule IDs for the active response.
- `Localfile Settings`
  - `<localfile>`: Contains local file configuration settings.
  - `<log_format>`: Specifies the log format for the local file.
  - `<location>`: Specifies the location of the local file.

These tags cover a wide range of configurations needed to integrate a new feature into the `ossec.conf` file. Depending on the specific feature, additional tags and settings may be required. To avoid any conflict, each time a modification is done which implies adding an active response, localfile, or new commands, the system verifies to:

- Avoid any duplicates
- If a similar tag is added, it adds only extra attributes
- Restructure in the right order the file after modification
- "Compress" tags by putting in the same row all rules IDs, paths, and so more.

To compress any tags, it is necessary to include it in:

```
TAG_TO_COMPRESS = {
    'syscheck': {'tags': ['ignore', 'directories']},
    'global': {'tags': ['white_list']},  # NOT USED
}
```

Compressing it will add all paths on the same line:

```
<list>etc/lists/amazon/aws-eventnames,etc/lists/audit-keys,..</list>
```

https://documentation.wazuh.com/current/user-manual/reference/ossec-conf/index.html

3.4.9 Methods and Structure of the Class

Main methods to consider are:

Method	Description
`add_section_tag_to_conf_file`	Adds a section tag to the configuration file
`synchronize_df_with_conf_file`	Synchronizes the dataframe with the configuration file
`verify_conf_file_via_bash`	Verifies the configuration file via bash

Table 9: Main Methods of Ossec.py

To add a section tag, the following instructions are necessary:

```python
active_response_to_add = """
<active-response>
    <command>isolate_infected_system</command>
    <location>local</location>
    <rules_id>100003</rules_id>
</active-response>
"""

# Step 3: Manager updates the ossec.conf and local_rules.xml
print("Manager updating ossec.conf and local_rules.xml...")
manager.add_section_tag_to_conf_file(active_response_to_add)
```

Listing 10: Example of an active response

> **To Go Further**
>
> We can implement a same process for `agent.conf` files on each VM to add extra custom rules.

Summurizing Ossec.py in one picture :

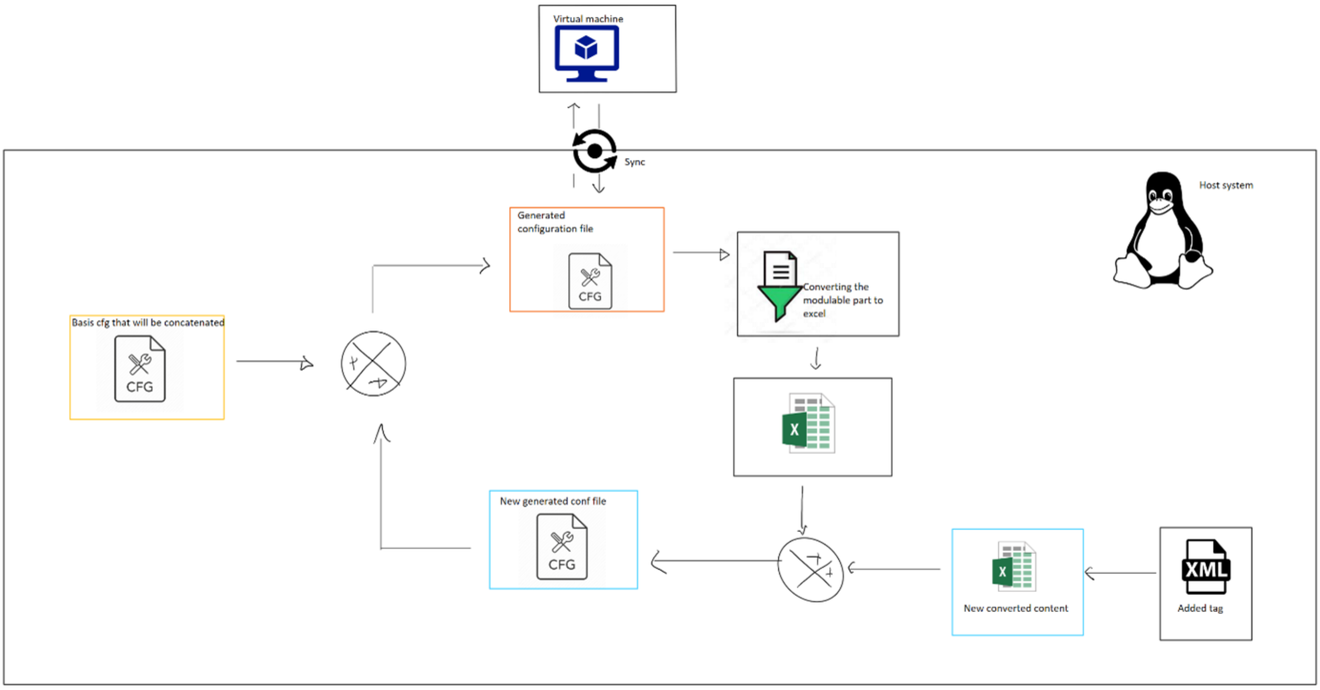

Virtual machine
Sync
Host system
Generated configuration file
CFG
Converting the modulable part to excel
Basis cfg that will be concatenated
CFG
New generated conf file
CFG
New converted content
XML
Added tag

Figure 11: Principle of Ossec.py

In one line, the system will:

- Recognize the type of tag,
- Add it gracefully,
- Synchronize the host version with the manager version — if the new ossec.conf is receivable.

This core class is essential, relying on the strategic idea of applying a transformation from a configuration file to its corresponding Excel representation.

To keep transformations manageable, the final configuration file is built by combining a generated configuration with a predefined base, as mentioned in 3.4.7.

With the ability to modify key configuration files — namely local_rules, decoders, and ossec.conf as presented in subsection 1.2.1 — we are now equipped to address our first cybersecurity focus: **File Integrity Monitoring**.

3.4.10 File Monitoring Class

What is File Monitoring?

According to the Wazuh's documentation, file integrity monitoring aims at monitoring specific files or folders to highlight any changes. Any modifications will then be logged and mentioned on the dashboard.
Main methods are:

Method	**Description**
check_fim_change	Checks for file integrity changes
configure_audit	Configures audit settings
manage_ip_in_hosts_allow	Manages IP in hosts allow
simulate_file_change	Simulates file change
test_who_data	Tests who owns data

Table 10: Main Methods of File Monitoring Class

This allows configuring the file monitoring system and to get enhanced logs about modification, we also need to configure who_data and audit.
Those tools provide more information on who did the changes.

Example : Let's say we want to monitor two specific folders and check for consistency; the following test will do it.
We need to:

- Configure ossec.conf
- Modify a file in the folder as another user
- Check for file change locally
- Check if Wazuh has detected it.

https://documentation.wazuh.com/current/user-manual/capabilities/file-integrity/index.html
https://documentation.wazuh.com/current/user-manual/capabilities/file-integrity/advanced-settings.html

Step 1: Define directory tags

```
# Define directory tags
directory_tag_wdata = {'text': monitored_path_wdata, 'check_all': 'yes', ...}
directory_tag = {'text': monitored_path, 'check_all': 'yes', 'whodata': 'yes'}
```

Step 2: Initialize Manager and Defender instances

```
# -------------------- INITIALIZATION --------------------
# Manager Instance
manager = Manager(name='wazidx1', ip_address='192.168.56.13')
# Defender Instance
defender = Defender(name='wazagent1', ip_address='192.168.56.14')
```

Step 3: Add FIM Configuration

```
print("Starting FIM setup...")
# Add FIM configuration
defender.add_fim_configuration(
    monitored_paths=[directory_tag, directory_tag_wdata],
    do_synchronize_with_VM=True,
    conf_path=PATH_TO_OSSEC_AGENT_BASE_DEFENDER
)
```

Step 4: Test FIM detection

```
# Test FIM
print("Testing FIM...")
defender.run_function_on_remote_host('simulate_file_change', monitored_path)
manager.run_function_on_remote_host('check_fim_change', monitored_path)
```

Step 5: Test Who-Data tracking

```
# Test Who-Data
print("Testing audit...")
defender.run_function_on_remote_host('test_who_data')
manager.run_function_on_remote_host('check_fim_change', monitored_path_wdata)

print("FIM and Who-Data setup completed successfully.")
```

3.4.11 Yara.py

The `Yara` class is designed to enhance malware detection using YARA integration and is common to every VM agent. Designed inspired by its related documentation.

The YARA Active Response module scans new or modified files whenever the Wazuh FIM module triggers an alert.
This modulable malware detector would recognize new detected threats once the signature of it is added. It needs every component presented before in figure 10 : Ossec, Xml and Fim classes.

https://documentation.wazuh.com/current/proof-of-concept-guide/detect-malware-yara-integration.html#detecting-malware-using-yara-integration

To configure `Yara`, we write simply:

Step 1: Start YARA setup

```
print("Starting YARA setup...")

# Configure audit on the manager
manager.configure_audit()
```

Step 2: Install YARA on the defender

```
# Add YARA configuration on the defender
defender.check_and_install_yara(conf_path=PATH_TO_OSSEC_AGENT_BASE_DEFENDER)
```

Step 3: Prepare and synchronize XML sections

```
# Configure manager with XML sections
group_section_str = """...
"""
decoder_section_str = """...
"""
Command_tag = """ ... """
manager.synchronize_xml_with_VM(decoder_sections_str, group_sections_str)
```

Step 4: Modify configuration dataframes

```
# -------------------- TEST 3: MODIFY DATAFRAMES --------------------
print("Loading existing ossec.conf configuration...")
dict_dfs = manager.dataframe_from_conf(file_path)

print("Modifying configuration dataframes...")
# Ossec Section
Section_tag = """ ... """
manager.add_section_tag_to_conf_file(section_tag)
```

Step 5: Generate new configuration file

```
# -------------------- TEST 4: GENERATE NEW CONFIGURATION --------------------
print("Updating...")
manager.synchronize_df_with_conf_file(dict_dfs, conf_path)
```

Step 6: Simulate the attack and finish

```
# Processing the attack
defender.run_bash_script_on_remote_host(NAME_MALWARE_SCRIPT)

# -------------------- TEST COMPLETED --------------------
print("All tests completed successfully!")
```

This kind of script demonstrates how **easily a new tool can be configured** across multiple entities to address a specific cybersecurity domain — in this case, **File Integrity Monitoring (FIM)**.

By **grouping these methods under a structured header**, a large language model (LLM) can autonomously generate the corresponding configuration script without difficulty.
We will explore in the **practical section** 5 how, in this scenario, the LLM is capable of *recognizing new threats* in a straightforward way and **reconfiguring the system automatically**.
A simple way to generate such a header for the LLM — to help it understand the software's configuration — is to use the following function:

```
save_hierarchy_to_file(Manager, 'methods.txt')
```

This function produces a hierarchical tree that organizes all available methods (*see tab 11* in appendix), which can then be used directly as an header for the LLM.
Below is the ultimate scenario that is simulated on the `main.py` script:

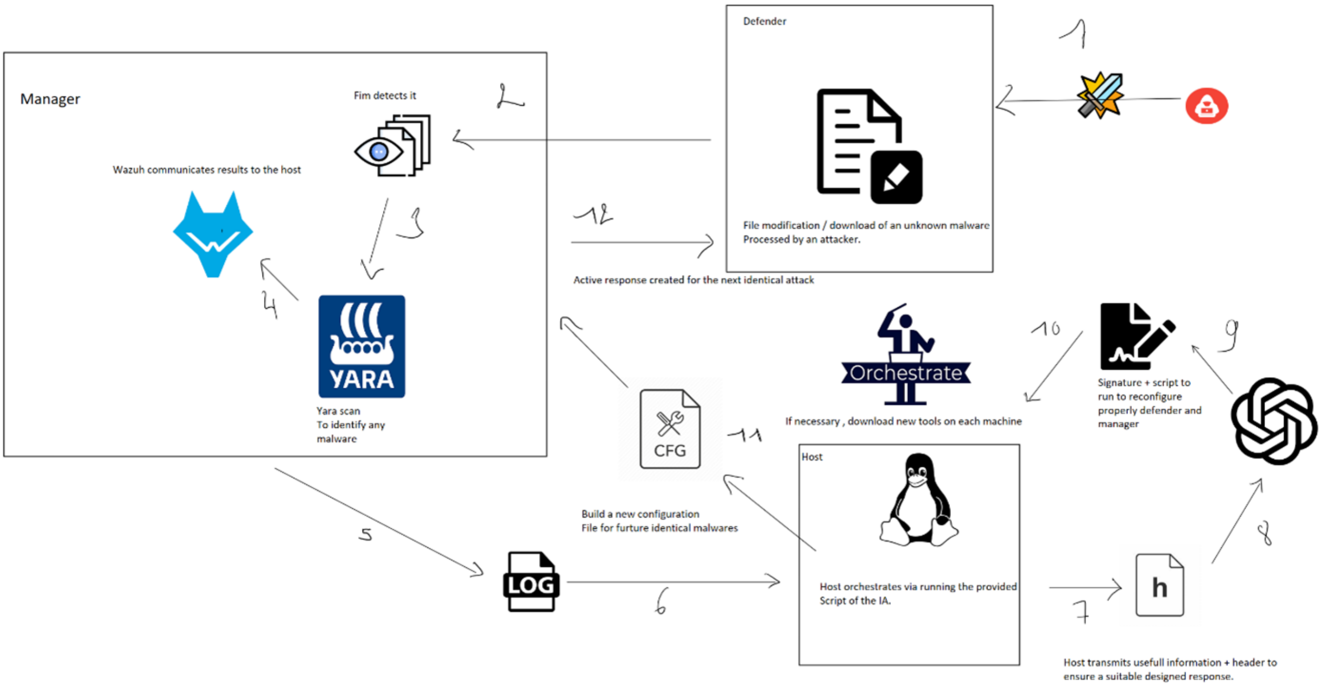

Defender
1
2
File modification / download of an unknown malware
Processed by an attacker.
Manager
Fim detects it
2
Wazuh communicates results to the host
3
4
YARA
Yara scan
To identify any
malware
12
Active response created for the next identical attack
Orchestrate
If necessary , download new tools on each machine
10
Signature + script to
run to reconfigure
properly defender and
manager
9
CFG
11
Build a new configuration
File for furture identical malwares
Host
5
LOG
6
Host orchestrates via running the provided
Script of the IA.
7
h
8
Host transmits usefull information + header to
ensure a suitable designed response.

Figure 12: Ultimate Scenario

Conclusion

As a conclusion, the architecture of the system helps to achieve:

- An elaborate architecture that progresses from low-level to high-level comprehension.
- An easy-to-set-up virtual machine (VM) system.
- A modular configuration of critical files (OSSEC and local files).

These concepts allow for a straightforward approach to address various cyber fields, and in our particular case: *malware detection*. By using file integrity monitoring coupled with YARA, we were able to design a simple scenario.
This scenario configures machines and indicates malware detection, ultimately generating an active response.

To Go Further

Consider the following points for further exploration:

- The possibility of configuring machines analogously for other cyber situations.
- The capability to generate a header to feed to a Large Language Model (LLM) each time a significant modification of the system is made.
- The necessity to clearly define the system's scope of action and its ability to modify the code.

The following sections are the most significant, serving as concrete evidence of both the theoretical and practical explanations provided earlier.

- We proceed with the analysis of the two phases previously defined in Sections 2 and 3.
- Command outputs are examined and cross-referenced with information displayed on the Wazuh dashboard to ensure consistency and verify authenticity.

4 Provisioning of the vagrant environment

The environment is quite tedious to set up and can take hours if you're not used to a Linux environment.

To install Vagrant and VirtualBox, guidelines are indicated in the README.
It is important to find a matching version between Vagrant, VirtualBox, and the mainline kernel.

- VirtualBox generates the VM.
- Vagrant orchestrates this generation via a program.

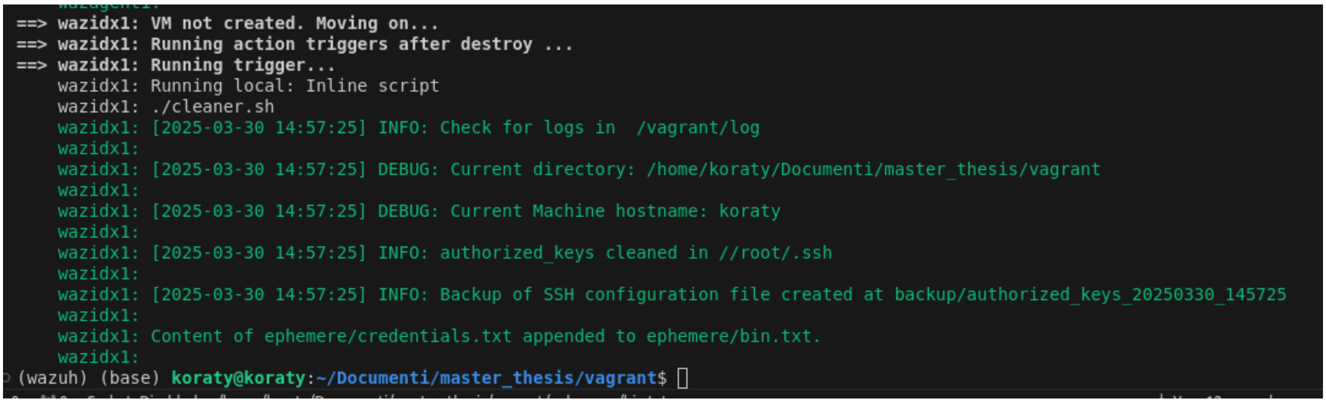
Our configuration is:

- Vagrant version 2.4.3
- VirtualBox version 7.1
- Kernel version 8.6.51

Once the host is configured, we need to ensure that the environment is cleared from previous configurations.

4.1 Cleaner

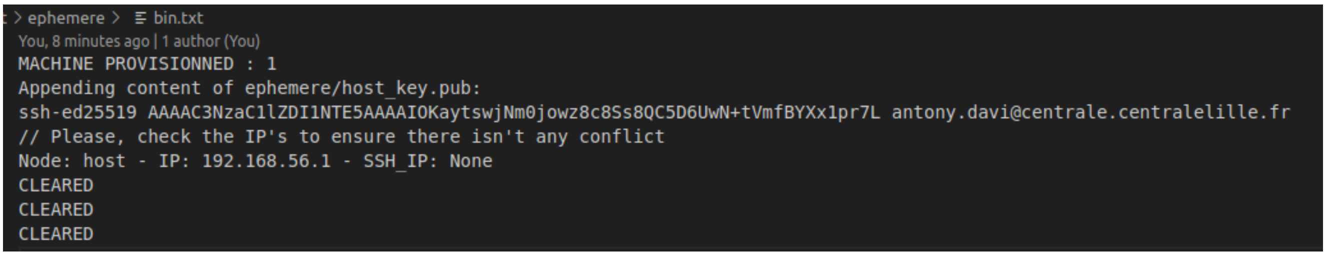
To do so: `sudo vagrant destroy`
More than destroying existing VMs, it will also execute `cleaner.sh`.



```
==> wazidx1: VM not created. Moving on...
==> wazidx1: Running action triggers after destroy ...
==> wazidx1: Running trigger...
    wazidx1: Running local: Inline script
    wazidx1: ./cleaner.sh
    wazidx1: [2025-03-30 14:57:25] INFO: Check for logs in  /vagrant/log
    wazidx1:
    wazidx1: [2025-03-30 14:57:25] DEBUG: Current directory: /home/koraty/Documenti/master_thesis/vagrant
    wazidx1:
    wazidx1: [2025-03-30 14:57:25] DEBUG: Current Machine hostname: koraty
    wazidx1:
    wazidx1: [2025-03-30 14:57:25] INFO: authorized_keys cleaned in //root/.ssh
    wazidx1:
    wazidx1: [2025-03-30 14:57:25] INFO: Backup of SSH configuration file created at backup/authorized_keys_20250330_145725
    wazidx1:
    wazidx1: Content of ephemere/credentials.txt appended to ephemere/bin.txt.
    wazidx1:
(wazuh) (base) koraty@koraty:~/Documenti/master_thesis/vagrant$ 
```

Figure 13: Output of the cleaner script showing the cleanup process.

Every previous information will be thrown to `bin.txt`, where a CLEARED tag will be written so that from the next `sudo vagrant up`, the cleaner won't be executed anymore :

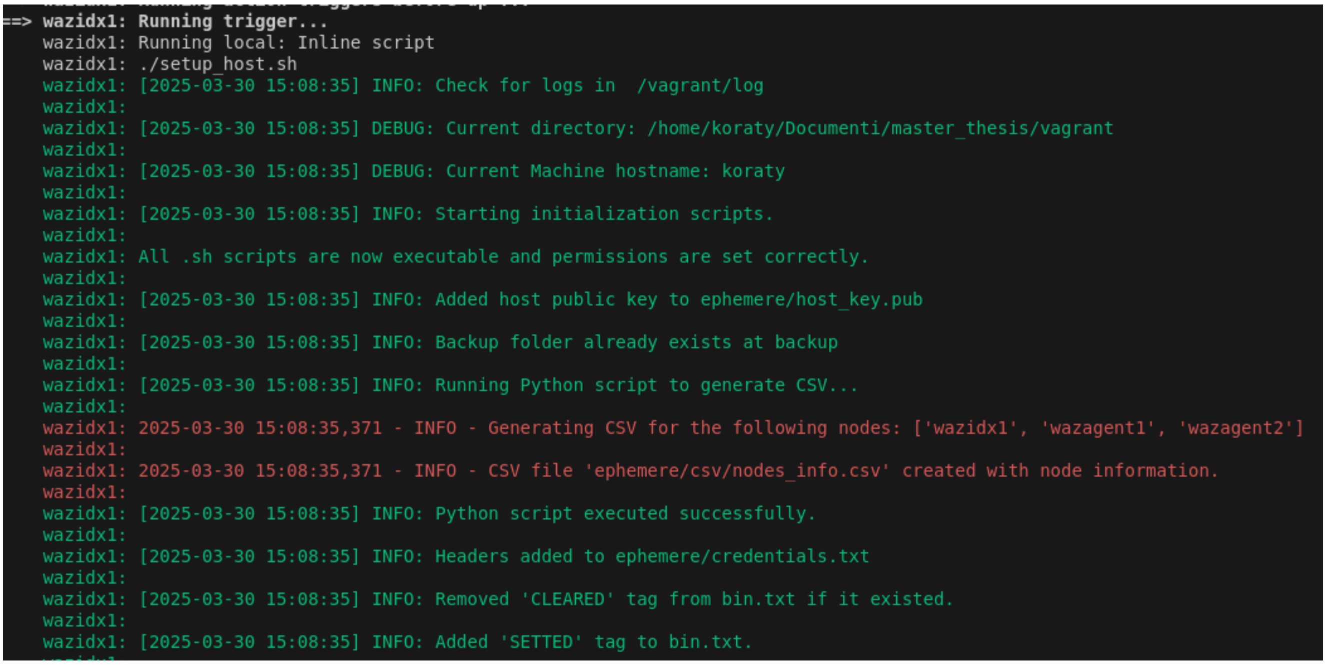


```
> ephemere > bin.txt
You, 8 minutes ago | 1 author (You)
MACHINE PROVISIONNED : 1
Appending content of ephemere/host_key.pub:
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIOKaytswjNm0jowz8c8Ss8QC5D6UwN+tVmfBYXx1pr7L antony.davi@centrale.centralelille.fr
// Please, check the IP's to ensure there isn't any conflict
Node: host - IP: 192.168.56.1 - SSH_IP: None
CLEARED
CLEARED
CLEARED
```

Figure 14: Content of bin.txt after cleanup.

4.2 Setup

We can then run a `sudo vagrant up`. As the environment was cleared, the system will understand that no machine was provisioned and will execute `setup.sh` as a first step (initial script type 2.2) to ensure executability of programs, folders existences ...

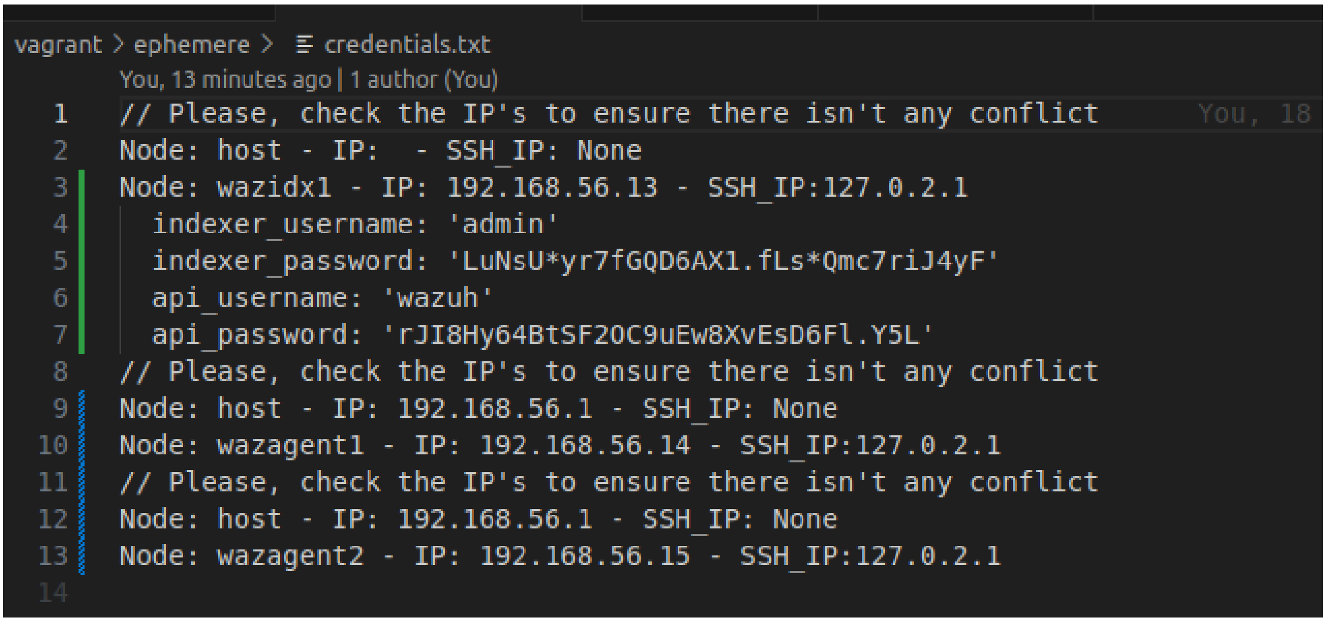


```
==> wazidx1: Running trigger...
    wazidx1: Running local: Inline script
    wazidx1: ./setup_host.sh
    wazidx1: [2025-03-30 15:08:35] INFO: Check for logs in  /vagrant/log
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] DEBUG: Current directory: /home/koraty/Documenti/master_thesis/vagrant
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] DEBUG: Current Machine hostname: koraty
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Starting initialization scripts.
    wazidx1:
    wazidx1: All .sh scripts are now executable and permissions are set correctly.
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Added host public key to ephemere/host_key.pub
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Backup folder already exists at backup
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Running Python script to generate CSV...
    wazidx1:
    wazidx1: 2025-03-30 15:08:35,371 - INFO - Generating CSV for the following nodes: ['wazidx1', 'wazagent1', 'wazagent2']
    wazidx1:
    wazidx1: 2025-03-30 15:08:35,371 - INFO - CSV file 'ephemere/csv/nodes_info.csv' created with node information.
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Python script executed successfully.
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Headers added to ephemere/credentials.txt
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Removed 'CLEARED' tag from bin.txt if it existed.
    wazidx1:
    wazidx1: [2025-03-30 15:08:35] INFO: Added 'SETTED' tag to bin.txt.
```

Figure 15: Output of the setup script showing initialization steps.

Comment: As written above, this will ensure the existence of necessary folders for further steps, and if not existent, it will create them. It will also add a SETTED tag to the `bin.txt` file to indicate that the setup was done before provisioning the first machine.

After this step, provisioning initiates, and a shortened version of all the logs during the execution is accessible in the Appendix C.3.



vagrant > ephemere > credentials.txt
You, 13 minutes ago | 1 author (You)
You, 18

```
// Please, check the IP's to ensure there isn't any conflict
Node: host - IP:  - SSH_IP: None
Node: wazidx1 - IP: 192.168.56.13 - SSH_IP:127.0.2.1
  indexer_username: 'admin'
  indexer_password: 'LuNsU*yr7fGQD6AX1.fLs*Qmc7riJ4yF'
  api_username: 'wazuh'
  api_password: 'rJI8Hy64BtSF2OC9uEw8XvEsD6Fl.Y5L'
// Please, check the IP's to ensure there isn't any conflict
Node: host - IP: 192.168.56.1 - SSH_IP: None
Node: wazagent1 - IP: 192.168.56.14 - SSH_IP:127.0.2.1
// Please, check the IP's to ensure there isn't any conflict
Node: host - IP: 192.168.56.1 - SSH_IP: None
Node: wazagent2 - IP: 192.168.56.15 - SSH_IP:127.0.2.1
```

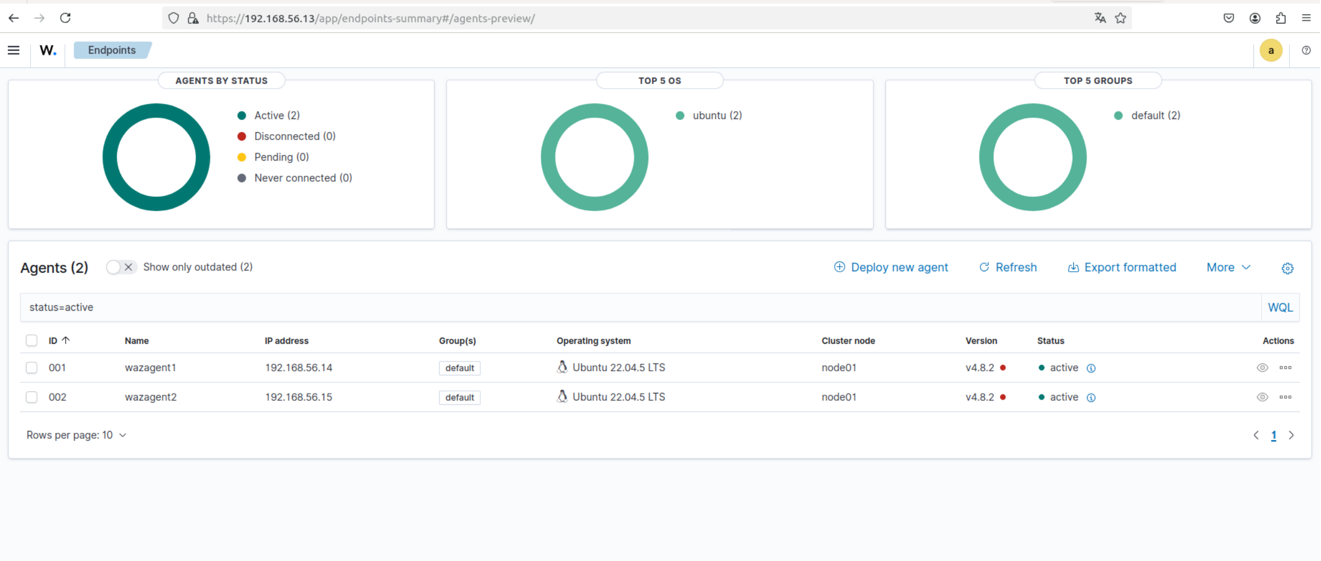
Figure 16: Content of credentials.txt with login information.

We can have a look at the `credentials.txt` that reunites every necessary information for login

IP addresses or SSH IP of every entity.

4.3 Wazuh Dashboard

As initial scripts occured and installed wazuh on the manager and endpoints (mentioned here 2.2.2), we are now ready to have a look to the *Wazuh Dashboard*. This one can be reached using *https://IPmanager*.
After logged in using `credentials.txt` , we can indeed see the presence of a manager and two agents:



status=active

ID ↑	Name	IP address	Group(s)	Operating system	Cluster node	Version	Status	Actions
001	wazagent1	192.168.56.14	default	Ubuntu 22.04.5 LTS	node01	v4.8.2 ●	● active	
002	wazagent2	192.168.56.15	default	Ubuntu 22.04.5 LTS	node01	v4.8.2 ●	● active	

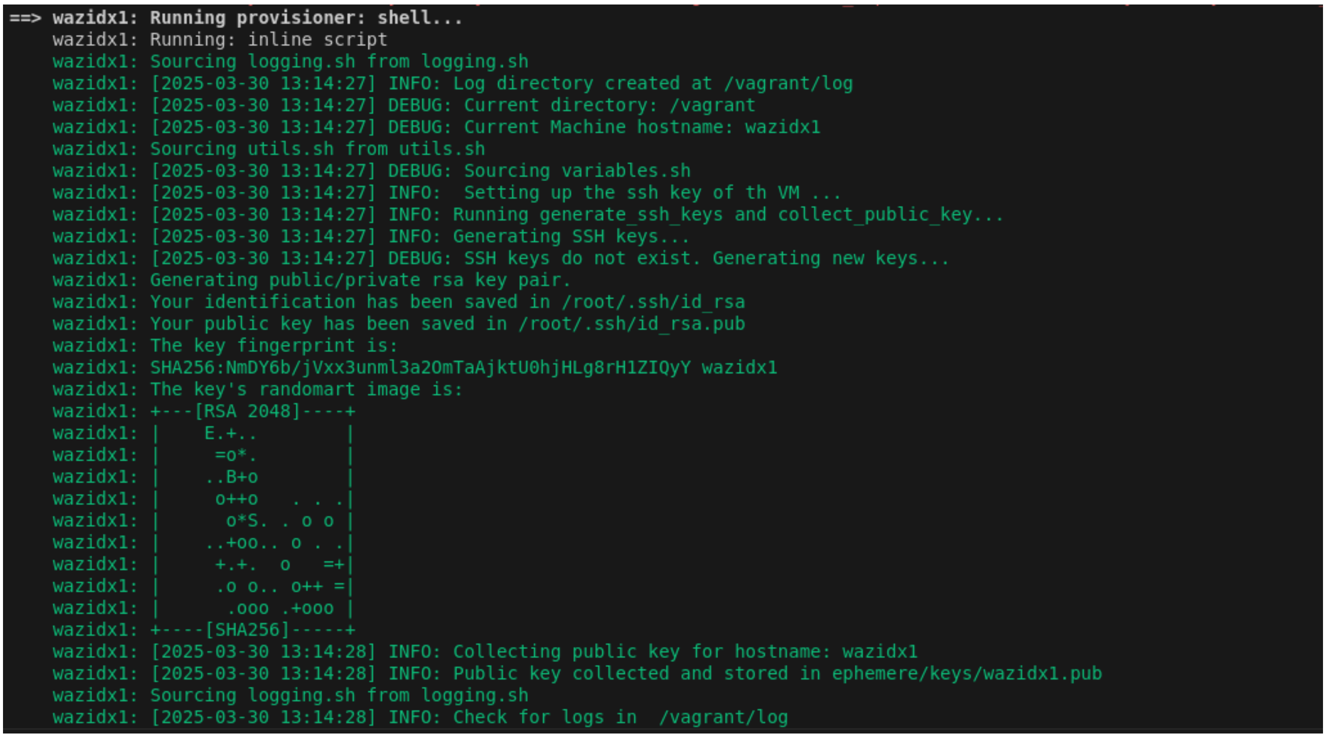
This dashboard gives a visual aspect of ongoing scenarios between elements , attacks - active responses or warning and further more.

4.4 SSH Keys

One crucial aspect is the effective working of SSH connections between the host and machines.
We saw above an example of an SSH connection from the host (home user) to the virtual machine. However, the most used SSH connection is from the root host to the root user on the VM and this will permit:

- Full control of the VM from an IA agent.
- Connect via VSCode to do code modifications easily.

As this procedure undergoes on each VM, below is an example of the parameterization of a key on the server:
Comment: We can see the key created and copied to the host system to be handled at the end of the provisioning to allow us to connect to each machine via VsCode as presented in the Appendix C.3.1.



```
==> wazidx1: Running provisioner: shell...
    wazidx1: Running: inline script
    wazidx1: Sourcing logging.sh from logging.sh
    wazidx1: [2025-03-30 13:14:27] INFO: Log directory created at /vagrant/log
    wazidx1: [2025-03-30 13:14:27] DEBUG: Current directory: /vagrant
    wazidx1: [2025-03-30 13:14:27] DEBUG: Current Machine hostname: wazidx1
    wazidx1: Sourcing utils.sh from utils.sh
    wazidx1: [2025-03-30 13:14:27] DEBUG: Sourcing variables.sh
    wazidx1: [2025-03-30 13:14:27] INFO:  Setting up the ssh key of th VM ...
    wazidx1: [2025-03-30 13:14:27] INFO: Running generate_ssh_keys and collect_public_key...
    wazidx1: [2025-03-30 13:14:27] INFO: Generating SSH keys...
    wazidx1: [2025-03-30 13:14:27] DEBUG: SSH keys do not exist. Generating new keys...
    wazidx1: Generating public/private rsa key pair.
    wazidx1: Your identification has been saved in /root/.ssh/id_rsa
    wazidx1: Your public key has been saved in /root/.ssh/id_rsa.pub
    wazidx1: The key fingerprint is:
    wazidx1: SHA256:NmDY6b/jVxx3unml3a2OmTaAjktU0hjHLg8rH1ZIQyY wazidx1
    wazidx1: The key's randomart image is:
    wazidx1: +---[RSA 2048]----+
    wazidx1: |    E.+..        |
    wazidx1: |     =o*.        |
    wazidx1: |    ..B+o        |
    wazidx1: |     o++o   . . .|
    wazidx1: |      o*S. . o o |
    wazidx1: |    ..+oo.. o . .|
    wazidx1: |     +.+.  o   =+|
    wazidx1: |     .o o.. o++ =|
    wazidx1: |      .ooo .+ooo |
    wazidx1: +----[SHA256]-----+
    wazidx1: [2025-03-30 13:14:28] INFO: Collecting public key for hostname: wazidx1
    wazidx1: [2025-03-30 13:14:28] INFO: Public key collected and stored in ephemere/keys/wazidx1.pub
    wazidx1: Sourcing logging.sh from logging.sh
    wazidx1: [2025-03-30 13:14:28] INFO: Check for logs in  /vagrant/log
```

Figure 17: SSH key generation process.

In this way, we can also have a greater look at the disk organization of each VM, in particular, we highlight the presence of the shared Vagrant folder between each entity.

To Go Further

While the Vagrant folder contains all the running code, a critical issue persists: it grants virtual machines unrestricted access to this code.
As a priority, complementary work should address managing VM access to specific files or folders, ensuring that only designated resources can be modified while others remain protected

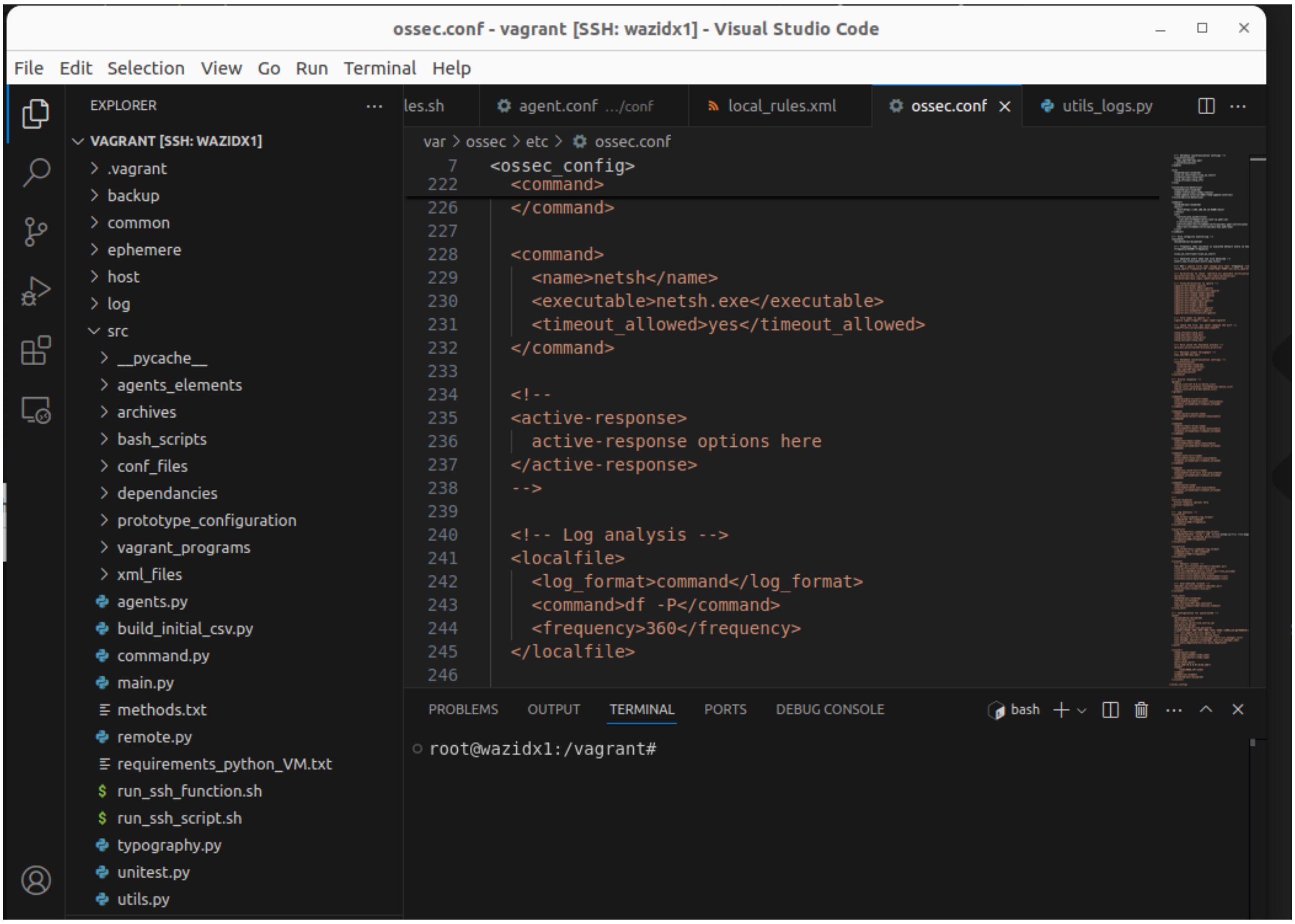
4.5 Tests & Scenario

Above a running manager and its agents, the provisioning also defines some tests to mention possible issues.

4.5.1 Testing suricata and ssh connections

We test ssh connections between host and machine , terminal logs results are visible in Appendix C.3.2.

The provisionning also installs Suricata, we do an Nmap scan to ensure its correct functioning:



```
<ossec_config>
  <command>
  </command>

  <command>
    <name>netsh</name>
    <executable>netsh.exe</executable>
    <timeout_allowed>yes</timeout_allowed>
  </command>

  <!--
  <active-response>
    active-response options here
  </active-response>
  -->

  <!-- Log analysis -->
  <localfile>
    <log_format>command</log_format>
    <command>df -P</command>
    <frequency>360</frequency>
  </localfile>
```

```
root@wazidx1:/vagrant#
```

Figure 18: Vagrant folder (on the left) accessible from `Wazidx1`

Nmap Scan initiation

```
Starting Nmap 7.80 ( https://nmap.org ) at 2025-03-30 14:22 UTC
Nmap scan report for wazagent2 (192.168.56.15)
Host is up (0.000046s latency).
Not shown: 3304 closed ports

PORT     STATE SERVICE
22/tcp   open  ssh
80/tcp   open  http

Nmap done: 1 IP address (1 host up) scanned in 6.12 seconds
```

Nmap Scan Output

```
Starting Nmap 7.80 ( https://nmap.org ) at 2025-03-30 14:22 UTC
Nmap scan report for wazagent2 (192.168.56.15)
Host is up (0.000046s latency).
All 3306 scanned ports on wazagent2 (192.168.56.15) are unfiltered

Nmap done: 1 IP address (1 host up) scanned in 6.13 seconds
```

Comment: The Nmap scan successfully identified open ports on the target system, including SSH and HTTP services. The scan results indicate that the network configuration is working correctly and that the Suricata installation is running. The latency and response times are within acceptable ranges, confirming the efficiency of the network setup.

4.5.2 Wazuh scenario

After connection between host and machine have been tested , we define three scenario where the principle is to address an attack from agent2 towards agent1.
The manager needs to spot it and generates an active response (blocking the IP) once the attack is detected.

During the provisioning, 3 scenarios are executed as mentioned above: *Alienvault* , *Bruteforce*, and *DVWA*.

For convenience, we focus only on the brute force scenario.

Installation & Configuration

```
[2025-03-30 17:28:48] DEBUG: crunch installed successfully.
[2025-03-30 17:28:48] DEBUG: Generating password list using crunch...
[2025-03-30 17:28:51] DEBUG: Password list generated successfully.
```

Checking Reachability of the Victim

```
[2025-03-30 17:28:51] DEBUG: Checking reachability of 192.168.56.14...
PING 192.168.56.14 (192.168.56.14) 56(84) bytes of data.
64 bytes from 192.168.56.14: icmp_seq=4 ttl=64 time=0.638 ms
--- 192.168.56.14 ping statistics ---
4 packets transmitted, 1 received, 75% packet loss, time 3174ms
rtt min/avg/max/mdev = 0.638/0.638/0.638/0.000 ms
[2025-03-30 17:29:04] DEBUG: 192.168.56.14 is reachable.
```

Attacking the Victim

```
[2025-03-30 17:29:04] DEBUG:Starting SSH brute force attack using hydra
Hydra v9.2 (c) 2021 by van Hauser/THC & David Maciejak
Hydra starting at 2025-03-30 17:29:04
[DATA] attacking ssh://192.168.56.14:22/
[STATUS] 20.00 tries/min, 20 tries in 00:01h, 1 to do in 00:01h, 4 active
1 of 1 target completed, 0 valid password found
Hydra finished at 2025-03-30 17:30:21
[2025-03-30 17:30:21] DEBUG: SSH brute force attack completed.
```

Checking Unreachability

```
[2025-03-30 17:30:21] DEBUG: Checking reachability of 192.168.56.14...
PING 192.168.56.14 (192.168.56.14) 56(84) bytes of data.
--- 192.168.56.14 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3085ms
[2025-03-30 17:30:34] DEBUG: Target is no more reachable.
```

Checking Logs of the Manager

```
[17:30:34] DEBUG: Cleaning up password list file...
[17:30:34] DEBUG: Password list file deleted successfully.
[17:30:34] INFO: Brute force attack successfully realized.
[19:30:34] INFO: Retrieving manager logs:
/var/ossec/logs/alerts/alerts.log and Rule ID: 5763
[17:30:35] INFO: Last log retrieved at time: 17:29:15
[17:30:35] INFO: Active Response: active-response/bin/firewall-drop
[17:30:35] INFO: Agent ID: 001
[17:30:35] INFO: Attack Technique: Brute Force
[17:30:35] INFO: Frequency: 8
[17:30:35] INFO: Script executed successfully: scenario_brute_force.sh
```

4.5.3 Wazuh dashboard identification

We can see that brute force scenario was run successfully and spot the attack using the Wazuh dashboard to ensure coherence between time, logs, and the type of attack:

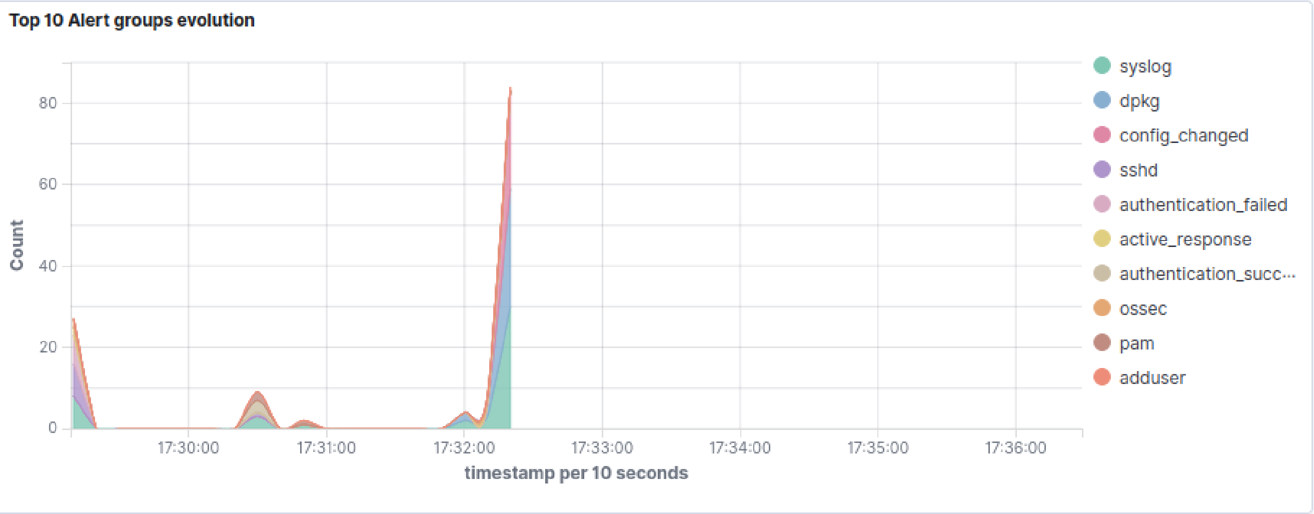
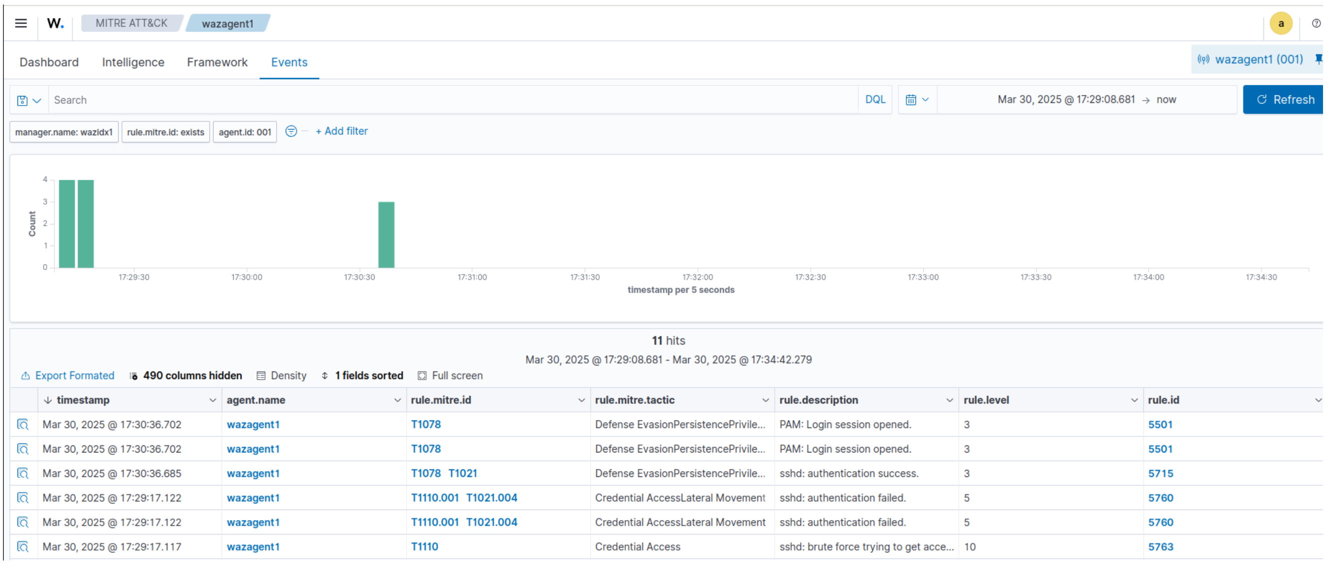


Figure 19: Overall alert on the Wazuh dashboard.

We spot the pic attack at 17:32 with syslog, sshd, and authentication failed as alert groups.

More than a graph, we can spot the written logs that we try to retrieve after each kind of attack: At the last line, we highlight the brute force attack with its associated rule ID and corresponding time.



11 hits

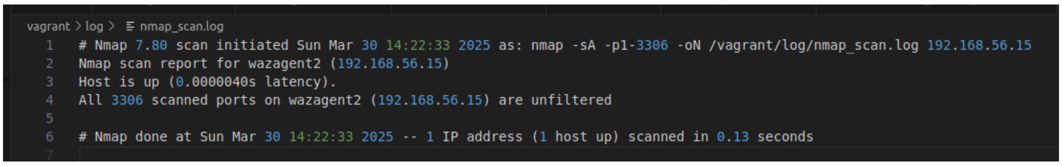
Mar 30, 2025 @ 17:29:08.681 - Mar 30, 2025 @ 17:34:42.279

↓ timestamp	agent.name	rule.mitre.id	rule.mitre.tactic	rule.description	rule.level	rule.id
Mar 30, 2025 @ 17:30:36.702	wazagent1	T1078	Defense EvasionPersistencePrivile...	PAM: Login session opened.	3	5501
Mar 30, 2025 @ 17:30:36.702	wazagent1	T1078	Defense EvasionPersistencePrivile...	PAM: Login session opened.	3	5501
Mar 30, 2025 @ 17:30:36.685	wazagent1	T1078 T1021	Defense EvasionPersistencePrivile...	sshd: authentication success.	3	5715
Mar 30, 2025 @ 17:29:17.122	wazagent1	T1110.001 T1021.004	Credential AccessLateral Movement	sshd: authentication failed.	5	5760
Mar 30, 2025 @ 17:29:17.122	wazagent1	T1110.001 T1021.004	Credential AccessLateral Movement	sshd: authentication failed.	5	5760
Mar 30, 2025 @ 17:29:17.117	wazagent1	T1110	Credential Access	sshd: brute force trying to get acce...	10	5763

Figure 20: Brute force attack spoted on the dashboard

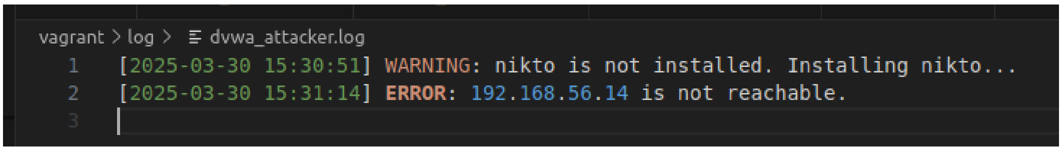
4.5.4 Logs

The logs were reported in a dedicated folder, which facilitates debugging. They are classified by scripts, and this method of classification aids in the debugging process. The logs report errors, warnings, and additional information, especially when the logs are too long to be displayed in the terminal.



```
# Nmap 7.80 scan initiated Sun Mar 30 14:22:33 2025 as: nmap -sA -p1-3306 -oN /vagrant/log/nmap_scan.log 192.168.56.15
Nmap scan report for wazagent2 (192.168.56.15)
Host is up (0.0000040s latency).
All 3306 scanned ports on wazagent2 (192.168.56.15) are unfiltered

# Nmap done at Sun Mar 30 14:22:33 2025 -- 1 IP address (1 host up) scanned in 0.13 seconds
```

Figure 21: Nmap log relatively to scanned ports on the target machine.



```
[2025-03-30 15:30:51] WARNING: nikto is not installed. Installing nikto...
[2025-03-30 15:31:14] ERROR: 192.168.56.14 is not reachable.
```

Figure 22: Error log indicating issues encountered during the the dvwa scenario

Comment: For instance, the last log indicates that one scenario during provisioning did not proceed as expected. The running website was not reachable and the attack did not occur. This makes sense because in a previous scenario the same IP was blocked by the manager (the victim's IP) as an active response. A potential solution would be to introduce a delay between each scenario to allow the IP to become reachable again.

Conclusion

- The script execution involved a systematic approach, starting with a clean setup of the environment.
- This was followed by the installation and configuration of necessary tools, including the Wazuh dashboard for monitoring.
- SSH keys were managed to ensure secure communication between machines.
- Various tests and scenarios were conducted to validate the setup, ensuring robustness and reliability throughout the process.

After the provisionning of the machines and their successful tests , we are now able to run the core of the software.

5 Execution of the prototype

This last part is divided into two:

- A unit test section that covers each crucial feature and ensures its correct functioning. All of these features were presented theoretically before in the section 3.
- A second part that puts these unit tests into practice in a first prototype, serving as the final product encompassing all elements, from provisioning to the software itself.

5.1 Unit Tests

Unit tests are run via `unitest.py`, those are about the following. The associated pseudocode is detailed in appendix E.1

- Ossec.conf (see Subsubsection 3.4.7)
- FIM setup (see Subsubsection 3.4.10)
- XML handling (see Subsubsection 3.4.3)
- YARA configuration (see Subsubsection 3.4.11)

Note :

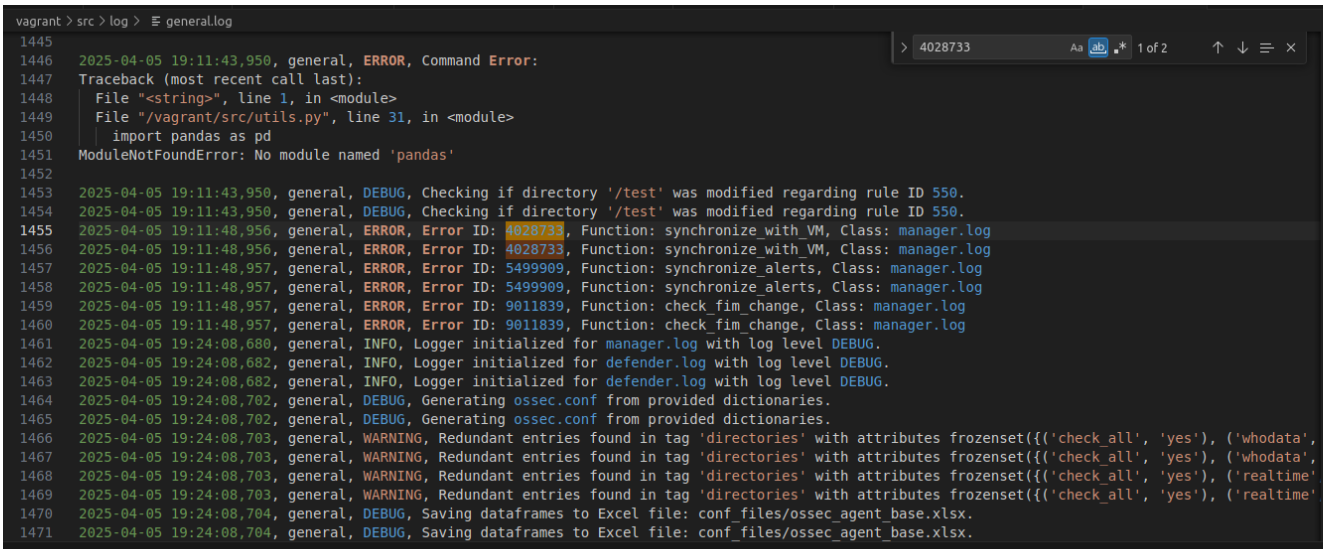
To ensure that tests aren't corrupted, we remove at the initialization any leftovers from previous manipulations to have pristine configuration files. We then ensure that these configuration files were well modified and that they efficiently work.

5.1.1 Logs

As the project is quite extensive, managing logs effectively is crucial. Analogous to the Vagrant provisioning in Subsection 4.5.4, Python logs are managed as follows:

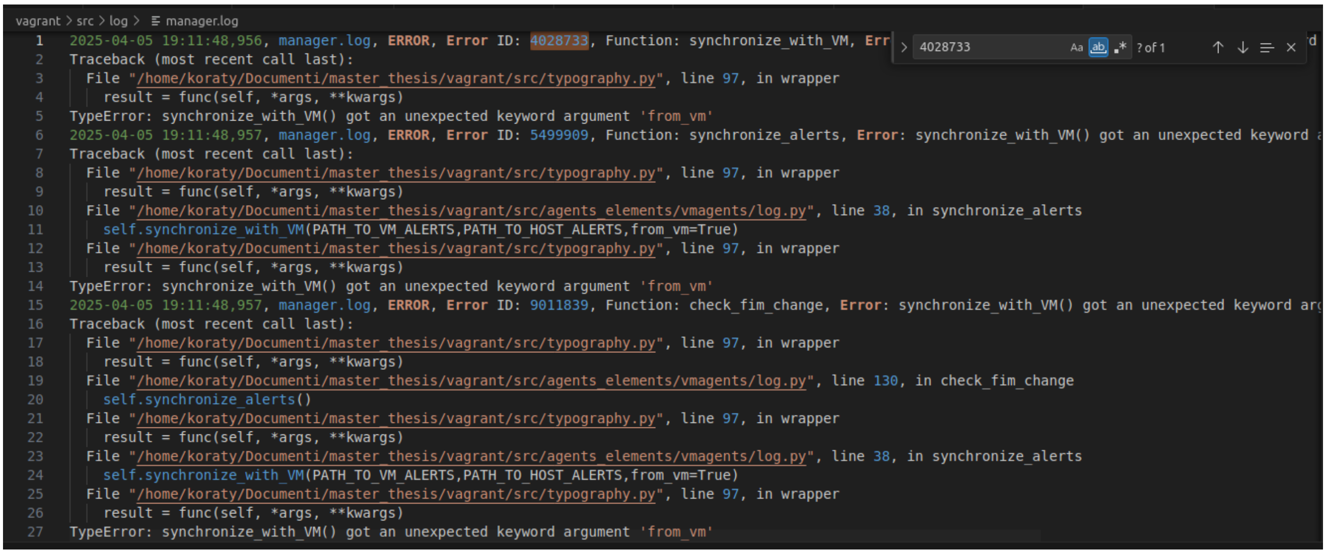
- Each error is assigned to an unique ID.
- When an error is reported, a general log entry is created with minimal details, including only the associated method and error ID writen in `general.log` were every other logs are reported (INFO/DEBUG/WARNING).
- The detailed error information is logged in the file associated with the Virtual Machine or entity where it occurred, which in our case can be either : `Manager.log`, `Attacker.log`, or `Defender.log`.

Below is an example of debugging using those logs where in the general logs, we can see an error with ID 4028733 from the manager, occurring in the function `synchronize_with_VM`.
We can then obtain more details on the following error by referring to the `manager.log`. Indeed, an extra argument has been parsed.
In this way, maintaining a detailed record of logs and errors from every machine aids in debugging and ensures the coherency of scenarios when implemented across multiple actors.



```

2025-04-05 19:11:43,950, general, ERROR, Command Error:
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/vagrant/src/utils.py", line 31, in <module>
    import pandas as pd
ModuleNotFoundError: No module named 'pandas'

2025-04-05 19:11:43,950, general, DEBUG, Checking if directory '/test' was modified regarding rule ID 550.
2025-04-05 19:11:43,950, general, DEBUG, Checking if directory '/test' was modified regarding rule ID 550.
2025-04-05 19:11:48,956, general, ERROR, Error ID: 4028733, Function: synchronize_with_VM, Class: manager.log
2025-04-05 19:11:48,956, general, ERROR, Error ID: 4028733, Function: synchronize_with_VM, Class: manager.log
2025-04-05 19:11:48,957, general, ERROR, Error ID: 5499909, Function: synchronize_alerts, Class: manager.log
2025-04-05 19:11:48,957, general, ERROR, Error ID: 5499909, Function: synchronize_alerts, Class: manager.log
2025-04-05 19:11:48,957, general, ERROR, Error ID: 9011839, Function: check_fim_change, Class: manager.log
2025-04-05 19:11:48,957, general, ERROR, Error ID: 9011839, Function: check_fim_change, Class: manager.log
2025-04-05 19:24:08,680, general, INFO, Logger initialized for manager.log with log level DEBUG.
2025-04-05 19:24:08,682, general, INFO, Logger initialized for defender.log with log level DEBUG.
2025-04-05 19:24:08,682, general, INFO, Logger initialized for defender.log with log level DEBUG.
2025-04-05 19:24:08,702, general, DEBUG, Generating ossec.conf from provided dictionaries.
2025-04-05 19:24:08,702, general, DEBUG, Generating ossec.conf from provided dictionaries.
2025-04-05 19:24:08,703, general, WARNING, Redundant entries found in tag 'directories' with attributes frozenset({('check_all', 'yes'), ('whodata',
2025-04-05 19:24:08,703, general, WARNING, Redundant entries found in tag 'directories' with attributes frozenset({('check_all', 'yes'), ('whodata',
2025-04-05 19:24:08,703, general, WARNING, Redundant entries found in tag 'directories' with attributes frozenset({('check_all', 'yes'), ('realtime'
2025-04-05 19:24:08,703, general, WARNING, Redundant entries found in tag 'directories' with attributes frozenset({('check_all', 'yes'), ('realtime'
2025-04-05 19:24:08,704, general, DEBUG, Saving dataframes to Excel file: conf_files/ossec_agent_base.xlsx.
2025-04-05 19:24:08,704, general, DEBUG, Saving dataframes to Excel file: conf_files/ossec_agent_base.xlsx.
```

Figure 23: General log error with ID 4028733.



```
2025-04-05 19:11:48,956, manager.log, ERROR, Error ID: 4028733, Function: synchronize_with_VM, Err
Traceback (most recent call last):
  File "/home/koraty/Documenti/master_thesis/vagrant/src/typography.py", line 97, in wrapper
    result = func(self, *args, **kwargs)
TypeError: synchronize_with_VM() got an unexpected keyword argument 'from_vm'
2025-04-05 19:11:48,957, manager.log, ERROR, Error ID: 5499909, Function: synchronize_alerts, Error: synchronize_with_VM() got an unexpected keyword a
Traceback (most recent call last):
  File "/home/koraty/Documenti/master_thesis/vagrant/src/typography.py", line 97, in wrapper
    result = func(self, *args, **kwargs)
  File "/home/koraty/Documenti/master_thesis/vagrant/src/agents_elements/vmagents/log.py", line 38, in synchronize_alerts
    self.synchronize_with_VM(PATH_TO_VM_ALERTS,PATH_TO_HOST_ALERTS,from_vm=True)
  File "/home/koraty/Documenti/master_thesis/vagrant/src/typography.py", line 97, in wrapper
    result = func(self, *args, **kwargs)
TypeError: synchronize_with_VM() got an unexpected keyword argument 'from_vm'
2025-04-05 19:11:48,957, manager.log, ERROR, Error ID: 9011839, Function: check_fim_change, Error: synchronize_with_VM() got an unexpected keyword arg
Traceback (most recent call last):
  File "/home/koraty/Documenti/master_thesis/vagrant/src/typography.py", line 97, in wrapper
    result = func(self, *args, **kwargs)
  File "/home/koraty/Documenti/master_thesis/vagrant/src/agents_elements/vmagents/log.py", line 130, in check_fim_change
    self.synchronize_alerts()
  File "/home/koraty/Documenti/master_thesis/vagrant/src/typography.py", line 97, in wrapper
    result = func(self, *args, **kwargs)
  File "/home/koraty/Documenti/master_thesis/vagrant/src/agents_elements/vmagents/log.py", line 38, in synchronize_alerts
    self.synchronize_with_VM(PATH_TO_VM_ALERTS,PATH_TO_HOST_ALERTS,from_vm=True)
  File "/home/koraty/Documenti/master_thesis/vagrant/src/typography.py", line 97, in wrapper
    result = func(self, *args, **kwargs)
TypeError: synchronize_with_VM() got an unexpected keyword argument 'from_vm'
```

Figure 24: Detailed error in `manager.log` showing an unexpected argument.

5.1.2 Ossec.conf

This unit-test aims at verifying satisfaction of the criteria presented in subsubsection 1.2.4 where we aims at automating the process of updating configuration files.
This unit-test performs the following steps:

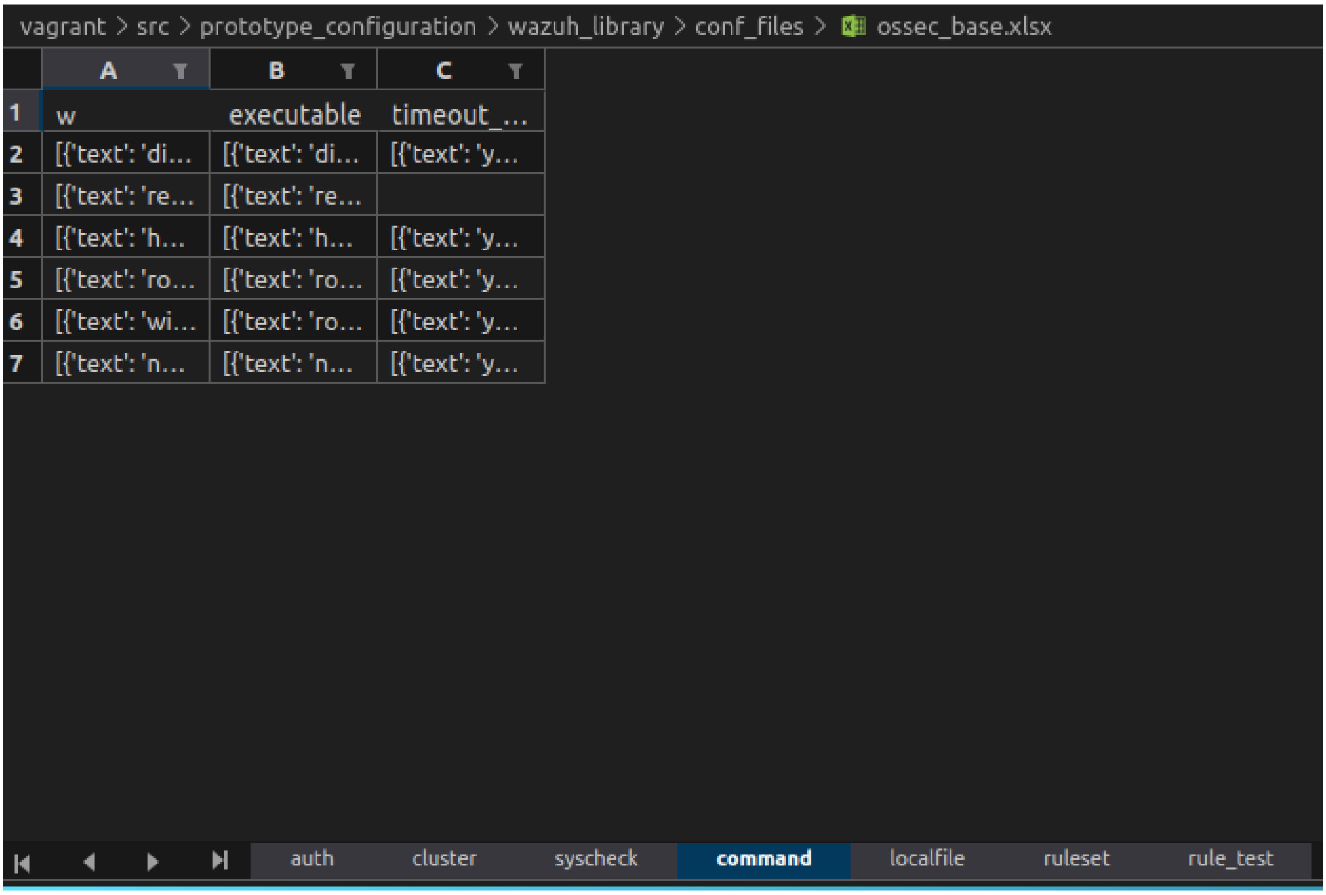
- It adds a section tag with commands, local file configurations, and active responses necessary for FIM configuration to the `ossec.conf` file , tags mentioned here 3.4.8.
- It then applies methods to add the section tag, converts it to an Excel file, and converts it back to an XML tag.
- Finally, it ensures the presence of the added tag using `assertEqual`.

```python
# Testing to add a section tag to an ossec_conf file
result = self.manager.add_section_tag_to_conf_file(section_tag)

# Load dataframes back from the Excel file
print("Reloading configuration data from Excel...")
dict_dfs = self.manager.dataframe_from_conf(PATH_TO_OSSEC_AGENT_TEST_MANAGER)

# Retrieving the command added in the ossec conf file
is_subset_command = expected_command.apply(lambda row: dict_dfs['command']..)

# Assertions to verify if ossec has been correctly modified #
self.assertEqual(is_subset_command, True)
```

Listing 11: Adding and Verifying Section Tag in ossec.conf

Excel Projection

Below is the Excel projection of the configuration file on an Excel board. The following Excel file is divided into tables for each tag:

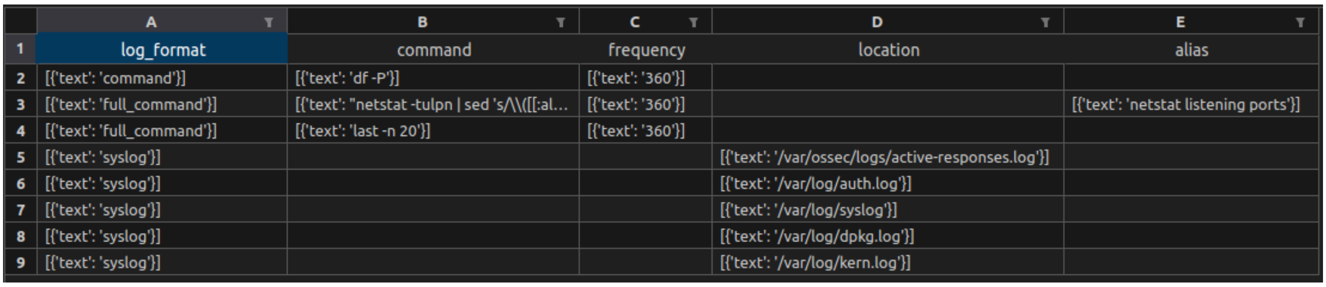


vagrant > src > prototype_configuration > wazuh_library > conf_files > ossec_base.xlsx

	A	B	C
1	w	executable	timeout_...
2	[{'text': 'di...	[{'text': 'di...	[{'text': 'y...
3	[{'text': 're...	[{'text': 're...	
4	[{'text': 'h...	[{'text': 'h...	[{'text': 'y...
5	[{'text': 'ro...	[{'text': 'ro...	[{'text': 'y...
6	[{'text': 'wi...	[{'text': 'ro...	[{'text': 'y...
7	[{'text': 'n...	[{'text': 'n...	[{'text': 'y...

Figure 25: Excel architecture of ossec conf

As we can see, this excel file is divided in several sheets that are related to the modifiable tags. We focus on the `localfile` section and compare both excel files of the first version and the modified version of osse.conf.

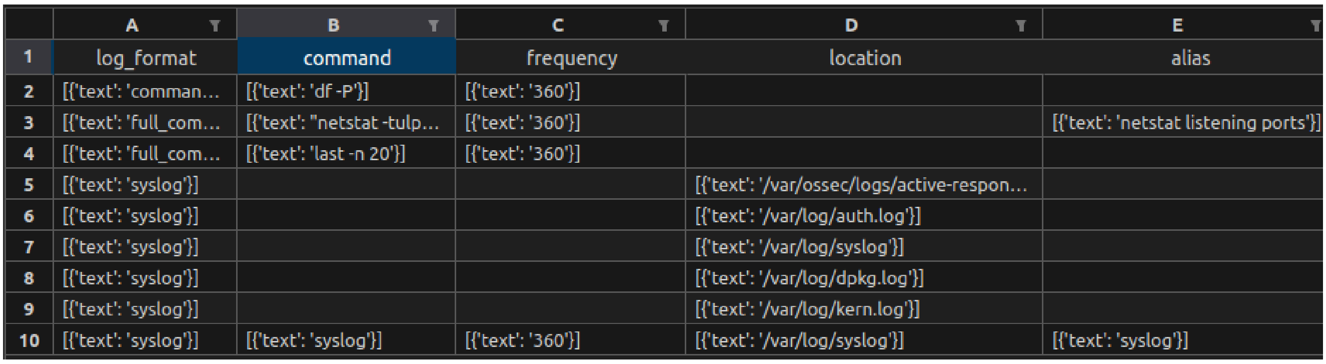


	A	B	C	D	E
1	log_format	command	frequency	location	alias
2	[{'text': 'command'}]	[{'text': 'df -P'}]	[{'text': '360'}]		
3	[{'text': 'full_command'}]	[{'text': "netstat -tulpn | sed 's/\\([[:al...	[{'text': '360'}]		[{'text': 'netstat listening ports'}]
4	[{'text': 'full_command'}]	[{'text': 'last -n 20'}]	[{'text': '360'}]		
5	[{'text': 'syslog'}]			[{'text': '/var/ossec/logs/active-responses.log'}]	
6	[{'text': 'syslog'}]			[{'text': '/var/log/auth.log'}]	
7	[{'text': 'syslog'}]			[{'text': '/var/log/syslog'}]	
8	[{'text': 'syslog'}]			[{'text': '/var/log/dpkg.log'}]	
9	[{'text': 'syslog'}]			[{'text': '/var/log/kern.log'}]	

Figure 26: Original `localfile` section in the Excel projection.

```
<localfile>
    <log_format>syslog</log_format>
    <command>syslog</command>
    <frequency>360</frequency>
    <location>/var/log/syslog</location>
    <alias>syslog</alias>
</localfile>
```

Listing 12: New `localfile` Section to add



	A	B	C	D	E
1	log_format	command	frequency	location	alias
2	[{'text': 'comman...	[{'text': 'df -P'}]	[{'text': '360'}]		
3	[{'text': 'full_com...	[{'text': "netstat -tulp...	[{'text': '360'}]		[{'text': 'netstat listening ports'}]
4	[{'text': 'full_com...	[{'text': 'last -n 20'}]	[{'text': '360'}]		
5	[{'text': 'syslog'}]			[{'text': '/var/ossec/logs/active-respon...	
6	[{'text': 'syslog'}]			[{'text': '/var/log/auth.log'}]	
7	[{'text': 'syslog'}]			[{'text': '/var/log/syslog'}]	
8	[{'text': 'syslog'}]			[{'text': '/var/log/dpkg.log'}]	
9	[{'text': 'syslog'}]			[{'text': '/var/log/kern.log'}]	
10	[{'text': 'syslog'}]	[{'text': 'syslog'}]	[{'text': '360'}]	[{'text': '/var/log/syslog'}]	[{'text': 'syslog'}]

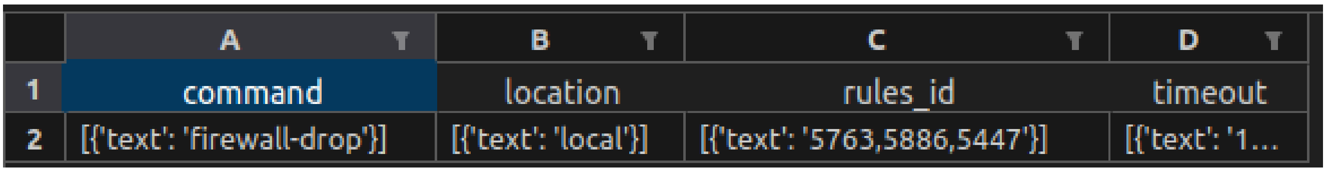
Figure 27: Updated `localfile` section in the Excel projection.

Additionally, the `active-response` section:

```
<active-response>
    <command>firewall-drop</command>
    <location>local</location>
    <rules_id>5763,5886,5447</rules_id>
    <timeout>180</timeout>
</active-response>
```

Listing 13: Original `active-response` Section

becomes:



	A	B	C	D
1	command	location	rules_id	timeout
2	[{'text': 'firewall-drop'}]	[{'text': 'local'}]	[{'text': '5763,5886,5447'}]	[{'text': '1...

Figure 28: Updated `active-response` section in the Excel projection.

5.1.3 Rules verification

To ensure that rules are correctly triggered when a related event occurs, we rely on a verification mechanism implemented in the `log.py` classes. This process includes:

- **Synchronizing** the `alerts.json` file located on the Wazuh manager with the host machine.
- **Parsing** the log file to search for a specific **Rule ID** and extract relevant event information.

This is achieved through the following private method `_get_rule_id`:

```
def _get_rule_id(self, rule_id: str = 550) -> dict:
""" Fetch the most recent alert matching a given rule ID. """

if not os.path.isfile(self.log_wazuh_path): return {}

last_log = None
with open(self.log_wazuh_path, ’r’) as file:
    for line in file:
        alert = json.loads(line)
        if alert.get("rule", {}).get("id") == rule_id:
            last_log = line

return json.loads(last_log) if last_log else {}
```

Listing 14: Simplified rule verification method

Description: This method retrieves the most recent alert that matches the specified rule's id.

It ensures that the detection mechanism is functioning correctly and that the alert contains the expected metadata, such as timestamps, source IPs, and event types.

An example of the extracted output is referenced in subsubsection 4.5.2.
This verification mechanism allows us to programmatically confirm that a rule has been triggered and that Wazuh is effectively monitoring and responding to relevant events

5.1.4 File integrity monitoring

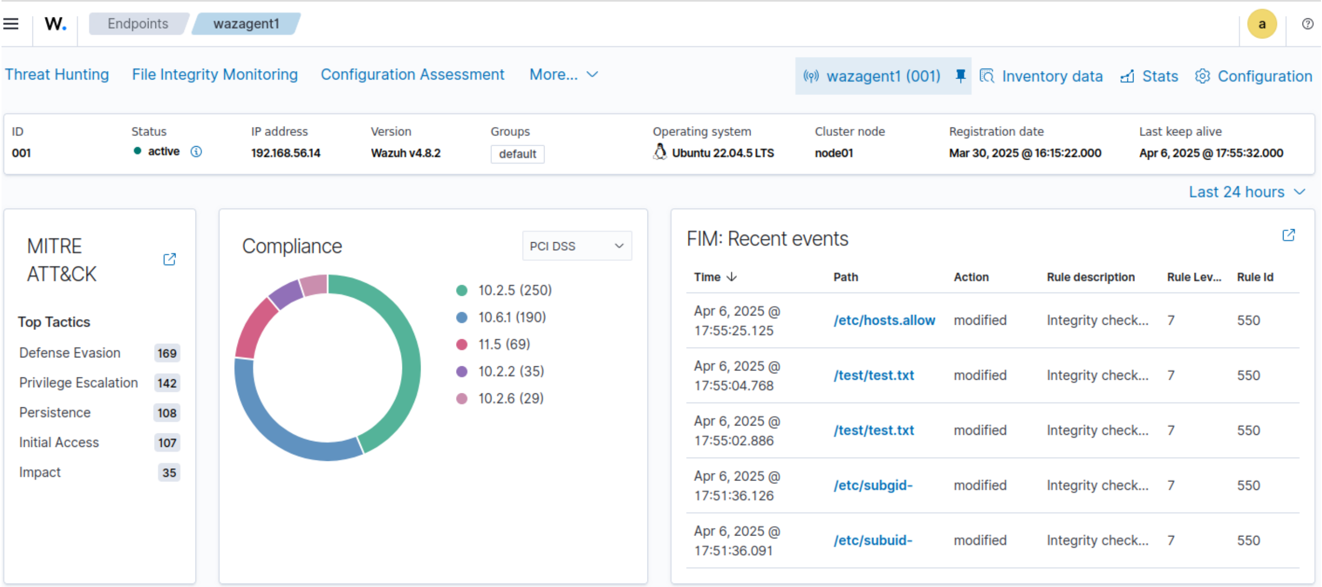
During this unit-test we address the following steps:

- Configure the file monitoring system by adding the monitored path to the `ossecc.conf` file as mentioned in this subsection.
- We then simulate a file change on this monitored path and check if an alert has been raised.
- We do the same regarding the who-data tool as explained in subsubsection 3.4.10.

To do so, we run the following explained test.

```
print("Starting FIM setup...")

# Add FIM configuration
self.defender.add_fim_configuration(monitored_paths=[directory_tag,
    directory_tag_whodata], do_synchronize_with_VM=True, conf_path=
    PATH_TO_OSSEC_AGENT_BASE_DEFENDER)

# Test FIM
print("Testing FIM...")
self.defender.run_function_on_remote_host('simulate_file_change',
    monitored_path)
self.assertEqual(self.manager.check_fim_change(monitored_path), True)

# Test Who-Data
print("Testing audit...")
self.defender.run_function_on_remote_host('test_who_data')
self.assertEqual(self.manager.check_fim_change(monitored_path_who_data), True)
```

This gives the following alert on the dashboard:



ID	Status	IP address	Version	Groups	Operating system	Cluster node	Registration date	Last keep alive
001	● active	192.168.56.14	Wazuh v4.8.2	default	Ubuntu 22.04.5 LTS	node01	Mar 30, 2025 @ 16:15:22.000	Apr 6, 2025 @ 17:55:32.000

Top Tactics	
Defense Evasion	169
Privilege Escalation	142
Persistence	108
Initial Access	107
Impact	35

Compliance	
● 10.2.5	(250)
● 10.6.1	(190)
● 11.5	(69)
● 10.2.2	(35)
● 10.2.6	(29)

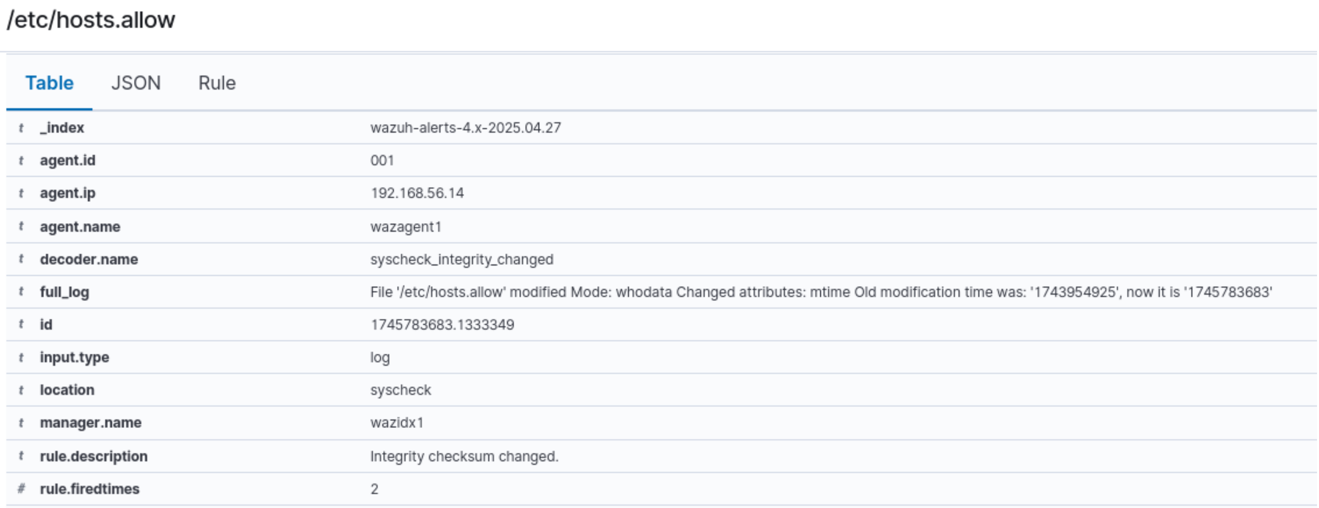
Time ↓	Path	Action	Rule description	Rule Lev...	Rule Id
Apr 6, 2025 @ 17:55:25.125	/etc/hosts.allow	modified	Integrity check...	7	550
Apr 6, 2025 @ 17:55:04.768	/test/test.txt	modified	Integrity check...	7	550
Apr 6, 2025 @ 17:55:02.886	/test/test.txt	modified	Integrity check...	7	550
Apr 6, 2025 @ 17:51:36.126	/etc/subgid-	modified	Integrity check...	7	550
Apr 6, 2025 @ 17:51:36.091	/etc/subuid-	modified	Integrity check...	7	550

Figure 29: Alert on the dashboard indicating file modification.

We indeed spot the file modification of `test.txt` at 17h55m matching with the logs detailed in appendix. We also find the who-data alert that indicates connection of another user to our agent1. We can go in further detail by clicking on it as presented in appendix E.3.
`Wazgent1` was modified and it raises the rule id 550 that then was spotted by the software to ensure consistency with the rule description "Integrity checksum changed".

Who-Data

During the test, another user has attempted to connect to our victim 'agent1'. We can correlate the output logs with the time of connection.
Thanks to Who-Data, we can obtain more information about the attacker.



/etc/hosts.allow

t	_index	wazuh-alerts-4.x-2025.04.27
t	agent.id	001
t	agent.ip	192.168.56.14
t	agent.name	wazagent1
t	decoder.name	syscheck_integrity_changed
t	full_log	File '/etc/hosts.allow' modified Mode: whodata Changed attributes: mtime Old modification time was: '1743954925', now it is '1745783683'
t	id	1745783683.1333349
t	input.type	log
t	location	syscheck
t	manager.name	wazidx1
t	rule.description	Integrity checksum changed.
#	rule.firedtimes	2

Figure 30: Detailed information on the modifier.

There are two kinds of OSSEC configurations: one for the agent and the other for the manager.

Their structures are slightly different, as are the associated checking tools. **To monitor file modifications, we need to modify the OSSEC configuration of the endpoint, which is the victim.**

As in any other process, in E.3, we always follow the same procedure:

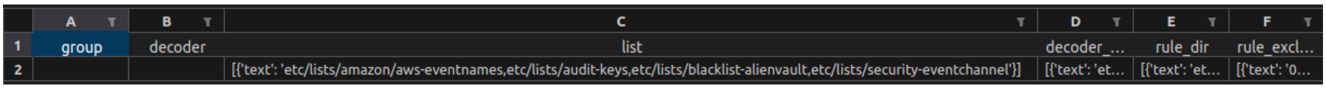
- Back up the files to be modified.
- Check the consistency of the new OSSEC configuration.
- Test it by applying a file modification.
- Retrieve the logs.

The ruleset added during the configuration of Who-Data was:

```
<ruleset>
    <list>etc/lists/amazon/aws-eventnames,etc/lists/audit-keys,etc/lists/blacklist-alienvault,etc/lists/security-eventchannel</list>
    <decoder_dir>etc/decoders,ruleset/decoders</decoder_dir>
    <rule_dir>etc/rules,ruleset/rules</rule_dir>
    <rule_exclude>0217-policy_rules.xml</rule_exclude>
</ruleset>
```

Listing 15: Original Ruleset for Who-Data

It will be translated into:



	A	B	C	D	E	F
1	group	decoder	list	decoder_...	rule_dir	rule_excl...
2			[{'text': 'etc/lists/amazon/aws-eventnames,etc/lists/audit-keys,etc/lists/blacklist-alienvault,etc/lists/security-eventchannel'}]	[{'text': 'et...	[{'text': 'et...	[{'text': '0...

Figure 31: Updated ruleset for Who-Data in the Excel projection.

This ruleset is added to ossec.conf and requires to verify ossec configuration. Below is the logs generated to test audit where :

- We create an user *Smith*
- We log in as *Smith* to the machine.

Who-Data logs test

```
Testing audit...
WHODATA
2025-04-06 17:55:11,005 - DEBUG - Running function configure_audit.
2025-04-06 17:55:11,005 - DEBUG - Checking if package 'auditd' is installed.
[CONFIGURING AUDIT]
2025-04-06 17:55:11,030 - INFO - Checking if 'wazuh_fim' rule is applied.
2025-04-06 17:55:20,925 - DEBUG - wazuh-agent restarted successfully.
2025-04-06 17:55:23,014 - DEBUG - auditd restarted successfully.
2025-04-06 17:55:25,021 - DEBUG - '/etc' configured for audit and rule wazuh_fim.
2025-04-06 17:55:25,023 - DEBUG - User 'smith' already exists.
2025-04-06 17:55:25,051 - DEBUG - Password for user 'smith' set to 'wazuh'.
2025-04-06 17:55:25,073 - DEBUG - Successfully logged in as 'smith'.
[... 17:55:25,073] - IP address '192.168.32.' rewritten in /etc/hosts.allow.
[... 17:55:30,367] -INFO - Directory '/etc' was modified as per rule ID 550.
It occurred at file /etc/hosts.allow.
```

5.1.5 Xml modification

This unit test is straightforward. It ensures that local rules and decoders are updated correctly by running the following commands:

```python
def test_xml_handling(self):
    """
    Configuring manager XML for YARA.
    """
    # Add the group section to the local_file XML
    self.manager.add_group_to_xml(self.group_section_str)
    # Add the decoder section to the decoder XML
    self.manager.add_decoder_to_xml(self.decoder_section_str)

    # Synchronize the XML with the VM
    result = self.manager.synchronize_xml_with_VM(decoder_sections_str=self.
   decoder_section_str, group_sections_str=self.group_section_str)

    # Check for successful modification of newly added rules
    self.assertEqual(result, True)
```

Listing 16: Unit Test for XML Handling

An important aspect is the decoder, as mentioned in Section 3.4.6, we will analyze the decoding process to ensure the consistency of the file.

The `test_group_sections` function is designed to test multiple group sections in a given XML string using the `wazuh-logtest` tool. This function generates testing logs based on the provided XML configuration, writes these logs to a temporary file, and then executes the `wazuh-logtest` command remotely to validate the logs against the configured rules.

Here is a brief overview of the function:

- **Input**: The function takes an XML string containing multiple group sections and a temporary path for saving the XML file.
- **Process**: It generates testing logs, writes them to a temporary file, and runs the `wazuh-logtest` command.
- **Output**: The function returns an exit status indicating whether all rules matched successfully.

In the unitest, we consider the following XML configuration for a YARA decoder:

```xml
<group name="yara_decoders">
    <decoder name="yara_decoder">
        <prematch>wazuh-yara:</prematch>
    </decoder>
    <decoder name="yara_decoder1">
        <parent>yara_decoder</parent>
        <regex>wazuh-yara: (\S+) - Scan result: (\S+) (\S+)</regex>
        <order>log_type, yara_rule, yara_scanned_file</order>
    </decoder>
</group>
```

Listing 17: YARA Decoder Configuration

Given the log entry `wazuh-yara: INFO - Scan result: rule1 file1`, the `wazuh-logtest` output look like this:

Wazuh-Logtest Output

```
**Phase 1: Completed pre-decoding.
        full event: 'wazuh-yara: INFO - Scan result: rule1 file1'
        timestamp: 'Apr 28 12:00:00'
        hostname: 'localhost'
        program_name: 'wazuh-yara'

**Phase 2: Completed decoding.
        decoder: 'yara_decoder'
        log_type: 'INFO'
        yara_rule: 'rule1'
        yara_scanned_file: 'file1'

**Phase 3: Completed filtering.
        Rule id: '100001'
        Level: '3'
        Description: 'Yara scan result matched.'
```

In case of success , the `xml` files are updated on the manager.

5.1.6 YARA

In the previous sections, we configured file monitoring, local rules files, and decoders. The next steps are:

- Install YARA on the endpoint.
- Configure the `ossec.conf` on the manager.
- Simulate a download of known malware.

This leads to the following test:

```python
def test_yara_configuration(self):
    """
    Detecting malware using YARA integration.
    [https://documentation.wazuh.com/current/proof-of-concept-guide/detect-malware-yara-integration.html#detecting-malware-using-yara-integration]
    """
    print("Configuring ossec.conf for YARA")

    # Install YARA on the endpoint
    self.defender.check_and_install_yara(conf_path=PATH_TO_OSSEC_AGENT_BASE_DEFENDER)

    # Add YARA configuration to ossec.conf
    section_tag_yara = """
    <command>
       ...
    </command>
    <active-response>
      ...
    </active-response>
    """
    self.manager.add_section_tag_to_conf_file(section_tag=section_tag_yara, file_path=PATH_TO_OSSEC_AGENT_TEST_MANAGER)

    # Simulate malware download
    self.defender.run_bash_script_on_remote_host(NAME_MALWARE_SCRIPT)

    # Verify alerts
    self.assertEqual(self.manager.check_alerts('100300'), True)
```

Listing 18: YARA Configuration Test

This test ensures that the YARA configuration is correctly set up and that the system can detect malware downloads.

YARA Rules

One important aspect is the installation and configuration of the YARA rules. This process can be challenging and typically requires the following:

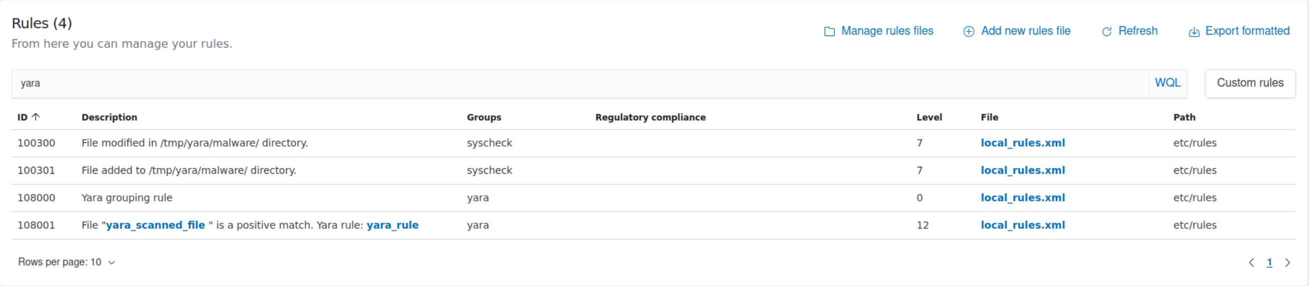
- The YARA tool.
- The associated rules.

Both need to be downloaded, correctly installed, and placed in the appropriate directory. This is automated using the `yara_setup.sh` script, which is summarized below:

```
# Check if YARA is already installed
if command -v yara >/dev/null 2>&1; then
    log "YARA is already installed. Checking for rules..."

    # Check if YARA rules file is present
    if [ -f "\$YARA_RULES_FILE" ]; then
        log "YARA rules already exist. Skipping installation."
        exit 0
    else
        log "YARA rules not found. Proceeding to download rules..."
    fi
else
    log "YARA not found. Proceeding with installation."
fi
```

Listing 19: YARA Setup Intro

Once the rules are installed, their presence and functionality can be verified on the dashboard, as shown in the following image:



Rules (4)

From here you can manage your rules.

ID ↑	Description	Groups	Regulatory compliance	Level	File	Path
100300	File modified in /tmp/yara/malware/ directory.	syscheck		7	local_rules.xml	etc/rules
100301	File added to /tmp/yara/malware/ directory.	syscheck		7	local_rules.xml	etc/rules
108000	Yara grouping rule	yara		0	local_rules.xml	etc/rules
108001	File "yara_scanned_file " is a positive match. Yara rule: yara_rule	yara		12	local_rules.xml	etc/rules

Figure 32: YARA rules configuration on the dashboard.

The dashboard displays the following key information for each rule:

- **ID**: The unique identifier for the rule.
- **Description**: A brief description of the rule's purpose.
- **Groups**: The group to which the rule belongs (e.g., `syscheck`, `yara`).
- **Regulatory Compliance**: Indicates whether the rule is related to regulatory compliance.
- **Level**: The severity level of the rule.
- **File**: The file in which the rule is defined.
- **Path**: The path to the rule file.

We retrieve the elements defined in the associated active response. This ensures that the YARA rules are correctly configured and operational, allowing for effective malware detection and compliance monitoring.

Downloading Malware

Once YARA is configured, it's time to start an attack simulation. The victim is forced to download malicious packets into the monitored folder. This is done using the `malware_downloader.sh` script. The following logs indicate the successful download of the malware samples:

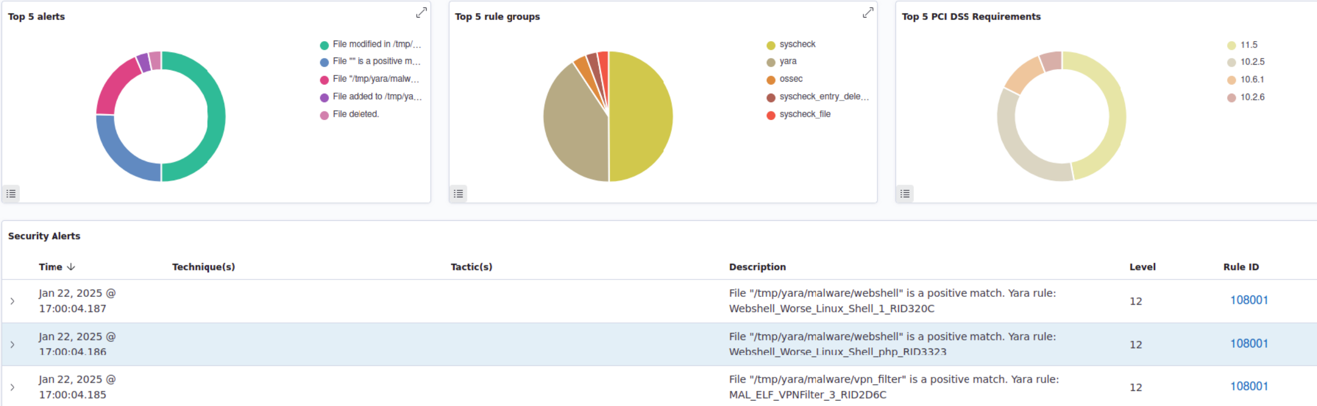
Malware Download Script Output

```
WARNING: Downloading Malware samples, please use this script with caution.
# Mirai: https://en.wikipedia.org/wiki/Mirai_(malware)
Downloading malware sample...
Done!
# Xbash: https://unit42.paloaltonetworks.com/...
Downloading malware sample...
Done!
# VPNFilter: https://news.sophos.com/en-us/...
Downloading malware sample...
Done!
# WebShell: https://github.com/SecWiki/WebShell-2/...
Downloading malware sample...
Done!
```

The overall logs are accessible in the appendix E.1.

Dashboard Analysis

Once the attack is complete, we can retrieve YARA logs that indicate the scan output and the associated active response (e.g., blocking the IP).

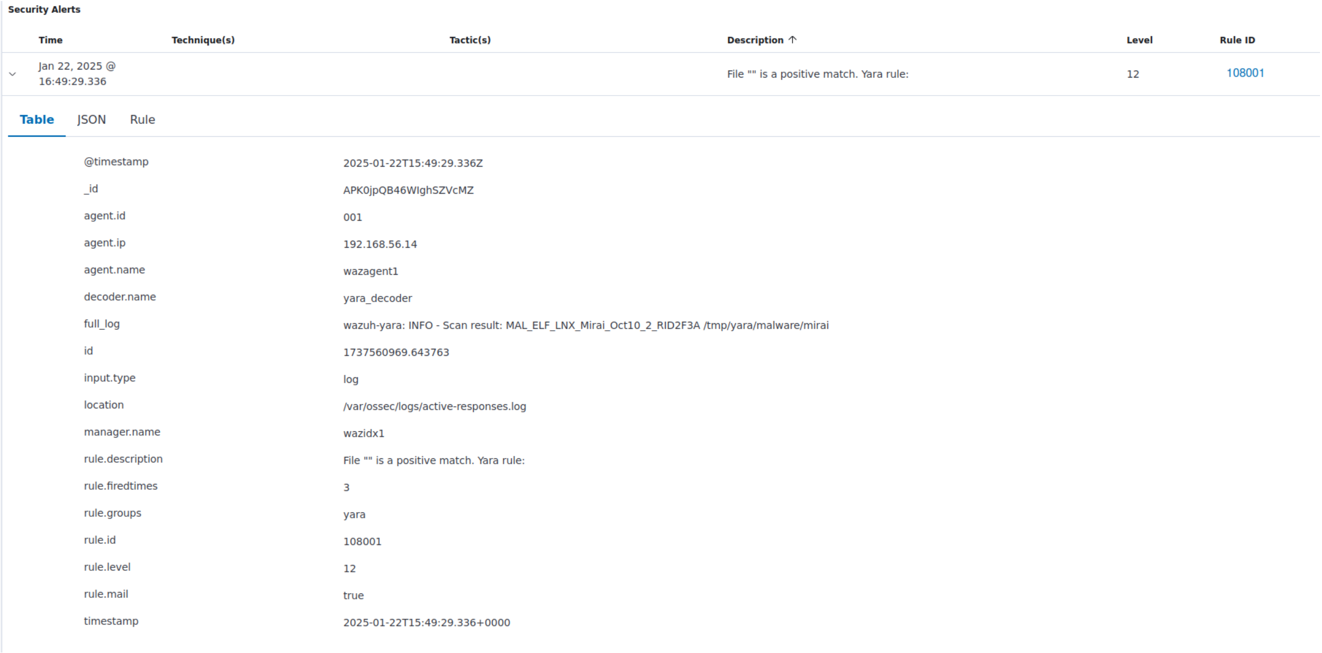


Security Alerts

Time ↓	Technique(s)	Tactic(s)	Description	Level	Rule ID
Jan 22, 2025 @ 17:00:04.187			File "/tmp/yara/malware/webshell" is a positive match. Yara rule: Webshell_Worse_Linux_Shell_1_RID320C	12	108001
Jan 22, 2025 @ 17:00:04.186			File "/tmp/yara/malware/webshell" is a positive match. Yara rule: Webshell_Worse_Linux_Shell_php_RID3323	12	108001
Jan 22, 2025 @ 17:00:04.185			File "/tmp/yara/malware/vpn_filter" is a positive match. Yara rule: MAL_ELF_VPNFilter_3_RID2D6C	12	108001

Figure 33: YARA attack spotted on the dashboard with rule ID 108001.

We can also obtain detailed information about the attack:



Security Alerts

Time	Technique(s)	Tactic(s)	Description ↑	Level	Rule ID
Jan 22, 2025 @ 16:49:29.336			File "" is a positive match. Yara rule:	12	108001

@timestamp	2025-01-22T15:49:29.336Z
_id	APK0jpQB46WIghSZVcMZ
agent.id	001
agent.ip	192.168.56.14
agent.name	wazagent1
decoder.name	yara_decoder
full_log	wazuh-yara: INFO - Scan result: MAL_ELF_LNX_Mirai_Oct10_2_RID2F3A /tmp/yara/malware/mirai
id	1737560969.643763
input.type	log
location	/var/ossec/logs/active-responses.log
manager.name	wazidx1
rule.description	File "" is a positive match. Yara rule:
rule.firedtimes	3
rule.groups	yara
rule.id	108001
rule.level	12
rule.mail	true
timestamp	2025-01-22T15:49:29.336+0000

Figure 34: Detailed view of the YARA attack.

This analysis demonstrates the powerful capabilities of YARA. With just a few lines of configuration, we were able to set up a machine to detect and respond to malicious downloads effectively.

During provisioning, as mentioned in subsubsection 2.2.3, initial tests and scenarios were conducted to ensure that Python-side software errors would not be related to misconfigurations in the Vagrant environment.

With these unit tests successfully completed, we can now write a simple scenario to demonstrate the power of automating processes using a multi-agent system enhanced by generative AI, particularly for malware downloading. Detailed logs and error records from each machine aid in debugging and ensure scenario coherence across multiple actors.

The following unit tests were performed to ensure system robustness:

- **XML unit test**: Verified that local rules and decoders were updated correctly.
- **Ossec conf unit test**: Verified that configuration file was updated correctly.
- **File Integrity Monitoring (FIM)**: Configured the file monitoring system, simulated file changes, and verified alerts.
- **Yara unit test**

These tests confirm our system's reliability and readiness for more complex scenarios.

5.2 Prototype

5.2.1 Content provided by the AI

As mentioned in the introduction, the real lever of such a system would be to get a reactive response in the case of an unidentified packet. `Main.py` implements a scenario where a Defender agent downloads a fake malware script, and a Manager agent adds a new YARA rule to detect and respond to the malware. The process involves several steps to ensure the system can automatically detect and isolate infected systems. This scenario was presented visually in the figure 12. Main aspects to consider are :

- **Fake Malware Script**: A fake malware script is defined to simulate malicious behavior, such as creating suspicious files and modifying system files.

```
fake_malware_script = """
#!/bin/bash
echo "This is a fake malware script!"
touch /tmp/suspicious_file
ping -c 4 nonexistentdomain.com
echo "127.0.0.1 fake.malware.domain" >> /etc/hosts
echo "Suspicious command executed"
"""
```

Listing 20: Fake Malware Script

Intervention of the AI: The AI provides the YARA rule, the associated active response and local rules to detect the fake malware script.

```
response = llm_api.generate(prompt="Provide a YARA rule and signature to detect the following malware behavior:\n" + fake_malware_script)
new_yara_rule = response['yara_rule']
active_response_to_add = response['active_response']
local_rules_to_add = response['local_rules']
```

Listing 21: Request YARA Rule and Signature from LLM

- **Active Response and Rules**: The agent generates an active response and local rules to be added to the OSSEC configuration and new YARA rule is also defined to detect the fake malware script.

```
active_response_to_add = """
<active-response>
    <command>isolate_infected_system</command>
    <location>local</location>
    <rules_id>100003</rules_id>
</active-response>
"""

local_rules_to_add = """
<group name="yara,malware">
    <rule id="100003" level="10">
        <decoder>yara</decoder>
        <options>no_full_log</options>
        <match>yara.rule=DetectFakeMalware</match>
        <description>Fake malware detected by YARA rule</description>
        <group>yara,malware</group>
    </rule>
</group>
"""
```

Listing 22: Active Response and Local Rules provided by the AI

```
new_yara_rule = """
rule DetectFakeMalware
{
    meta:
        description = "Rule to detect the fake malware script"
        author = "Your Name"
        date = "2023-10-01"

    strings:
        \$malware_string1 = "This is a fake malware script!"
        \$malware_string2 = "Creating a suspicious file..."
        \$malware_string3 = "Simulating network activity..."
        \$malware_string4 = "Simulating modification of a system file..."
        \$malware_string5 = "Simulating execution of a suspicious command
   ..."

    condition:
        any of (\$malware_string*)
}
"""
```

Listing 23: New YARA Rule Provided by AI

5.2.2 Main.py

Initialization

```
manager = Manager(name='wazidx1', ip_address='192.168.56.13')
defender = Defender(name='wazagent1', ip_address='192.168.56.14')
```

Step 1: Defender Downloads Fake Malware Script / Class Remote

```
defender.run_bash_script_on_remote_host('copy_malware.sh')
```

Step 2: Check for Path Modified / Class Log

```
Get the last log entry for rule ID 550

alert = manager._get_rule_id("550")

if alert:
# Extract the path from the alert
path = alert.get("syscheck", {}).get("path")
```

Step 3: Intervention of the AI / Class Evaluator

```
Request YARA rule and signature from the LLM

# Read the content of the file at the monitored path
with open(path, 'r') as file:
    file_content = file.read()

# Request YARA rule and signature from the LLM
response = llm_api.generate(prompt=f"Provide a YARA rule and
signature to detect the following malware behavior:\n{file_content}")

new_yara_rule = response['yara_rule']
active_response_to_add = response['active_response']
local_rules_to_add = response['local_rules']
```

Step 4: Manager Adds New YARA Rule / Class Yara

```
manager.add_yara_signature(new_yara_rule)
```

Step 5: Manager Updates Configuration Files / Class XML

```
manager.add_section_tag_to_conf_file(active_response_to_add)
manager.add_group_to_xml(local_rules_to_add)
```

Step 6: Defender Downloads Script Again

```
defender.run_bash_script_on_remote_host('copy_malware.sh')
```

Step 7: Check Rules and Active Response / Class FIM

```
assert(manager.check_alerts('100300'), True)
```

In summary, this scenario demonstrates the integration of all previously discussed components:

- It utilizes the Agent classes, specifically the Manager and Defender, to orchestrate the process. The Remote class facilitates the execution of scripts on remote systems, while the Log class monitors logs generated by the Manager on the Wazuh dashboard. The Evaluator class, though not fully developed, plays a role in the evaluation process.
- The XML and Ossec classes handle the configuration of necessary files. The FIM class is employed to monitor specified paths and detect changes, and the Yara class, though less detailed, is used to add signatures for newly detected malware.
- As mentioned earlier, these classes follow a hierarchical order, and calling one may invoke several others (see Appendix 39).

6 Conclusion

6.1 Project

Conclusion

Thanks to a provisioned **Vagrant** environment and a well-structured **Python class system**, we were able to address the initial thesis problem with only a few lines of code (`Main.py`, as introduced in Section 1.1.1.

The host running this environment is capable of:

- Detecting malware
- Scanning it and requesting signatures and headers from an AI module
- Making autonomous modifications to the system
- Testing and verifying those modifications

The thesis explores:

- An analysis of the problem
- A theoretical explanation of the Vagrant provisioning and supporting software
- The execution of the full environment with detailed logs and screenshots from the dashboard

Moreover, this architecture can be adapted to other contexts. Its strength lies in its **versatility** and **self-healing capability**, potentially marking the beginning of a new era in automated frameworks.

To Go Further

To improve the system, we could consider the following enhancements:

- **Configure the Evaluator Class**: Define predefined prompts for the AI and include attributes such as an HEADER that would include configuration files and class methods. This will enable the creation of complete scripts that can orchestrate additional scenarios (e.g., adding tools) and provide a versatile response to various threats.
- **Generate PDF Reports**: Implement the capability to generate PDF reports (scans) detailing the proposed scripts and the modifications that will be made.
- **Dynamic Orchestration**: Enhance the evaluator to orchestrate scenarios in the context of game theory (highlighted in appendix D.2). Currently, scenarios are script-based, but in a more advanced configuration, the system can become dynamic. Agents can operate with autonomy and modify their content independently, putting real game theory concepts into practice.

6.2 Greetings

A special thanks to Saverio Milo, who served as my tutor during this thesis at Braintech. His availability and expertise were invaluable in helping me meet deadlines and overcome various challenges.
He was particularly helpful in:

- Introducing me to core concepts of cyber-security. As a mechatronic engineer with a background in development and Python programming, I had never previously ventured into this domain.
- Assisting with debugging advanced issues beyond my knowledge, especially those involving low-level programming.

His dedication to students and his friendly demeanor were deeply appreciated.

I would also like to thank Silvio Massimino, my project manager, who allowed me to complete the thesis in time, scheduling regular meetings, and ensured I always worked under suitable conditions during my time at the company.

Finally, **my gratitude goes to the entire Braintech staff**, whose cheerfulness and sense of team cohesion made this experience enjoyable and productive.

6.3 Glossary

6.3.1 A Rule

In Wazuh, it is a key component used to detect specific patterns or conditions in monitored data, such as logs or events. When these conditions are met, the rule triggers alerts or actions, helping to identify security threats or policy violations. Rules can be customized, grouped, and assigned severity levels to prioritize responses. They are essential for automating security monitoring and integrating with other security tools. It is mainly related to signatures or headers of threats.

6.3.2 Active Response

An active response is an automated action taken by a security system to mitigate threats or anomalies. It involves predefined steps like blocking traffic or isolating systems to contain risks and gather forensic data. These responses are customizable and integrated into Wazuh.

7 Annexes

A Overview

A.1 Timeline and project management

The project spanned over 4 operational months , 2 weeks of preliminary work and a 1 month equivalent to do the report (after JAN).
It and can be divided into 3 main parts :

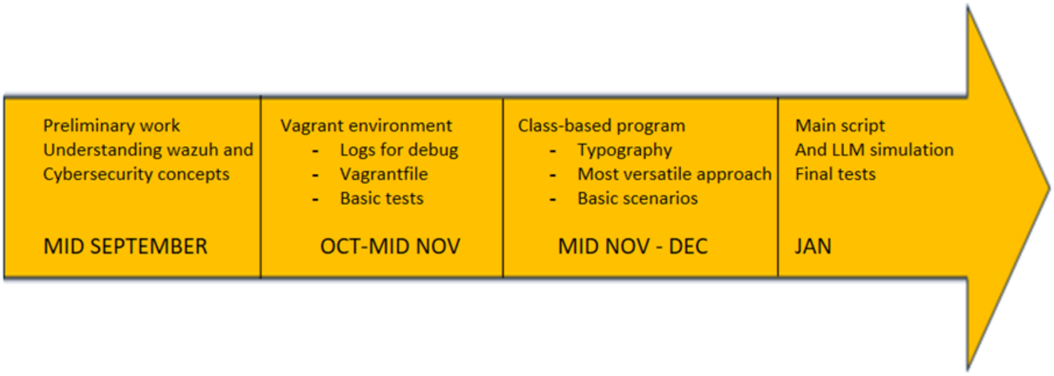
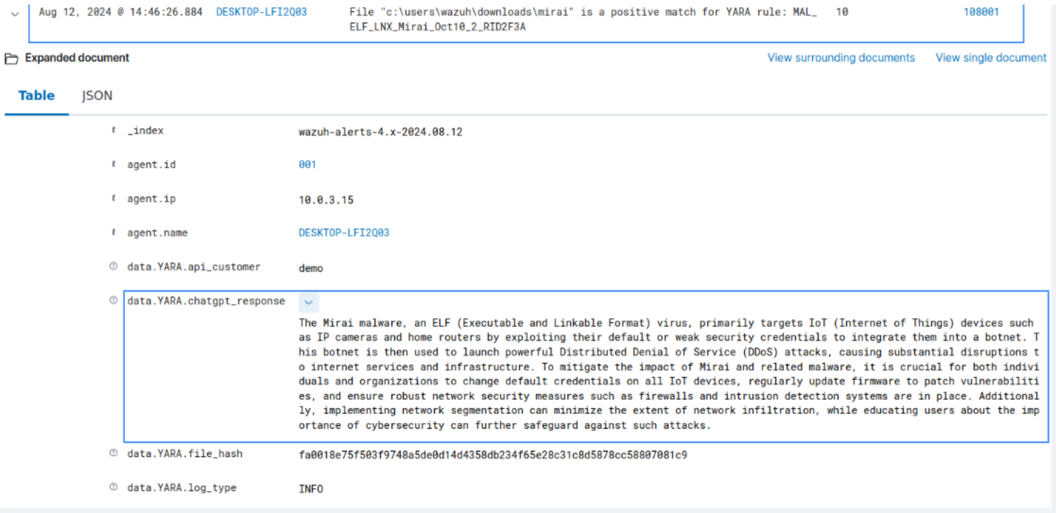


Figure 35: Project Timeline and Management

- The preliminary work was necessary to understand the main concepts in cybersecurity, despite having a background in programming.
- Setting up the Vagrant environment required significant learning time as this concept was new.
- Developing the Python class-based software was complex and required strategic planning rather than just knowledge. In January, unit tests and the main script were designed, consisting of a small scenario where the software corrects itself to handle a specific attack effectively.
- The subsequent sections will provide a more detailed approach to each part of the project.

A.2 LLM enrichment tutorial

In this tutorial, the focus is on LLM enrichment regarding file modifications (FIM) using a tool called YARA, which detects malicious packets. The logs from YARA are parsed to a language model like ChatGPT, which provides a detailed response on the malware and proposes a solution.



```
Aug 12, 2024 @ 14:46:26.884   DESKTOP-LFI2Q03   File "c:\users\wazuh\downloads\mirai" is a positive match for YARA rule: MAL_   10   108001
                                                ELF_LNX_Mirai_Oct10_2_RID2F3A
```

_index	wazuh-alerts-4.x-2024.08.12
agent.id	001
agent.ip	10.0.3.15
agent.name	DESKTOP-LFI2Q03
data.YARA.api_customer	demo
data.YARA.chatgpt_response	The Mirai malware, an ELF (Executable and Linkable Format) virus, primarily targets IoT (Internet of Things) devices such as IP cameras and home routers by exploiting their default or weak security credentials to integrate them into a botnet. This botnet is then used to launch powerful Distributed Denial of Service (DDoS) attacks, causing substantial disruptions to internet services and infrastructure. To mitigate the impact of Mirai and related malware, it is crucial for both individuals and organizations to change default credentials on all IoT devices, regularly update firmware to patch vulnerabilities, and ensure robust network security measures such as firewalls and intrusion detection systems are in place. Additionally, implementing network segmentation can minimize the extent of network infiltration, while educating users about the importance of cybersecurity can further safeguard against such attacks.
data.YARA.file_hash	fa0018e75f503f9748a5de0d14d4358db234f65e28c31c8d5878cc58807081c9
data.YARA.log_type	INFO

Figure 36: LLM Enrichment Process

B SSH Key Generation Procedure

The following steps outline the procedure for creating and using SSH keys:

1. **Generate SSH Keys**: Use the `ssh-keygen` command to create a private and public key pair.

   ```
   ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
   ```

   This command generates a private key (e.g., `id_rsa`) and a public key (e.g., `id_rsa.pub`) in the `.ssh` directory.

2. **Copy the Public Key to the Remote Server**: Add the public key to the remote server's `/.ssh/authorized_keys` file.

   ```
   ssh-copy-id user@remote_host
   ```

   Alternatively, manually append the public key to the `authorized_keys` file on the remote server.

3. **Authenticate Using the Private Key**: When logging in, the server uses the public key to encrypt a challenge, which your private key decrypts to authenticate.

4. **Secure Access**: Keep your private key secure and consider using a passphrase for added security.

5. **Manage Keys**: Regularly update and manage your keys to maintain security.

C Vagrant Provisionning

Vagrantfile pseudocode

1. **SET configuration variables:**

   PARAMS

   MANAGER, CPUS_MANAGER, MEM_MANAGER, CPUS_AGENT, MEM_AGENT, MANAGER_NAME, AGENT1_NAME, AGENT2_NAME

2. **SET UP the host machine:**

   INIT

   Installing necessary packages on the host.

3. **SET Vagrant box to "ubuntu/jammy64"**

4. **DEFINE list of nodes with parameters:**

   PARAMS

   NODES = [MANAGER, AGENT1, AGENT2]

5. **ASSIGN IPs to nodes:**

   USUAL

   Each VM has its own IP, careful to not choose an already taken IP.

6. **FOR EACH node IN NODES:**

   USUAL

   - DEFINE Vagrant VM configuration
   - CONFIGURE VirtualBox provider settings
   - PROVISION common requirements
   - GENERATE and DISTRIBUTE SSH keys
   - IF node type is "server":
     - Configure the manager on Wazuh
   - IF node type is "agent":
     - Configure the agent on Wazuh

7. **DEFINE specific triggers to run after destroying or upping a VM**

C.1 Cron jobs

The following cron job has been implemented to ensure synchronization between `agent.conf` on the host and on the manager:

```

    source sourcer.sh variables.sh
    LOG_FILE=$PATH_TO_CRON_JOB_LOGS
    # Synchronize agent.confs between host and server
    check_file_existence $LOG_FILE
    * * * * * root inotifywait -m -e modify "$PATH_TO_VICTIM_AGENT_CONF_FILE" | while read; do synchronize_file "$PATH_TO_VICTIM_AGENT_CONF_FILE" "$PATH_TO_SERVER_VICTIM_AGENT_CONF_FILE" >> "$PATH_TO_CRON_JOB_LOGS" 2>&1; done

    # Add additional cron jobs here as needed
    # SURICATA #
    # sudo tee /etc/cron.d/config_customs_rules_update.sh <<EOF
    # 0 3 * * * root /vagrant/VM/configs/config_customs_rules_update.sh
    EOF

```

Listing 24: Cron job

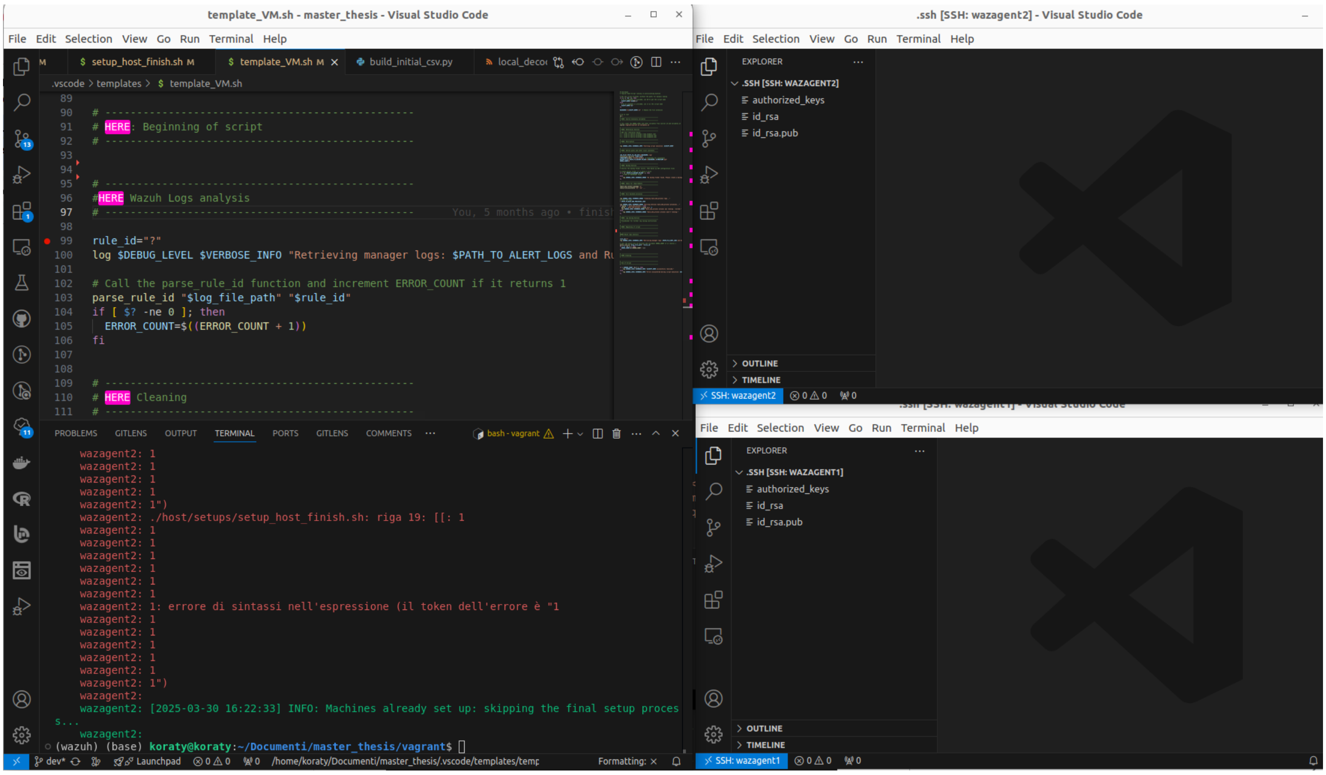
C.2 cleaner.sh

```
# Cleaning script: cleaning.sh
    #!/bin/bash

    # Ensure the script runs only once
    if [ -f "CLEARED" ]; then
        echo "Cleanup already performed."
        exit 0
    fi

    echo "Backing up SSH configurations..."
    cp -r ~/.ssh /path/to/backup/

    echo "Clearing specific files..."
    find /path/to/keys -name "*.txt" -o -name "*.pub" -exec cat {} \; >> /path/to/central_file
    find /path/to/keys -name "*.txt" -o -name "*.pub" -exec rm {} \;

    echo "Removing .tar files..."
    rm -f /path/to/logs/*.tar

    echo "Deleting log files..."
    rm -f /path/to/logs/*.log

    echo "Cleanup complete."
    touch "CLEARED"
```

Listing 25: cleaner.sh

C.3 Execution

C.3.1 Ssh connection via Virtual Studio Code



template_VM.sh - master_thesis - Visual Studio Code
File Edit Selection View Go Run Terminal Help
setup_host_finish.sh M
template_VM.sh M
build_initial_csv.py
local_decod
.vscode > templates > template_VM.sh
PROBLEMS GITLENS OUTPUT TERMINAL PORTS GITLENS COMMENTS
bash - vagrant
dev* Launchpad
/home/koraty/Documenti/master_thesis/.vscode/templates/temp
Formatting: ×
.ssh [SSH: wazagent2] - Visual Studio Code
EXPLORER
.SSH [SSH: WAZAGENT2]
authorized_keys
id_rsa
id_rsa.pub
OUTLINE
TIMELINE
SSH: wazagent2
.ssh [SSH: wazagent1] - Visual Studio Code
.SSH [SSH: WAZAGENT1]
SSH: wazagent1

```

# ----------------------------------------------------
# HERE: Beginning of script
# ----------------------------------------------------


# ----------------------------------------------------
#HERE Wazuh Logs analysis
# ----------------------------------------------------        You, 5 months ago • finish

rule_id="?"
log $DEBUG_LEVEL $VERBOSE_INFO "Retrieving manager logs: $PATH_TO_ALERT_LOGS and Ru

# Call the parse_rule_id function and increment ERROR_COUNT if it returns 1
parse_rule_id "$log_file_path" "$rule_id"
if [ $? -ne 0 ]; then
  ERROR_COUNT=$((ERROR_COUNT + 1))
fi


# ----------------------------------------------------
# HERE Cleaning
# ----------------------------------------------------
```

```
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1")
    wazagent2: ./host/setups/setup_host_finish.sh: riga 19: [[: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1: errore di sintassi nell'espressione (il token dell'errore è "1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1
    wazagent2: 1")
    wazagent2:
    wazagent2: [2025-03-30 16:22:33] INFO: Machines already set up: skipping the final setup proces
s...
    wazagent2:
○ (wazuh) (base) koraty@koraty:~/Documenti/master_thesis/vagrant$ 
```

Figure 37: VSCode connection to virtual machines.

C.3.2 SSH Tests

```
wazagent2: [2025-04-06 17:26:54] DEBUG: Current Machine hostname: koraty
wazagent2: [2025-04-06 17:26:54] INFO: Testing SSH connection from both sides
    ...
wazagent2: [2025-04-06 17:26:54] INFO: Check for logs in  /vagrant/log
wazagent2: [2025-04-06 17:26:55] DEBUG: The NODES are : wazidx1
wazagent2: [2025-04-06 17:26:55] INFO: VM hostname found: wazidx1
wazagent2: [2025-04-06 17:26:55] INFO: Testing SSH connection from root to
    wazidx1...
wazagent2: SSH connection successful from root to wazidx1
wazagent2: [2025-04-06 17:26:55] INFO: VM hostname found: waagent1
wazagent2: [2025-04-06 17:26:55] INFO: Testing SSH connection from root to
    waagent1...
wazagent2: SSH connection successful from root to waagent1
wazagent2: [2025-04-06 17:26:55] INFO: VM hostname found: waagent2
wazagent2: [2025-04-06 17:26:55] INFO: Testing SSH connection from root to
    waagent2...
wazagent2: SSH connection successful from root to waagent2
```

Listing 26: Testing ssh connection

D Src folder

D.1 Overall Methods

Class	Methods
Manager (Level 0)	`__init__`
Vmagent (Level 1)	`_create_ossec_conf_attributes`, `_create_remote_attributes`, `_create_xml_handler_attributes`
Fim (Level 2)	`check_fim_change`, `configure_audit`, `manage_ip_in_hosts_allow`, `simulate_file_change`, `test_who_data`
Log (Level 3)	`check_alerts`, `extract_time`, `get_rule_id`, `is_time_consistent`, `parse_rule_id`
Remote (Level 4)	`check_remote_file`, `remove_remote_file`, `restart`, `run_function_on_remote_host`, `run_remote_command`, `synchronize_with_VM`
Agents (Level 5)	`_create_instance`, `_detect_valid_types`, `get_all_instances`, `get_attacker`, `get_defender`, `get_evaluator`, `get_manager`, `initialize`, `interact_with_llm`, `modify_ossec`, `start`, `stop`
Command (Level 6)	`_run_subprocess`, `_sanitize_command`, `check_and_install_packages`, `kill_related_subprocesses`, `run_command`, `run_command_with_pipe`
Typography (Level 7)	`to_dataframe`
Ossec_conf (Level 3)	`add_active_response`, `add_command`, `add_fim`, `add_fim_configuration`, `add_local_file`, `check_in_tag`, `compress_elements`, `compress_tree`, `compress_xml`, `create_nested_elements`, `dataframe_from_conf`, `extract_section`, `extract_wodle_section`, `generate_ossec_conf`, `load_dataframes`, `print_node_text`, `process_element`, `save_dataframes`, `synchronize_df_with_conf_file`, `verify_conf_file_via_bash`
Xml_handler (Level 4)	`_format_save_xml`, `_validate_xml_file`, `add_decoder_to_xml`, `add_group_to_xml`, `load_xml_file`, `save_xml_file`
Yara (Level 2)	`check_and_install_yara`

Table 11: Overall Methods

D.2 Non-Developed Folders

These folders weren't developed but can be a great prospect for further development.

```
[DIR] integrations
    \- mininet_integration.py
```

To integrate other actors in the system that may not be virtual machines but simply stranger entities that could create noise and simulate a more realistic environment.

```
[DIR] dependencies
    .
    .
    |- __init__.py
    |- graph.py
    \- llm_integration.py
```

For further improvements:

- To have a graph representation of the system
- To integrate an LLM

E Prototype execution

E.1 Yara installation logs

Yara installation

```
[2025-04-28 16:04:11] DEBUG: Downloading YARA source code...
[2025-04-28 16:04:14] DEBUG: Extracting YARA archive...
[2025-04-28 16:04:14] DEBUG: Starting YARA build process...
[2025-04-28 16:05:21] INFO: YARA build process completed successfully.
[2025-04-28 16:05:21] DEBUG: libyara.so.9 found.
[2025-04-28 16:05:21] DEBUG: Verifying YARA installation...
[2025-04-28 16:05:21] DEBUG: YARA installed successfully. Version: 4.2.3
[2025-04-28 16:05:21] DEBUG: Downloading YARA detection rules...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3298k  100 3298k  100    93   352k      9  0:00:10  0:00:09  0:00:01  488k
[2025-04-28 16:05:31] INFO: YARA rules downloaded successfully.
[2025-04-28 16:05:31] DEBUG: Copying yara.sh file to the correct location...
[2025-04-28 16:05:31] DEBUG: Changing yara.sh ownership and permissions...
[2025-04-28 16:05:31] DEBUG: yara_setup.sh completed successfully.
```

Unit Test Pseudocode

SETUP

- Initialize Manager and Defender instances.
- Set path to ossec testing conf file.
- Define section tags and raw XML strings for testing.
- Remove section tags from ossec.conf file before testing to ensure a clean state.

TEST OSSEC CONF HANDLING

- Test conversion from tag to dataframe.
- Assert expected output matches the result.
- Test adding a section tag to ossec.conf file.
- Reload configuration data from Excel.
- Assert ossec.conf has been correctly modified.

TEST FIM SETUP

- Define monitored paths and directory tags.
- Add FIM configuration.
- Test FIM and Who-Data functionality.

TEST XML HANDLING

- Add group and decoder sections to XML.
- Synchronize XML with VM.
- Check for working modification of new added rules.

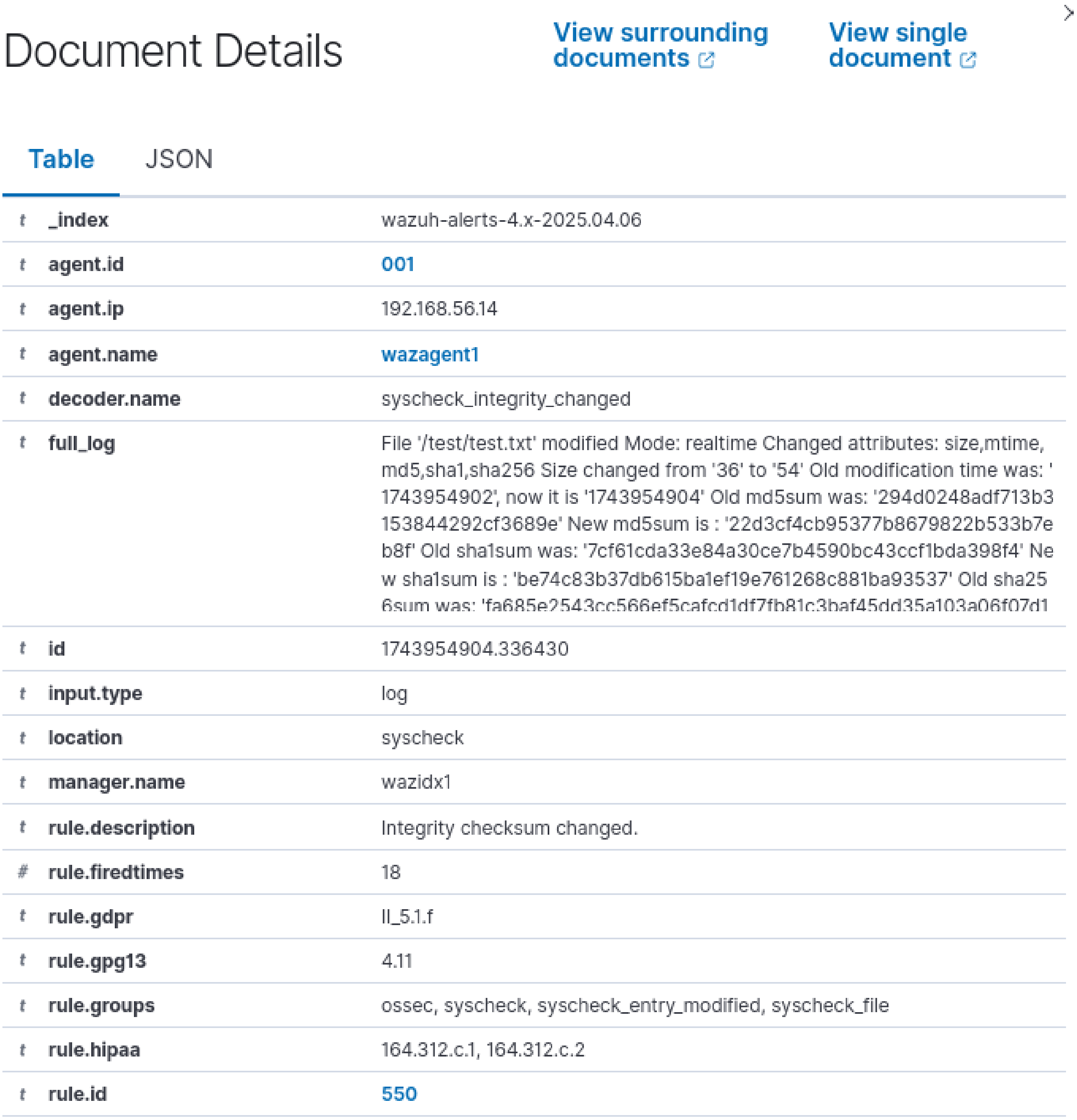
TEST YARA CONFIGURATION

- Configure ossec.conf for YARA.
- Add YARA configuration on the endpoint.
- Generate updated ossec.conf XML.
- Run bash script on remote host.
- Check alerts.

RUN TESTS

- Create a test suite with specific methods.
- Run the test suite.

E.2 FIM detail modification



Document Details

t	_index	wazuh-alerts-4.x-2025.04.06
t	agent.id	001
t	agent.ip	192.168.56.14
t	agent.name	wazagent1
t	decoder.name	syscheck_integrity_changed
t	full_log	File '/test/test.txt' modified Mode: realtime Changed attributes: size,mtime, md5,sha1,sha256 Size changed from '36' to '54' Old modification time was: '1743954902', now it is '1743954904' Old md5sum was: '294d0248adf713b3153844292cf3689e' New md5sum is : '22d3cf4cb95377b8679822b533b7eb8f' Old sha1sum was: '7cf61cda33e84a30ce7b4590bc43ccf1bda398f4' New sha1sum is : 'be74c83b37db615ba1ef19e761268c881ba93537' Old sha256sum was: 'fa685e2543cc566ef5cafcd1df7fb81c3baf45dd35a103a06f07d1
t	id	1743954904.336430
t	input.type	log
t	location	syscheck
t	manager.name	wazidx1
t	rule.description	Integrity checksum changed.
#	rule.firedtimes	18
t	rule.gdpr	II_5.1.f
t	rule.gpg13	4.11
t	rule.groups	ossec, syscheck, syscheck_entry_modified, syscheck_file
t	rule.hipaa	164.312.c.1, 164.312.c.2
t	rule.id	550

Figure 38: Detailed view of the file modification alert.

E.3 FIM logs

FIM Setup Logs

```
Starting FIM setup...
17:54:52,526 - Generating ossec.conf from provided dictionaries.
17:54:52,527 - Redundant entries found in tag 'directories' with attributes
frozenset({('check_all', 'yes'), ('whodata', 'yes')}). Removed duplicates.
Saving dataframes to Excel file: conf_files/ossec_agent_base.xlsx.
Backup saved to conf_files/backup/ossec_agent_base.xlsx.20250406175452
Removed old backup: conf_files/backup/ossec_agent_base.xlsx.20250406173117
17:54:52,653 - Saving ossec.conf to conf_files/ossec_agent_base.conf.
Successfully saved the .conf file at conf_files/ossec_agent_base.conf
17:54:52,661 - Synchronizing with VM...
```

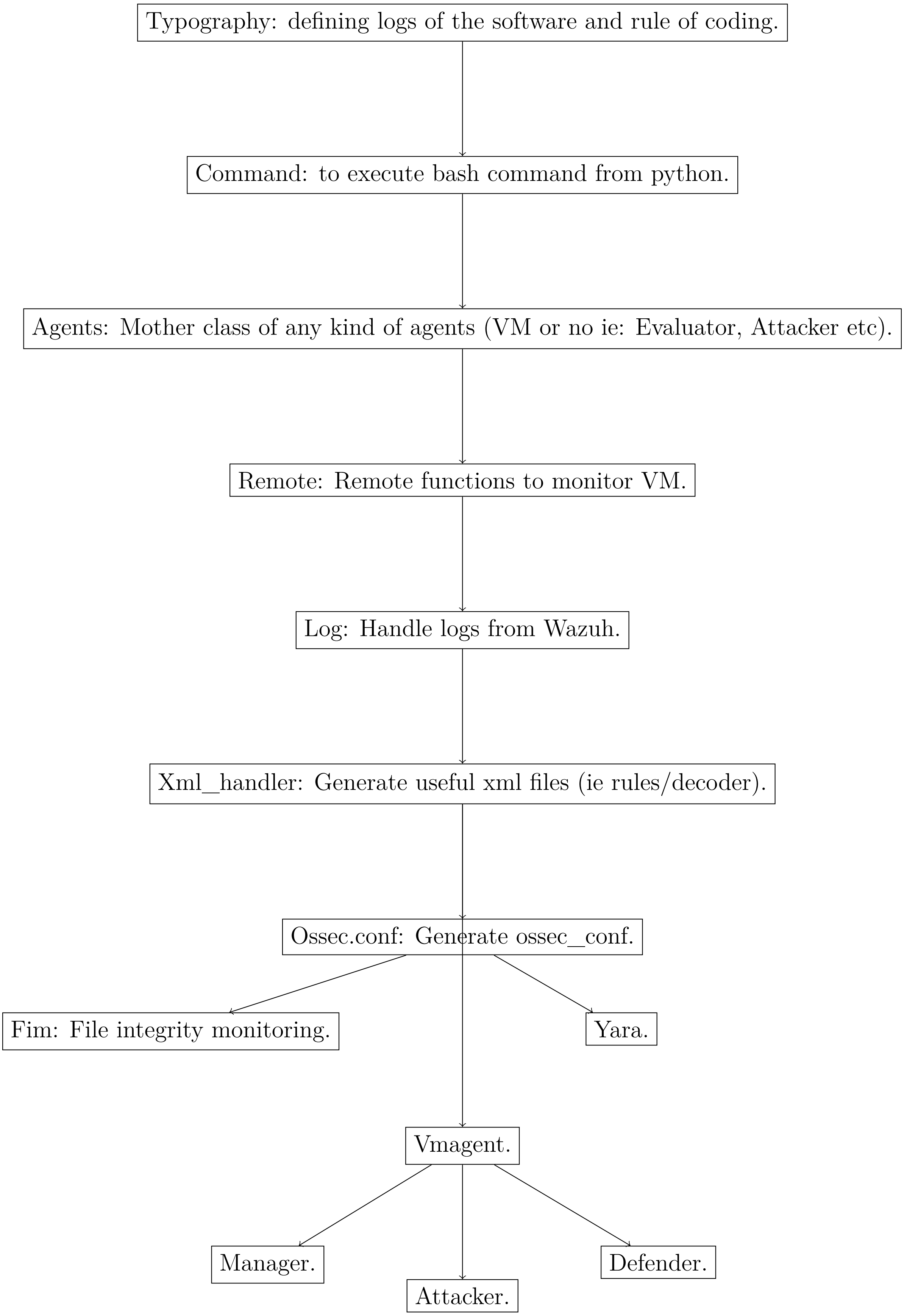
Verifying change of ossec before making the changes.

```
[17:54:53] Received conf_file path: /vagrant/src/conf_files/ossec_agent_base.conf
[17:54:53] Verifying configuration with: wazuh-logcollector
[17:54:53] Command 'wazuh-logcollector' verified successfully.
[17:54:53] Verifying configuration with: wazuh-modulesd
[17:54:53] Command 'wazuh-modulesd' verified successfully.
[17:54:53] Verifying configuration with: wazuh-agentd
[17:54:53] Command 'wazuh-agentd' verified successfully.
[17:54:53] Verifying configuration with: wazuh-syscheckd
[17:54:53] Command 'wazuh-syscheckd' verified successfully.

[17:54:53] Successfully synchronized /vagrant/src/conf_files/ossec_agent_base.conf wit
/var/ossec/etc/ossec.conf - see
[17:54:53] Restarting service...
[17:55:01] Successfully restarted wazuh-agent.
[17:55:01] Service restarted successfully
```

Simulating a Change

```
17:55:04,779 - Checking if directory '/test' was modified regarding rule ID 550.
17:55:09,785 - Starting VM synchronization for file: log/alerts.json to
wazidx1:/var/ossec/logs/alerts/alerts.json.
17:55:10,347 - Synchronization successful:
17:55:10,347 - Running function _get_rule_id for rule ID 550
17:55:10,355 - Directory '/test' was modified as per rule ID 550.
It occurred at file /test/test.txt.
```

Figure 39: Overall class-based configuration of the system


Typography: defining logs of the software and rule of coding.
Command: to execute bash command from python.
Agents: Mother class of any kind of agents (VM or no ie: Evaluator, Attacker etc).
Remote: Remote functions to monitor VM.
Log: Handle logs from Wazuh.
Xml_handler: Generate useful xml files (ie rules/decoder).
Ossec.conf: Generate ossec_conf.
Fim: File integrity monitoring.
Yara.
Vmagent.
Manager.
Attacker.
Defender.