Master Thesis

Reinforcement and modernization of the MISP open source application

Thomas Lacroix

Academic Year 2024–2025

Final year internship done in partnership with

CIRCL - Computer Incident Response Center Luxembourg

in preparation for the engineering diploma of TELECOM Nancy

Internship supervisor: Sami Mokaddem

Academic supervisor: Jannik Dreier

Master Thesis

Reinforcement and modernization of the MISP open source application

Thomas Lacroix

Academic Year 2024–2025

Final year internship done in partnership with

CIRCL - Computer Incident Response Center Luxembourg

in preparation for the engineering diploma of TELECOM Nancy

Thomas Lacroix
11b, Rue du Général Duroc
54 000, Nancy
06 46 05 70 23
thomas.lacroix@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

CIRCL - Computer Incident Response Center Luxembourg
122, Rue Adolphe Fischer
L-1521, Luxembourg
+352 274 00 98 625

Internship supervisor: Sami Mokaddem

Academic supervisor: Jannik Dreier

Acknowledgements

First of all, I would like to express my gratitude to the team of Luxembourg House of Cybersecurity, particularly the human resources department, for their warm welcome and support throughout my internship.

I would also like to thank Alexandre Dulaunoy, head of CIRCL, for the trust he placed in me and for welcoming me into his team.

I would especially like to thank Sami Mokaddem, my internship supervisor, for his constant support, the time he devoted to me, and his availability and empathy throughout my internship.

I would also like to thank Jannik Dreier, my university supervisor, for his availability and valuable advice, which helped me to successfully complete this internship and write my thesis.

Finally, I would like to thank all the members of CIRCL whom I had the opportunity to work with on a daily basis and who welcomed me with kindness: Cédric, David, Michael, Andras, Quentin, Luciano, Sascha, Deborah, Christian, Aurélien, and Gérard.

Contents

1 Introduction

Cybersecurity threats are becoming more frequent and sophisticated in today's world, which is becoming more and more digital. Data breaches, ransomware attacks, and other malicious activities that can cause serious harm are a constant risk for governments, companies, and individuals. The need for efficient tools for detecting, evaluating, and disseminating threat intelligence across borders and organizations is growing along with our dependence on digital infrastructures.

By creating and maintaining open-source tools that facilitate information sharing among organizations regarding cybersecurity threats, CIRCL (Computer Incident Response Center Luxembourg) plays a crucial role in this effort. One of these tools is MISP Threat Sharing, a popular threat intelligence platform made to enhance cyberattack detection and prevention via cooperative data exchange. The work of CIRCL benefits a worldwide user community that depends on timely and accurate information to strengthen their security systems.

During my internship at CIRCL, I contributed to this mission by both developing new features and fixing issues within the MISP platform. In the first part of my work, I focused on enhancing the application's functionality and stability. In the second part, I designed and executed tests to ensure proper synchronization of data across multiple interconnected MISP instances. This work was essential for validating that critical threat intelligence could be reliably shared across organizations using the platform.

2 Internship context

2.1 Company overview

CIRCL, created on 1 January 2008, is the national CERT for the private sector, municipalities and non-governmental entities in Luxembourg. CIRCL is therefore a public service operated by the LHC (Luxembourg House of Cybersecurity), a Luxembourg public entity. LHC hosts other public services such as NC3 (National Cybersecurity Competence Center) and is funded by several Luxembourg ministries and unions, including those responsible for the economy, education and family affairs. The service currently consists of 14 people, including security researchers, incident analysts and developers [5]. Among the key tasks to be performed by CIRCL are:

- Provide a systematic response to incidents: coordination, analysis, correlation, investigation and remediation.
- Minimise impacts: losses, data theft or service disruption at national level.
- Share knowledge and alerts with ICT users in Luxembourg and internationally.

One of CIRCL's main activities is therefore the creation and sharing of open-source software. This software has various objectives, such as sharing information on threats for MISP, searching for technical information on CVE vulnerabilities for Vulnerability Lookup, and analysing information leaks on the Internet with the AIL (Analysis Information Leak) framework.

2.2 Internship Issue

2.2.1 Initial project description

The objective of this internship was to participate in the modernisation of MISP Threat Sharing, one of the main applications developed by CIRCL. This application was constantly in need of new features and fixes, while ensuring the reliability of the code it provided. Initially, I contributed to the development and improvement of the application's frontend and backend. Then, I worked on the CI/CD pipeline to enhance code reliability by implementing secure coding practices and automated testing. The work was carried out in an open source environment, with Pull Requests submitted to present achievements to the project.

2.2.2 Personal Objective

During this internship, I set myself two goals. The first was to develop my technical skills in software development. This was possible because I worked on an important application for which I had to acquire a deep technical understanding of its architecture. I also learned for the first time how to perform automated testing on an application via the CI/CD pipeline. The second goal was to develop my knowledge of cyber threats by working in an environment where the flow and sharing of threat information is constant.

2.3 Technologies used

2.3.1 CakePHP

MISP is a web application based on the **CakePHP** framework, a framework that primarily uses PHP but also other web development languages such as HTML/CSS and JavaScript. CakePHP allows for the management of business logic (such as event management, attributes, users, organisations), interactions with the database, and the generation of views.

2.3.2 Python

Python is an interpreted, versatile programming language widely used in cybersecurity and data analysis. In this project, I used **PyMISP**, a Python client for the MISP REST API, to develop and run test scripts to verify the proper functioning of synchronization between multiple MISP instances.

2.3.3 Docker

Docker is a containerisation technology that allows MISP and its dependencies to be packaged and run in isolated, reproducible environments. In the context of development or testing, Docker facilitates the rapid deployment of a MISP instance with MariaDB, Redis, and associated modules. This approach simplifies the setup of multiple instances for testing synchronization scenarios without polluting the host system environment.

2.3.4 LXC

LXC (Linux Containers) is a lightweight virtualisation technology used to run MISP in an environment closer to that of a virtual machine. Unlike Docker, LXC provides finer process separation and a complete file system. In some CI configurations, LXC allows you to simulate MISP deployments close to production.

2.3.5 GitHub Actions

GitHub Actions is a continuous integration and deployment (CI/CD) platform integrated with GitHub. It has long been used to automate MISP testing with each new contribution (pull request). GitHub Actions workflows allow you to run PHP, Python, or Shell scripts to ensure that features are not broken by changes.

2.3.6 Forgejo Actions

Forgejo Actions is the equivalent of GitHub Actions for the decentralised software forge *Forgejo*, a community fork of Gitea. It allows you to run CI/CD workflows locally or in a private instance.

3 State of the art of MISP

3.1 Project history

In 2011, Christophe Vandeplas, a cybersecurity analyst, wanted to solve a rather annoying problem in the sharing of indicators of compromise. Most of them were shared by email or PDF document, which made their analysis and use quite time-consuming. To remedy this, he decided to develop his own tool capable of providing this information. Christophe Vandeplas first presented an initial version of his CyDefSIG (Cyber Defense Signatures) project to the Belgian armed forces, where he worked. NATO then heard about the project and began assigning developers to it. The project's reputation began to spread throughout various CERTs in Europe, including CIRCL, which is now the organization employing the project's developers. As for the software license, it was collaboratively decided to place it under the Affero GPL license so that the code could be publicly shared, thus ensuring the application's longevity. The project is funded by the European Union (through the Connecting Europe Facility) and the Computer Incident Response Center Luxembourg. Nowadays, the MISP application is much more than just a malware indicator sharing tool, from the numerous tools such as PyMISP to the various formats supported, the entire project is now called MISP Threat Sharing.

3.2 Open source development and community collaboration

MISP is an open source project actively developed and maintained on the GitHub platform. Being open source allows anyone to inspect the code and suggest improvements in addition to the four developers working within CIRCL. This collaborative development model is manifested through *pull requests*, in which contributors detail the changes they have made and their motivations. These *pull requests* are then reviewed by the main maintenance developers, who ensure the quality of the code and its compliance with the project's objectives before it is integrated.

In addition to code contributions, the *Issue* section of the repository plays a crucial role in the community's collaborative work. Users can report bugs, suggest new features, or ask questions for clarification. This open section enables a transparent development cycle, where decisions and corrections are visible to all interested parties.

3.3 Technical Architecture

The MISP architecture is composed of several entirely open source components, which allows to have a finished product that is as transparent as possible and fully community-driven.

3.3.1 Web Interface

At the heart of the application is the web interface, built on the **CakePHP** framework, an open source MVC framework written in PHP. This choice allows for a clear separation between data models, application logic, and the user interface. The use of CakePHP facilitates fast development, consistent code structuring, and long-term maintainability. It also allows for the integration of additional modules and APIs, which extend MISP's functionality without compromising its fundamental stability.

3.3.2 Data storage

Persistent data storage in MISP uses a relational database management system **MariaDB**. It is a fully open source fork of MySQL, wihch offers high performance and full compatibility with existing MySQL tools and libraries.

In addition to relational storage, MISP uses **Redis**, an in-memory key-value database engine. Redis is used to cache frequently accessed data, manage sessions, and store temporary objects, which significantly improves the responsiveness of the application. Using a hybrid model like this find the rigth balance between durability and data access, which is important in production deployments where data integrity and performance are essential.

3.3.3 Background workers

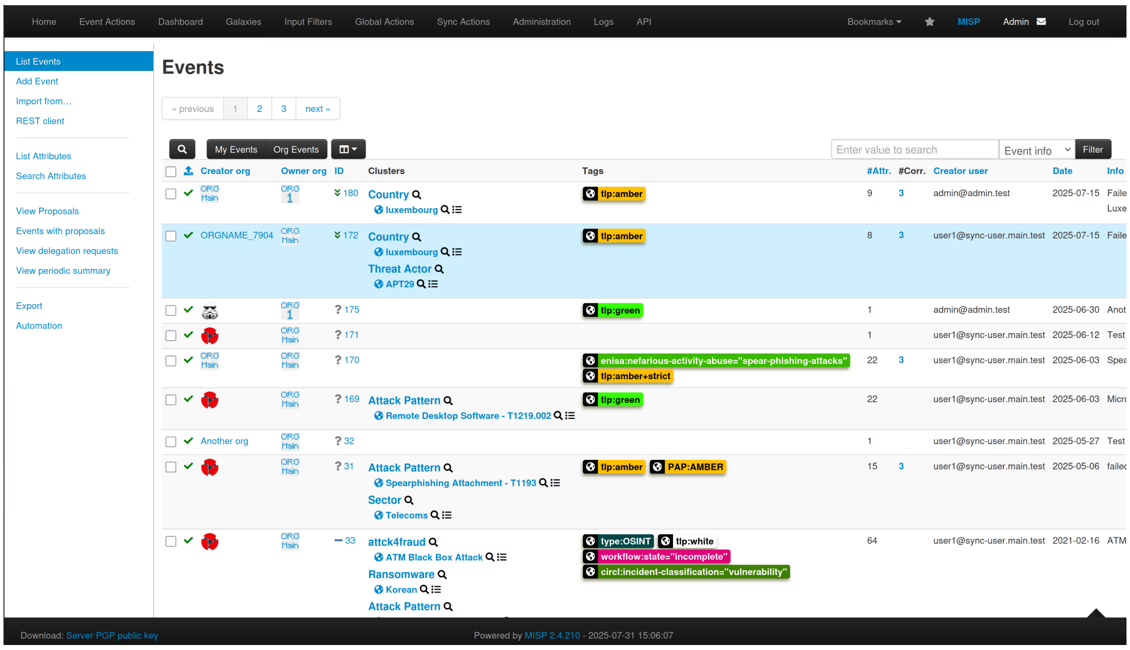
To manage tasks that cannot be executed in the main web process, MISP integrates a system of background workers managed by **supervisord**, a lightweight process control system. These workers are responsible for asynchronous tasks, such as synchronizing events between different instances or delivering notifications.

3.4 Main features

3.4.1 Identity and Access Management

Before discussing the practical features of the application, it is important to understand how access management is handled in MISP. Each deployment of the MISP application on a machine is called an instance. On each instance of the application, every user necessarily belongs to an organization. An organization is a group of users. For example, if we take the case of the financial sector sharing community managed by CIRCL, CIRCL is the host organization of the instance and is able to add an organization for each company in the financial sector that wishes to have access to this instance. In addition, each user in an organization has their own role: Org admin (control over users in their own organization), User (can create, edit, and view events according to their rights), Read Only (can only view events).

After a user logs into the application, they are automatically redirected to the list of all events present on the application (see Figure 3.1). This view allows users to quickly view information about an event, such as its creator, associated clusters, and tags.



Creator org	Owner org	ID	Clusters	Tags	#Attr.	#Corr.	Creator user	Date	Info
ORG Main	ORG 1	180	Country luxembourg	tlp:amber	9	3	admin@admin.test	2025-07-15	Faile Luxe
ORGNAME_7904	ORG Main	172	Country luxembourg Threat Actor APT29	tlp:amber	8	3	user1@sync-user.main.test	2025-07-15	Faile
	ORG 1	? 175		tlp:green	1		admin@admin.test	2025-06-30	Anot
	ORG Main	? 171			1		user1@sync-user.main.test	2025-06-12	Test
ORG Main	ORG Main	? 170		enisa:nefarious-activity-abuse="spear-phishing-attacks" tlp:amber+strict	22	3	user1@sync-user.main.test	2025-06-03	Spea
	ORG Main	? 169	Attack Pattern Remote Desktop Software - T1219.002	tlp:green	22		user1@sync-user.main.test	2025-06-03	Micr
Another org	ORG Main	? 32			1		user1@sync-user.main.test	2025-05-27	Test
	ORG Main	? 31	Attack Pattern Spearphishing Attachment - T1193 Sector Telecoms	tlp:amber PAP:AMBER	15	3	user1@sync-user.main.test	2025-05-06	failed
	ORG Main	— 33	attck4fraud ATM Black Box Attack Ransomware Korean Attack Pattern	type:OSINT tlp:white workflow:state="incomplete" circl:incident-classification="vulnerability"	64		user1@sync-user.main.test	2021-02-16	ATM

Download: Server PGP public key

Powered by MISP 2.4.210 - 2025-07-31 15:06:07

Figure 3.1: Landing page of the application (here a training instance)

3.4.2 Key items

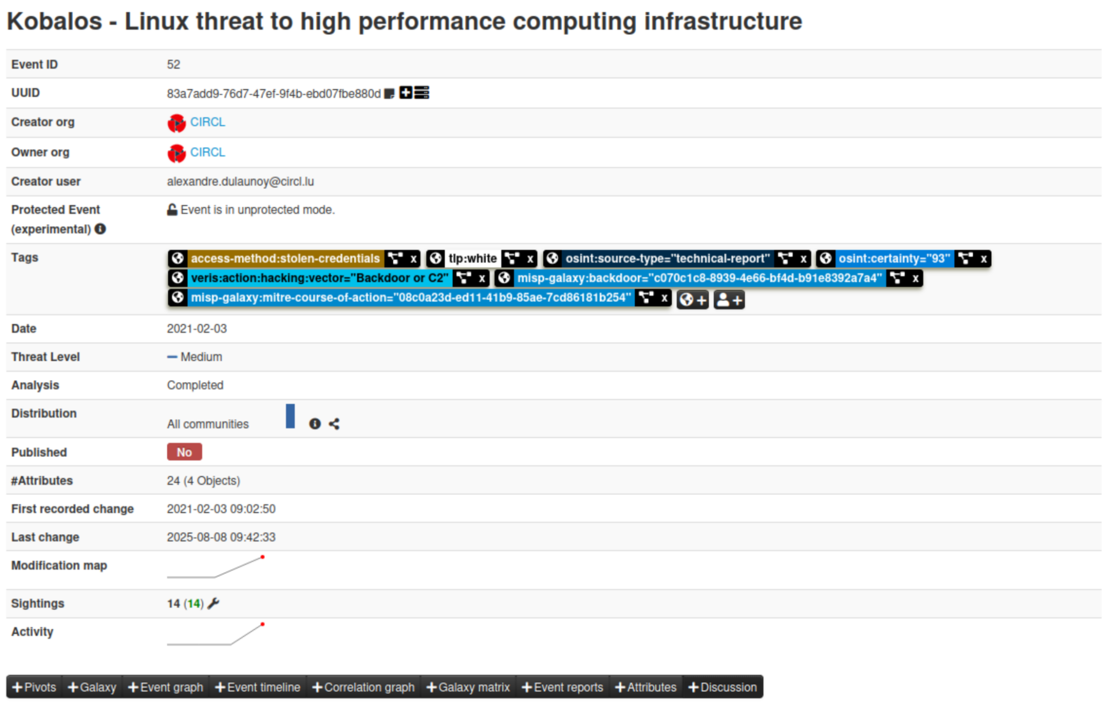
In MISP, the central object is the **Event**. An event represents an incident or situation and contextually groups together all the information related to it. Each event has basic metadata, such as its owner, access rights, creation date, and sharing level (see Figure 3.2).

An event can contain several other types of objects, including [3]:

- **Attributes**: these are atomic objects that represent information (e.g., IP address, domain name, file hash). They can be enriched by taxonomies and specify whether they are intended to provide human context or to be used in an automated manner (see Figure 3.3).
- **Objects**: these are groupings of attributes based on a template For example, a *file* object may contain a file name, an MD5 hash, an SHA1 hash, etc.
- **Tags and Taxonomies**: these are standardized labels based on common vocabularies, used to classify or qualify data.
- **Galaxies and Galaxy Clusters**: these are advanced labels, enriched with metadata. *Galaxies* represent a conceptual set (e.g., groups of attackers, malware, countries), and *clusters* are specific instances of them (see Figure 3.4)) .
- **Event Graph**: a graphical representation showing the relationships between the different entities contained in the event.

- **Event Timeline**: a chronological representation of the data contained in the event, allowing its temporality to be visualized.
- **Event Report**: a text document (supporting the Markdown format) allowing the event or incident to be described in more detail, with narrative context.

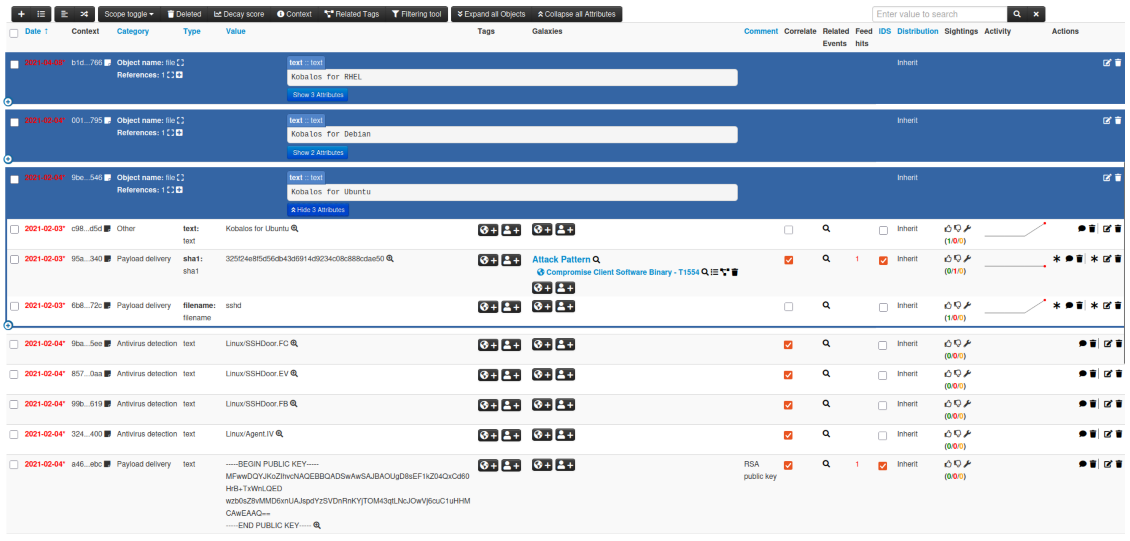
These different objects, when combined, enable MISP to effectively represent, structure, and enrich cybersecurity-related information. The values stored in the attributes can then be exported to network protection equipment (firewalls, intrusion detection systems, etc.) in order to automate and strengthen defense measures.



Kobalos - Linux threat to high performance computing infrastructure

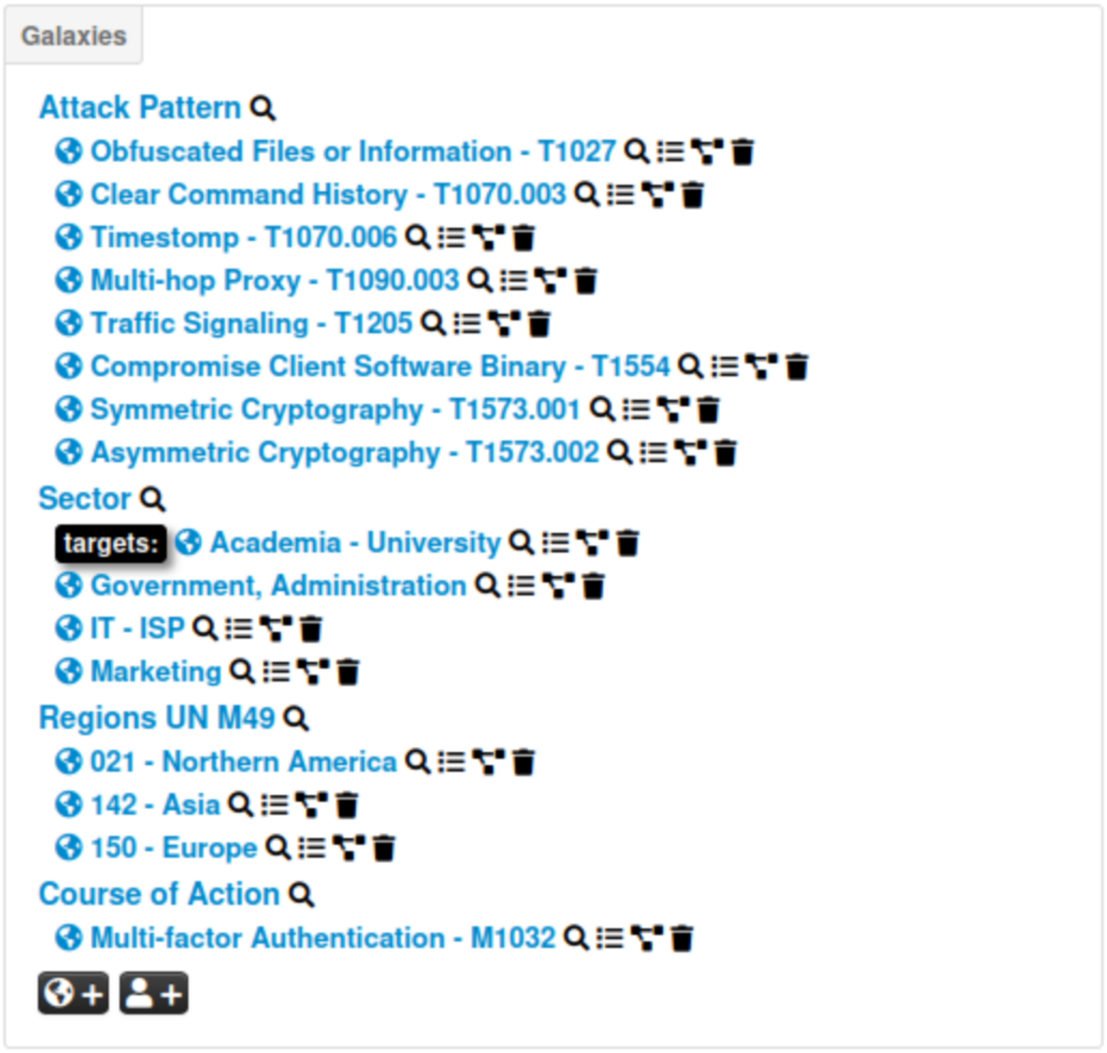
Event ID	52
UUID	83a7add9-76d7-47ef-9f4b-ebd07fbe880d
Creator org	CIRCL
Owner org	CIRCL
Creator user	alexandre.dulaunoy@circl.lu
Protected Event (experimental)	Event is in unprotected mode.
Tags	access-method:stolen-credentials x tlp:white x osint:source-type="technical-report" x osint:certainty="93" x veris:action:hacking:vector="Backdoor or C2" x misp-galaxy:backdoor="c070c1c8-8939-4e66-bf4d-b91e8392a7a4" x misp-galaxy:mitre-course-of-action="08c0a23d-ed11-41b9-85ae-7cd86181b254" x
Date	2021-02-03
Threat Level	Medium
Analysis	Completed
Distribution	All communities
Published	No
#Attributes	24 (4 Objects)
First recorded change	2021-02-03 09:02:50
Last change	2025-08-08 09:42:33
Modification map	
Sightings	14 (14)
Activity	

Figure 3.2: Example of basic information associated with an Event in MISP



Date	Context	Category	Type	Value	Tags	Galaxies	Comment	Correlate	Related Events	Feed hits	IDS	Distribution	Sightings	Activity	Actions
2021-04-08*	b1d...766	Object name: file References: 1		text :: text Kobalos for RHEL Show 3 Attributes								Inherit			
2021-02-04*	001...795	Object name: file References: 1		text :: text Kobalos for Debian Show 2 Attributes								Inherit			
2021-02-04*	9be...546	Object name: file References: 1		text :: text Kobalos for Ubuntu Hide 3 Attributes								Inherit			
2021-02-03*	c98...d5d	Other	text: text	Kobalos for Ubuntu				☐			☐	Inherit	(1/0/0)		
2021-02-03*	95a...340	Payload delivery	sha1: sha1	325f24e8f5d56db43d6914d9234c08c888cdae50		Attack Pattern Compromise Client Software Binary - T1554		✓		1	✓	Inherit	(0/1/0)		
2021-02-03*	6b8...72c	Payload delivery	filename: filename	sshd				☐			☐	Inherit	(1/0/0)		
2021-02-04*	9ba...5ee	Antivirus detection	text	Linux/SSHDoor.FC				✓			☐	Inherit	(0/0/0)		
2021-02-04*	857...0aa	Antivirus detection	text	Linux/SSHDoor.EV				✓			☐	Inherit	(0/0/0)		
2021-02-04*	99b...619	Antivirus detection	text	Linux/SSHDoor.FB				✓			☐	Inherit	(0/0/0)		
2021-02-04*	324...400	Antivirus detection	text	Linux/Agent.IV				✓			☐	Inherit	(0/0/0)		
2021-02-04*	a46...ebc	Payload delivery	text	-----BEGIN PUBLIC KEY----- MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAOUgD8sEF1kZ04QxCd60 HrB+TxWnLQED wzb0sZ8vMMD6xnUAJspdYzSVDnRnKYjTOM43qtLNcJOwVj6cuC1uHHM CAwEAAQ== -----END PUBLIC KEY-----			RSA public key	✓		1	✓	Inherit	(0/0/0)		

Figure 3.3: Example of attributes and objects linked to an Event in MISP


Galaxies

Attack Pattern
- Obfuscated Files or Information - T1027
- Clear Command History - T1070.003
- Timestomp - T1070.006
- Multi-hop Proxy - T1090.003
- Traffic Signaling - T1205
- Compromise Client Software Binary - T1554
- Symmetric Cryptography - T1573.001
- Asymmetric Cryptography - T1573.002

Sector
- targets: Academia - University
- Government, Administration
- IT - ISP
- Marketing

Regions UN M49
- 021 - Northern America
- 142 - Asia
- 150 - Europe

Course of Action
- Multi-factor Authentication - M1032

Figure 3.4: Example of galaxies associated with an Event in MISP

3.5 Synchronization

3.5.1 How it works

So far, we have seen that MISP is an application that allows you to enter information about an IT incident and then export it to network protection equipment. This leaves us with one last fundamental objective to address: information sharing. Each organization can have its own instance of the application, and it is possible that some organizations may agree to share information between their instances. This is why the concept of synchronization between instances/servers was developed according to the conditions shown in Figure 3.5. If server B wants to share data with server A, here are the steps to follow:

- Step 1: Add OrgB as a local organization on ServerA (OrgB.ServerA) using OrgB's existing uuid from their local organization on ServerB.
- Step 2: Add a Sync User (e.g. syncuser@OrgB.ServerA) in the organization OrgB.ServerA on MISP ServerA.
- Step 3: Set up a sync server on MISP ServerB using the key (called Authkey) from the sync user (syncuser@OrgB.ServerA) created on MISP ServerA.

Once these steps have been completed, it is possible to perform synchronization actions from server B to server A. In other words, it is possible to send data from B to A (*PUSH*) or to receive data from A to B (*PULL*). It is also entirely possible to set up a mirror setup on A for even more efficient data sharing.

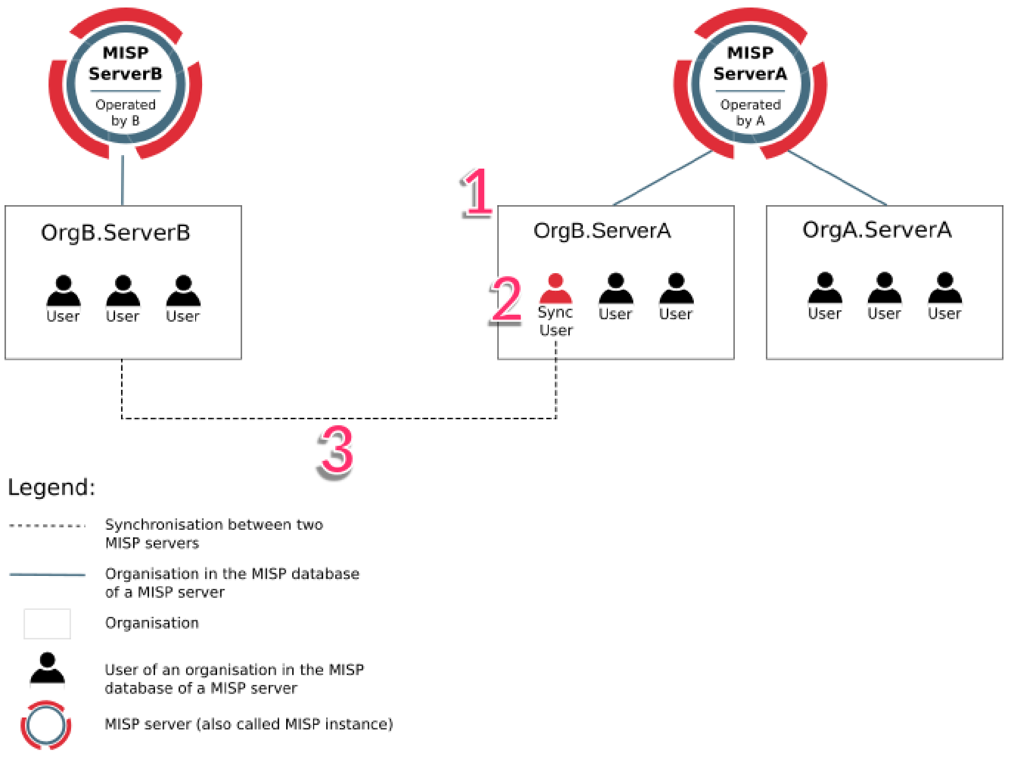

MISP ServerB
Operated by B
MISP ServerA
Operated by A
1
OrgB.ServerB
User
User
User
OrgB.ServerA
2
Sync User
User
User
OrgA.ServerA
User
User
User
3
Legend:
Synchronisation between two MISP servers
Organisation in the MISP database of a MISP server
Organisation
User of an organisation in the MISP database of a MISP server
MISP server (also called MISP instance)

Figure 3.5: Diagram showing the synchronization setup between two instances [2]

3.5.2 Additions and synchronization options

Adding a new instance as a synchronization server is a relatively simple process (see Figure 3.6). First, simply enter the URL where the instance is hosted and give it a name. Then define the host organization associated with the remote instance. This organization can be:

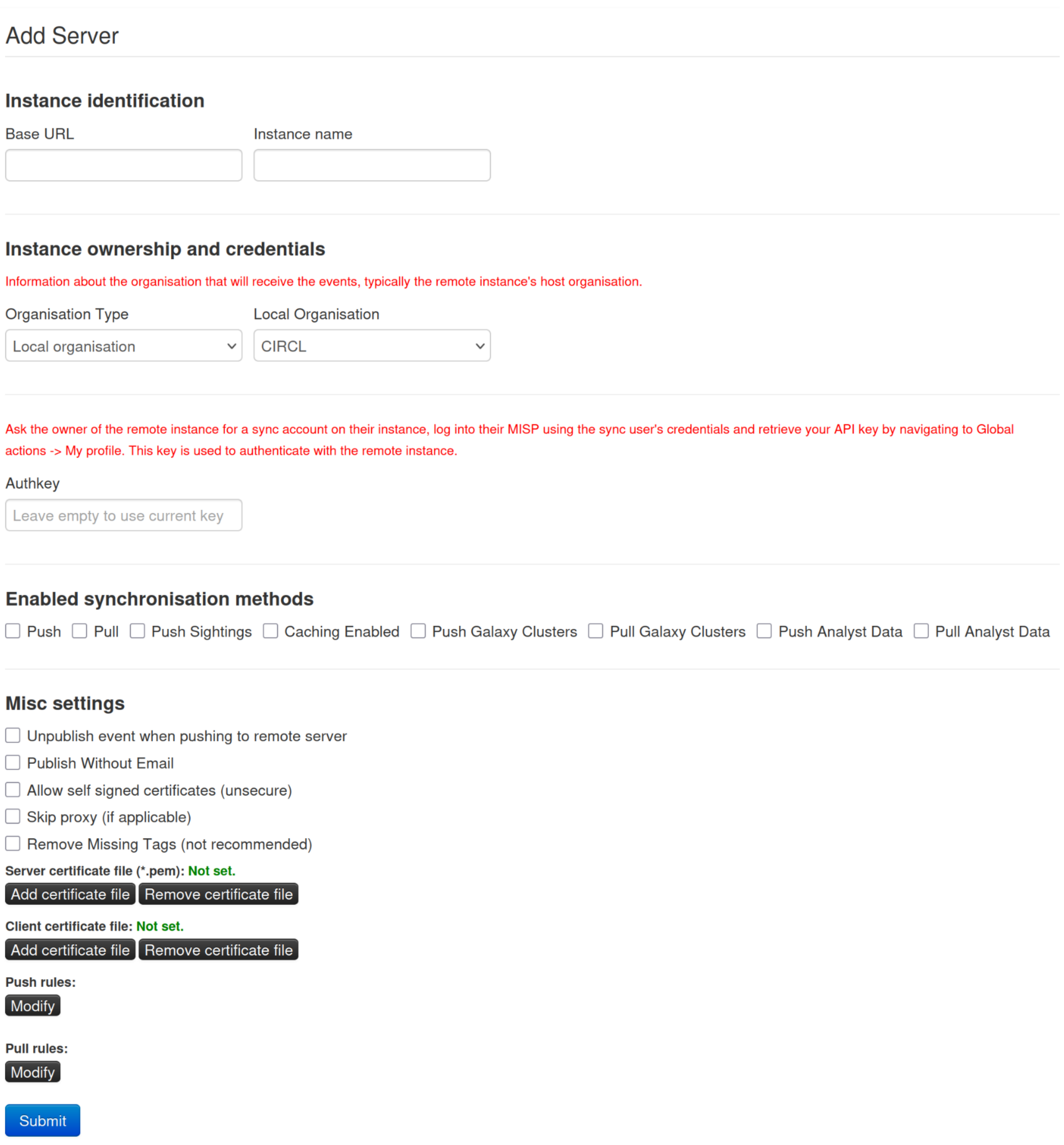
- already defined locally on the instance
- from remote organizations discovered during a previous synchronization process with another server
- completely unknown, in which case it is necessary to provide its *uuid*

Once this step has been completed, it is possible to configure the different synchronization methods authorized with this instance:

- enable *PUSH* and/or *PULL* events
- authorize *PUSH* of *sightings*
- enable *PUSH* and/or *PULL* of *Galaxy Clusters*
- enable *PUSH* and/or *PULL* of *Analyst Data* (comments left on attributes)

Other more technical settings are also available. Finally, it is possible to define *PUSH*/*PULL* rules. These rules allow, for example:

- some tags to be associated to indicate whether or not data should be synchronized
- synchronization to be restricted to certain organizations, or inversely, to exclude certain organizations


Add Server

Instance identification

Base URL	Instance name

Instance ownership and credentials

Information about the organisation that will receive the events, typically the remote instance's host organisation.

Organisation Type	Local Organisation
Local organisation	CIRCL

Ask the owner of the remote instance for a sync account on their instance, log into their MISP using the sync user's credentials and retrieve your API key by navigating to Global actions -> My profile. This key is used to authenticate with the remote instance.

Authkey: Leave empty to use current key

Enabled synchronisation methods

☐ Push ☐ Pull ☐ Push Sightings ☐ Caching Enabled ☐ Push Galaxy Clusters ☐ Pull Galaxy Clusters ☐ Push Analyst Data ☐ Pull Analyst Data

Misc settings

☐ Unpublish event when pushing to remote server
☐ Publish Without Email
☐ Allow self signed certificates (unsecure)
☐ Skip proxy (if applicable)
☐ Remove Missing Tags (not recommended)

Server certificate file (*.pem): Not set.

Client certificate file: Not set.

Push rules:

Pull rules:

Figure 3.6: Setup page to synchronize with another MISP instance

3.5.3 Data distribution level

In MISP, each piece of data (event, attribute, cluster, etc.) is associated with a distribution level. This distribution level allows you to filter data sharing. The different distribution levels are:

- **0 - Your organization only**: The information is only visible to the organization that created it.
- **1 - This community only**: The information is shared only with organizations present locally on the MISP instance.
- **2 - Connected communities**: The information can be shared with servers connected via synchronization (partner MISP instances).
- **3 - All communities** : The information can be shared with all connected servers, including the servers of connected servers.

- **4 - Sharing group** : The information is shared only with a defined sharing group (see explanation in the following section), which allows precise control over which entities can access the information.

To better understand the difference between the Connected communities and All communities levels, let's take a concrete example. Suppose there are three MISP instances: A, B, and C, where A is connected to B, and B is connected to C, but A is not directly connected to C.

If an event E is created on instance A with a distribution level of 2 (Connected communities), then when synchronizing with B, the event will be transmitted to B and its distribution level will automatically be lowered to 1 (This community only). As a result, the event will not be relayed further, and C will not receive it.

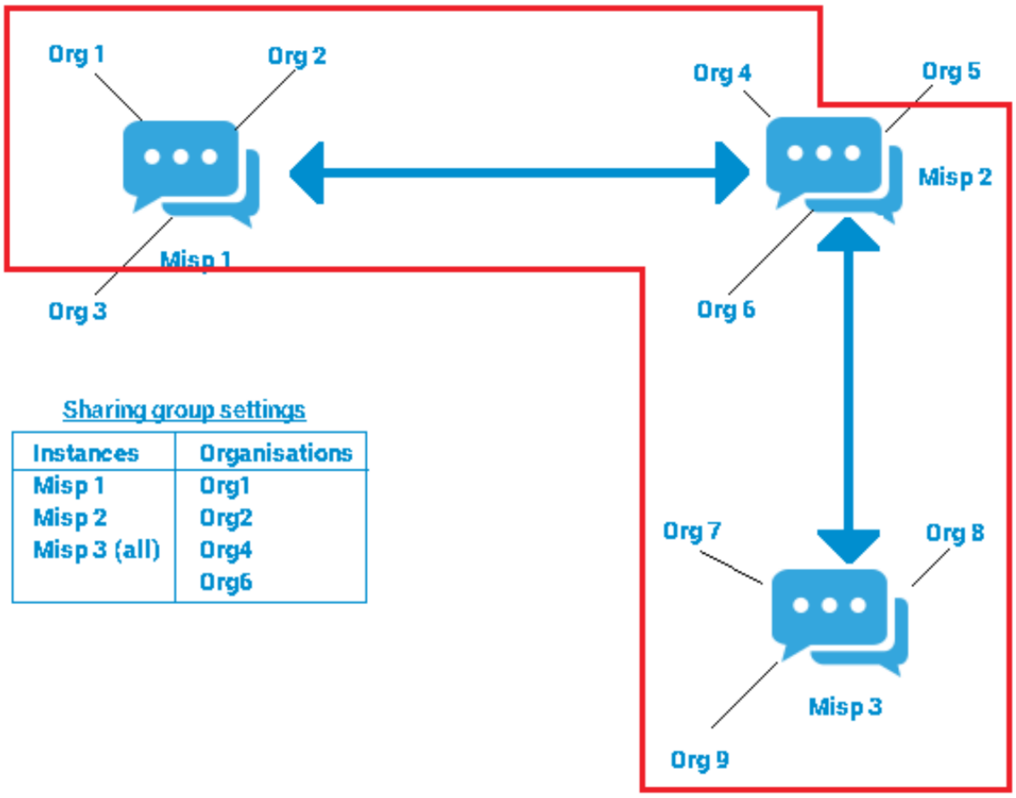
However, if the distribution level of event E on A is set to 3 (All communities), then this level is retained on B, allowing B to synchronize it with C in turn. Thus, the event can propagate across all connected instances, even indirectly.

3.5.4 Sharing groups

Sharing groups represent an additional level of distribution in MISP, allowing for more precise control over data sharing. They offer the possibility to define precisely which organizations, within which instances, can access a given piece of information.

For example, in Figure 3.7, the *sharing group* includes organizations *Org1*, *Org2*, *Org4*, *Org5*, as well as all organizations in *MISP3*. These entities can thus exchange data between themselves on instances *MISP1*, *MISP2*, and *MISP3*.

The most common use cases for sharing groups are to create reusable thematic subgroups in MISP that share events or to set up ad hoc sharing scenarios (e.g., multiple organizations involved in a specific incident that want to collaborate). In general, sharing groups add a level of complexity for the users involved, as well as a performance overhead on the data associated with them.



Sharing group settings

Instances	Organisations
Misp 1	Org1
Misp 2	Org2
Misp 3 (all)	Org4
	Org6

Figure 3.7: Example of a sharing group between organizations from three MISP instances [2]

3.6 Ecosystem and extensibility

From the beginning of its development, MISP was designed to integrate with other cybersecurity ecosystems so that it could be extended according to user needs. Several mechanisms enable this, such as data enrichment, import/export in multiple formats, third-party integrations, and dedicated modules.

3.6.1 Data enrichment

In MISP, it is possible to enrich the data already present in an event manually or automatically via a module. Here are a few examples of enrichment:

- A file hash can be sent to VirusTotal to obtain detection rates or metadata.
- An IP address can be resolved based on its geolocation or queried by Shodan to obtain some information about exposed services.
- A domain name can be subject to passive DNS queries to identify associated subdomains.

3.6.2 Import and export formats

MISP natively supports a large number of import and export formats, to operate easily with other platforms and tools. Supported formats include:

- **STIX (1.x and 2.x)**: a widely used standard for structured cyber threat intelligence.
- **OpenIOC**: an XML-based format initially developed by Mandiant.
- **CSV and JSON**: for integration into custom workflows and data pipelines.
- **Bro/Zeek IDS** and **Snort/Suricata**: for direct integration with intrusion detection and prevention systems.

3.7 Comparison with other solutions

MISP is not the only application for sharing threat intelligence. There are several other solutions, both open source and commercial, each offering different features depending on the intended use cases, integration requirements, or business models adopted. To name a few

- **OpenCTI**: an open source graph-oriented platform designed to represent complex entities (groups, campaigns, vulnerabilities, etc.) in line with the STIX 2.1 standard. It facilitates contextual analysis thanks to its advanced visualizations.
- **TheHive**: an incident management tool often coupled with Cortex for automated enrichment. It can integrate threat intelligence feeds, but is not focused on inter-agency sharing.

- **IBM X-Force Exchange**: a commercial platform offering a large catalog of enriched threats and integration capabilities with IBM security tools.
- **ThreatConnect**: an enterprise-oriented solution combining threat intelligence, process automation (SOAR), and collaboration.

Among all these solutions, MISP stands out for its fully open source model, its widespread adoption within CERTs, CSIRTs, and government communities, and above all for its native ability to synchronize events between instances.

4 Functionality development and correction

The first part of my internship consisted of developing the MISP application in order to better understand and explain its features. To do this, I used the *Issue* section of the GitHub repository. This section is used by both the development team and the community to submit ideas for new features and report bugs in the application. I was assigned certain issues that I had to resolve by submitting a pull request, and for some issues, I submitted a pull request myself for bugs and features that had not yet been reported.

4.1 Extended events

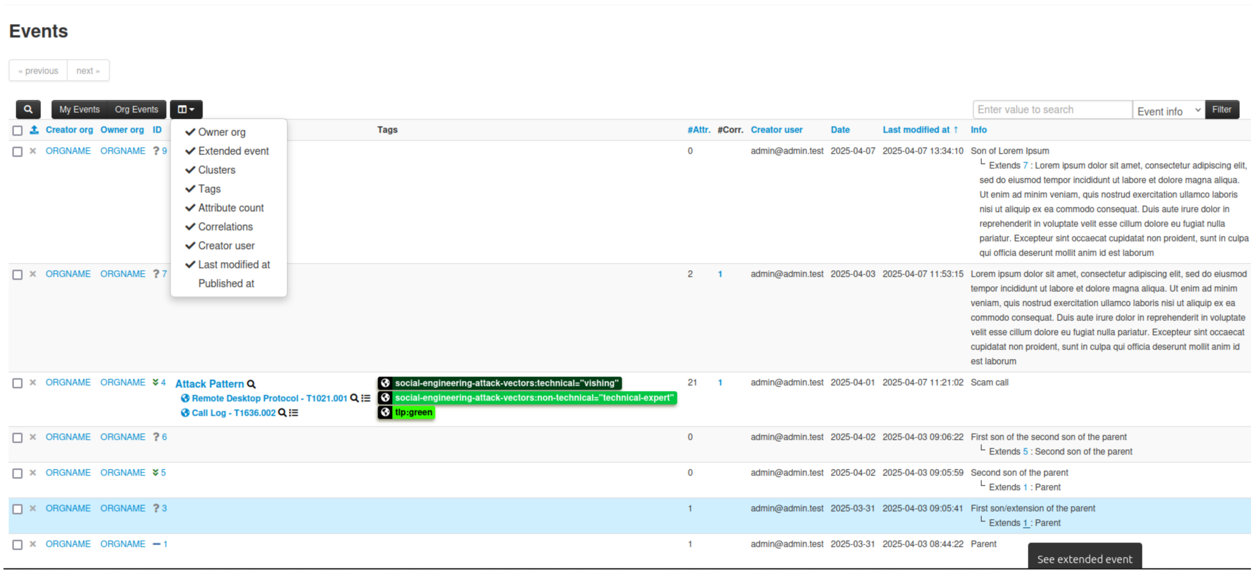
4.1.1 Definition

The first feature I worked on during my internship was extended events. This feature allows you to create an event that enriches an existing event without having to modify it. The goal is to facilitate collaboration between different organizations while respecting the sharing rules and editing rights of each user [1].

In concrete terms, an extended event follows the classic structure of an MISP event, but it only contains additions (attributes, objects, tags, etc.) related to an initial event, referenced by its uuid. One of the major advantages of extended events is that they allow clear traceability of contributors, while avoiding risky manipulation of source events. They also address certain sensitive use cases, such as the management of confidential or embargoed information, by isolating it in a separate but related event.

4.1.2 Feature request: whether or not to display the extended event in the index

The first issue I worked on was to implement an option to visually display on the event index (see Figure 4.1) whether one event extends another. This feature is important because it allows a user to see at a glance which events in their instance extend others. Specifically, I added a symbol to the description of the index for these events, and if the user activates a filter, they see the description of the extended event instead of the symbol.



Creator org	Owner org	ID		Tags	#Attr.	#Corr.	Creator user	Date	Last modified at ↑	Info
ORGNAME	ORGNAME	? 9			0		admin@admin.test	2025-04-07	2025-04-07 13:34:10	Son of Lorem Ipsum └ Extends 7 : Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum
ORGNAME	ORGNAME	? 7			2	1	admin@admin.test	2025-04-03	2025-04-07 11:53:15	Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum
ORGNAME	ORGNAME	≫ 4	Attack Pattern Remote Desktop Protocol - T1021.001 Call Log - T1636.002	social-engineering-attack-vectors:technical="vishing" social-engineering-attack-vectors:non-technical="technical-expert" tlp:green	21	1	admin@admin.test	2025-04-01	2025-04-07 11:21:02	Scam call
ORGNAME	ORGNAME	? 6			0		admin@admin.test	2025-04-02	2025-04-03 09:06:22	First son of the second son of the parent └ Extends 5 : Second son of the parent
ORGNAME	ORGNAME	≫ 5			0		admin@admin.test	2025-04-02	2025-04-03 09:05:59	Second son of the parent └ Extends 1 : Parent
ORGNAME	ORGNAME	? 3			1		admin@admin.test	2025-03-31	2025-04-03 09:05:41	First son/extension of the parent └ Extends 1 : Parent
ORGNAME	ORGNAME	— 1			1		admin@admin.test	2025-03-31	2025-04-03 08:44:22	Parent

Figure 4.1: Index of events with extended event display

4.1.3 Feature request: flag to filter with restsearch request

The second solution I worked on was to add an extended/extending parameter to the /events/restSearch URI of the MISP REST API 4.1. I also updated the API documentation to explain how to use these new parameters.

The PHP function shown in listing 4.2 allows filtering when the extended parameter is used in the query. This function is more complex than the one used to filter events that extend others, because the necessary information is not directly accessible in the event object.

In fact, to identify extended events (or those that are not, depending on the value of the parameter), we first perform an initial query to the database to retrieve all events whose extends_uuid field is not null. This provides a list of extended events, but it is then necessary to remove duplicate UUIDs, as the same event can be extended multiple times.

A second query is then performed to retrieve the id/uuid pairs for all events. We can then isolate the events whose ids match (or do not match) the uuids obtained previously, depending on the value specified by the user.

```
curl -k \
     -X POST https://localhost/events/restSearch \
     -H "Authorization: API_KEY" \
     -H "Accept: application/json" \
     -H "Content-type: application/json" \
     -d '{"returnFormat":"json", "org": "12345", "published": false, "
    extending":false}'
```

Listing 4.1: Example of a POST API request to a REST endpoint

```
public function set_filter_extended(&$params, $conditions, $options)
    {
        if (!isset($params['extended'])) {
            return $conditions;
        }
        //If extended is an array, it means that the user is filtering for both extended and not extended events
        if (is_array($params['extended']) && in_array(1, $params['extended']) && in_array(0, $params['extended'])) {
            return $conditions;
        } else {
            $extended = filter_var($params['extended'], FILTER_VALIDATE_BOOLEAN);
        }
        //Step 1 - Extract the UUIDs of the events that are extended and remove duplicates
        $targetUuids = array_unique($this->find('column', array(
            'fields' => array('Event.extends_uuid'),
            'conditions' => array('Event.extends_uuid !=' => ''),
            'recursive' => -1
        )));
        //If there is no event with extends_uuid(extending), there is basically no event extended
        if (empty($targetUuids)) {
            $conditions['AND'][] = array('Event.id' => -1);
            return $conditions;
        }
        //Step 2 - Extract the UUIDs and ids of all events
        $allEvents = $this->find('list', array(
            'fields' => array('Event.uuid', 'Event.id'),
            'recursive' => -1
        ));
        //Step 3 - Fetch the events that are extended or not extended
        $linkedEventIds = array();
        if ($extended) {
            foreach ($targetUuids as $uuid) {
                if (isset($allEvents[$uuid])) {
                    $linkedEventIds[] = $allEvents[$uuid];
                }
            }
        } else {
            foreach ($allEvents as $uuid => $id) {
                if (!in_array($uuid, $targetUuids, true)) {
                    $linkedEventIds[] = $id;
                }
            }
        }
        if (empty($linkedEventIds)) {
            $conditions['AND'][] = array('Event.id' => -1);
        } else {
            $conditions['AND'][] = array('Event.id' => $linkedEventIds);
        }
        return $conditions;
    }
```

Listing 4.2: Filter function used for extended flag

4.2 Warning lists

4.2.1 Definition

MISP *warninglists* are lists of known indicators, generally associated with potential false positives, errors, or irrelevant information. The name of this object can be misleading, as a *warninglist* does not refer to cyber threats, but rather to items that are best ignored or treated with caution.

By default, MISP offers a large number of warninglists that users can activate as needed, particularly to identify potentially unreliable attributes. Users also have the option of creating their own custom warninglists.

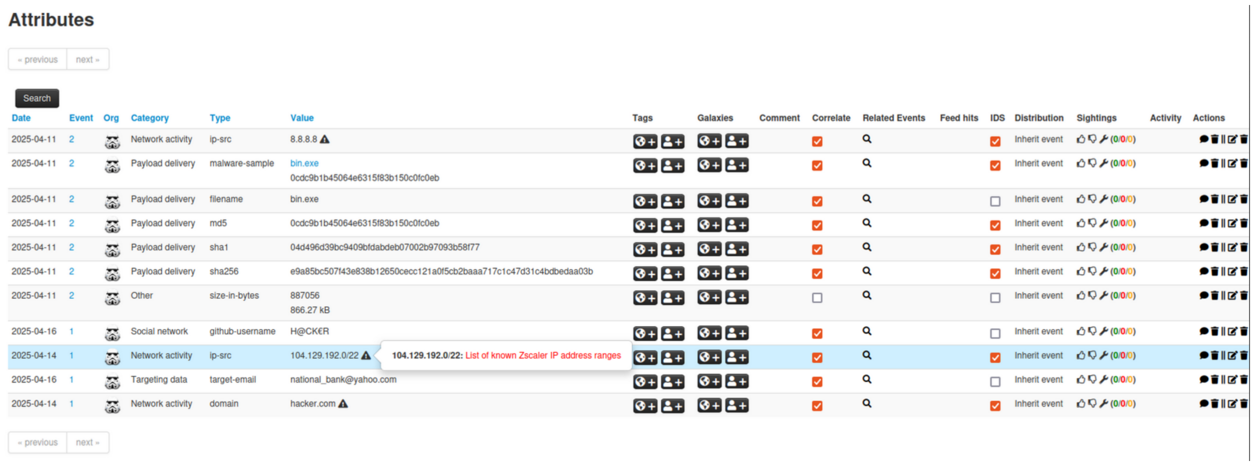
4.2.2 Filtering on the attribute index

MISP offers a summary index listing all the attributes associated with all the events present in the instance. On this interface, it is possible to filter attributes according to several criteria: by category (IP address, hash, etc.), by associated tags, or by their raw value.

However, it was not possible from this index to exclude attributes present in a warning list, or even to view whether they belonged to one.

To solve this, I added a checkbox to the search interface. When enabled, it triggers the use of two existing flags, *includeWarninglistHits* and *enforceWarninglist*, to ensure correct filtering in the rest of the code.

The correct use of these parameters has also made it possible to add a visual indicator in the index, explicitly signaling the attributes present in a *warninglist* (see Figure 4.2).


Attributes

« previous | next »

Search

Date	Event	Org	Category	Type	Value	Tags	Galaxies	Comment	Correlate	Related Events	Feed hits	IDS	Distribution	Sightings	Activity	Actions
2025-04-11	2		Network activity	ip-src	8.8.8.8 ⚠				✓			✓	Inherit event	(0/0/0)		
2025-04-11	2		Payload delivery	malware-sample	bin.exe 0cdc9b1b45064e6315f83b150c0fc0eb				✓			✓	Inherit event	(0/0/0)		
2025-04-11	2		Payload delivery	filename	bin.exe				✓			☐	Inherit event	(0/0/0)		
2025-04-11	2		Payload delivery	md5	0cdc9b1b45064e6315f83b150c0fc0eb				✓			✓	Inherit event	(0/0/0)		
2025-04-11	2		Payload delivery	sha1	04d496d39bc9409bfdabdeb07002b97093b58f77				✓			✓	Inherit event	(0/0/0)		
2025-04-11	2		Payload delivery	sha256	e9a85bc507f43e838b12650cecc121a0f5cb2baaa717c1c47d31c4bdbedaa03b				✓			✓	Inherit event	(0/0/0)		
2025-04-11	2		Other	size-in-bytes	887056 866.27 kB				☐			☐	Inherit event	(0/0/0)		
2025-04-16	1		Social network	github-username	H@CK€R				✓			☐	Inherit event	(0/0/0)		
2025-04-14	1		Network activity	ip-src	104.129.192.0/22 ⚠ **104.129.192.0/22:** List of known Zscaler IP address ranges				✓			✓	Inherit event	(0/0/0)		
2025-04-16	1		Targeting data	target-email	national_bank@yahoo.com				✓			☐	Inherit event	(0/0/0)		
2025-04-14	1		Network activity	domain	hacker.com ⚠				✓			✓	Inherit event	(0/0/0)		

« previous | next »

Figure 4.2: Index of attributes for a misp training instance. Here three attributes have been identified in warning lists.

4.2.3 Explicit display of the warninglist category

Currently, from the event view, when an attribute matches an entry in an activated *warninglist*, a red box is displayed to indicate the *warninglists* concerned.

However, MISP distinguishes between two types of *warninglists*:

- **False Positive**: generic indicators, often found in the analyzed data, but not very relevant in the context of threat detection;
- **Known Identifier**: indicators specific to the owner of the MISP instance (such as internal IP ranges or legitimate certificates), useful for avoiding internal false positives.

So I implemented a second visual insert, which is displayed according to the category of the *warninglist* to which an attribute corresponds. This makes it possible to visually distinguish attributes corresponding to *False Positives* from those linked to *Known Identifiers* (see Figure 4.3).

In addition, from a more backend perspective, I added the ability to filter event attributes based on the warninglist category from the event view. Until now, filtering only allowed you to differentiate between attributes present or not present in a warninglist, without distinguishing between the warninglist categories.

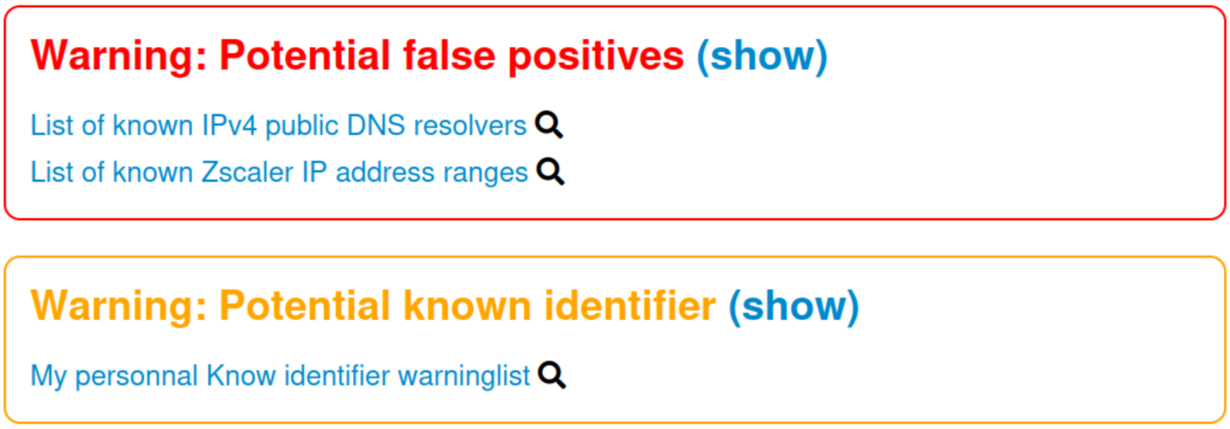

Warning: Potential false positives (show)
List of known IPv4 public DNS resolvers
List of known Zscaler IP address ranges
Warning: Potential known identifier (show)
My personnal Know identifier warninglist

Figure 4.3: New interface element (in yellow) showing the explicit separation warninglists according to their category

4.3 Roles

4.3.1 Definition

As briefly mentioned above, users of a MISP instance are assigned a role that determines the scope of their rights within the application. By default, MISP offers six main roles:

- **Site Admin**: superuser of the instance, with full access (user management, viewing of all data, etc.);

- **Org Admin**: administrator of a specific organization, can manage users, events, and logs related to their organization;
- **Sync User**: role reserved for users serving as synchronization points between instances, via dedicated authentication keys;
- **Publisher**: authorized to publish (set something as ready to be synchronized) events;
- **Read-only**: read-only access, without modification rights;
- **User**: standard user with limited permissions, depending on the configuration of their role.

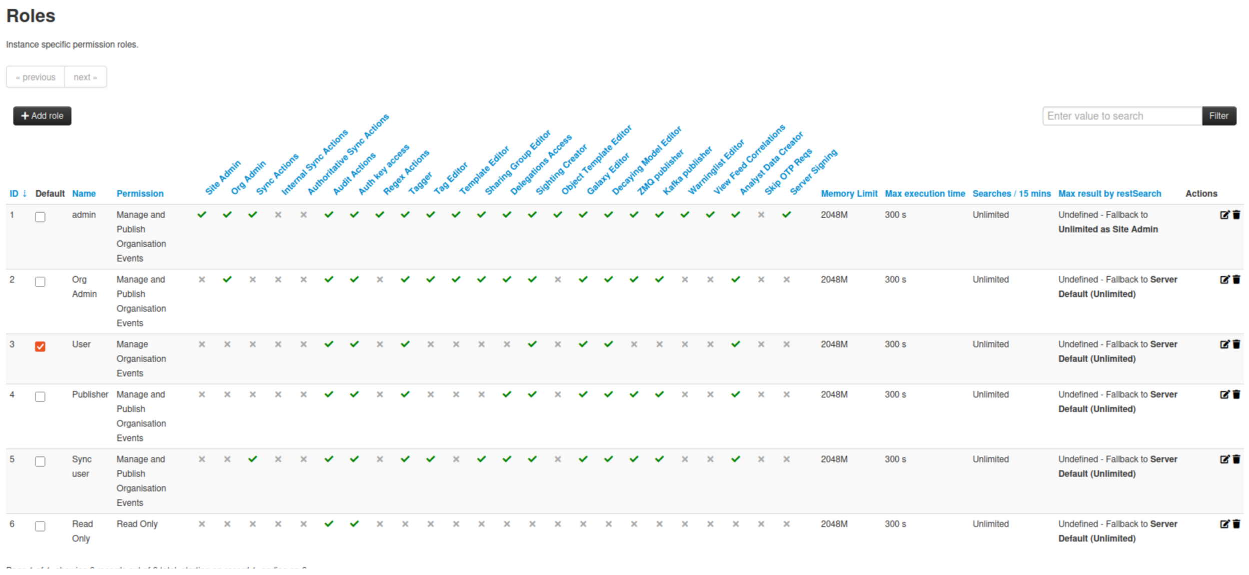
The administrator of a MISP instance can, of course, create new custom roles by combining the various available rights as needed.

4.3.2 Limiting the number of results from a restSearch

As a reminder, any user can query the MISP REST API via routes such as /events/restSearch to obtain a large amount of data. In the case of an instance containing a large volume of events, overly broad or repeated queries can quickly lead to overload or even denial of service.

To mitigate this risk, I have introduced a new option when creating (or modifying) a role, allowing you to set a maximum limit on the number of results returned per query, depending on the role of the user making the query (see Figure 4.4

If no explicit limit is defined for a given role, the global value configured at the instance level (by the administrator) will be used by default.


Roles

Instance specific permission roles.

« previous | next »

+ Add role

Enter value to search | Filter

ID ↓	Default	Name	Permission	Site Admin	Org Admin	Sync Actions	Internal Sync Actions	Authoritative Sync Actions	Audit Actions	Auth key access	Regex Actions	Tagger	Tag Editor	Template Editor	Sharing Group Editor	Delegations Access	Sighting Creator	Object Template Editor	Galaxy Editor	Decaying Model Editor	ZMQ publisher	Kafka publisher	Warninglist Editor	View Feed Correlations	Analyst Data Creator	Skip OTP Reqs	Server Signing	Memory Limit	Max execution time	Searches / 15 mins	Max result by restSearch	Actions
1	☐	admin	Manage and Publish Organisation Events	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	2048M	300 s	Unlimited	Undefined - Fallback to **Unlimited as Site Admin**	
2	☐	Org Admin	Manage and Publish Organisation Events	×	✓	×	×	×	✓	✓	×	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	×	×	✓	×	×	2048M	300 s	Unlimited	Undefined - Fallback to **Server Default (Unlimited)**	
3	☑	User	Manage Organisation Events	×	×	×	×	×	✓	✓	×	✓	×	×	×	×	✓	×	✓	✓	×	×	×	×	✓	×	×	2048M	300 s	Unlimited	Undefined - Fallback to **Server Default (Unlimited)**	
4	☐	Publisher	Manage and Publish Organisation Events	×	×	×	×	×	✓	✓	×	✓	×	×	×	✓	✓	×	✓	✓	✓	✓	×	×	✓	×	×	2048M	300 s	Unlimited	Undefined - Fallback to **Server Default (Unlimited)**	
5	☐	Sync user	Manage and Publish Organisation Events	×	×	✓	×	×	✓	✓	×	✓	✓	×	✓	✓	✓	×	✓	✓	✓	✓	×	×	✓	×	×	2048M	300 s	Unlimited	Undefined - Fallback to **Server Default (Unlimited)**	
6	☐	Read Only	Read Only	×	×	×	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	2048M	300 s	Unlimited	Undefined - Fallback to **Server Default (Unlimited)**	

Figure 4.4: Default roles index of a MISP instance

4.4 Minor additions

REST API

As I had the opportunity to work with the MISP REST API for the tasks mentioned above, I was able to identify a few bugs and features that could be easily implemented.

In MISP, most API requests allow you to specify a *returnFormat* flag to indicate the format you expect in the response. However, some users may not be aware that this option exists or may simply be more accustomed to using the standard HTTP mechanism (the *Accept* header) to define the desired return format.

That is why I made changes to the source code to take this possibility into account. If the *returnFormat* flag is not specified, then the response format will be determined by the value of the *Accept* header, provided that it corresponds to one of the formats officially supported by MISP (such as JSON, XML, CSV, etc.).

When frequently using the REST client integrated into MISP, I noticed that the curl command automatically generated for the user never contained the -k option, even though the skip SSL validation option was checked. I therefore modified the code so that it adds the flag, allowing the user to execute the command directly without encountering an error message.

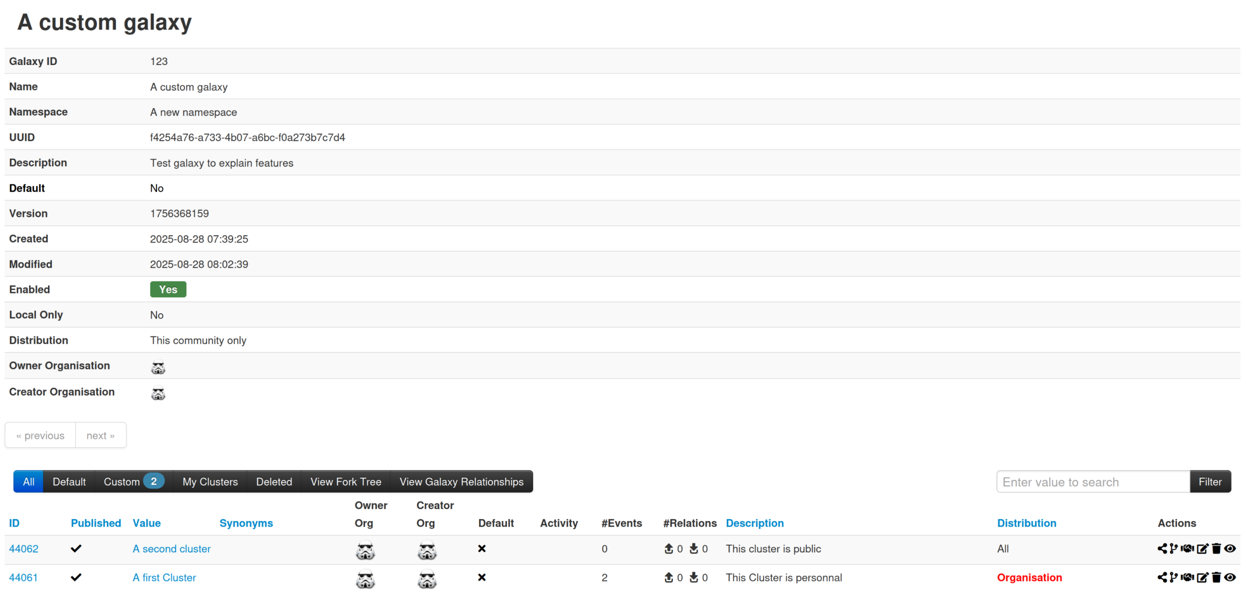
Event view

At the top of Figure 3.3, you can see a list of black buttons representing shortcuts for performing various actions on the attributes and objects of an event. However, the shortcut action for adding attributes directly from a file in a supported format (OpenIOC, ThreatConnect, MISP JSON, CSV, etc.) was not present. I therefore implemented a dedicated button to perform this operation.

Getting started with Galaxies

Like events, galaxies and galaxy clusters are objects that can be shared during synchronization. Galaxies and their clusters also have their own distribution level, with the galaxy level always being more restrictive than the cluster level. Figure 4.5 shows a detailed view of a galaxy, with essential information about each of its clusters.

A cluster, and by extension the galaxy to which it belongs, can be attached to an event, but also more specifically to an attribute. The detailed view of a cluster, illustrated in Figure 4.6, shows how many events this cluster is attached to. However, before my intervention, this view did not specify the number of attributes to which the cluster was linked. If the cluster is attached to an event, it is automatically associated with all of that event's attributes. If the user clicks on the number of linked attributes, they are redirected to the attribute index page (Figure 4.2), automatically filtered on the corresponding cluster.

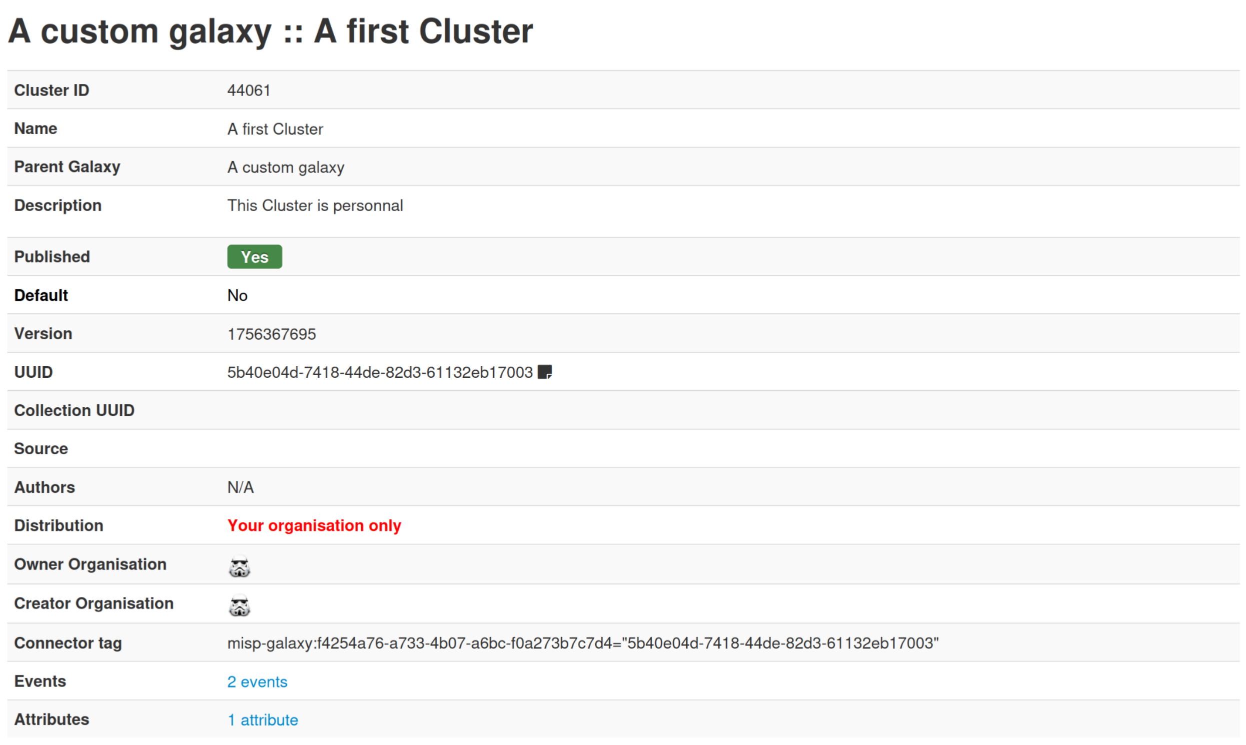


A custom galaxy

Galaxy ID	123
Name	A custom galaxy
Namespace	A new namespace
UUID	f4254a76-a733-4b07-a6bc-f0a273b7c7d4
Description	Test galaxy to explain features
Default	No
Version	1756368159
Created	2025-08-28 07:39:25
Modified	2025-08-28 08:02:39
Enabled	Yes
Local Only	No
Distribution	This community only
Owner Organisation	
Creator Organisation	

ID	Published	Value	Synonyms	Owner Org	Creator Org	Default	Activity	#Events	#Relations	Description	Distribution	Actions
44062	✓	A second cluster				×		0	0 0	This cluster is public	All	
44061	✓	A first Cluster				×		2	0 0	This Cluster is personnal	Organisation	

Figure 4.5: View of a custom MISP galaxy with 2 cluster



A custom galaxy :: A first Cluster

Cluster ID	44061
Name	A first Cluster
Parent Galaxy	A custom galaxy
Description	This Cluster is personnal
Published	Yes
Default	No
Version	1756367695
UUID	5b40e04d-7418-44de-82d3-61132eb17003
Collection UUID	
Source	
Authors	N/A
Distribution	Your organisation only
Owner Organisation	
Creator Organisation	
Connector tag	misp-galaxy:f4254a76-a733-4b07-a6bc-f0a273b7c7d4="5b40e04d-7418-44de-82d3-61132eb17003"
Events	2 events
Attributes	1 attribute

Figure 4.6: View of a MISP cluster indicating the number of attributes associated with it

First tests

Finally, to begin the second part of my internship focused on automated testing, I wrote two new tests for the continuous integration (CI) pipeline. These tests, written in Python using the *pymisp* library, validate some of the features mentioned above:

- The first test creates several events linked together (via the *extended* attribute) and verifies the accuracy of the REST API responses according to the different possible values of the *extended* parameter: *0*, *[0,1]*, *true*, etc.

- The second test generates different types of *warninglists*, then creates an event with attributes corresponding to these lists. It verifies that the attributes are correctly detected as belonging to an active *warninglist* and that they are correctly associated with their category (*False Positive* or *Known Identifier*).

4.5 Review of my contribution to misp-core

This first part of my internship allowed me to gain a better understanding of the application's main features and their technical implementation. I had the opportunity to contribute to both the user interface, by improving the display of extended attributes and warning lists, and the backend, by strengthening the filtering of the attribute index and adding new query flags for the REST API.

On this last point, I gained a lot of technical knowledge by learning how such features are implemented in a large-scale application. These contributions have improved the usability, security, and traceability of MISP, while consolidating its functional robustness.

Finally, by writing the first automated tests, I prepared the ground for the second part of my internship, which focuses on testing synchronization.

5 Implementation of synchronization tests

5.1 Problem definition

As mentioned in the state of the art of the MISP application, one of its main strengths resides in its ability to allow different instances to synchronise their data, which facilitates efficient information sharing. However, to date, CIRCL has not yet set up a test environment or continuous integration (CI) pipeline to comprehensively verify the proper functioning of all these synchronization mechanisms.

The objective assigned to me is therefore to set up a pipeline capable of running a series of tests with each new commit to the project to ensure that the synchronization features continue to function correctly. To do this, it will be necessary to deploy several instances of MISP on the same machine so that they can interact with each other via the synchronization system.

5.2 Choice of technologies used

Currently, the MISP project uses GitHub Actions (GitHub's continuous integration solution) to run tests on the application's basic features.

However, GitHub Actions imposes limited resources for public projects. Since synchronization tests can be resource-intensive, we opted for an alternative solution that does not impose such constraints.

5.2.1 Code hosting platform

Before I began my study, the CIRCL team explained that they had a server hosting an instance of Forgejo to back up their various projects locally. Forgejo is a fully open source, self-hosted code management platform. One of its features allows automatic synchronization every X minutes with a project hosted on another platform (such as GitHub).

I therefore began by studying the various technologies that enable continuous integration to be implemented locally with Forgejo. Initially, I looked at independent CI solutions such as Woodpecker, Drone CI, and Jenkins. However, I discovered that starting in July 2023, a new version of Forgejo will offer its own continuous integration solution: Forgejo Actions.

Although this technology is still relatively new—which means there is limited documentation and few examples available—I chose to use it. The main reason is that the syntax of Forgejo Actions workflows is very similar to that of GitHub Actions, which will greatly facilitate my work later on.

5.2.2 Runner containerization

In Forgejo, the *runner* is the element responsible for executing the jobs defined in the continuous integration workflows. It interprets the steps in the configuration file (e.g., compilation, testing, deployment) and executes them in an isolated environment.

The *runner* can be configured to run in different environments, including a Docker container or an LXC container. I chose to use an LXC container because it offers a better compromise between isolation and performance. Unlike Docker, which imposes a certain level of network and system abstraction, LXC provides an environment closer to a traditional system, while retaining the advantages of isolation. This makes it particularly suitable for complex scenarios such as the deployment of multiple MISP instances that need to communicate with each other via internal network interfaces. In addition, using LXC allows for better control over the system configuration of each instance (network, services, volume management), which is an important advantage when testing synchronization between multiple MISP instances.

Figure 5.1 illustrates how my local Forgejo setup works. The Forgejo server runs in a Docker container, which I have linked to a runner installed in a separate LXC container. Communication between the runner and the server is achieved using a connection key and by correctly configuring the LXC broadcast address.

When a commit is made to a repository where Forgejo Actions is enabled, the runner automatically creates a temporary LXC container based on an Ubuntu distribution to run the pipeline.

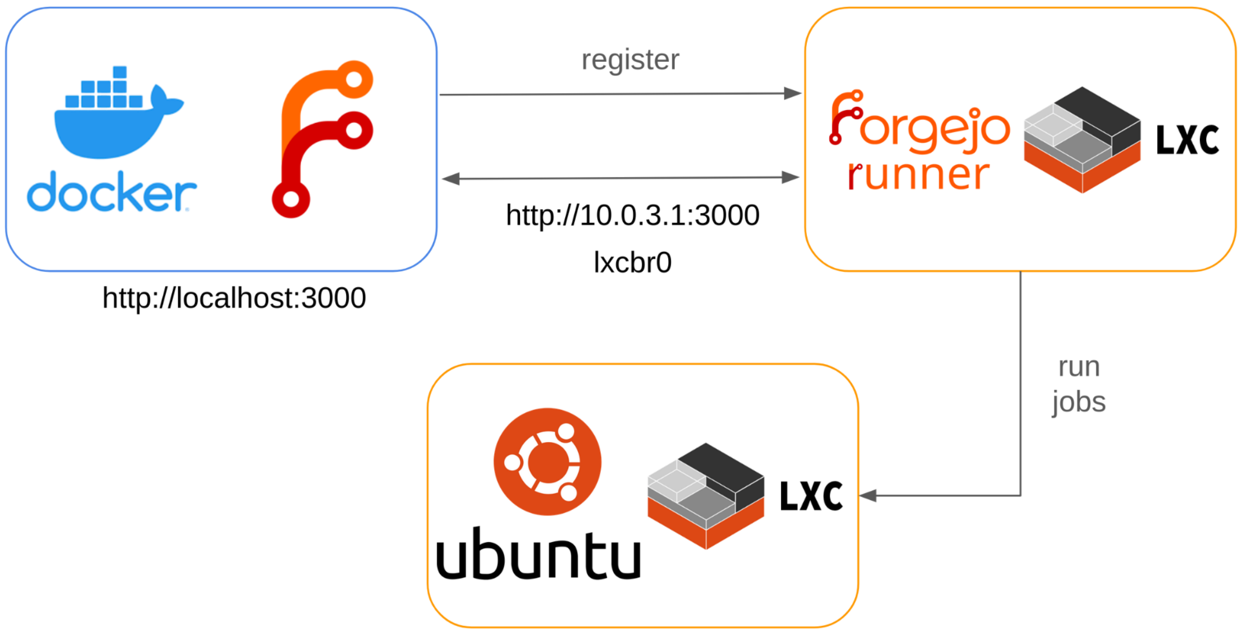


Figure 5.1: Local setup of a forgejo instance and a runner

5.3 Migration from GitHub pipeline to Forgejo

5.3.1 Explanation of Github CI pipeline for MISP

When a commit is made to the MISP project's GitHub repository, GitHub automatically triggers a CI process if it detects one or more workflows defined in the repository. These workflows are described in YAML files located in the project's *.github/workflows/* directory.

Each YAML file can contain one or more **jobs**. A job corresponds to a sequence of instructions to be executed in a given environment. These jobs are themselves composed of several **steps**, which are executed linearly. Each step is generally independent and corresponds to a shell command (e.g., installing dependencies, launching services, running tests, etc.).

Specifically, in the case of MISP, the GitHub pipeline begins by defining the operating system on which the jobs will be executed (in this case, a Linux distribution). It then installs all the dependencies necessary for the application to function properly, specifying the compatible versions.

Docker containers are then launched for the services required by MISP, such as MariaDB for the database and Redis for queue management. The MISP application is then installed and configured automatically, similar to a manual installation via an installation script.

Finally, test files are run. These tests simulate user behavior and verify that the application's critical features are working properly. If any of the tests fail, the pipeline stops and displays an error message detailing the cause of the problem.

5.3.2 Modification apported for the Forgejo CI

The first step in the Forgejo pipeline is to specify the environment in which the job will run. Here, since we want to ensure good isolation, reproducibility, and flexibility, we do not run the job directly on the runner, but in a secondary container deployed on that runner. To do this, we use the *runs-on: lxc* label.

In order to use the same image as the one used in the GitHub Actions pipeline, we specify *container:image: ubuntu:jammy.*

Another important difference is that, unlike GitHub Actions, we cannot launch Redis and MariaDB in separate Docker containers. Using an LXC container therefore requires manual installation of these services inside the container.

Finally, some GitHub Actions repositories, such as *action/checkout*, have a fork compatible with Forgejo, but this is not the case for the majority of actions. I therefore cloned these repositories locally on my Forgejo instance so that I could use them directly in the pipeline.

This new pipeline runs in roughly the same time as those on GitHub, in less than ten minutes.

5.4 Setting up multiple local instances

5.4.1 Issue encountered

Once the deployment pipeline for a single MISP instance was finalized, I began writing a first version of a pipeline that would allow multiple instances to be deployed in parallel. Once this version was functional, I wrote a basic synchronization test. However, each run of my test took about ten minutes for the pipeline to complete. This made the test cycle slow and inefficient. To avoid this constraint, I decided to create a script capable of quickly deploying multiple local instances of MISP.

5.4.2 Deploying multiple instances

In a previous project, I had the opportunity to use an additional MISP repository called *misp-docker*, which allows you to easily deploy an instance via Docker. So I reused the *docker-compose* provided in this repository, adapting it so that a loop automatically deploys multiple instances. Figure 5.2 shows the technical architecture of a local deployment of multiple instances of misp with docker.

For each instance, the script generates:

- a separate application container and database container (e.g., misp_1, misp_2, ...)
- a separate folder for configuration files (e.g. /instance_1/config, /instance_2/config, ...)
- a unique HTTP port (8081, 8082, ...)

This deployment step is the most time-consuming part of the script (mainly due to the deployment of MISP containers). Once all instances have been started, a manual step is still required: the user must log in to each instance to retrieve the API key for their administrator account, which is essential for subsequent API calls.

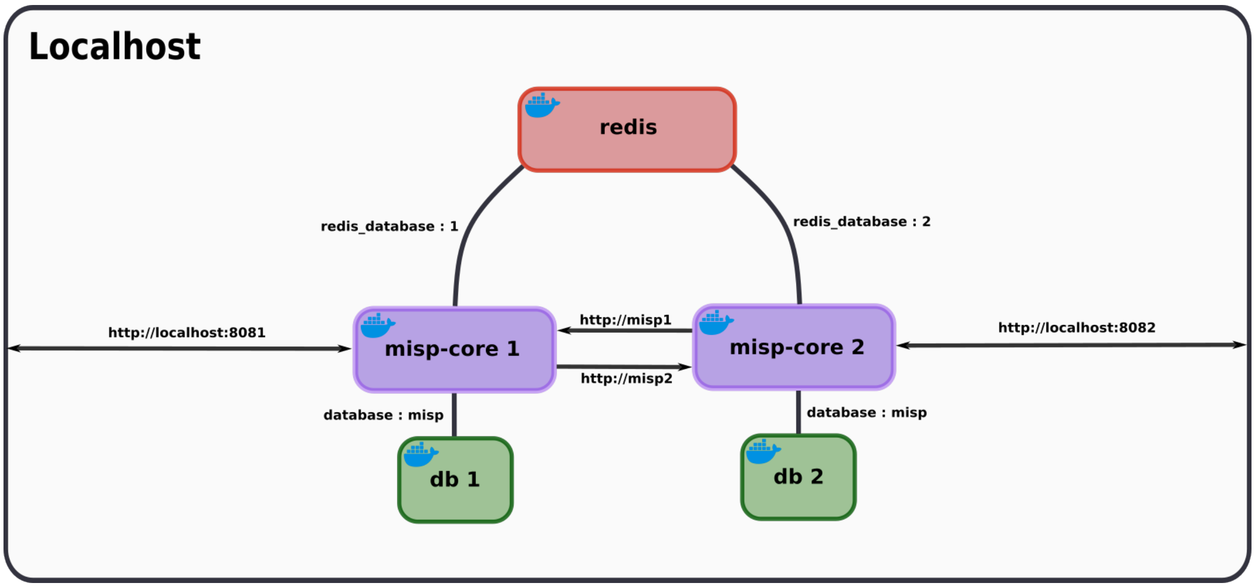


Figure 5.2: Technical architecture for local deployment of 2 MISP instances

5.4.3 Setting up synchronization between instances

Once the administrator keys have been retrieved, the script asks the user to define the synchronization topology. Specifically, for each instance, you must specify which other instances will be considered servers. For example, if instance 1 is to synchronize with instances 2 and 5, the user must enter: 1   5.

The script then proceeds as follows:

- **Creation of organizations**: on each instance, it creates X organizations (where X is the number of instances), ensuring that organization X shares the same *uuid* on all instances.
- **Definition of the host organization**: for each instance X, organization X becomes the host organization. An *orgAdmin* user is also created to manage objects.
- **Creation of synchronization users**: for each user-defined relationship (e.g., instance 1 –> servers 2 and 5), the script creates a *syncUser* in organization 1 on instance 2, and another on instance 5.
- **Retrieving API keys**: the synchronization user keys are then used to configure the remote servers.

It is important to avoid using URLs such as *http://localhost:808X* when creating servers. Instead, use the internal address of the Docker network, in the form *http://misp_X*.

Finally, to avoid conflicts, the script configures a separate Redis ID for each instance.

5.4.4 Additional features

Optionally, the script also allows you to:

- automatically deploy a *sharing group* on the first instance of the topology
- establish an internal connection between the last two instances to test local object sharing

5.5 Development of synchronization tests

5.5.1 Technology and architecture for testing

Like the other application integration tests, my synchronization tests will be written in Python. This task will be made easier by the fact that MISP has its own Python library, PyMISP. This library makes it easier to call the API using pre-built functions [4]. In addition to the PyMISP library, I use the *unittest* library to write test classes. Furthermore, since MISP synchronization can be performed at different levels, I decided to separate the tests into thematic files in order to improve the scalability and readability of the code.

Listing 5.1 below illustrates the general structure of a test. To create and modify the various objects used in the test, I use a user of type *Org_admin*, included in the list *misps_org_admin*. I do not use the default user, namely the super administrator (*misp_site_admin*), because the latter has extended rights allowing them to see certain information that is inaccessible to other users.

However, to perform synchronization actions (server retrieval, *PUSH* and *PULL* operations), it is necessary to use the instance's super administrator, as they are the only one with access to these server action settings.

```python
class TestLockedStatus(unittest.TestCase):
     def testLockedStatusOnPush(self):
        """
        Verifies that the 'locked' attribute of an event is correctly set to True when the event is pushed.
        Ensures that the event cannot be modified on the target MISP instances after synchronization.
        """
        source_instance = misps_org_admin[0]
        # Create a new event on the source instance
        event = create_event('\n Event for locked status on push')
        event.distribution = 2  # Set distribution to 'Connected Community'
        event = source_instance.add_event(event, pythonify=True)
        check_response(event)
        self.assertIsNotNone(event.id)
        uuid = event.uuid
        publish_immediately(source_instance, event, with_email=False)
        time.sleep(2) # the event to propagate it to linked instances
        # Retrieve the server configurations linked to the source instance
        servers = misps_site_admin[0].servers()
        servers_id = get_servers_id(servers)
        if not servers_id:
            raise Exception("No server configuration found for the source instance")
        # For each linked target instance, verify that the event exists and is locked
        linked_server_numbers = extract_server_numbers(servers)
        for target_index in linked_server_numbers:
            target_instance = misps_org_admin[target_index - 1]
            search_results = target_instance.search(uuid=uuid)
            self.assertGreater(
                len(search_results), 0,
                f"Event not found on MISP_{target_index} after push"
            )
            for result in search_results:
                self.assertTrue(result['Event']['locked'], f"Event on MISP_{target_index} is not locked")
        # Attempt to update the event on each target instance and verify that modification is not allowed
        for target_index in linked_server_numbers:
            target_instance = misps_org_admin[target_index - 1]
            # Try to add an attribute to the locked event
            event_to_update = target_instance.get_event(event, pythonify=True)
            event_to_update.add_attribute('text', 'This should not be allowed')
            target_instance.update_event(event_to_update, pythonify=True)
            # Ensure the event was not modified
            updated_event = target_instance.search(uuid=uuid)
            self.assertNotEqual(
                len(updated_event[0]['Event']['Attribute']), 2,
                f"Event on MISP_{target_index} was modified despite being locked"
            )
        # Cleanup: remove all test events and blocklists from all instances
        for instance in misps_site_admin:
            purge_events_and_blocklists(instance)
```

Listing 5.1: Example of a synchronization test

5.5.2 Topology for testing

Figure 5.3 shows the topology chosen for setting up synchronization tests. It consists of seven instances. On each instance, the organization with the same number is defined as the host organization, while the other organizations are also present locally. This topology allows us to evaluate all the particularities that may arise during synchronization. From **MISP 1**, for example, it is possible to test the propagation of an event across multiple instances when its distribution is set to *All communities*. We can also check the functionality of *sharing groups*.

Between **MISP 5** and **MISP 6**, synchronization is unidirectional, unlike other connections, which are bidirectional. This choice is necessary: if two instances were connected in bidirectional mode, as soon as an event was pushed, it would be automatically sent back by the remote instance, thus preventing the *PULL* functionality from being tested correctly.

Finally, the **MISP 6** and **MISP 7** instances are configured as internal instances. In practical terms, this means that they share the same host organization, which allows local tags and *clusters* to be shared during synchronization.

In the next section, I will present the various tests that have been set up. Note that I will only present the version for the *PUSH* mechanism, as the version for the *PULL* mechanism is quite similar.

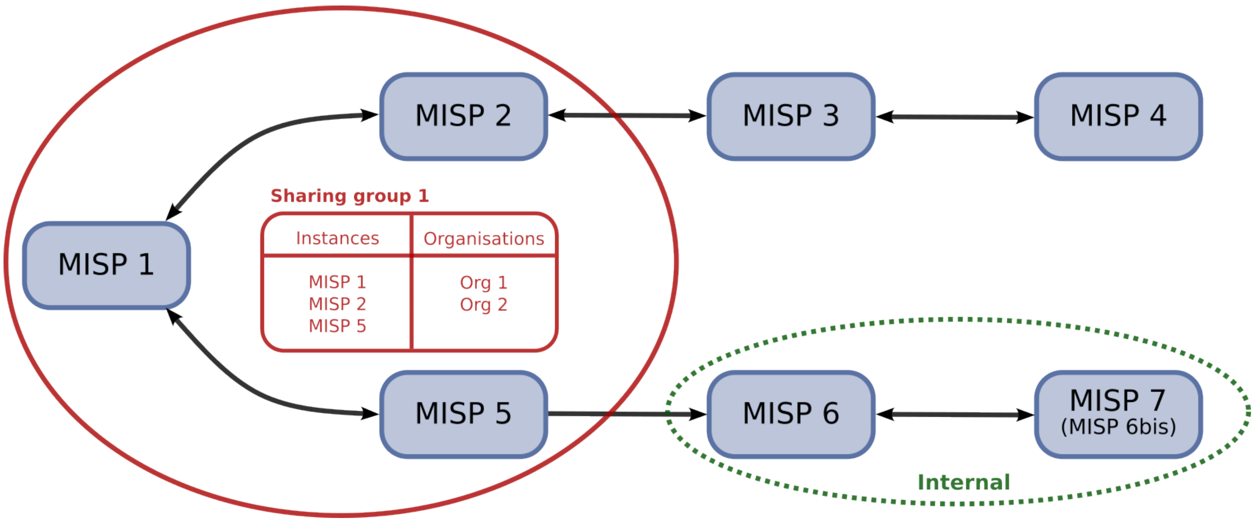


Figure 5.3: Setup scenario for synchronization tests

5.5.3 Details of the various tests implemented

Propagation tests on the topology

This test consists of publishing an event whose distribution is defined on *Connected communities* from each instance of the topology, in order to verify that it is properly propagated to the connected servers. The objective, in addition to validating the *PUSH* and *PULL* mechanisms, is to ensure that the connections between the different instances are working properly.

Publication tests

I also implemented a publication test to verify that an event is only synchronized at the appropriate time. When an event has just been created and the user performs a *PUSH ALL*, it should not be synchronized immediately. For this to happen, the event must be published: once published, it is then automatically pushed to the connected servers, without any additional action on the part of the user. As a reminder, publishing serves to confirm that an event is complete and ready to be shared.

Tests on synchronization methods

As mentioned earlier (see Figure 3.6), there are other synchronization methods besides simple event propagation.

- **Synchronization of sightings**: *Sightings* are metadata that a user can add to an event or attribute to indicate whether they have actually observed that data. I set up an initial test to verify that the addition of a sighting is correctly propagated between servers.
- **Synchronization of analyst data**: it is also possible to synchronize analyst data, which corresponds, for example, to comments added by an analyst to a piece of data to express an opinion or provide additional context. I implemented a test consisting of adding an *analyst data* to an event in order to validate its correct synchronization,
- **Synchronization of galaxy clusters**: the synchronization of *galaxy clusters* is independent of events (distribution is visble on 4.5). When a user creates a galaxy cluster and publishes it, it is automatically synchronized with the connected servers (if the option is enabled). I therefore implemented a test simulating this operation.

Tests on event modification

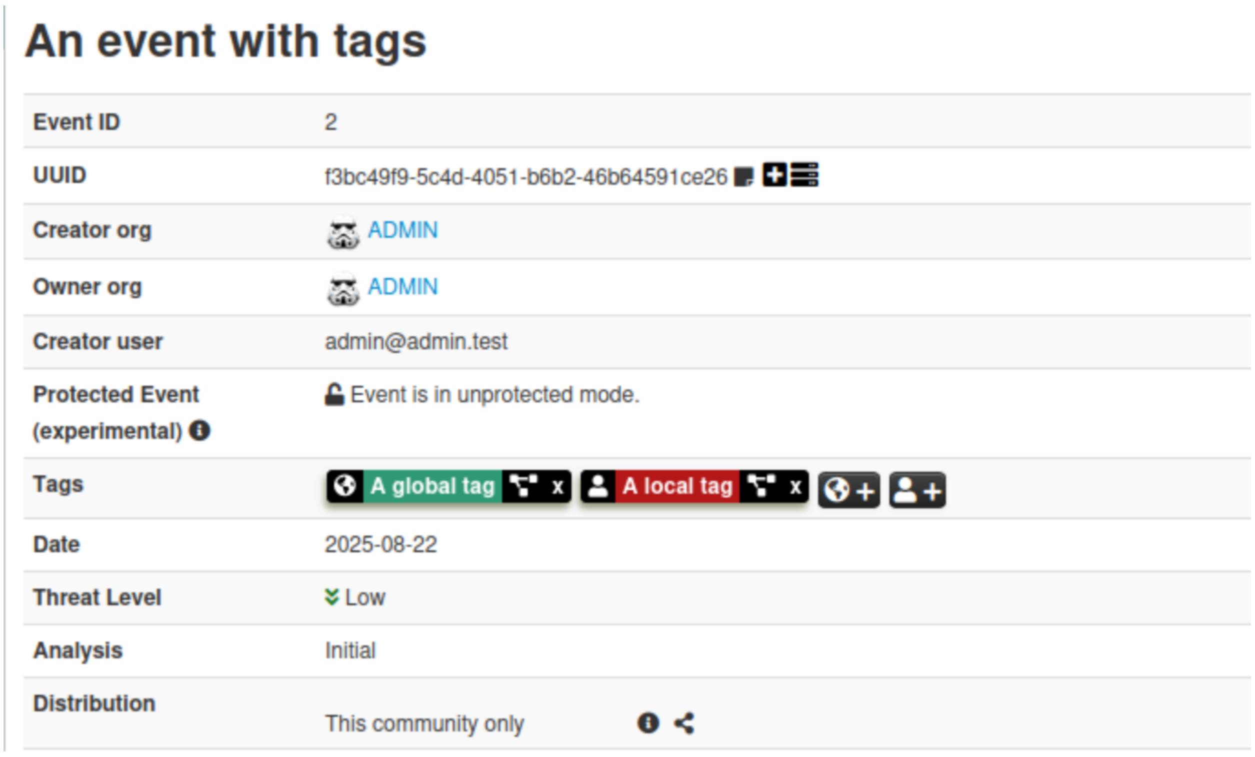
Beyond simple event synchronization, it is also important to verify that any changes made to an event are correctly propagated. For example, if the description of an event is modified, this update must be reflected on the connected servers. To do this, I created a test function that publishes an event, then modifies the *event.info* field before republishing the event, in order to verify that the modification is taken into account during synchronization. Inversely, it should not be possible to modify an event on the instance that synchronized it, so I have another test that verifies this (see Listing 5.1) by checking in particular whether the locked flag has been set to True.

Event enrichment tests

Event enrichment can be achieved in several ways:

- **Adding attributes**: I performed an initial basic test consisting of adding an attribute to an event, then checking that it appeared correctly after synchronization;
- **Adding objects**: similarly, I added a complete object to an event, then checked after publication that all the information in this object was correctly synchronized on the target instances.

- **Adding tags**: I also tested the synchronization behavior of *tags*. Unlike other objects, a tag can be attached in two different ways: - globally, in which case it must be synchronized with the associated event, - or locally, in which case it must not be synchronized (see Figure 5.4).
- **Adding galaxies**: same behavior as tags.
- **Adding an event report**: finally, I implemented a test that consists of creating an *event report* linked to an event, then verifying that this report is correctly synchronized after the corresponding event is published.



An event with tags

Event ID	2
UUID	f3bc49f9-5c4d-4051-b6b2-46b64591ce26
Creator org	ADMIN
Owner org	ADMIN
Creator user	admin@admin.test
Protected Event (experimental)	Event is in unprotected mode.
Tags	A global tag x A local tag x
Date	2025-08-22
Threat Level	Low
Analysis	Initial
Distribution	This community only

Figure 5.4: Event with a global tag and a local tag

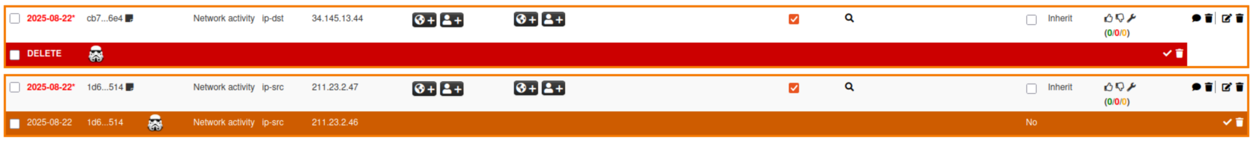
Tests on attribute modification

Several tests were also implemented on attributes:

- **Modification of an attribute**: as with events, I modified the value of an attribute and then checked, after synchronization, that the modification had been taken into account.

- **Deleting an attribute**: I wrote a test that deletes an attribute from an event, then republishes the event to verify that the attribute is also deleted on the connected servers.

- **Proposals for modification and deletion**: it is also possible to make *proposals* on attributes (see Figure 5.5). I therefore implemented tests similar to the previous ones to verify that the synchronization of modification and deletion *proposals* works correctly.



```
2025-08-22*  cb7...6e4  Network activity  ip-dst  34.145.13.44   Inherit  (0/0/0)
DELETE
2025-08-22*  1d6...514  Network activity  ip-src  211.23.2.47    Inherit  (0/0/0)
2025-08-22   1d6...514  Network activity  ip-src  211.23.2.46    No
```

Figure 5.5: Example of proposals to delete and modify an attribute

Distribution tests

Distribution tests have two main objectives: (1) to verify whether an event is actually synchronized on the target instance based on its distribution level, and (2) to check the distribution level applied to the event once it has been transferred to the target instance. The detailed behavior is illustrated in Tables 5.1 and 5.2.

Distribution on A	**Result on B**
Your organization only	The event is not transferred.
This community only	The event is not transferred.
Connected communities	The distribution level is reduced to *This community only* on B.
All communities	The distribution level remains *All communities*.

Table 5.1: Case of **Push**: Instance A ⟶ Instance B

Distribution on A	**Result on B**
Your organization only	The event is imported only if the synchronization user belongs to the organization that owns the event on A. The distribution level remains *Your organization only*.
This community only	The distribution level is reduced to *Your organization only* on B.
Connected communities	The distribution level is reduced to *This community only* on B.
All communities	The distribution level remains *All communities*.

Table 5.2: Case of **Pull**: Instance B ⟵ Instance A

Just like events, galaxies and galaxy clusters have their own distribution level and, consequently, their own synchronization mechanism. However, they obey the same synchronization rules as those presented in the tables above.

An additional rule applies, however: since a galaxy cluster is an instance of a galaxy, its distribution level must be less than or equal to that of the galaxy to which it belongs in order for synchronization to be possible. For example, a cluster defined with the Connected communities distribution (level 2) within a galaxy whose distribution is Your organization only (level 0) cannot be synchronized.

Tests on internal servers

Synchronization between internal instances follows the same synchronization cases as those described above, but also has certain specific features.

First, sharing and distribution level evolution obey specific rules. I have therefore set up a test to verify this behavior. These new rules are summarized in tables 5.3 and 5.4.

Another notable difference concerns the possibility of modifying an event received via a *PUSH* or *PULL* mechanism. To validate this behavior, I implemented a test that verifies that the *locked* field is set to *False* after synchronization, thus ensuring that the event remains editable on the internal instance.

Finally, unlike synchronization with a traditional server, local elements are expected to be shared between internal instances. I therefore developed specific tests confirming that:

- **local tags** are synchronized correctly
- **local galaxies** follow the same behavior and are also propagated

Distribution on A	**Result on B**
Your organisation only	Event/object/attribute not pushed if triggering push of already locally (on instance A) published event. Event/object/attribute synced on publication of an event, even if the organisation publishing is not the host organisation of the instance
This community only	Event/object/attribute distribution stays 'This community only' on B.
Connected communities	Event/object/attribute distribution stays 'Connected communities' on B
All communities	Event/object/attribute distribution stays 'All communities on B'

Table 5.3: Case of internal **Push**: Instance A → Instance B

Distribution on A	**Result on B**
Your organisation only	Event/object/attribute pulled in only if the sync user is member of the event's owner organisation. Event distribution stays 'Your organisation only' on B
This community only	Event/object/attribute distribution decreased to 'Your organisation only' on B
Connected communities	Event/object/attribute distribution decreased to 'This community only' on B
All communities	Event/object/attribute distribution stays 'All communities' on B

Table 5.4: Case of internal **Pull**: Instance B ← Instance A

Sharing Group Tests

In order to validate the proper functioning of *Sharing Groups*, I developed a test consisting of creating an event with a distribution level set to *Sharing Group 1* (see Figure 5.3), then publishing it.

In accordance with the definition of this sharing group, the event must be synchronized on the **MISP**, **MISP 2**, and **MISP 5** instances, but remain accessible only to organizations that are members of this group, namely **Org 1** and **Org 2**.

5.6 Pipeline for deploying multiple instances of MISP

Now that synchronization tests are running correctly locally, it's time to return to the initial goal: to be able to integrate and run them directly within a Forgejo pipeline. However, to make this possible, I need to slightly adapt my method of deploying instances.

This is because the Forgejo pipeline runs in an LXC container. In this context, it is not possible to launch Docker containers inside it to host, for example, the MariaDB database or the Redis service. We must therefore opt for a classic installation of these services directly on the LXC container (see Figure 5.6)

With this new configuration:

- all MISP instances share the same host (*mysql_host*)
- but each has its own dedicated database (*mysql_database*)

Regarding Redis, the configuration remains broadly similar to that used locally, except that it is no longer run in a Docker container. However, there is one limiting factor: Redis only allows 15 separate databases to be configured on the same host. However, a MISP instance currently requires two Redis databases (one for MISP's internal tasks and one for *SimpleBackgroundJobs*). This means that a maximum of seven MISP instances can be deployed simultaneously in this test configuration (14 Redis databases used out of 15 available). Despite this constraint, this number of instances is still sufficient to cover most of the test scenarios needed to evaluate synchronizations.

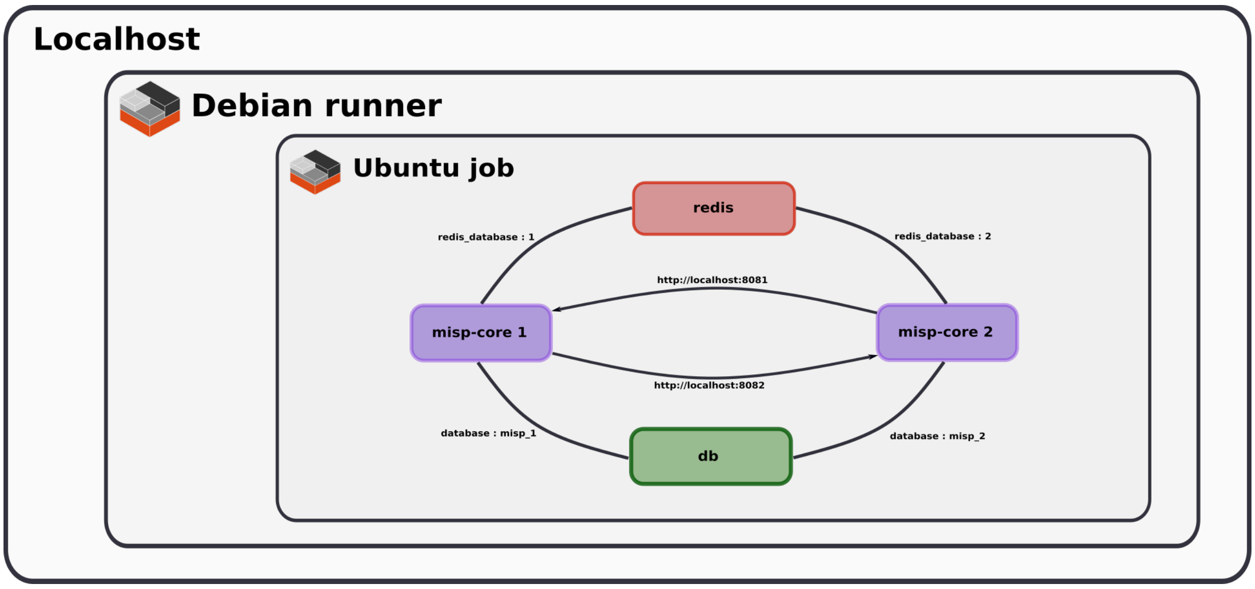


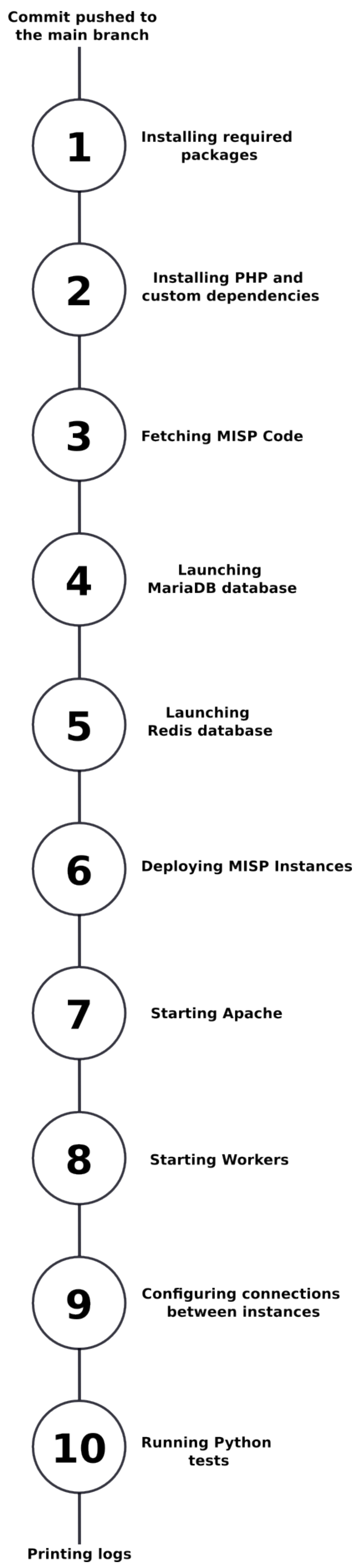
Figure 5.6: Technical architecture for pipeline deployment of 2 MISP instances

To recap, I will outline the step-by-step operation of the pipeline (shown in Figure 5.7).

First, I start with a blank Ubuntu container, on which I install all the modules necessary for MISP to function. Next, I retrieve the latest version of the MISP code, then I launch the necessary services (Redis, MariaDB) directly on the container.

Once these services are available, I deploy my different MISP instances, activate Apache to enable communication with them, and then start the workers to manage background tasks (such as publishing events). I then make the necessary API calls to configure the connections between my instances. Finally, I run my Python tests to verify the synchronization features.

However, I had to remove certain steps from the classic pipeline, such as Update JSON (which fills empty MISP instances with basic data). This step took too long (about three minutes per instance) and was not essential for testing the synchronization features.


Commit pushed to the main branch
1
Installing required packages
2
Installing PHP and custom dependencies
3
Fetching MISP Code
4
Launching MariaDB database
5
Launching Redis database
6
Deploying MISP Instances
7
Starting Apache
8
Starting Workers
9
Configuring connections between instances
10
Running Python tests
Printing logs
Figure 5.7: Lightly simplified version of the pipeline for deploying multiple MISP instances

5.7 Result analysis

At this stage, 53 synchronization tests have been implemented to cover different data sharing scenarios. Among them, 10 fail during execution:

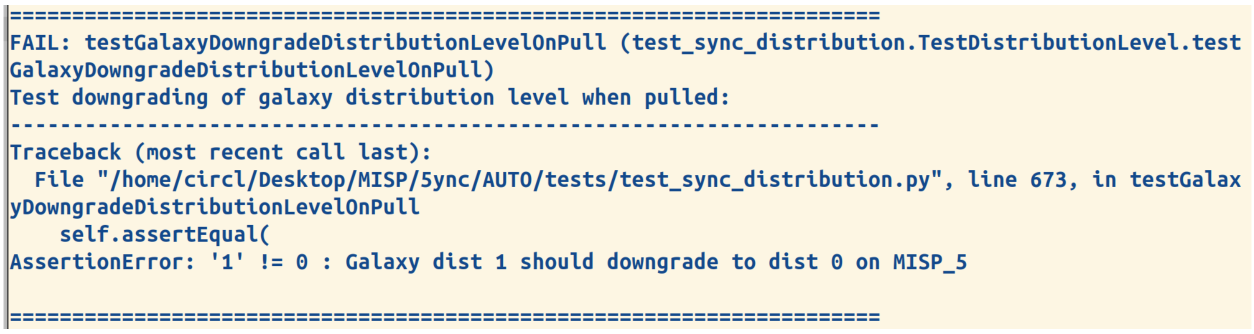
2 failures are related to an anomaly in the processing of API requests, concerning the acceptance and rejection of proposals on an attribute. This anomaly was quickly corrected by the development team.

Two other failures can be considered false positives. As in other test scenarios, I had assumed that analyst data and sightings were synchronized as soon as an event was published. In reality, however, an explicit push all action by the administrator is required:

- **Non-synchronization**: an *analyst data* attached to an event is not synchronized during a *PUSH* action to a server.
- **Non-synchronization**: A *sighting* attached to an event is not synchronized during a *PUSH* action to a server.

The last six failures correspond to real data synchronization issues between instances of the application:

- **Non-synchronization**: a local tag attached to an event is not synchronized during a *PULL* action between internal servers.
- **Non-synchronization**: a local *galaxy cluster* attached to an event is not synchronized during a *PULL* action between internal servers.
- **Incorrect synchronization (vulnerability)**: a *galaxy cluster* with a distribution of *Your Organization Only*, but belonging to a *galaxy* with a distribution of *This Community Only*, is nevertheless synchronized during a *PULL*, when it should not be.
- **Incorrect synchronization (vulnerability)**: a *galaxy* whose distribution is *This Community Only* is shared during a *PUSH* if it contains clusters with the distributions *Connected Communities* or *All Communities*.
- **Incorrect distribution update (vulnerability)**: a *galaxy* with a *This Community Only* distribution does not change to *Your Organization Only* after a *PULL*, which allows another instance to also perform a *PULL* (see Figure 5.8).
- **Incorrect distribution update (vulnerability)**: a *galaxy* with a *Connected Communities* distribution does not switch to *This Community Only* after a *PUSH*, which still allows this instance to perform a *PUSH*.



```
======================================================================
FAIL: testGalaxyDowngradeDistributionLevelOnPull (test_sync_distribution.TestDistributionLevel.test
GalaxyDowngradeDistributionLevelOnPull)
Test downgrading of galaxy distribution level when pulled:
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/circl/Desktop/MISP/5ync/AUTO/tests/test_sync_distribution.py", line 673, in testGalax
yDowngradeDistributionLevelOnPull
    self.assertEqual(
AssertionError: '1' != 0 : Galaxy dist 1 should downgrade to dist 0 on MISP_5

======================================================================
```

Figure 5.8: Example output for a failed synchronization test

It should be noted that these errors mainly concern specific scenarios that are relatively rarely used by most users. However, identifying them highlights the importance of regularly checking all synchronization mechanisms, including those that are used less frequently.

From a quantitative point of view, the complete execution of the Python tests takes an average of 650 seconds. Adding the phase of building and deploying instances via the pipeline, the total time between a *git push* and obtaining the results is around 20 minutes (see Appendix A). When designing my tests, I opted for a division strategy to improve readability, but factoring certain tests could save execution time.

In terms of functional coverage, the tests carried out cover approximately 90% of MISP's synchronization features. The remaining cases mainly concern more complex scenarios, in particular using sharing groups in more advanced scenarios (sharing groups on galaxy clusters, several complex sharing groups on the same instance, etc.).

6 Project Management

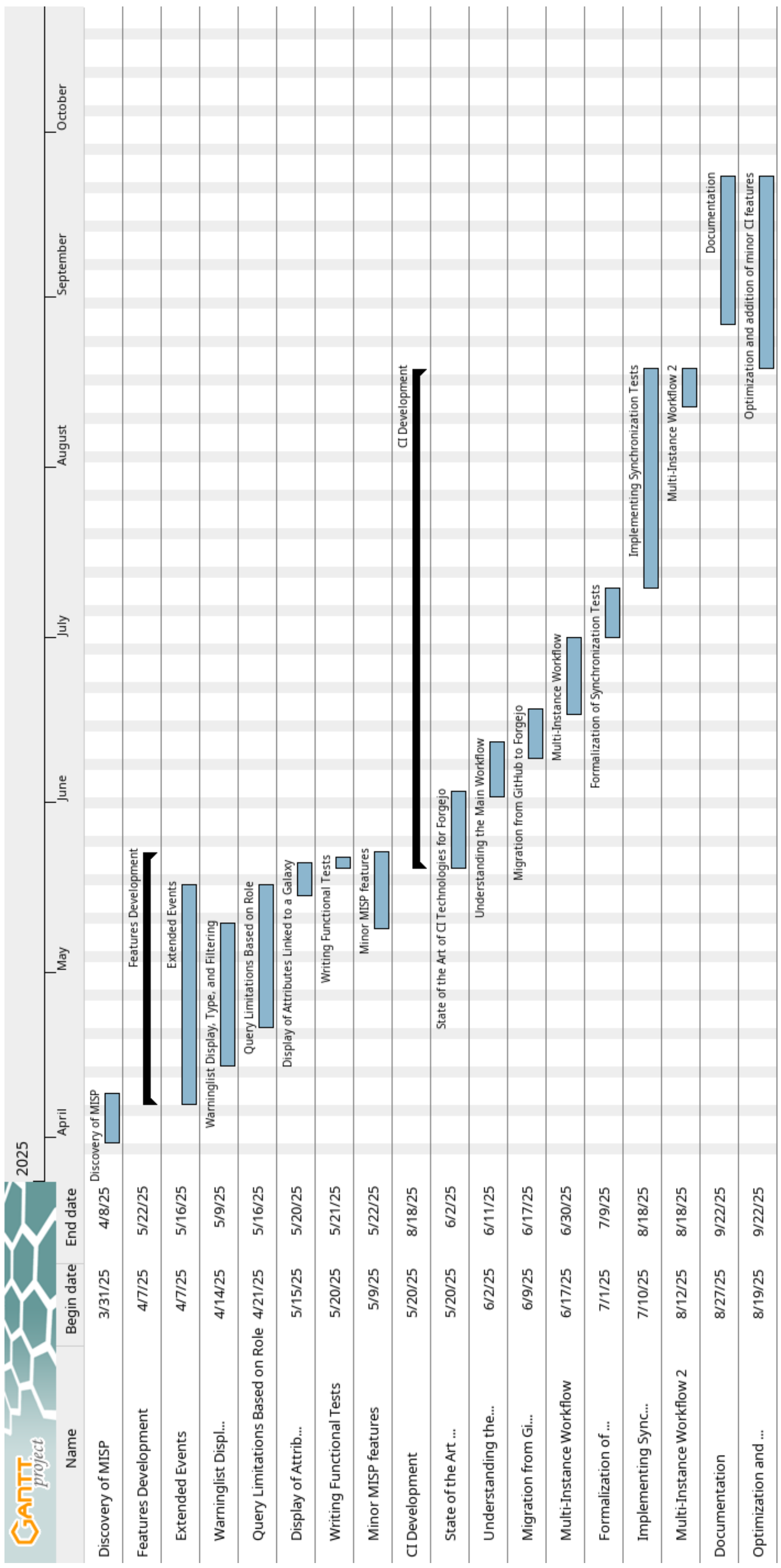
6.1 Working method

Most of the work carried out during this internship was done independently. For the first part, I started by learning how to use and understand how MISP works on my own, with the help of the various documentation available. Then, if I encountered any problems, I could ask Sami for additional information if necessary.

For the second part, Sami initially gave me a goal to achieve, and I was then free to choose the method to achieve it. However, we did a progress review about once a week, so that I could present my progress and we could discuss the methods I had chosen.

6.2 Gantt

The following Gantt chart (see Figure 6.1) summarizes the tasks I completed and the time I spent on them. I had to update it several times due to unforeseen circumstances and the actual time spent on some tasks.



Name	Begin date	End date
Discovery of MISP	3/31/25	4/8/25
Features Development	4/7/25	5/22/25
Extended Events	4/7/25	5/16/25
Warninglist Displ...	4/14/25	5/9/25
Query Limitations Based on Role	4/21/25	5/16/25
Display of Attrib...	5/15/25	5/20/25
Writing Functional Tests	5/20/25	5/21/25
Minor MISP features	5/9/25	5/22/25
CI Development	5/20/25	8/18/25
State of the Art ...	5/20/25	6/2/25
Understanding the...	6/2/25	6/11/25
Migration from Gi...	6/9/25	6/17/25
Multi-Instance Workflow	6/17/25	6/30/25
Formalization of ...	7/1/25	7/9/25
Implementing Sync...	7/10/25	8/18/25
Multi-Instance Workflow 2	8/12/25	8/18/25
Documentation	8/27/25	9/22/25
Optimization and ...	8/19/25	9/22/25

Figure 6.1: Gantt chart for my internship

7 Project outlook

This internship highlighted several errors in the synchronization of the MISP application. The first step is therefore to correct the eight cases of incorrect data synchronization revealed by my tests.

Furthermore, as the objective of these tests is to run automatically each time the code is modified (push on a branch of the repository), it will be necessary to deploy a dedicated Forgejo runner on the CIRCL local server, accessible at `https://helga.circl.lu/`. This runner will be responsible for executing the various pipelines:

- the main pipeline, which verifies the proper functioning of an instance
- as well as the one I developed, which allows checking the correct synchronization between several instances

The tests I have implemented can also serve as models for MISP developers when writing future tests, particularly when adding new synchronization-related features.

Finally, as presented in section 5.4, I also designed a script to deploy multiple MISP instances. Beyond its initial use—creating and testing synchronization scenarios—this script can be reused in other practical contexts requiring the deployment of multiple MISP instances.

8 Conclusion

The main objective of this internship was to ensure the reliability of cybersecurity information sharing between different MISP instances, an application dedicated to the management and dissemination of *Cyber Threat Intelligence*. To achieve this goal, I first contributed to the core of the application by making various functional improvements. I then worked on setting up a test topology to validate all the synchronization mechanisms I had learned to master. Finally, I wrote a series of tests in Python that revealed several synchronization errors. From an automation perspective, all that remains is to deploy a worker on the CIRCL local server to finalize the integration of the tests into the CI/CD chain.

More generally, this internship allowed me to acquire many skills. I participated for the first time in the development of an open source project, and I learned how to design and implement a continuous integration chain from start to finish. I also had the opportunity to contribute to an application that is a major player in cybersecurity in Europe.

Finally, this experience reinforced my decision to focus my professional career on projects that combine software development and cybersecurity. It showed me that these two fields are not separate, but more and more indissociable in modern information system protection solutions.

Bibliography / Webography

[1] CIRCL. Introducing the new extended events feature in misp. https://www.misp-project.org/2018/04/19/Extended-Events-Feature.html. Visited on August 21, 2025. 19

[2] CIRCL. Misp - user guide a threat sharing platform. https://www.circl.lu/doc/misp/book.pdf. Visited on August 21, 2025. 12, 16, 55

[3] CIRCL. Misp concepts cheat sheet. https://www.misp-project.org/misp-training/cheatsheet.pdf. Visited on August 21, 2025. 9

[4] CIRCL. Pymisp's documentation. https://pymisp.readthedocs.io/en/latest/. Visited on August 21, 2025. 34

[5] CIRCL. Rfc 2350 circl - the cert for the private sector, communes and non-governmental entities in luxembourg. https://www.circl.lu/mission/rfc2350/. Visited on August 21, 2025. 3

List of Figures

List of Tables

Listings

Glossary

Affero GPL The Affero General Public License (AGPL) is a free license that, like the GNU GPL, guarantees users the right to use, modify, and redistribute software, but also requires the publication of modified source code when software is used via a network (such as a web service). 7

API Application Programming Interface: Interface enabling different software applications to interact and exchange data securely 4, 8, 20, 24, 25, 26, 27, 32, 33, 34, 42, 44, 59

CERT Computer Emergency Response Team 3

CI/CD Continous Integration/Continous Delivery 3, 4, 5

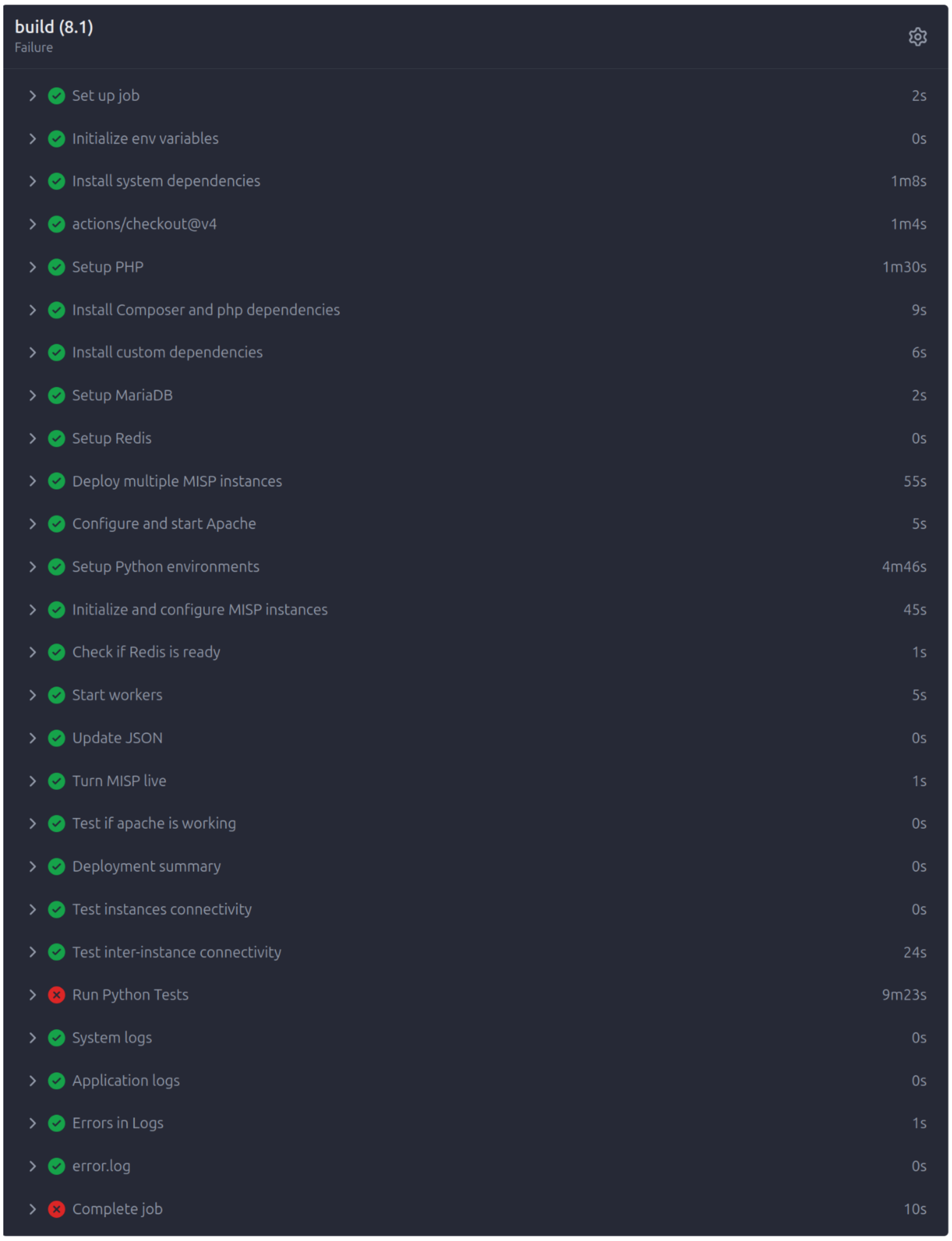
CIRCL Computer Incident Response Center Luxembourg 1, 3, 7, 8, 29, 49

MVC Model(data and business logic management)-View (user interface)-Controller (user action processing logic) 8

uuid Universally unique identifier (128-bit label used to uniquely identify objects) 12, 13, 19, 20, 33

Appendix

A Forgejo pipeline



```
build (8.1)
Failure

> ✓ Set up job                                  2s
> ✓ Initialize env variables                    0s
> ✓ Install system dependencies               1m8s
> ✓ actions/checkout@v4                       1m4s
> ✓ Setup PHP                                1m30s
> ✓ Install Composer and php dependencies       9s
> ✓ Install custom dependencies                 6s
> ✓ Setup MariaDB                               2s
> ✓ Setup Redis                                 0s
> ✓ Deploy multiple MISP instances             55s
> ✓ Configure and start Apache                  5s
> ✓ Setup Python environments               4m46s
> ✓ Initialize and configure MISP instances    45s
> ✓ Check if Redis is ready                     1s
> ✓ Start workers                               5s
> ✓ Update JSON                                 0s
> ✓ Turn MISP live                              1s
> ✓ Test if apache is working                   0s
> ✓ Deployment summary                          0s
> ✓ Test instances connectivity                 0s
> ✓ Test inter-instance connectivity           24s
> ✗ Run Python Tests                         9m23s
> ✓ System logs                                 0s
> ✓ Application logs                            0s
> ✓ Errors in Logs                              1s
> ✓ error.log                                   0s
> ✗ Complete job                               10s
```

Figure A.1: Graphical display of the execution status of the Forgejo pipeline with synchronization tests

Résumé

Ce document présente mon stage de fin d'études réalisé au sein du CIRCL, dans le cadre de ma formation à TELECOM Nancy. Le CIRCL est un service public d'assistance en cas d'incident informatique pour les entreprises, qui développe également plusieurs outils open-source. Parmi eux figure MISP, une application permettant de documenter des cybermenaces et de les partager facilement. C'est sur cette application que j'ai travaillé durant l'intégralité de mon stage.

Dans un premier temps, j'ai contribué à l'implémentation de nouvelles fonctionnalités et à la correction de bugs, afin de me familiariser avec l'architecture et le fonctionnement de l'application. Une fois MISP bien maîtrisé, j'ai pu aborder la seconde partie de mon stage, consistant à tester le bon fonctionnement de la synchronisation des données entre différentes instances de l'application, et ce à chaque commit effectué sur le dépôt principal.

Pour atteindre cet objectif, j'ai conçu une pipeline d'intégration continue sur Forgejo, un gestionnaire de dépôts de code hebergeable en local. Cette pipeline déploie automatiquement plusieurs instances de MISP dans un conteneur LXC, puis exécute une série de tests simulant divers scénarios de synchronisation pouvant survenir en conditions réelles.

Mots-clés : Open-source development, Cyber threat intelligence, Data synchronization, Continuous integration, Automated testing

Abstract

This document presents my end-of-studies internship at CIRCL, as part of my training at TELECOM Nancy. CIRCL is a public service that provides assistance to companies in the event of IT incidents and also develops several open-source tools. These include MISP, an application that allows cyber threats to be documented and easily shared. I worked on this application throughout my internship.

Initially, I helped implement new features and fix bugs in order to familiarize myself with the application's architecture and functionality. Once I had mastered MISP, I was able to move on to the second part of my internship, which consisted of testing the proper functioning of data synchronization between different instances of the application, each time a commit was made to the main repository.

To achieve this goal, I designed a continuous integration pipeline on Forgejo, a locally hosted code repository manager. This pipeline automatically deploys multiple instances of MISP in an LXC container, then runs a series of tests simulating various synchronization scenarios that could occur in real-world conditions.

Keywords : Open-source development, Cyber threat intelligence, Data synchronization, Continuous integration, Automated testing