A coordinated framework for cyber resilient supply chain systems over complex ICT infrastructures

D4.4 Security and Certification IT2 integration

Document Identification			
Status	Final	**Due Date**	28/02/2023
Version	1.0	**Submission Date**	15/03/2023

Related WP	WP4	**Document Reference**	D4.4
Related Deliverable(s)	D4.1, D4.2, D4.3	**Dissemination Level (*)**	PU
Lead Participant	ATOS	**Lead Author**	Jorge Martinez
Contributors	POLITO, STS	**Reviewers**	STS (Grigorios Kalogiannis), POLITO (Cataldo Basile)
			Telefónica (Jose Manuel Manjon Caliz)

Keywords:
SACM, EDC, SCM, certification, IT-2

This document is issued within the frame and for the purpose of the FHISY project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 952644. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

Document Information

List of Contributors	
Name	Partner
Antonio Alvarez	ATOS
Jorge Martinez	ATOS
Cataldo Basile	POLITO
Dimitris Deyannis	STS
Georgios Chatzivasilis	STS
Chrysanthos Chrysanthou	STS

Document History			
Version	Date	Change editors	Changes
0.1	01/01/2023	ATOS	First draft and contribution scheduling
0.11	07/01/2023	STS	Contribution to section 2
0.12	07/01/2023	POLITO	Contribution to section 3
0.2	07/01/2023	ATOS	Contribution to sections 1 and 5
0.21	07/01/2023	ATOS	Initial merge
0.22	16/02/2023	POLITO	Refinements to section 3
0.3	21/02/2023	ATOS	Initial merge and refinements to sections 1 and 5
0.31	24/02/2023	POLITO	Reviewer comments
0.32	24/02/2023	STS	Reviewer comments
0.33	24/02/2023	Telefonica	Reviewer comments
0.34	27/02/2023	POLITO	Addressed reviewer comments
0.35	27/02/2023	STS	Addressed reviewer comments
0.4	27/02/2023	ATOS	Addressed reviewer comments
0.49	28/02/23	ATOS	QA
0.5	06/03/2023	ATOS	Addressed QA comments
0.51	07/03/2023	STS	Addressed QA comments
0.52	07/03/2023	POLITO	Addressed QA comments
0.57	09/03/2023	ATOS	Addressed answers to QA
1.0	15/03/2023	ATOS	FINAL VERSION TO BE SUBMITTED

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Jorge Martinez (ATOS)	14/03/2023
Quality manager	Juan Andrés Alonso (ATOS)	15/03/2023
Project Coordinator	Antonio Alvarez (ATOS)	15/03/2023

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
AH	Authentication Header
CNF	Conjunctive Normal Form
DH	Diffie-Hellman
DNF	Disjunctive Normal Form
EC	Event Calculus
ECE	Evidence Collection Engine
EDC	Enforcement and Dynamic Configuration
ESP	Security Payload
F2F	Farm To Fork
FRF	FISHY Reference Framework
HLP	High-Level Policy
I2NSF	Interface to Network Security Functions
ICMP	Internet Control Message Protocol
IKE	Internet Key Exchange
IRO	Intent-based Resilience Orchestrator and Dashboard
LLC	Low-Level Configurations
MAC	Media Access Control
MLP	Medium-Level Policy
NAT/NAPT	Network Address Translation/Network Address and Port Translation
NED	Network Edge Device
NSF	Network Security Function
PMEM	Predictive MaintenancE Monitoring
R&P	Register and Planner
ReM	Remediation Module
RII	Recipe Instruction Interpreter
RSA	Rivest, Shamir y Adleman
SACM	Security Assurance and Certification Management
SADE	Securing Autonomous Driving Function at the Edge

Abbreviation / acronym	Description
SCM	Security and Certification Manager
SCP	Secure CoPy
SeCaM	Secure Capability Model
SIA	Secure infrastructure Abstraction
TIM	Trust & Incident Manager
TIR	Threat Intelligence Report
TM	Trust Manager
UML	Unified Modelling Language
XFRM	Transform Module
XMI	XML Metadata Interchange
XSD	XML Schema Definition
YAML	Yet Another Markup Language, YAML Ain't Markup Language

Executive Summary

This document is the continuation of deliverable D4.3 [1] and describes the final integration phase, final implementations and final design adjustments for the Security and Certification Manager (SCM) module, composed by the Enforcer and Dynamic Configuration (EDC) on one side and the Security Assurance Certification Manager (SACM) on the other side, during the FISHY project. The latest incremental updates and advances since design and implementation phase of IT-2 are reported in this document.

The SACM verifies the security of services, using provided evidence. In this document there is a detailed description of the internal architecture and the components, including: the Asset Loader Module with an in-depth view of the related Security Assurance Model (Assets and Security Properties); the Vulnerabilities Loader Module; and with new details for the Auditing Module and Event Captor Module (addition of Event Calculus concept into the rules). The update on the Use Cases related work describes the status of the integrations and deployments. It is recommended for a better understanding of this section to have a prior reading/knowledge of FISHY deliverables D4.3 [1], D4.2[2], and D4.1[3].

Concerning the EDC, in this document the following information is included: detailed description; architecture diagrams; data models and workflows of the six EDC components. These components are, namely: Security Capability Manager (SeCaM); NSF Language Manager; Register and Planner; Controller; Enforcer; and Remediation Module (ReM). For the data models, there is an in-depth view of the High-Level Policies (HLP) rules (that specify security requirements); a detailed description of the design process of the Secure Capability Model (that has security controls) and the related Information Model; and the definition of Network Security Functions (NSF). The use case developments and integrations are explained in detail. This section about EDC can be read as a self-contained text, as it brings the relevant information from past deliverables for the reader to have a full picture.

1 Introduction

1.1 Purpose of the document

Deliverable D4.4 documents the integration during the second iteration (IT-2) of the Security and Certification Manager (SCM) components, SACM and EDC, due M30 as per the Description of Action (DoA) planning. It also marks the end of WP4. The design and implementation of EDC and SACM is part of T4.1 and T4.2 and is reported in D4.3 [1], while the integration and latest modifications are carried out in T4.3 and reported in this deliverable D4.4.

1.2 Relation to other project work

Deliverable D4.4 builds on top of D4.3[1] (M26, October 2022), where the design and implementation of EDC and SACM, as SCM components, was reported, and is the closing deliverable for WP4. The information contained in the FISHY deliverables D4.2[2] and D4.1[3] is relevant to this deliverable.

WP3 delivers at M30 the final version of TM that is ready to be integrated with the SCM components described in this deliverable.

WP5 takes both WP3 and WP4 results and outputs and integrates them within the FISHY Reference Framework (FRF), with other components.

WP6 monitors the achievements of the work described here, including integration and application to use cases of the EDC and SACM for the three use cases of FISHY.

1.3 Structure of the document

This document is structured in four major chapters

Chapter 2 presents information about the Security Assurance and Certification Management (SACM), the final designs (architecture, components and interfaces), final implementation and integration as well as an update on the use cases status.

Chapter 3 presents information about the Enforcement and Dynamic Configuration (EDC), the final designs (architecture, components and interfaces), implementation and integration as well as an update on the use cases status.

Chapter 4 presents information on the links with WP5 and WP6.

2 Security Assurance and Certification Management

2.1 Architecture

The STS Security and Certification Management (SACM) is a comprehensive system of models, processes, and tools that verifies the security of services. It uses various types of evidence to prove their compliance with the necessary security requirements and award the appropriate certificate. Additionally, the SACM tool will be used to monitor key components of the ICT infrastructure, with the Evidence Collection Engine allowing for a certifiable view of the security level of the ICT system. Furthermore, the mechanisms will enable the design of audit procedures for the ICT supply chain and automate the security certification models that adhere to security standards, service level agreements, and legal and regulatory obligations (e.g., GDPR). Figure 1 shows the SACM architecture.

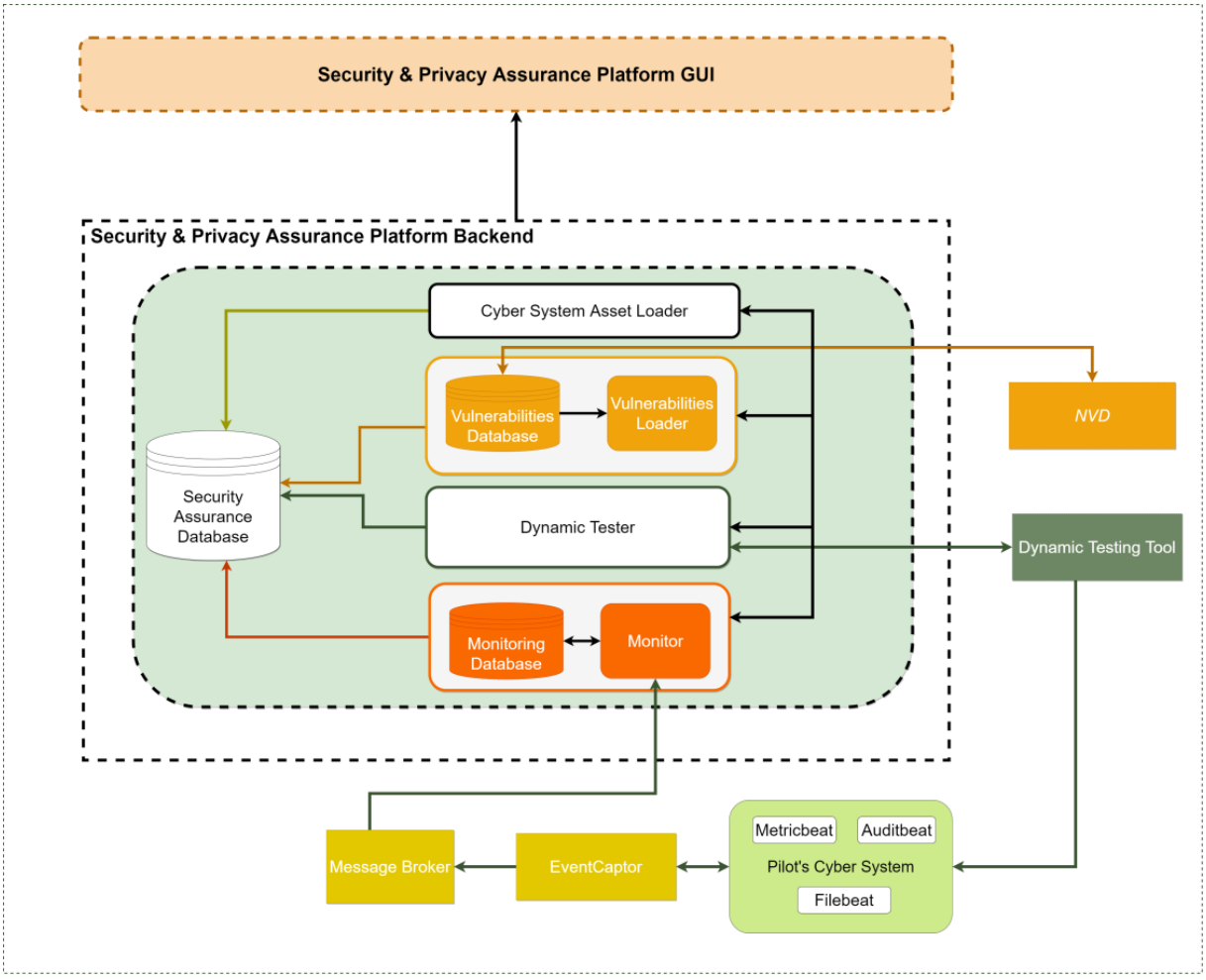

Figure 1 - SACM architecture

2.2 Components

The STS Security Assurance platform can combine active monitoring and dynamic testing to ensure that security measures are correctly and effectively implemented. It can be connected to multiple systems through specialized probes (like event trackers and testing tools) to gather the evidence

needed for assurance and certification reviews. The system works by establishing models that determine the evidence that needs to be gathered from the system and how it should be evaluated (for example, what conditions it should meet) to assess the accuracy and efficacy of the security controls in place. It also allows for the evaluation of temporal event patterns and rules that can be used to detect signature or abnormal patterns.

2.2.1 Asset Loader Module

The Asset Loader Module is the component that receives the asset model of the organization's cyber system. This model includes the assets and their security properties, threats that could breach those properties, and the security controls that protect the assets. The Assurance Model, which is used to define the organization's data, is provided by STS in an Excel file. This file is parsed by the Asset Loader Module, which automatically creates the model for the target organization. The Asset Loader Module is an essential component of the SACM. It stores all the information about the organization, its assets, projects, assessment criteria, and assessment profiles. It is also responsible for receiving and processing the security assurance model for the target organization, which includes details about the organization's assets and the associated security properties, the threats that might compromise these properties, and the security controls used to prevent this.

The Asset Loader Module is based on the SACM Assurance Model, and it can be incorporated into FISHY using either the GUI of the SACM platform by manually entering the model data, or by filling out an Excel file that contains the necessary information, which can then be parsed using the SACM GUI.

2.2.2 Vulnerabilities Loader Module

The Vulnerabilities Loader Module is the part of the assurance platform that loads the known vulnerabilities of the identified assets and updates the assurance platform accordingly. It consists of two sub-components, the Vulnerabilities Loader and the Vulnerabilities Database. To perform its functions, the Vulnerabilities Loader Module uses OpenVas[4], an open-source vulnerability assessment framework/scanner, while the known vulnerabilities are loaded through a daily update feed provided by Greenbone[5].

2.2.3 Monitoring - Auditing Module

This component is a runtime monitoring engine built in Java that provides an API for setting up the rules that need to be monitored. It is composed of two submodules: the monitoring database and the monitor. The module's job is to receive the runtime events from the application's monitored properties and then provide the monitoring results. These outcomes are stored in the monitoring database while the monitor submodule is the main component of the auditing module that determines if a monitoring rule is either broken or fulfilled.

2.2.4 Evidence collection - Event Captor Module

The Evidence Collection Engine or Event Captor Module is employed to provide ongoing, real-time surveillance and evaluation of the FISHY platform's security status. It collects data from multiple layers and sources and aggregates it in real-time, resulting in a comprehensive understanding of each monitored component's security posture. This module utilizes incoming data from event captors, which are software components that form rules based on collected data and triggered events, sending them to the Monitoring Module for assessment. The event captor module is integrated as a lightweight dockerized agent/container in the systems that must be monitored. The agents gather evidence from various sources (e.g., network traffic, security logs, system logs, etc.) and wrap it in an

event format the monitor understands. The events are then transmitted to the monitor via the event collector.

The Event Captor is a tool used to collect data and events from multiple sources, like Elastic Search, Beats, and Logstash, and create rules or a set of rules based on the collected data. The rules are then sent to the Monitoring Module for evaluation, and the Event Captor is initiated through the REST calls from the monitoring module.

2.3 Interfaces

Table 1 - SACM inbound interfaces

Origin	Data	Type of communication
Smart Contracts	Raw data from Smart Contracts	AMQP
FISHY Appliance	Raw data connected for analysis	AMQP/ElasticSearch

Table 2 - SACM outbound interfaces

Destination	Data	Type of communication
FISHY Dashboard	GUI of the SACM	SACM GUI has been properly integrated as part of the Dashboard, and has been deployed successfully. It needs some testing in terms of interaction
Smart Contracts	Auditing Module of the SACM sends data to Smart Contracts	RabbitMQ
Central Repository	Reasoning results. SACM can store to the central repository its reasoning result	REST API

2.4 Description of work done

One of the major works done for the need of FISHY Platform was the transformation of the reasoning mechanism of the auditing module to incorporate the Drools[6] logic engine. The reasoning mechanism of the auditing module is modelled in event calculus, a logical language for representing and reasoning about actions and their effects as time progresses.

The initial version of the auditing module was based on an XML-based language called Event Calculus (EC)-Assertion[7], a first-order temporal logic language. To incorporate the concepts of Event Calculus into the Drools logic engine, we had to create an object-oriented structure that expresses the fundamental ideas of Event Calculus in Drools syntax. This task was difficult due to the need to capture the predicates (Happens, HoldsAt) and Axioms of event calculus in the Drools language, as well as to maintain memory safety in the code. To address this issue, we took reference from Cerbere[8], a Jess[8] tool production system designed to perform online causal, temporal, and epistemic reasoning based on the Event Calculus

2.5 Current status of components and interfaces

Table 3 - SACM inbound interfaces status

Origin	Status
Smart Contracts	Started. Currently reading data from smart contracts and performing reasoning
FISHY Appliance	Not started

Table 4 - SACM outbound interfaces status

Destination	Status
FISHY Dashboard	GUI in the dashboard, has been deployed successfully. Need's some testing in terms of interaction
Smart Contracts	Started
Central repository	Started, expected to be completed by M32 (April 2023)

2.6 Update on use cases

Currently, most of the major components of the SACM platform are developed and integrated into the FISHY platform while pending are few interfaces of the latter that will be concluded in terms of integration in the following up months, integration with Central Repository of FISHY has also started and is expected to be completed by M32, and inclusion of new security rules.

During this time, the SACM component was installed and set up for the FISHY Dashboard, complete with an auditing system that uses Event Calculus and Drools. The integration of the SACM within the FISHY Dashboard has been completed, however, adjustments to the GUI are expected before M34 (June 2023) that will not affect the structure of FISHY or the primary components of SACM. The Asset Loader component has been configured for the other use cases of FISHY/pilots and the security metrics already included in the SACM tool will be used for the rest of the supply chain scenarios. Additionally, a new security metric/rule for using smart Contracts in supply chains has been developed and will be finished before M32.

3 Enforcement and Dynamic Configuration

This section presents the current version of the EDC, which, except for minor modifications and adaptations needed for integration purposes, is the final one. The EDC implements all the features that have been designed and developed. The only part currently a work in progress is related to deploying the generated configurations into the target NSF. The integration with the NED and SIA is not completed yet; it is under test for the demonstrators of the use cases.

This deliverable aims to be the only reference document about the EDC, the only one to use for future references to this component and to avoid interested readers to jump into four documents to reconstruct the details about the EDC. Hence, this deliverable supersedes the past deliverables D4.1[3], D4.2[2], and D4.3[1] where past version of this component has been described.

This document may report entire sentences from the past deliverables, when the content still applies to the final EDC version. To increase readability, we avoided to use quotes and continuous citations and references. The table below reports the relevant sections and the source deliverable.

Table 5 - Comparative table with D4.1, D4.2 and D4.3

Section in D4.4	Section in D4.1, D4.2, D4.3	What is new?
3.1 Architecture and workflows	D4.1, section 3.1 Architecture and components	D4.1 workflows have been expanded and updated according to the latest WP2 results
3.2.1 High-level policy model	D4.1, section 3.3.1 High-level policy model, and D4.2, section 2.2.1.2 HLP Definition	No changes (latest version in D4.2)
3.2.2 Low-level configuration settings (aka configurations)	D4.1, section 3.3.2 Low-level configuration settings (aka configurations)	Slightly updated section to include information on Medium Level Policies MLP
3.2.3 Landscape model	D4.1, 3.3.3 The Security Capability model and Medium-level policies (aka abstract configurations)	No changes
3.3 The capability model (SeCaM)	D4.1, section (fraction) 3.3.3 The Security Capability model and Medium-level policies (aka abstract configurations)	Slightly updated section fraction to include information on specification of policies and model and NSFCatalogue class
3.3.1 Requirements	D4.2, section 2.1.1 Requirements	No changes
3.3.2 The information model	D4.1, section (fraction) 3.3.3 The Security Capability model and Medium-level policies (aka abstract configurations) and D4.2, section (fraction) 2.1.2 The information model	Slightly updated section fraction to include information on the decorator pattern and about automatically generating an NSF's abstract language
Section in D4.4	**Section in D4.1, D4.2, D4.3**	**What is new?**

3.3.3 Supported NSFs	D4.2, section 2.1.4 Supported NSFs	Updated section to include information on Ethereum and F2F use case, filtering rules and SADE use case and the manual operator and SONAE use case
3.4.1 Register and Planner	D4.1, section 3.5 EDC Component: Register and Planner, D4.2, section 2.3.1 Register and Planner	Slightly updated section to include additional information on the APIs
3.4.2 Enforcer	D4.1, section 3.6 EDC Component: Enforcer, D4.2, section 2.3.3 Enforcer and D4.3, section 3.1.2 Enforcer	Slightly updated section to include information about the installation instructions and APIs
3.4.2.1 Translator module	D4.2, section (fraction) 2.1.3 Capability model: management workflow and formats	Updated section fraction to include information on the translation rules, NSF Catalogue, translation of capabilities to NSFs.
3.4.3 Controller	D4.1 section 3.4 EDC Component: Controller, D4.2, sections 2.1.3 Capability model: management workflow and formats and 2.3.2 Controller, D4.3 section 3.1.1 Controller	Slightly updated section to include information about the installation instructions and APIs
3.4.4 Security Capability Management	D4.2, section (fraction) 2.1.3 Capability model: management workflow and formats	Slightly updated section fraction to include APIs for the XSDConverter, and to present it as an independent component, while it was presented in the Security Capability workflow in past deliverables.
3.4.5 NSF Language Manager	D4.2, 2.1.3 Capability model: management workflow and formats	Slightly updated section to present it as an independent component, while it was presented in the Security Capability workflow in past deliverables.
3.4.6 Remediation Module	D4.3, section 3.1.4 Remediation Module	No changes, editorial updates

3.1 Architecture and workflows

Figure 2 presents the architecture of the EDC, including all the components and data they manage.

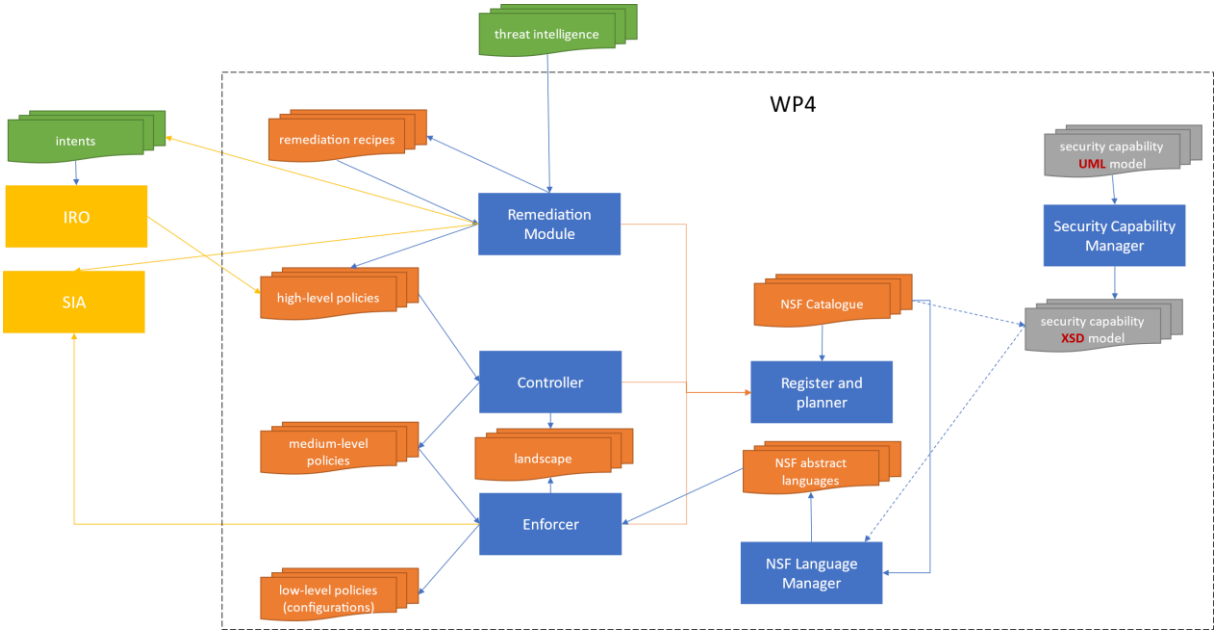

Figure 2 - The EDC architecture

The components of EDC are six (represented as blue rectangles in Figure 2).

The **Security Capability Manager** is a tool that transforms the security capability model, represented in the more general and elegant UML model, into more usable formats. In particular, in the FISHY architecture, the Security Capability Manager converts the security capability UML model into an equivalent XMLSChema model.

The **NSF Language Manager** is a tool that can generate from an NSF Catalogue all the abstract languages that specify medium-level policies for all the NSFs available in the NSF Catalogue.

The **Register and Planner** (or simply **Planner, R&P**) is the component that manages the information related to the NSF available in a given domain (e.g., the list of available firewalls, and VPN gateways). The R&P is the Front-end for accessing information about the instances in the NSF Catalogue, which is represented according to the capability model. It allows importing a catalogue of NSFs and exposes services that implement the features required by the other components, including optimally querying, comparing, and selecting the NSFs based on their capabilities.

The **Controller** refines high-level policies into medium-level configurations.

The **Enforcer** implements the translation functions, i.e., it converts medium-level policies in the knowledge base into device-specific configuration settings, named in FISHY low-level configurations. Moreover, it pushes the generated configurations into the proper NSFs of the target network. Additionally, it may also request changes in the target network if policies are not enforceable or to remediate risks identified by any WP3 threat intelligence tools.

The **Remediation Module (ReM)** recommends actions to mitigate threats other FISHY tools (e.g., PMEM, TIM) have identified. It proposes Remediation Recipes, which are sequences of actions represented in an abstract format tailored to the threat and target network. When a user decides to enforce a Remediation Recipe, the ReM implements the selected Recipe, which may result in

creating or editing high-level policies and intents and changes to the target network. The other FISHY components then manage all the modifications proposed by the ReM. For instance, configurations are obtained by refinement and translation performed by the other EDC tools, Controller and Enforcer, with the help of the IRO.

3.1.1 EDC Workflows

Several workflows describe the features provided by the EDC to the FISHY reference architecture. These workflows are fully compliant with the D2.4[9] description. However, while D2.4 presents them from a more holistic point of view, this deliverable focus on the WP4 tasks and adds more details.

3.1.1.1 Security Capability Data Model preparation workflow (edc-wf1)

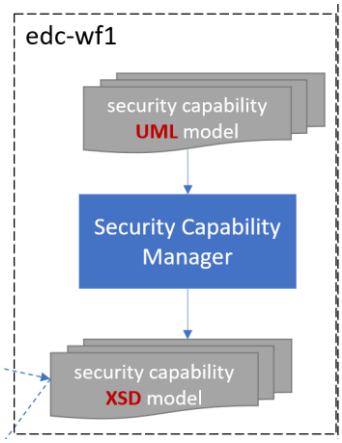

Figure 3 - Security Capability Data Model preparation workflow

The update or extension of the security capability model requires the updates of the UML model. The UML model of the security capability model uses state-of-the-art design patterns aiming for compact, effective, and elegant modelling of the NSF security capabilities. The design is made more sophisticated as one primary requirement is to satisfy a model-driven approach for the translation. That is, the information needed to translate the medium-level configurations into low-level configurations (settings directly usable by the NSFs) is directly provided into the model.

The level of precision of a UML model cannot be matched with other modelling languages that have been developed for more practical uses, such as XMLSchema, JSON Schema, and YAML. However, directly using UML models in practice is usually cumbersome and does not have extensive tool support and libraries available in the main programming languages. Therefore, in FISHY, we had to maintain more forms of the same model to satisfy modelling and implementation needs.

To avoid the above scenario, the Security Capability Manager allows us to only maintain the UML model, as the ground truth, and obtain from the UML model more usable formats. In particular, the rest of the EDC works with XML data; thus, the Security Capability Manager implements a translation module that transforms the UML model (passed as an XML[1] file) into an XMLSchema, which is named Security Capability Data Model. However, the Security Capability Manager can be easily extended to translate the UML into more formats.

Hence, every time changes are performed to the UML model, a new version of the XSD needs to be generated using the Security Capability Manager. However, changes to the UML model are only rarely made. During the FISHY project, after an initial design phase, during which we made the first

[1] XMI (XML Metadata Interchange) is the XML file format used for representing UML class diagrams

design of the model to cover the case of packet filters, we only had to update the UML model a few times. In particular, we updated the UML model when we added the support for describing the IPsec security controls (once when we added support for Strongswan[10], then when we added XFRM[11], and a third time when we homogenized the two models) and when we added support for the application layer filters (when modelling Squid[12]).

This workflow is triggered by calling the Security Capability Manager, which is available both as a .jar and accessible through an API (see Figure 3). This workflow needs to be invoked manually every time there is an update of the Security Capability Model, which is expected to happen rarely.

3.1.1.2 NSF Catalogue management workflow (edc-wf2)

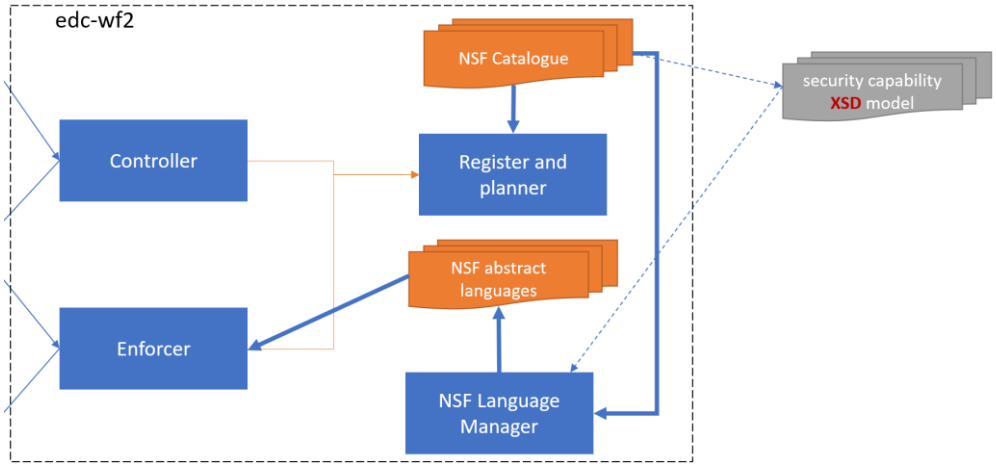

Figure 4 - NSF Catalogue management workflow

In the FISHY scenarios, every company that will use the EDC to enforce security policies in their network automatically has to define the set of NSFs they intend to use. These NSFs need to be described in an **NSF Catalogue**. In the FISHY architecture, the NSF Catalogue is an XML file that must comply with the types defined in the Security Capability Data Model. The NSF Catalogue contains two types of information: the NSF, all their capabilities, and the mapping of the NSF capabilities into their NSF-specific language.

The NSF Catalogue can be manually created; however, it is suggested to start from a FISHY-provided template, including the already supported security controls. When needed, as happened for the use cases, the NSF Catalogue can be extended for describing ad hoc security controls (e.g., the Ethereum authorization module of the F2F web app) or to add specific features (writing human-understandable textual files for the firewalls operators that have to configure their filtering devices manually).

Once an NSF is specified in the NSF Catalogue, it is possible to obtain its **abstract configuration language**, one abstract configuration language for each NSF in the NSF Catalogue (see Figure 4). These abstract configuration languages are XMLSchema files that define the format and syntax of the medium-level policies of the NSF they are intended to.

The first time the NSF Catalogue is created, three components must be made aware of this event: the R&P; the Controller; and the Enforcer. The R&P will provide the other components with information about the NSFs available in that specific domain; thus, it must load the NSF Catalogue. The Enforcer will also need to access the abstract languages to give meaning to the medium-level policies it has to translate and mapping capabilities into NSF-specific language translation rules to perform the translation. On the other hand, the Controller does not need to access this information as it accesses NSF capabilities info directly by querying the R&P.

Of course, the same workflow must be performed each time the NSF Catalogue is updated.

This workflow needs to perform the following steps:

- (1) Call the R&P API method to reload the NSF Catalogue, which will regenerate its internal XML DB with the new catalogue information. The same result can be obtained by substituting in the R&P Catalog folder (both the service is used as a docker or standalone) the NSFCatalogue.xml file and restarting the service.
- (2a) Call the NSF Language Generator to generate the abstract NSF languages. Two methods are available. The first method generates the languages for all the NSF in the NSF Catalogue, and the second method receives the name of the NSF as input and only generates the corresponding abstract language. Since generating the languages requires several seconds (5 to 60 seconds depending on the capabilities it owns) for each NSF to process, the second method is preferrable in case local updates to the NSF Catalogue have been made.
- (2b) Call the Enforcer method to pass the XML of an NSF-specific abstract language (will overwrite existing languages). The same result can be obtained by updating the XML files in the Languages folder (both the service is used as a docker container or standalone) of the Enforcer and restarting the service.

This workflow is manually invoked only when there is a change into the NSF Catalogue. The generated files need to be copied in the Enforcer server without the need for restarting the service.

3.1.1.3 High-level policy refinement and Medium-level Policy translation workflow (edc-wf3)

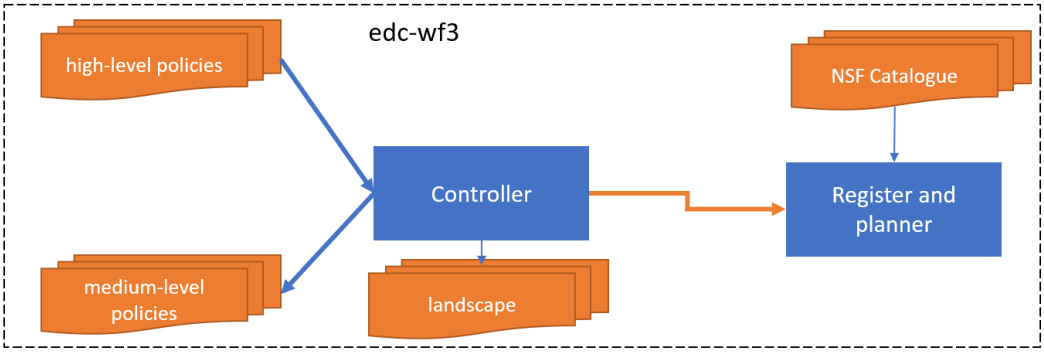

Figure 5 - High-level policy refinement and Medium-level Policy translation workflow

The EDC's main task is refining high-level policies into low-level configurations passing through an intermediate format, the medium-level policies (see Figure 5). The EDC involves three components for translating policies: the Controller and the Enforcer, which perform the translations, and the R&P, which provides information about the NSF for which the translation is performed.

This workflow starts every time a new High-Level Policy is inserted into the policies section of the Central Repository. This insertion triggers the sending of a message that is broadcasted into the policies topic.

The Controller is registered into the policies topic and reacts to a message informing about the availability of new policies (or the update of existing ones) by fetching the referred HLP policy.

The Controller generates the medium-level policies. It analyses the HLP type and fields to determine the security capabilities needed to enforce them. Next, it analyses the landscape for which the HLP needs to be translated and identifies all the NSF that may be used to implement the HLP. It selects the NSFs to use or proposes to place more on precise points of the landscape more NSFs according to a refinement strategy. The Controller has two operating modes. It can automatically select the best refinement strategy for the HPL according to an internal decision engine (the default mode) or provide a list of alternatives to the user that has to select one refinement strategy, a GUI is available to this purpose. Once the NSFs to configure are determined, the Controller processes the HLP fields to generate the medium-level policy for all the involved NSFs.

The Controller stores the generated MLPs in the "mlp" section of the Central Repository for all the generated NSFs.

3.1.1.4 Medium-level Policy translation workflow (edc-wf4)

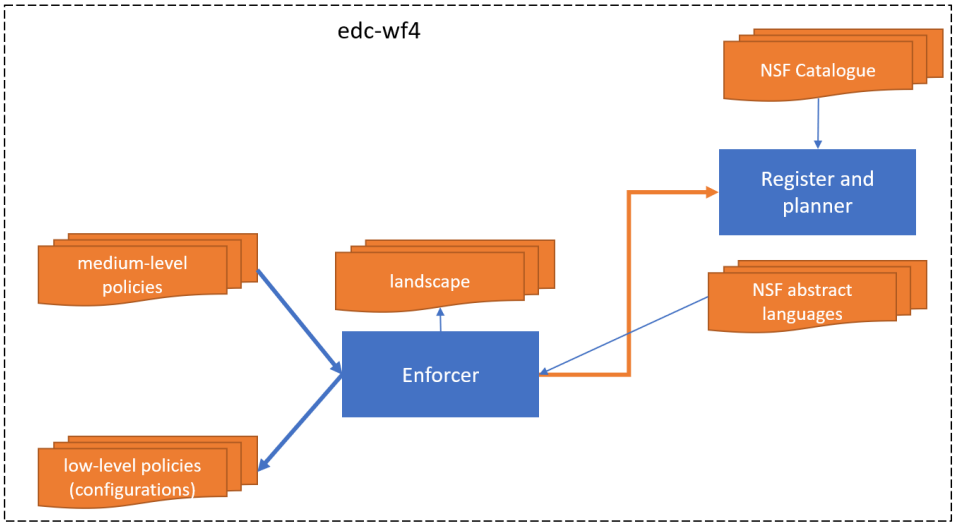

Figure 6 - Medium-level Policy translation workflow

This workflow starts every time a new Medium-Level Policy (MLP) is inserted in the mlp section of the Central Repository (see Figure 6). This insertion triggers the sending of a message that is broadcasted into the "mlp" topic.

The Enforcer is registered into the "mlp" topic and reacts to messages about the availability of new medium-level policy (or the update of an existing one) by fetching the referred MLP policy.

The Enforcer uses the abstract language of the NSF referred to by the MLP and then translates the MLP into a low-level configuration for that NSF.

The Enforcer stores the generated low-level configuration in the "llc" section of the Central Repository.

3.1.1.5 Enforcement workflow (edc-wf5)

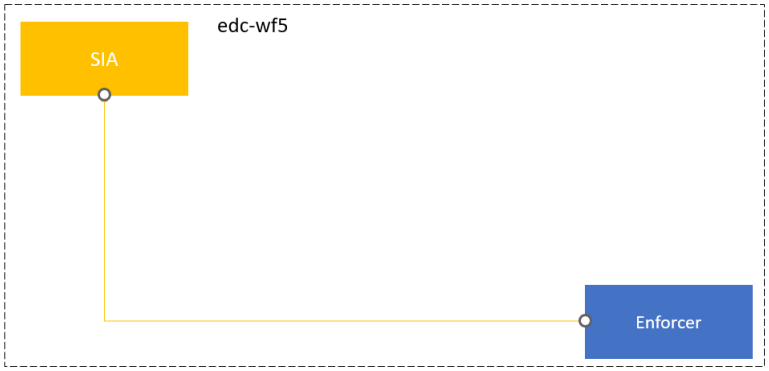

Figure 7 - The enforcement workflow

This workflow starts every time a new low-level configuration is stored in the "llc" section of the Central Repository (see Figure 7). This insertion triggers the sending of a message broadcasted into the "llc" topic.

The LLC-GUI, a frame into the single GUI-dashboard, is registered in the "llc" topic and reacts to messages informing about the availability of new low-level configurations (or the update of an existing one). It adds the name of the NSF to configure to a list of pending configurations. When clicking on the NSF name, the user can inspect both the low-level configuration to push and the MLP from which it derives.

When the user selects the push operation, the Enforcer pushes to the SIA the information about the NSF to configure and the configuration to store.

The final part of this workflow, which actually configures the NSF, is currently under development for the NSFs in the use cases.

3.1.1.6 Remediation proposal workflow

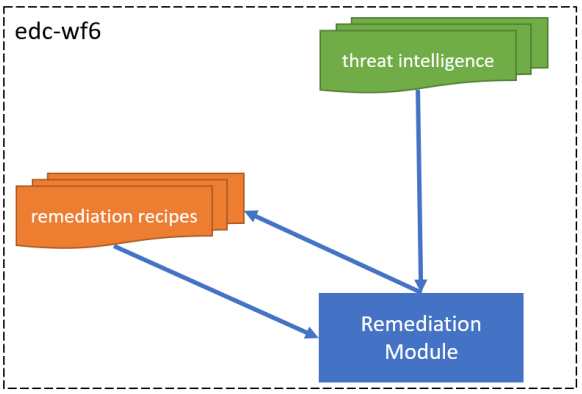

Figure 8 - Remediation workflow

The EDC also includes the Remediation Module, a component that adds more high-level features to the refinement and translation services offered by the Controller and Enforcer.

This workflow starts every time a new message is broadcasted into the threat-intelligence topic (see Figure 8).

The Remediation Module reads the threat intelligence data reported in the message. It parses all the data and verifies if it contains Remediation Recipes to react to the security event reported in the message. If there are no applicable Recipes, the workflow stops.

If some Remediation Recipes are available in the Remediation Module to mitigate the security event reported, the Remediation Module evaluates the appropriateness and effectiveness and determines a mitigation score.

The Remediation Module then writes a remediation-proposal message into the remediations section of the Central Repository that contains a list with the *n* highest score remediations (default *n=3*).

3.1.1.7 Remediation proposal selection

This workflow starts every time a new remediation-proposal is broadcasted on the remediations topic.

The Remediation-GUI, another dashboard frame, is registered in the remediations topic and reacts to messages informing about the availability of new remediation proposals. It shows the listed proposals. When clicking on the remediation Recipe name, the user can see the description of the recipes and the Recipe itself.

When the user selects a Recipe to enforce and pushes the operation, the Remediation-GUI broadcasts into the remediations topic the selected remediation recipe (the ID of the proposal and the internal ordinal number of the proposals) as a new remediation-selected message.

3.1.1.8 Remediation proposal enforcement

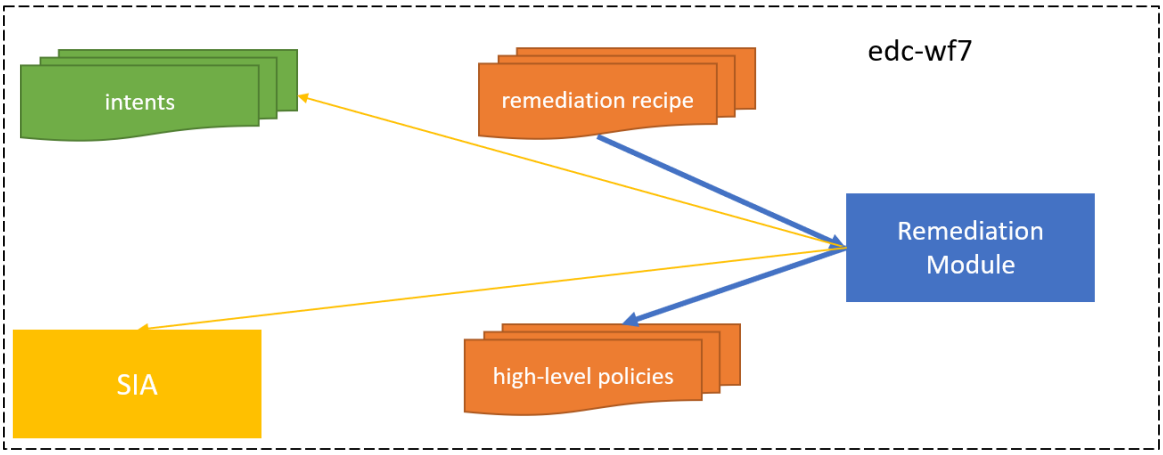

Figure 9 - Remediation enforcement

This workflow (see Figure 9) starts every time a new remediation-selected message is broadcasted into the mitigations section message into the remediations topic.

The Remediation Module reads the remediation proposal information and the ordinal of the selected Recipe. It then executes the Recipe. Different scenarios may happen after the execution of a Recipe. As a favorite approach, the Recipe that proposes to reconfigure the security policies will insert a new intent that the IRO will process. This is fully compliant with the FISHY workflows in WP2 and will be the method used in the demonstration.

Moreover, we have investigated the possibility of improving reaction times, bypassing some of the steps of the FISHY workflow as a sort of reflex arc. Therefore, some recipes may directly inject new HLP policies into the policies section of the Central Repository. These HLP are saved results of a previous IRO compilation, thus they are IRO-validated but allow saving time that may be lost waiting for human re-inspection and re-approval.

Finally, we have also implemented Recipes that require changes in the landscape. The changes proposed are described with a simple graph-based language (substitute, add, or delete nodes and edges of the network layout). This scenario is currently under finalization.

3.2 Data models

3.2.1 High-level policy model

The high-level policies (HLP) are used to specify security requirements the administrators want to be enforced in their ICT system. HLP represent at the same time the output of some FISHY tools, in particular, the IRO Intent Compiler and some TIM components, and the input of the EDC refinement process.

The FISHY project has decided to adopt a policy model based on authorization statements, one of the most used approaches for describing high-level policies, which has already been favorably used in another EC-funded project named SECURED[13].

An HLP is a triple subject-action-object with the following structure:

[subject] action object [(field_type,value) ... (field_type,value)]

Where:

- ***subject*** is the entity (e.g., username, IP address, wallet ID) that needs to access or perform some operation on an object and may be omitted if the policy is applied to the user that defines the HLP.
- ***action*** is the operation to be performed on the object (e.g., allow access, notify).
- ***object*** is the entity (e.g., Internet traffic, all the traffic) target of the action.
- ***(field_type,value)*** is an optional condition that adds specific constraints to the action (e.g., time, content type, traffic type). The value part is a string with a specific format depending on the field type.

HLP policies are defined in XML. This format allows an exhaustive validation of the high-level policies via an ad-hoc XML Schema.

For instance, an HLP used to ban the IP address 1.2.3.4 is:

```
<hspl>
	<subject type="ip_address">1.2.3.4</subject>
	<action>no_authorise_access</action>
	<object>all_traffic</object>
</hspl>
```

In IT-2, the biggest addition to the HLP model is a set of reactive policies. These policies are characterized by one or more enabling conditions that trigger their execution. This mechanism allows the FISHY tools to implement dynamic configurations that are executed only when a specific change in the network occurs.

For instance, the following reactive HLP can be used to ban a wallet ID wallet1 when it establishes more than ten connections in 30 seconds:

```
<reaction id="reaction1">
	<enabling-conditions>
		<threshold>
			<subject type="wallet_id">wallet1</subject>
			<value>10</value>
			<period>30</period>
			<time>second</time>
		</threshold>
		…
	</enabling-conditions>
	<hspl>
		<subject type="wallet_id">wallet1</subject>
		<action>no_authorise_access</action>
		<object>all_traffic</object>
	</hspl>
<reaction>
```

It is important to note that the EDC is not a monitoring component, but its job is only to produce low-level configurations when an HLP or a Threat Information Report (TIR) is fetched from the Central Repository. The triggering of a reactive policy is hence not performed by the EDC.

3.2.2 Low-level configuration settings (aka configurations)

The low-level configuration settings have a primary requirement. They must be understood and **enforceable by the target NSF**.

The enforceability is guaranteed because they will be obtained as the translation of Medium-Level Policies that are aware of the features (i.e., the capability) owned by the NSFs. Therefore, in FISHY, we aim at properly wrapping the raw configuration settings, which can be pushed without further processing into the NSF, with essential management information.

An elementary example of what can be a useful wrapping of configurations is presented below:

```
<llconf nsf="iptables" format = ASCII-text" filename="iptables.rules4">
      -A FORWARD -m state -m state --state ESTABLISHED,RELATED -j ACCEPT
      -A FORWARD -p tcp =d 192.168.0.1/24 -dport 80,443 -j ACCEPT
[..]
</llconf>
<llconf nsf="fw1" format = "b64encoded-binary" filename="bin.rules">
      bajh234jsdf02nsdkf[..]shdjf==
</llconf>
```

3.2.3 Landscape model

The landscape model used is the one defined by the PoSecCo project and used also by the SECURED project. During the project, we never experienced the need for additional features; the original ones have been sufficient for the project's purposes.

Additional information about the landscape specification language can be found in the PoSecCo deliverable D4.2[14].

3.3 The capability model (SeCaM)

The development of the capability model is a crucial point in EDC development. Indeed, almost all the components of the EDC depend on this model. We report here the requirements and how these have been considered for the SeCaM design.

The security capabilities are an abstraction of what a (network) security control can do regarding security policy enforcement. Security capabilities have a central role in several EDC operations.

First, in FISHY, we have decided that being coherent with international standards and models is a method to increase the visibility and impact of our research and results. Therefore, we decided that the security capability model must follow the initial design of a capability model provided by the IETF I2NSF (Interface to Network Security Functions) Working Group[15][16]. Quoting from a draft produced by this working group:

> "*Security Capabilities are independent of the actual security control mechanisms that will implement them. Every NSF should be described with the set of capabilities it offers. Security Capabilities enable security functionality to be described in a vendor-neutral manner. That is, it is not necessary to refer to a specific product or technology when designing the network; rather, the functions characterized by their capabilities are considered. Security Capabilities are a market enabler, providing a way to define customized security protection by unambiguously describing the security features offered by a given NSF.*"

The I2NSF proposed an Information Model intended to provide:

"*a formal description of NSF functionality. Capabilities enable unambiguous specification of the security capabilities available in a (virtualized) networking environment and their automatic processing by means of computer-based techniques. This includes enabling the security controller to properly identify and manage NSFs, and allow NSFs to properly declare their functionality so that they can be used in the correct way.*"

The capability model focuses on six main concepts derived from the analysis of the generic features security controls offer to specify configurations, as already proposed in the I2NSF WG draft[17]. The first four concepts describe the creation of a rule.

- The *conditions* describe the features available at the NSF to identify the target (e.g., the traffic) of the policy (e.g., conditions on the IP addresses or regex on the HTTP MIME type).
- The *actions* are the operations that an NSF can perform on individual elements (e.g., deny packets or encrypt flows).
- Finally, the model describes the *events* that allow triggering the rules' evaluation for specific classes of security controls. Currently, the need for events has not been highlighted in the FISHY use cases.
- The last field is related to the *condition clause*. It serves to express that a rule is activated when all the conditions evaluate to true (DNF, Disjunctive Normal Form) or just one condition is satisfied (CNF, Conjunctive Normal Form).

The following two concepts explain how to build the desired security policy from individual rules.

- The *resolution strategies* describe how the control behaves when more than one rule applies to the same entity. For instance, the first matching rule applies the rule's action at the highest priority when more rules apply.
- Support for *default actions* allows determining how the control behaves when no rule matches the element. For instance, it is essential to know if the control allows specifying the "deny all" default action, as implemented by most firewalls, or not.

The last six bullets represent the 6-tuple needed for describing the security capability of an NSF.

3.3.1 Requirements

The following requirements have been identified for the definition of the Security Capability Model:

- The SeCaM describes all the conditions of the NSF that will be considered relevant by the Consortium.
- The SeCaM describes all the actions of the NSF that will be considered relevant by the Consortium.
- The SeCaM describes all the supported resolution strategies supported by the NSF that will be considered relevant by the Consortium.
- The SeCaM introduces all the features needed for the specification of policies (evaluation clauses, default actions).
- The abstract policy language for configuring an NSF is obtained by transformation from an instance of the SeCaM describing it. Equivalently:
    - There must be no need to explicitly define an NSF's abstract language if its security capability description is already available.
    - The capability description is the only information needed to define the abstract language of an NSF.
- NSFs having the same security capability description will have the same abstract language.

- All the policies for an NSF will also be valid for all the NSFs that own at least all the security capabilities owned by the first NSF; that is, an abstract policy represented for an NSF will also be valid for all the NSFs that own at least all the security capabilities used in the policy.
- The Register & Planner component requires a root element to store the NSFs available in a specific FISHY domain.
- The instructions for translating the abstract policies of an NSF in configuration settings using its proprietary format must be represented in the security capability model, that is:
  - A translator will need no additional code to support a new NSF, and
  - The security capability model makes it possible to describe how the translator will generate the actual low-level configuration in the information model.

3.3.2 The information model

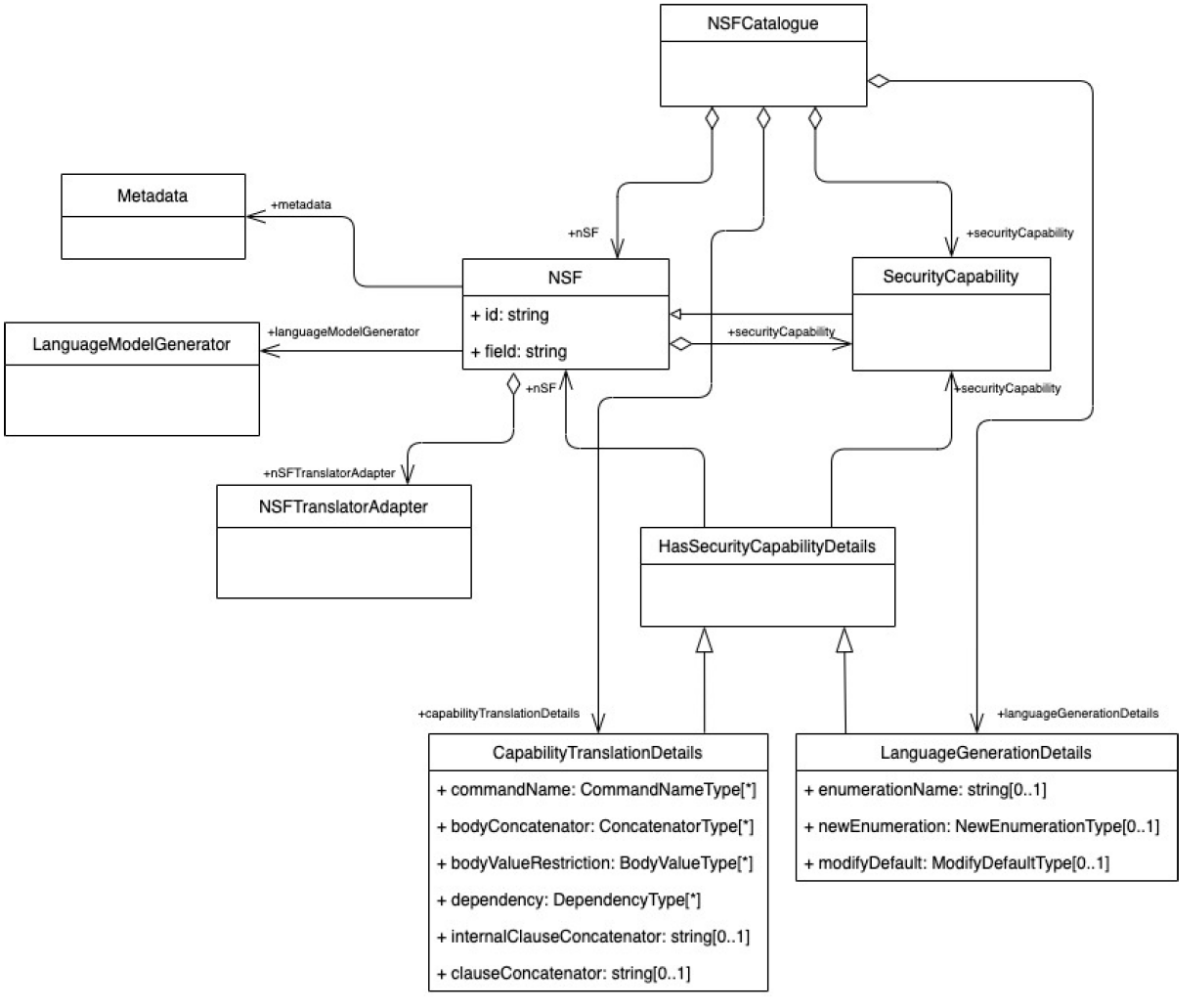

Figure 10 - The Security Capability Model (Information Model)

Figure 10 describes the current status of the design of the security capability model. The security capability model is composed of two classes (*NSF* and *SecurityCapability*), which are abstractions of a Network Security Function and a generic security capability, and an association class (*HasSecurityCapabilityDetail*), to link them according to the *decorator pattern* ". The model also contains the *NSFCatalogue* class, the central aggregator of NSF instances, and the root element of the capability part of the repository.

The decorator pattern provides additional information (as an association class) when capabilities are "attached" to the decorated objects. For this model, this pattern is important to define how capabilities map to device-specific settings. This pattern may be used to add capabilities both statically, when an NSF is described, and dynamically, during network operations. Thanks to the decorator pattern, the model will ease the description of the capabilities by supporting templates, allowing merging them and individually adding and subtracting features from a previous description. For instance, the description of the capabilities of iptables can be determined by extending the description of a generic packet filter and a generic filter on TCP states, i.e., by adding more features to these generic templates. Furthermore, the v2.0 of an NSF can be obtained starting from the description of the v1.0 of the NSF by adding or removing individual capabilities.

The capability model serves to build the **formal model of the abstract configurations** and automate the **translation of abstract configurations** represented in MLP into the low-level configuration settings (i.e., to implement some of the functionalities of the Enforcer). To this purpose, the information model includes classes that allow the generation of the abstract language of an NSF described using its capabilities and translating MLP into low-level configuration settings.

As an additional requirement, we have selected the EDC design to minimize the semantics required by the refinement and translation processes to work with NSF capabilities. In particular, the only hardcoded semantics must be the one associated with the previously mentioned 6-tuple. The internal implementations of the Controller and Enforcer must only be aware of the six categories of capabilities inherited from the I2NSF WG approach to provide refinement and translation services.

This translation must not rely on the NSF-specific code. The MLP are **vendor-independent specifications of the policies** that a given NSF must enforce. In short, MLP expresses configuration directives for the NSF with a generic language instead of the NSF-specific language and settings. Indeed, the security capabilities describe the security control's features regarding security policy enforcement. Thus, capabilities can be mapped to the parameters, fields, and options that can be enabled and configured using a set of configuration directives in the target NSF language. The language generation and translation operations adopt the *adapter pattern* to allow NSFs to be uniformly accessed to perform the translation into low-level settings, even though they may have different capabilities, languages, and interfaces. This pattern allows a unique translation implementation without the need for NSF-specific code.

We automatically generate an NSF's abstract language from the description of its capabilities. The current design includes the *GenerationConstraint* class, a subclass of the association class *HasSecurityCapabilityDetail* to customize the generation of the language (e.g., to customize enumerates, limit the available options). The *LanguageGenerator* class is a singleton that, starting from an NSF instance, generates its abstract language. It also interprets the *GenerationConstraint* instances used to associate the capabilities to the NSF. The *HasSecurityCapabilityDetail* class has been expanded into a (normal) class associated with two separate associations to the NSF they characterize:

- *HasSecurityCapabilityDetail.nsf* and
- *HasSecurityCapabilityDetail.securityCapability* associations.

Indeed, the association classes are certainly elegant at the modelling level. However, they are not supported in almost all the implementation-level formats and tools (e.g., XMLSchema, databases).

The subclasses of *HasSecurityCapabilityDetail* have been organized based on the functions that will use the information they store. There are two subclasses:

- *CapabilityTranslationDetails* reports information the translator will use to generate the low-level configurations.
- *LanguageGeneratorDetails* reports information for the language generator to be used when generating the abstract language associated with the NSF.

Both the language generation and translation operations adopt the *adapter pattern* to allow NSFs to be uniformly accessed to perform the translation into low-level settings, even though they may have different capabilities, languages, and interfaces. This pattern allows employing a unique implementation of the translator without the need for NSF-specific code. This translation must be implemented to support the addition of new NSFs without requiring the having to write new code.

3.3.3 Supported NSFs

Currently, the data models allow describing the first categories of security controls:

- Controls able to filter traffic up to layer 4 and layer 7 of the ISO/OSI stack.
- Channel protection controls, like devices for the creation of IPsec-based VPN, which use cryptography to apply integrity, data authentication, and confidentiality properties to data transferred between two network entities.
- A simple authorization module for the Ethereum web app of the F2F use case.
- A human-understandable textual format for specifying layer4 filtering rules to be used in the Securing Autonomous Driving Function at the Edge (SADE) use case.

The list now includes:

- Iptables[18], a very famous and effective filtering control available in the Linux distributions (only the filtering modules, no NAT/NAPT)
- XFRM, the native IPsec configuration for Linux platforms
- Strongswan, one of the most spread IPsec and IKE implementations
- squid (http://www.squid-cache.org/), a caching proxy for the web that also has an application layer filtering module
- Ethereum authorization module, a simple authorization module available at the Ethereum web app of the F2F use case to ban or allow Distributed Identities or Wallet IDs from accessing the web app.
- SONAE manual operator, a virtual device that has been created to allow exporting the rules generated by the IRO+Controller into a human-readable format. Despite the FISHY demonstration and objectives, SONAE manifested the will to approve and then manually insert the rules into their firewall, to consider the use of FISHY tools also after the project end.

Moreover, the model supports the following generic security controls (i.e., they are abstractions of security control that do not correspond to existing products; thus, the translation is not needed):

- Generic packet filter implementing the 5-tuple paradigm (i.e., Allow and Deny based on conditions on the IP source and destination addresses, IP Protocol Type, source, and destination ports).
- Packet filter with full TCP stateless and stateful filtering
- Generic IPsec-based channel protection module supports both Authentication Header (AH) and Encapsulation Security Payload (ESP) with both tunnel and transport modes.
- Generic IKEv1 module.
- Generic IKEv2 module.

These generic NSFs serve as templates and are helpful in defining standard features to compare the NSFs.

Supporting these security controls required the modelling of several entities, which have been listed below:

- (*actions*) two filtering actions (allow/deny) and some variants (reject-with ICMP, allow-but-log).
- (*actions*) a data model for describing IPsec actions and features
    - AH vs. ESP, and the general model to describe the parts of the packets (payload or header) covered by the protection.
    - Transport mode vs tunnel mode, and the general model to describe different forms of encapsulation that have been introduced in several IETF RFCs.
    - The cryptography information needed to create IPsec Security Associations (e.g., MAC algorithms, symmetric encryption algorithms and modes, authenticated encryption).
- (*actions*) integration of the IPsec actions and features data model to describe IKE-based key-agreement capabilities (DH, RSA, manual specification of security associations).
- (*conditions*) conditions on all the parameters for the most used protocols headers, ranging from layer2 (e.g., MAC addressed) up to layer4 (IP, TCP, UDP) header fields and some session-level protocols (e.g., TLS handshake)
    - Exact match conditions, based on predefined enumerates (e.g., the IP Protocol Type Condition), where an order cannot be specified.
    - Conditions on integer values (e.g., TCP ports) as well as the possibility to specify single values, ranges, and lists.
- (*conditions*) filters to specify stateful conditions on specific protocols (e.g., on the TCP three-way handshake) and networking data (e.g., max bandwidth, max number of connections).
- (*conditions*) filters to specify conditions (e.g., on HTTP protocol fields and payload) based on string match conditions and regular expressions.
- (*resolution strategies*) the First Matching Rule resolution strategy and its variants (e.g., rule chains and Last Matching Rule).
- (condition clause evaluation) DNF and CNF logical formulas.

The model has been validated against the first existing security controls. The validation of the model for a given NSF included the test for:

- The possibility to specify all the features owned by the NSF (i.e., its capabilities).
- The possibility to express a valid configuration written in the NSF-specific configuration language using its abstract language (automatically generated from its capabilities).
- The possibility to translate a policy for an NSF written using its abstract language into a valid configuration written in the NSF-specific configuration language (accepted as valid by the NSF, semantics manually validated by experts).

3.4 Components

3.4.1 Register and Planner

The R&P must provide the FISHY architecture with the following features:

- Read as input an XML file, namely the NSF Catalogue
- Answer to queries to information in the NSF Catalogue:
    - Given an NSF name, return the capabilities it owns.
    - Given a set the capabilities, return a list containing all the NSFs present in the Catalogue that own all of them.
    - Given a set of MLP policies, return a list containing all the NSFs present in the Catalogue that own all of them.

        - The returned NSFs will not be abstract, which means that there are translation details for them and are not just generic devices (e.g., generic packet filter implementing the standard five-tuple).
    - Given two NSFs and the set of capabilities they own *n1* and *n2*, return the set comparison relation among them, either:
        - *n1* contains *n2*,
        - *n2* contains *n1*,
        - *n1* and *n2* intersect, but none of them contains the other,
        - *n1* and *n2* are disjoint.
    - Given an NSF, return the list of NSFs in the catalogue that own at least the same capabilities as the input NSF.
- Expose a (web) service that allows interested entities to access the results of the previous queries.

The requirements analysis highlighted that using an XML Database was a good choice for implementing this component. XML Databases are optimized to search and query data that are represented in the XML ecosystem.

Therefore, at the core of the Register and planner, there is an XML DB, BASEX[19], which has the following features:

- lightweight, robust, high-performance, and scalable XML Database.
- stores, queries and processes textual (XML, HTML, JSON, CSV, others) and binary data
- uses XQuery 3.1 processo[20] with the full support of the W3C Update and Full Text extensions[20]
- exposes a GUI that provides a XQuery editor for writing complex applications and provides various visualizations to explore data interactively
- implements a RESTXQ[21] that enables Web Application development in XQuery

The following query on the NSFCatalogue have been modelled with their XQuery format[22]:

- *two_nsf_comparison.xq*: “receives two NSF names and checks if their Security Capability Sets are equivalent, their intersection is empty, or one is contained in the other one”
    - *nsf_a*: string
    - *nsf_b*: string
- *one_nsf_comparison.xq*: “receives an NSF name and returns the list of NSFs that provide the same Security Capabilities provided by the input NSF”
    - *nsf_a*: string
- *rule_nsf_search.xq*: receives the path of a RuleInstance file and returns the NSF that is able to implement the “policies stated within the above file”
    - rule_instance_path: string (filename)
- *capa_set_search.xq*: “receives a set of Security Capabilities and returns which NSF provides those items”
    - capa_set: comma separated list of strings
- *overall_search.xq*: returns the list “of all the available Security Capabilities arranged by type”.

Only another open-source XML DB is available, eXist[23]. It has been analyzed and considered then discarded for its limitations on the XQuery engine and also due to a performance comparison. Moreover, the BASEX web engine was a very helpful feature for our purposes.

3.4.1.1 API

The APIs exposed by the R&P are just links to the queries whose syntax is

*http://rp-url:8984/rest?run=**query_name**&**param1**=value[&**param2**=value]*

Using CURL on a register and planner available at

Comparing two NSFs:

```
curl -i "https://url-rp:8984/rest?run=two_nsf_comparison.xq&nsf_a=StrongSwan&nsf_b=XFRM" -u admin:admin
```

Look for alternative NSFs:

```
curl -i "https:// url-rp:8984/rest?run=one_nsf_comparison.xq&nsf_in=StrongSwan" -u admin:admin.
```

Find the NSFs that can implement a given rule:

```
curl -i "https:// url-rp:8984/rest?run=rule_nsf_search.xq&rule_instance_path=RuleInstance.xml" -u admin:admin
```

Find the NSF that own a set of capabilities:

```
curl -i "https:// url-rp:8984/rest?run=capa_set_search.xq&
capa_set=SourcePortConditionCapability,IpDestinationAddressConditionCapability" -u
admin:admin
```

Download the list of all the available capabilities:

```
curl -i "https:// url-rp:8984/rest?run=overall_search.xq" -u admin:admin
```

Installation instructions for the Register and Planner can be found in Annex A – EDC Register and Planner installation instructions.

3.4.2 Enforcer

The Enforcer is composed of two sub-components: the MLP rule translator and the enforcement modules.

3.4.2.1 Translator module

The translator is a complex Java-based application, named NSFTranslator, which is in charge of interpreting what is written in a medium-level policy and transforming all the abstract configuration settings into the NSF-specific configuration settings.

Since the security capability model has been designed to adhere to a model-driven approach, all the translation rules are written into the description of the NSF (stored in an NSF Catalogue). Therefore, the translator is an interpreter of the NSFTranslationDetails classes, association classes between a SecurityCapability and an NSF, that determine how to translate individual capabilities for a given NSF. Moreover, the translator takes additional translation instructions from the NSF class, which has a field to provide information at the policy or rule level. For instance, the NSF includes strings used to determine the headers and trailers of a policy or individual rules.

The NSFTranslator takes as input the XMLSchema describing the NSF Catalogue, the XML describing the security capability model, and the name of an NSF in the Catalogue, and the XML describing a valid policy for the NSF. It outputs the low-level configuration for the input NSF. The low-level

configuration contains the configuration file for the NSF and some FISHY-related management information.

The NSFTranslator reads and understands the CapabilityTranslationDetails instances that describe how to translate capabilities written in the abstract language for a target NSF.

```
java –jar NSFTranslator.jar catalogue_filename.xml
          policy.xml [output_filename.xsd]
```

For the NSFTranslator, we are considering several additional optional parameters that allow writing fixed leading or trailing strings for each rule and for the whole policy.

This translation is also available from a web service, which renders the translation accessible to a service accessible with an API.

3.4.2.2 Enforcement module

This module comprises different sub-modules, one for each technology used for pushing the configurations into the target NSF. The information about the technology used for deploying the configurations is saved in a separate Deployment Configuration File. There is one Deployment Configuration File for each of the NSFs stored in the NSFCatalogue. We preferred using separate files as the NSF Catalogue is intended just for the policies.

Deployment Configuration Files follow a "Type: value" structure. An example of a Deployment Configuration File is reported below:

```
NSF: iptables-on-server1
Deployment Type: SSH
SSH-User: deployer
SSH-PWD: strong-pwd
Target Path: /etc/iptables/rules.v4
Command: iptables-restore < /etc/iptables/rules.v4; netfilter-
persistent save
```

The SSH sub-module copies the low-level configurations generated via SCP (Secure Copy) on the target folder of the NSF to configure. Then, it connects via SSH and executes the reconfiguration Command.

The file contains the information that is needed

- The username to use for the SSH connection
- The password of that user
- The path where to store the configuration file
- The filename to use when storing the configuration file
- The name of the command to execute to force the use of the just-stored configuration file

3.4.2.3 APIs

Two APIs methods are available to use this service, described using the Python requests format

- Upload, uploads the file to translate

```
requests.post('http://127.0.0.1:6000/translator', file=file, data={'upload':'Upload'}
```

  - *file* the name of the file to upload, it is expected to contain a medium-level policy to translate. No checks (e.g., validation against the XSD of the NSF language) will be performed with this method.

- Translate, runs the translation process on the last uploaded file

```
requests.post('http://127.0.0.1:6000/translator',
    data={'translate':'Translate','destnsf':destination_nsf})
```

- *destnsf* is an optional parameter that overrides the NSF information in the last uploaded XML file containing the medium-level policy (this option is currently disabled and may be dropped, waiting for the last updates in the integration with the SIA/NED).

Installation instructions for the Enforcer can be found in "Annex B - EDC Enforcer Installation instructions".

3.4.3 Controller

The Controller is the component that refines high-level policies into medium-level policies. The process consists of the following:

- Giving semantics to HLP statements by analyzing all the subjects, objects, and optional fields.
- determining the capability that an NSF should own to enforce what is implied by the HLP statement analysis.
- identifying the NSFs in the landscape that can enforce the required policy.
- selecting the NSFs to configure according to a refinement strategy (e.g., defence in depth, minimization of the security control processing times) or proposing the addition of new NSFs (taken from the NSF Catalogue) to remediate non-enforceable policies.
- deriving the medium-level policy of all the selected NSFs.

The Controller completes its refinement tasks by performing an enrichment process as a forward reasoning task. To this purpose, a forward reasoning engine is at the core of the component. In particular, we have preferred the CLIPS[2] forward reasoning because it is open source, efficient, and implements the RETE algorithm[24]. Other open-source forward reasoners have been considered; however, with its ClipsPyPython porting[25], CLIPS has been proven more effective and easier to integrate into our design. CLIPS supports and transparently manages a Knowledge Base where all the facts are stored.

The reasonings (inferences) are modelled as template rules. Template rules are formed by preconditions, which are expressions to be evaluated on the data on the knowledge base. If the preconditions are met, the template rules perform operations defined in the body of a template rule. The operations that can be performed are direct insertion of facts into the knowledge base in the form of assertions or, since we use ClipsPY, as invocations of Python functions. The possibility to call Python functions extends the features of the forward reasoner as a whole programming language, and all the Python libraries can be used. In the end, the function called will store data in the knowledge base.

Template rules are called as soon as their preconditions are met and in an optimal way. Indeed, for this purpose, CLIPS implements the RETE algorithm, which ensures excellent performance compared to ad hoc solutions.

3.4.3.1 Template rules

The rules interpret the meaning of the HLP field.

HLP processing rules

The following rule determines that if the action of the HLP is about authorization, the system must be ready to implement filtering rules:

```
(defrule filtering
```

[2] https://clipsrules.net/

```
            (or (hspl (action "is authorized to access"))
                (hspl (action "is not authorized to access"))
            )
            =>
                (assert (Case filtering))
        )
```

This second rule, on the other hand, characterize the action to enforce on filtering devices

```
        (defrule filtering-association1
            (hspl (action "is not authorized to access"))
            =>
                (assert (RejectActionCapability))
    )
```

The following rule determines the security protocol to use:

```
        (defrule protection-association
            (hspl (action "protect confidentiality integrity"))
            =>
                (protection-algorithm-sel confidentiality-integrity)
                (assert (DataAuthenticationActionCapability))
                (assert (IpProtocolTypeConditionCapability esp, ah))
        )
```

The following rule specifies that if there is an association between an entity (subject or object) and Distributed ID information, any rule involving them will be enforced by NSFs owning the DistributedIDConditionCapability:

```
(defrule synelixis-use-case-did
        ?f1 <- (DID: ?did)
        =>
            (assert (DistributedIDConditionCapability ?did))
            (retract ?f1)
    )
```

Data analysis rules

The following rule executes a Python function, named *subject-analysis,* which determines all the information about the subject of the HLP statement. It queries the facts in the knowledge base and determines the information that characterizes the subject. For instance, the function identifies if the subject can be associated with IPs, URLs, and other information that allow writing policies (e.g., the Wallet ID of the Ethereum web app. All the associations are added to the DB. The association is also performed by looking at the landscape description.

```
      (defrule subject-association
            (not (hspl (subject "")))
            =>
                (subject-analysis)
        )
```

The following rule specifies that when the subjects and objects have been properly characterized and associations are linking them to the landscape, an analysis of the paths connecting them needs to be

started. This analysis will highlight all the NSFs in the path and, by querying the R&P, their suitability for enforcing the HLP in input.

```
(defrule start-path-searching
    ?f1 <- (Sources done)
    ?f2 <- (Destinations done)
    =>
        (retract ?f1 ?f2)
        (start-path-search)
)
```

3.4.3.2 Architecture and workflow

When a new policy is stored in the Central Repository, a message is broadcasted on the "policies" topic. These messages are read by the Controller that is registered to that topic.

The Controller workflow is straightforward. When the message is broadcasted concerning adding or modifying a new HLP statement, the Controller reads it from the Central Repository.

The downloaded HPL is passed to the refinement engine, which

- Builds an instance of the ClipsPy reasoning engine.
- Parses the HLP statements and fills the HLP template of the ClipsPY reasoning engine, which is passed to the reasoning engine.
- Executes the reasoning engine, which will fire all the template rules until no more facts can be inferred.
- Executes the fact printer, which extracts from the KB all the facts that are relevant for the generation of MLP policies.
- Generates the MLP rules for all the involved NSFs based on the extracted facts.

3.4.3.3 APIs

The Controller is accessible as a web service and usable with a GUI. The Controller has been implemented using the Flask framework[26]. It exposes several API methods to control its features, described using the CURL syntax. Moreover, it has a GUI that helps users implement interactive refinement, were decisions about the strategy to use to select the security controls to use need to be taken.

To upload the HSPL.xml file that contains the High-Level Security Policy:

```
curl -X POST -F file=@path/to/HSL.xml http://url-controller:5000/upload
```

where HSL.xml is the file containing the high-level policy

To upload the file that contains the company information:

```
curl -X POST -F file=@path/to/landscape_file
        http:// url-controller:5000/upload_database.
```

landscape_file is the file containing the description of the networked landscape where the policy needs to be enforced.

To execute the interactive refinement using the last uploaded HLP and the landscape files:

```
curl -X GET http:// url-controller:5000/refinement.
```

The list of NSFs to configure is presented to the user. The user selects the NSFs to configure (the system will ensure that the selections guarantee the enforceability of the policy) and post its selections to the Controller, which continues the enrichment of the information.

To execute the refinement in a non-interactive way:

```
curl -X GET http:// url-controller:5000/refinement_no_gui.
```

In this case, the NSFs to configure are selected using the default strategy and defense in depth (in case multiple alternative NSFs are involved, they are all configured. Later, the user can decide to avoid implementing the individual MLPs.

The generation of the MLP policies from the facts stored in the last execution of the reasoning engine is possible by executing the following method:

```
curl -X GET http:// url-controller:5000/converter
```

It will produce in output the list of the configured NSFs, which can be saved in this way

```
curl -X GET http://localhost:5000/converter > configured_NSFs.json
```

To download a single RuleInstance file generated by converter.py:

```
curl -X GET -O http:// url-controller:5000/download/RuleInstance_name.xml
```

The name of the file to download follows a standard concatenation "RuleInstance_" + "NSFname", where the NSFName can be obtained from the converter. To help download all the rule instances, the following script can be used:

```
RuleInstance_downloader.sh configured_NSFs.json
```

where configured NSFs.json is the output of the converter.

Installation instructions for the Controller can be found in Annex C – EDC Controller installation instructions

3.4.4 Security Capability Management

This component is in charge of translating the XMI file, with the UML model of the security capabilities, into other formats that are more usable in practice.

While it is not the most modern approach, as more compact representations are usually preferred nowadays (e.g., JSON), the XML ecosystem is well recognized, well supported with tools, libraries for all the programming languages, validators, parsers, and everything one may need when writing programs that use XML data. Last but not least, XML databases exist and are optimized to store XML objects efficiently, which seems the best way to implement the Register and Planner.

This component takes as input the XMI and produces as output the XSD containing all the classes in the UML model and the defined types. It is written in Java. The syntax for the translation would be the following:

```
java –jar xmiconverter.jar input_filename.xmi [output_filename.xsd]
```

3.4.4.1 APIs

This component can be manually invoked using Java; however, it is also available as an additional service exposed by the Enforcer. The API to access this feature is

- *Upload*, uploads the XMI file to convert

```
requests.post('http://127.0.0.1:6000/converter', files=file, data={'upload':'Upload'})
```

- *file* is the name of the file to upload; it is expected to contain a valid XMI file containing the security capability UML model

- *generate*, runs the transformation of the XMI into the XMLSchema on the last uploaded file

```
requests.post('http://127.0.0.1:6000/converter', data={'generate':'Generate'})
```

3.4.5 NSF Language Manager

This component is in charge of generating the abstract languages for the NSFs in the NSFCatalogue. The generation of the specific abstract language of an input NSF is generated by an Ad-hoc tool named LanguageModelGenerator. This component is a wrapper for a program named LanguageGenerator, written in Java. The syntax for the language generation would be the following:

```
java –jar LanguageGenerator.jar catalogue_filename.xml
          nsfName [output_filename.xsd]
```

The LanguageModelGenerator takes as input the XMLSchema describing the Catalogue, the XML describing the security capability model, and the name of an NSF in the Catalogue. This tool exports an XMLSchema that defines the syntax of the policies the NSF can enforce, i.e., its abstract language specification.

3.4.5.1 API

This component can be manually invoked using Java; however, it is also available as an additional service exposed by the Enforcer. The API to access this feature is

- Upload, uploads the XMI file to convert

```
requests.post('http://127.0.0.1:6000/converter', files=file, data={'upload':'Upload'})
```

  - *file* is the name of the file to upload; it is expected to contain a valid XMI file containing the security capability UML model

- *generate*, runs the transformation of the XMI into the XMLSchema on the last uploaded file

```
requests.post('http://127.0.0.1:6000/converter', data={'generate':'Generate'})
```

3.4.6 Remediation Module

The Remediation Module (ReM) recommends actions to mitigate the increased network-centred threats that other FISHY tools (e.g., PMEM, TIM) have identified.

The information about the risks to mitigate is reported in a Threat Intelligence Report (TIR) on the topic remediations of the RabbitMQ broker.

The solutions recommended by ReM are named Remediation Recipes (or simply Recipes). Recipes are sequences of actions represented in an abstract format identified by a name.

The changes proposed are:

- modifications to the landscape (e.g., adding new security controls); and
- changes to the (network) Security policies, i.e., new intents or HSPL statements stored in the repository. For maximum coherence of the FISHY framework, intents should be preferred.

Following the FISHY high-level policy, changes are not automatically enforced by ReM. Instead, changes are shown in a dashboard to the FISHY roles that approve them. A fully automatic reaction can be selected via ReM configuration (see workflow edc-wf6 in Figure 8).

These changes will trigger the appropriate workflows, as explained in Sections 3.1.1.7 and 3.1.1.8.

3.4.6.1 Remediation Recipes

All the Remediation Recipes are characterized by:

- a set of *labels* that indicate the threat scenario they address.
- a set of *enabling constraints* that allow understanding when a Recipe is applicable. For instance, Recipes report all the information necessary for their correct deployment and the security capabilities that need to be enforced (e.g., a layer 7 filter), which may not be available in the network.
- the set of remediation *deployment instructions*, written in an abstract language, programmatically states all the steps needed to remediate the identified risks.

Figure 11 presents an example of deployment instructions for a ReM Recipe. This Recipe remediates an ongoing attack against a specific host in the network. It proposes inserting a control to drop a specific payload in the path between the attacker and the target host. For example, this can be useful if the impacted host has become part of a botnet, and the Command and Control messages between them exhibit a specific string that can be filtered to disrupt the communication between the bot and the botnet master.

The language describes different concepts, reported here using different colours.

The keyword representing operations are reported in green. These operations are available as they are exposed either from the FISHY framework or any of its components

- adding security controls (e.g., add_firewall),
- modifying the configuration of specific security controls (e.g., add_filtering_rule),
- modifying the network layout or flows,

The keywords that represent language-specific concepts are reported in red. They are introduced to satisfy the language-required features, e.g.:

- results from past computations (e.g., found_node),
- placeholders for predefined concepts (e.g., new_node),
- inputs from FISHY threat intelligence are reported in orange (e.g., impacted_host_ip),
- information related to the security capability model is reported in blue (e.g., security_capability=UrlRegexCapability).

The actions proposed are tailored to the actual landscape of the target network. In this case, if a security control with payload filtering capability is already present in the path from the impacted host to the attacker, the existing control is reconfigured to filter the target payload; otherwise, a new security control is placed in front of the impacted host and is appropriately configured with the needed filtering rules.

The target landscape is described using a Landscape Description Language, a graph-based representation of the network layout, which describes both nodes (their attributes, e.g., capabilities) and edges. This representation is prone to be imported with graph libraries available for the main programming languages (e.g., iGraph on Python). Currently, it is represented as a simple text file that follows the specification defined during a past EC-funded project (SECURED[13]). Furthermore, we are investigating the possibility of avoiding using network graphs if the network flows (e.g., the SDN ones) are expressive enough for our needs.

```
list_paths from impacted_host_ip to 'attacker'
iterate_on path_list
find_node of type 'l7filter' in iteration_element
if not present
add_firewall behind impacted_host_ip in iteration_element with security_capability=UrlRegexCapability
found_node=new_node

add_HSPL (subject="attacker", object=" impacted_host_ip", action="is not authorized to access") to found_node
endif
end iteration
```

Figure 11 - Example recipe of the ReM

ReM will implement the following workflow (see Figure 11), which can be divided into two phases:

- Remediation proposal selection.
- Remediation proposal enforcement.

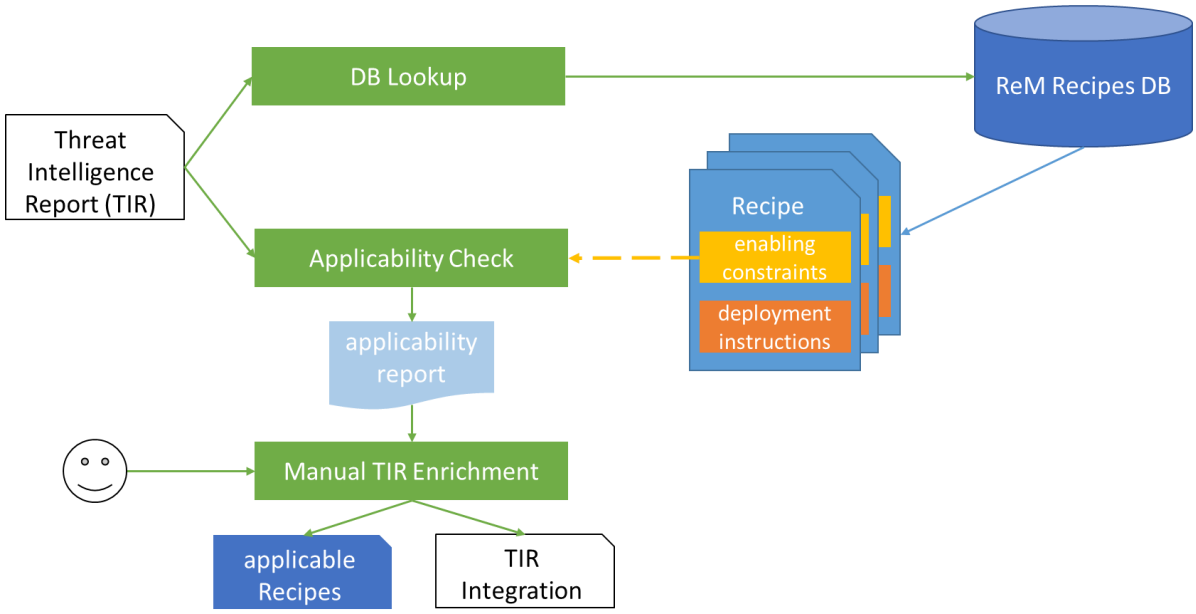

Figure 12 - ReM workflow

In the Remediation proposal selection workflow, the ReM tool listen to the "remediations" topic of the RabbitMQ broker, it reads and parses the information in the broadcasted message, then uses the information in the TIR to look up into a database where all the remediation recipes are stored (*DB lookup*). All the remediation recipes are labelled according to a standard set of categories to allow finding an easier match with threat intelligence information. For instance, the Recipe to mitigate the risks from a malware infection is labelled as 'malware' and further refined with more specific data, such as 'botnet' or 'ransomware'. If needed, custom recipes against specific malware strains may be added to increase the efficacy of the proposed mitigations. The broadcasted messages (at least the ones used in the project use cases) will include the same set of labels, allowing very fast filtering of the recipes.

The recipes produced by the DB lookup phase are all Recipes that can mitigate a specific set of threats. Nonetheless, depending on the target landscape, these recipes may not be deployable in the current threat scenario. For this purpose, an additional step is performed, named Applicability Check, where all the enabling constraints are evaluated. The applicability report lists all the directly applicable recipes. When a Recipe is not applicable, the tools reports the constraints that were not satisfied.

An extension has been developed to the tool which is not used in the use cases, as they aim at reaching full automation. A user analyzing the report can provide additional information enabling additional recipes (e.g., missing IP/URL information or missing information about honey pot networks). The additional data provided by the users are saved in a data structure developed for the purpose, named TIR integration (e.g., within the ReM subcomponent). In this way, these data are neither forgotten nor merged with official information coming from the tool. This information can be integrated into future versions of the FISHY framework, however, the work to integrate this feature only consists in the development of new GUIs that allow inserting the missing data. Currently, the TIR integration can be provided by passing files containing missing information in form of type=value information.

The second workflow, the Remediation proposal enforcement workflow, starts when the user decides on the remediation recipe to enforce. At this point, the Recipe Deployment Engine reads and starts interpreting the deployment instructions.

Initially, all the generic concepts are made concrete with the TIR (and the TIR Integration information). In the Recipe in Figure 11, the generic concepts "*impacted_host_ip*" and "*attacker*" are associated with their actual IP addresses, and this information is made available to the rest of the EDC components.

Then, a Remediation Recipe Interpreter executes the deployment instructions and generates as output, depending on the selected Recipe:

- the HSPL policies or intents to enforce.
- (Optionally, if the networked scenario where the remediation takes place needs it) a set of suggested changes to the landscape. Examples of these changes are:
    - moving network nodes to a different position in the network
    - removing nodes from the network
    - changing the network connectivity (either by connecting a node to a different network or by redefining the flows by changing the routing information)
    - adding nodes (e.g., an NSF)

3.4.6.2 Interfaces with other components and subcomponents

At the current stage of design and development, the ReM tool interacts with the following FISHY components:

- (WP3) TIM writes in the exchange topic of the broker the information discovered by the threat intelligence.
- (WP5) NED will process the instructions provided as output by this subcomponent to remediate the TIM-identified threat scenario.

Figure 13 shows the inputs and outputs that allow interaction with the FISHY components.

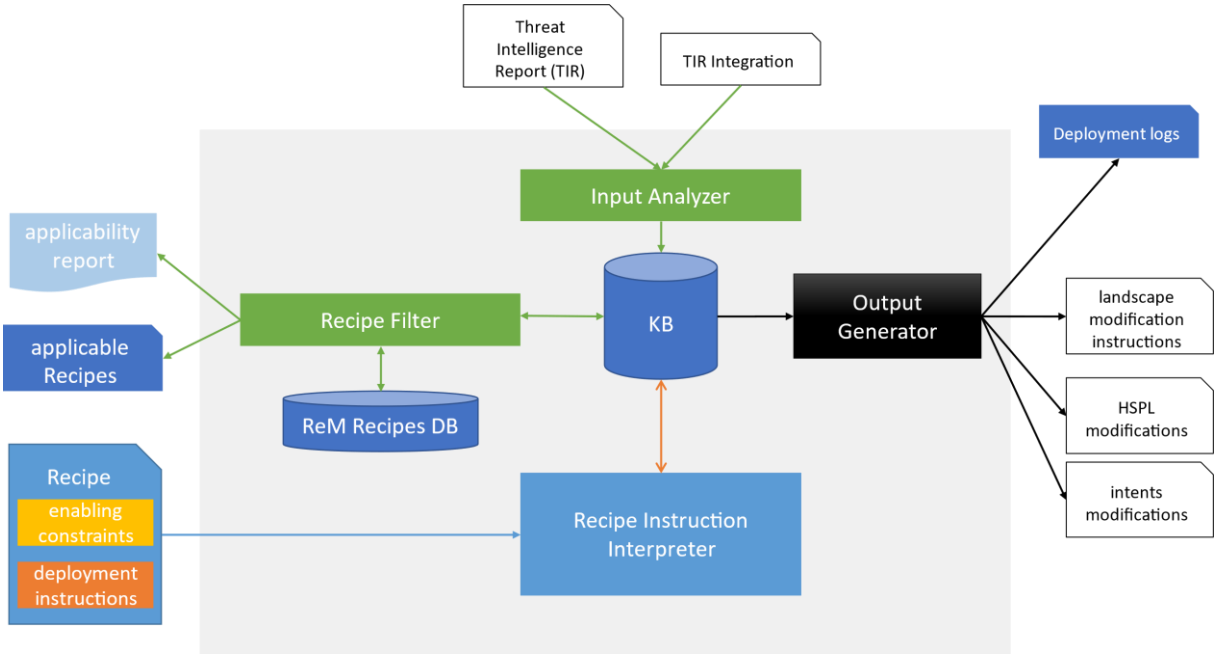

Figure 13 - ReM components and interfaces

3.4.6.3 Architecture

This section provides insights into the Recommendation and Remediation tool, particularly detailing the sub-modules constituting the tool and the exchanged communications (see Figure 12).

The Input Analyzer is tasked with the interpretation of the Threat Intelligence Report. It extrapolates the information needed to enrich the recipe instructions with concrete information from the TIR. This information includes the labels that allow filtering recipes and the IP addresses (and possibly the TCP/UDP ports) of the impacted hosts and the attacker.

The input Analyzer stores the interpreted information in an internal ReM Knowledge Base (KB), which stores all the data produced by all the ReM tool components.

The Input Analyzer is also the module that parses and stores the additional information the user provides as TIR Integration into the KB.

The Recipe Filter is the module that reads from the KB the TIR and TIR integration. It extracts the labels and the additional information that allows for categorizing the threats. At the current development and integration stage with threat intelligence, the enabling constraints are limited to checking a set of labels. The Consortium has discussed further extensions (e.g., the use of ML techniques); however, they have been discarded as they do not add enough value to the ReM technology. Therefore, the use case will use this label-matching features.

From the extracted information, this module selects the Recipe that applies. More precisely, this module queries the ReM Recipe DB and:

- extracts the applicable Recipes.
- checks the satisfaction of the enabling constraints associated with selected Recipes.
- produces the Applicability Report.
- produces the Applicable Recipe list.

The Applicability Report is then produced. The Applicability report is currently a list of Recipes and the information that the Recipe Filter was unable to collect or verify the Enabling Constraints. Examples of Enabling constraints are:

- check for the presence of specific attributes (e.g., the IP addresses of the Command and control for malware infections).
- need for specific security capabilities (e.g., if the Security Capability Catalogue contains an element able to filter by URLs).

The recipes in the Applicability Report for satisfying all the constraints are listed in the Applicable Recipe List. A subset of the Recipes in the Applicable Recipe List is then broadcasted into the remediations topic of the broker. Currently, the default number of Recipes to propose is set to 3, but it can be changed by updating the ReM configuration files.

The Rem oversees deploying the Recipe that it receives as input. When the user selects a recipe among the applicable ones presented into the dedicated Dashboard GUI, the Recipe to deploy is received from the remediations topic of the broker.

The Recipe Instruction Interpreter (RII) interprets the deployment instructions in the Recipe. This module concretizes instructions using the information contained in the Knowledge Base.

The result of the Recipe execution is also stored in the KB. It consists of the following:

- changes to the landscape, such as:
  - adding new nodes, including adding new security controls/NSFs.
  - deleting edges and disconnecting nodes from the network.
  - moving nodes to different networks/subnetworks.
- changes to the HSPL policies.

When the interpretation of all the deployment instructions is complete, the Output Generator reads the internal KB and provides the output in a format that is usable for other FISHY components and to the FISHY users; in particular, it produces:

- the needed modifications to the landscape, including, for example, new NSFs that must be deployed and the necessary alterations to the connections among network nodes, which will be sent to the SIA for their deployment.
- the HSPL policies or the intents to enforce.
- a set of deployment logs describing all the actions taken to remediate the risk.

This module is currently a simple interface to the KB, which queries data and reformats them. When the data formatting is completed, it writes the result in the proper section of the Central Repository.

Installation instructions for the Remediation Module can be found in Annex D - EDC Remediation Module installation instructions

Instructions about the Kubernetes pod can be found in Annex E - Kubernetes pod

3.5 Interfaces

Table 6 - EDC inbound interfaces

Origin	Data	Type of communication
Central repository	Controller is registered to the policies topic, Enforcer is registered to the mlps topic, ReM is registered to the remediations topic. Controller reads HLPs. Enforcer reads MLPs.	RabbitMQ broadcaster.

Table 7 - EDC outbound interfaces

Destination	Data	Type of communication
FISHY Dashboard	Relevant new policies Decisions that controller proposes to the user. Remediations that the ReM proposes to the user.	Web services (sending plain text messages, nothing graphical).
Central Repository	Controller writes MLPs. Enforcer writes LLCs. ReM writes intents or HLPs.	REST APIs
SIA (SIA-NBI)	EDC interacts with the SIA NBI to interact with the NFVI resources in the FISHY domains. It is based in the ETSI SOL0005, using OSM.	REST API (Based in ETSI SOL0005)

3.6 Description of work done

Since the last deliverable reporting the activities done, the following activities have been performed:

- implementation of new parts of the EDC components
  - Design of the enforcement module of the Enforcer
  - novel recipes for the ReM
  - first versions of the simple ReM and Enforcer GUIs
- components improvements
  - major code refactoring of the Controller and ReM,
  - Java part of Security Capability Management, NSF Language Support tools, and the Enforcer Translator restructured,
  - minor code refactoring for the Enforcer,
  - improvements of log and output quality,
  - added missing features needed to support the use cases.
- integration tasks
  - Integration with the central repository, changes requests for the Central Repository sections.
  - Integration with RabbitMQ, development of the EDC components consumers and producers.
- deployment
  - full Dockerization of all the components.
  - definition of a Pod for Kubernetes to install all the EDC components at once.
- use cases
  - Synelixis: update of the use case to the latest EDC component versions.
  - SONAE: analysis of the use case and decision about how to use the EDC(+IRO) in their scenario, creation of the remediation strategies, development and testing of the features needed.

  - SONAE: formal definition of the SONAE operator virtual device, testing and updates of the system to achieve full support.
  - CAPGEMINI: planning and feasibility analysis for the integration of the EDC and IRO into the use case, identification of the policy categories to support, test of the features needed, planning activities for integration.
- model of the software networks
  - analysis of Kubernetes and several plugins completed.
  - progresses in the modelling activities.
  - design of a general-purpose solution to dynamically update software networks with the security controls needed to remediate security issues.
  - implementation in progress.

Update on use cases:

The Synelixis use case employs the IRO+EDC to react to a set of well-known security incidents. The EDC components are fully integrated. The workflows will be completed as soon as the full Central Repository integration will be terminated.

The SONAE use case will employ the IRO+EDC. The workflows have been defined. The EDC will help remediate security incidents. Full integration in their platform still work in progress but, at the current status, does not pose significant challenges.

The IRO+EDC can also benefit the CAPGEMINI use case. Discussions have been made to include it and the task is feasible. It does not appear necessary to prove EDC applicability. In the next weeks it will be decided. Also, for this use case, full integration is straightforward and does not require additional task compared to the other two use cases.

3.7 Current status of components and interfaces

Table 8 - EDC inbound interfaces status

Origin	Status
Central repository	Read HLP policies, MLP and LLC sections of the Central Repository to be completed, EDC application logic ready

Table 9 - EDC outbound interfaces status

Destination	Status
FISHY Dashboard	GUI available but not integrated yet
Central Repository	Write HLP policies, MLP and LLC sections of the Central Repository to be completed, EDC application logic ready
SIA (SIA-NBI)	Currently under implementation. Proposed as a feature to be implemented in OSM source code. Feature accepted, currently in the design phase. Under development. Designed an Enforcer component to manage all the cases, design

Destination	Status
	finished, under implementation

4 Road ahead upon conclusion of WP4

4.1 WP5: Integration in the FISHY Platform

The integration of the EDC with the WP5 has two different aspects: policy deployment and forcing landscape modifications.

The Enforcer is the component in charge of deploying the LLC generated. Now the deployer component of the Enforcer interacts with Open Source Mano through standard APIs to deploy configurations. Moreover, other mechanisms that do not rely on WP5 have been developed (e.g., pushing configurations generated via IRO+EDC though SSH).

On the other hand, our studies to model software networks formally (to exploit their flexibility to increase the security level and increase the effectiveness of reactions) are still ongoing. It is being highly challenging to change the landscape programmatically, i.e., via calls to APIs (e.g., with Kubernetes or OSM). Advancements in the study of the Kubernetes ecosystem have been achieved, with focus on the FISHY sandbox first and eventually the FRF. However, Kubernetes proved to be a complex environment, where the existence of several plugins, the complexity of the interactions of these plugins with the host (in particular iptables), and the youth of this technology make cumbersome having the full control of the platform. Alternative scenarios have been conceived for supporting the use cases and to mitigate the risks of not fully achieving the expected outcomes. However, this research is still ongoing and the project partners firmly believe it can be an important leap to improve network-level security and can lead to important publications.

4.2 WP6: Piloting

Currently SACM is fully integrated in the FISHY Dashboard, using its own Keykloack mechanism for user authentication. Towards this direction, SACM will fully corporate and integrate with the Keykloack mechanism provided by the FISHY Dashboard, in the following months.

In terms of pilot integration, SACM already from the beginning of the project provided the mechanisms for integration with the different pilots and addressing their individual uses cases. In more details:

- For the F2F use case, SACM and its Evidence Collection Engine (ECE) have been integrated to detect 2 out of the 4 F2F attacks and it will be expanded until the end of project to support all F2F attacks. Furthermore, SACM and its ECE have been successfully integrated with Smart Contracts component provided by SYN. Furthermore, SACM will engage the availability and confidentially rules already deployed under its Auditing component to monitor services of the F2F premises in terms of downtime availability and user accessibility.
- For the WBP-trust use case, SACM offers the availability monitoring rule through the FISHY Dashboard. For the SADE use case, STS is currently focused on establishing a tailored metric rule, to facilitate monitoring of IOT devices using network traffic as a parameter. Bilateral calls with the respective pilot are ongoing.
- For the SADE use case, as stated before, the SACM component already includes several security metrics-rules that can be monitored. All available rules cover the Confidentiality-Integrity-Availability model; however, a clear state of the monitored assets must be declared to the asset loader of the SACM in order to facilitate the latter rules. Integration with the SADE use case is in progress and will be concluded in following months.

Currently, EDC is deployed in the F2F use case and is under deployment into the SADE use case. It is integrated into the workflows; however, full technical integration will be achieved when the Central Repository will include all the needed policy sections. Few technical problems still need to be faced for a full integration. None of these tasks have been evaluated as risky because they are technically feasible and easy, do not require much effort, and there are already examples of pieces of code that solve the same issues for other EDC or other WPs' components. The Central Repository misses the sections for storing MLP, LLC, and remediations. However, most of the code needed has been already implemented for reading and storing intents. Data from/to the Rabbit MQ topics broadcasting information about the CRUD operation on policy artifacts still need to be finished. However, similar examples have been addressed to read and push information on the exchange topic used by threat intelligence. The two simple GUIs used by the EDC (Enforcer and ReM) need to be integrated in the Dashboard. The Keycloack mechanism has been already developed however, the final integration steps will be performed in the next days.

In both scenarios, the EDC remediates attacks discovered by TIM (whose information are available in the exchange topic of RabbitMQ) by proposing intents or HLP that mitigate the risks. Once a mitigation is selected, the EDC components refine security policies into low-level configurations and stored into the Central Repository. In the F2F use case, the internal mechanisms download and deploy the generated configurations. In the SADE, the configurations are deployed thanks to the Enforcer mechanisms.

5 Conclusions

This deliverable documents the integration phase for the SACM and EDC during IT-2. The designs and implementations for these tools are final.

This deliverable marks the end of WP4 and most of the Use Cases-related advancements for EDC and SACM, and the final integration, validation and testing for WBP, F2F and SADE Use Cases will be reported at pending WP5 (D5.2) and WP6 (D6.4) deliverables. SACM is fully integrated in the FISHY Dashboard, almost fully integrated with the Central Repository and the Asset Loader is also deployed.

For the EDC, the Information Model is slightly updated, and there are two new components that previously were part of a larger component, the NSF Language Manager and the Security Capability Management, both previously part of the Security Capability.

The EDC is fully deployed into the Synelixis Use Case, almost fully deployed into the SONAE Use Case and ongoing integration for the CAPGEMINI Use Case. The workflows are fully defined and compliant with WP2 results, and all the processes affecting NSFs, NSF Catalogue, HLPs and MLPs and are defined and documented.

The deployment of EDC alongside IRO, for the automatic enforcement of actions, is one of the strong value propositions of FISHY and is exploited in the Use Cases.

In this deliverable a special emphasis is given to the SCM APIs, as it a key point of the integration with the rest of the FISHY Platform. The APIs definition and explanation include code examples and a "how-to" guide:

- Register and Planner: compares NSFs and finds NSFs based on rules/capabilities
- Enforcer: translates into NSF language
- Controller: transforms HLPs to MLPs, load template rules, that interprets the meaning of the HLP field, including HLP processing rules and Data analysis rules
- Security Capability Management: transforms XMI data using the internal models
- NSF Language Manager: generates abstract language from NSFs

This deliverable verifies the completion of milestone "MS19 FISHY Sec.&Cert. blockintegrated (IT-2)".

6 References

[1] **G. Kalogiannis**, "FISHY D4.3 Security and Certification Manager components design and implementation (IT-2)," 2022.

[2] **J. F. Ruiz**, "FISHY D4.2 Security and Certification Manager IT1," 2021.

[3] **C. Basile**, "FISHY D4.1 Security and Certification Manager," 2021.

[4] "**OpenVas**", [Online]. Available: https://www.openvas.org.

[5] **"Greenbone"**, [Online]. Available: https://www.greenbone.net/en/.

[6] "**Drools**", [Online]. Available: https://www.drools.org/.

[7] **M. Shanahan**, "The Event Calculus Explained," [Online]. Available: https://www.doc.ic.ac.uk/~mpsha/ECExplained.pdf.

[8] **D. P. A. C. Y. A. Theodore Patkos**, "An Event Calculus Production Rule System for Reasoning in Dynamic and Uncertain Domains," [Online]. Available: https://arxiv.org/abs/1512.04358.

[9] **J. D. F. C. Admela Jukan**, "FISHY D2.4 Final Architectural design and technology," 2023.

[10] "**Strongswan**", [Online]. Available: https://www.strongswan.org/.

[11] "**XFRM**", [Online]. Available: https://man7.org/linux/man-pages/man8/ip-xfrm.8.html.

[12] "**Squid**", [Online]. Available: http://www.squid-cache.org/.

[13] "**SECURED**", [Online]. Available: https://www.secured-fp7.eu/.

[14] **B. e. al**, "PoSecCo D4.2 – STRUCTURAL LANDSCAPE META-MODEL," 2011. [Online]. Available: https://cordis.europa.eu/docs/projects/cnect/9/257129/080/deliverables/001-D42LandscapeMetaModel.pdf.

[15] [Online]. Available: https://datatracker.ietf.org/wg/i2nsf/about/.

[16] [Online]. Available: https://tools.ietf.org/html/draft-ietf-i2nsf-capability-05.

[17] **I. W. draft**. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-i2nsf-capability-05.

[18] [Online]. Available: https://netfilter.org/news.html.

[19] [Online]. Available: https://en.wikipedia.org/wiki/BaseX.

[20] "**XQuery 3.1 processor**", [Online]. Available: https://www.w3.org/TR/xquery-31/.

[21] "**RESTXQ**", [Online]. Available: https://docs.basex.org/wiki/RESTXQ.

[22] **A. Aurelio Cirella. Rel. Cataldo Basile**, "An abstract model of NSF capabilities for the automated," Politecnico di Torino, Corso di laurea magistrale in Ingegneria , 2022.

[23] **eXist**. [Online]. Available: http://exist-db.org/exist/apps/homepage/index.html.

[24] **C. Forgy**, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, vol. 19, p. 17–37, 1982.

[25] "**ClipsPyPython porting**" [Online]. Available: https://clipspy.readthedocs.io/en/latest/.

[26] "**Flask Framework**", [Online]. Available: https://flask.palletsprojects.com/en/2.2.x/.

[27] "**Kind**", [Online]. Available: kind: https://github.com/kubernetes-sigs/kind.

[28] **Minikube**. [Online]. Available: https://minikube.sigs.k8s.io/docs/ .

Annexes

Annex A – EDC Register and Planner installation instructions

If the R&P needs to be installed individually, these are the instructions to follow.

1. Clone the current repository:

```
git clone git@github.com:torsec/NSF-Catalogue.git
```

2. Move into project folder:

```
cd NSF-Catalogue
```

3. Build Dockerfile:

```
docker build -t register_and_planner .
```

4. Run the Dockerfile containing RegisterAndPlanner server:

```
docker run -p 8984:8984 register_and_planner
```

At this point, R&P server should be listening on http://localhost:8984

Annex B - EDC Enforcer Installation instructions

If the Enforcer needs to be installed individually, the instructions are as follows:

1. Clone the current repository:

```
git clone git@github.com:torsec/security-capability-model.git
```

2. Move into the enforcer folder:

```
cd security-capability-model/enforcer
```

3. Build Dockerfile:

```
docker build -t enforcer .
```

4. Run the Dockerfile containing the proposed Enforcer:

```
docker run -p 8080:5000 enforcer
```

At this point, the Enforcer should be listening on http://localhost:8080

The GitHub repository contains examples showing policies for a security control set and the generated configuration settings.

Annex C – EDC Controller installation instructions

If the Controller needs to be installed individually, these are the instructions to follow.

1. Clone the current repository:

```
git clone git@github.com:torsec/refinement-engine
```

2. Move into the enforcer folder:

```
cd security-capability-model/refinement_engine
```

3. Build Dockerfile:

```
docker build -t controller .
```

4. Run the Dockerfile containing the proposed Controller:

```
sudo docker run -p 5000:5000 --add-host=host.docker.internal:host-gateway controller
```

At this point, the Enforcer Controller be listening on http://localhost:5000

Annex D - EDC Remediation Module installation instructions

If the Remediation Module needs to be installed individually, these are the instructions to follow.

1. Clone the current repository:

```
git clone git@github.com:frank7y/fishy_remediator.git
```

2. Move into project folder:

```
cd fishy_remediator
```

3. Build Dockerfile:

```
docker build -t remediator .
```

4. Run the Dockerfile containing the proposed Enforcer:

```
docker run -p 8080:5555 remediator
```

Annex E - Kubernetes pod

The EDC is also deployed as a pod for Kubernetes.

Architecture

The pod is made of 4 containers:

- Security Capability Model, (branch stable)
- Refinement engine, (branch pod)
- NSF Catalogue
- Remediator engine, fishy-remediator

Requirements

Kind [27] is a tool for running local Kubernetes clusters using Docker container "nodes".

Docker

Even though this mini-guide is tailored for deploying the pod on a Kind Kubernetes cluster, you can run it just as well on any other Kubernetes cluster, such as those created with Minikube [28], or even native clusters.

1. Clone the containers repositories

Make sure to select the correct branch for each project.

2. Build Docker containers

The --platform flag is needed on Apple Silicon Macs. ARM versions of the containers don't work because of some dependencies missing ARM binaries.

```
docker build --platform linux/amd64 -t nsf-catalogue .
docker build --platform linux/amd64 -t secap .
docker build --platform linux/amd64 -t fishy-remediator .
docker build --platform linux/amd64 -t refeng .
```

Show images in the Docker image registry

Check that the four images appear

```
docker image ls
```

3. Create the Kubernetes Kind cluster

```
kind create cluster
```

4. Load the Docker images in the Kind's cluster image registry

```
kind load docker-image fishy-remediator secap nsf-catalogue refeng
```

Show images in the Kind cluster image registry

Check that the four images appear

```
docker exec -it kind-control-plane crictl images
```

5. Spawn the pod

```
kubectl apply -f pod.yml
```

The shell where commands have been executed will automatically attach to the fishy-remediator container.

6. Attach the terminal to the newly spawned pod.

At any moment, it is possible to exit from the shell session with the pod (i.e. the fishy-remediator container by default) and re-attach later via the following command:

```
kubectl attach -it poli-remediator
```

Local ports can be forwarded to Pod ones.

In this way, container services are accessible directly on the machine in which the Kind cluster is running.

```
# kubectl port-forward <kubernetes-resource-name> <locahost-port>:<pod-port>
kubectl port-forward poli-remediator 6000:6000 # Security Capability Model
kubectl port-forward poli-remediator 5000:5000 # Refinement engine
kubectl port-forward poli-remediator 8984:8984 # NSF Catalogue
```