ANASTACIA

Advanced Networked Agents for Security and Trust Assessment in CPS/IoT Architectures

D3.4

Final Security Enforcement Manager Report

This deliverable presents the results of ANASTACIA Task 3.1, which aims to manage the final stage of the enforcement of security requirements established in high-level terms, i.e. security policies, through the ANASTACIA architectural components. Such a final stage is intended to perform security policies refinement from high-level to medium-level, medium-level conflict detection and dependencies identification, and finally, medium-level policies translation and policies management issues notification.

Distribution level	PU
Contractual date	30.09.2019 [M33]
Delivery date	30.09.2019 [M33]
WP / Task	WP3 / T3.1
WP Leader	UMU
Authors	Alejandro Molina Zarca, Jorge Bernal Bernabé, Antonio Skarmeta (UMU), Bagaa Miloud (AALTO)
EC Project Officer	Carmen Ifrim carmen.ifrim@ec.europa.eu
Project Coordinator	Softeco Sismat SpA Stefano Bianchi Via De Marini 1, 16149 Genova – Italy +39 0106026368 stefano.bianchi@softeco.it
Project website	www.anastacia-h2020.eu

ANASTACIA has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement N° 731558 and from the Swiss State Secretariat for Education, Research and Innovation.

This document only reflects the ANASTACIA Consortium's view.
The European Commission is not responsible for any use that may be made of the information it contains.

Table of contents

Index of figures

Index of tables

PUBLIC SUMMARY

This deliverable describes the outcomes of task 3.1, which is in charge of designing and developing algorithms, protocols & mechanisms required for policy refinement, translation as well as conflicts and dependencies detection in the Policy Interpreter of the ANASTACIA architecture.

The main objective of the task is to carefully behold the interactions among IoT objects and the ANASTACIA architecture components to ensure that security requirements are met in an end-to-end fashion. Those security requirements are established in high-level terms, namely in the form of high-level security policies affecting all or a selected set of objects, or even defining a global desire to defend privacy or other security aspects. This task also develops a policy conflict and dependencies detector. Once a policy or a set of policies are defined, the conflict detector analyses them in order to detect different kind of conflicts and dependencies. The Policy Interpreter (in the past Enforcement Manager) then is in charge of refine, translate and manage security policies at different levels of abstraction.

In this sense, this particular deliverable is the report regarding the Security Policy Interpreter module of the ANASTACIA framework, including its goals, design (such as interfaces, processes, relationships within the rest of Architectural components), main features, as well as the latest advances in its development.

1 INTRODUCTION

1.1 AIMS OF THE DOCUMENT

This document is part of ANASTACIA WP3 "Policy Enforcement and Run Time Enablers", which aims to design and develop algorithms, protocols & mechanisms that form the intelligence of the Security Enforcement Manager processes, addressed by the policy interpreter, the ANASTACIA security orchestrator and the Security Enforcement Enablers. Provide effective co-ordination between various and heterogeneous policy nodes by specifying (within the architecture) constraints and trade-offs at the micro level, to create robustness, efficiency and performance at the policy. Explore the opportunities that NFV and SDN jointly offer in intelligently coping with security threats against IoT services and enable the orchestration of network and cloud resources in a security policy-driven fashion.

Concretely, this deliverable is scoped in Task 3.1 of WP3, which aims to carefully behold the interactions among IoT objects and the ANASTACIA architecture components in order to ensure that security requirements are met in an end-to-end fashion. Those security requirements are established in high-level terms, namely in the form of policies affecting all or a selected set of objects, or even defining a global desire to defend privacy or other security aspects. This task will also develop an inference engine to enable security policy analysis.

The Policy Interpreter will be in charge of mapping policies defined for the flows and E2E communications to a collection of security properties to be deployed for dealing with security aspects required by objects, without altering their normal operations. Security policies will be enforced by the SDN controllers and the NFV MANO orchestrator at the Control Plane of the system. The management/orchestration of the security policies across the different components of the ANASTACIA architecture is carried out by the ANASTACIA Security Orchestrator, defining close coordination between this task and the other two tasks of WP3.

The main goal of this particular deliverable is to provide the outcomes regarding policy the Policy Editor Tool and the Security Policy interpreter modules of the ANASTACIA framework, including the design, interfaces, features as well as the development.

This document is structured as follow: Section 2 provides a state of the art of current policy refinement solutions, techniques and related solutions. Section 3 provides a discussion on progress beyond the state of the art. Section 4 gives an overview of the policy enforcement manager modules design, contextualizing the policy interpreter in the ANASTACIA framework. Section 5 is the core of the deliverable, since it defines the policy enforcement process, intended to translate the high-level policy intents or "desires" to specific configurations enforceable in the underneath infrastructure (either physical or virtual system). Section 6 describes the implementation of the policy interpreter, policy editor tool, security enablers provider as well as an overview of the security orchestrator implementation which will be described more in detail in deliverable D3.5. Finally, section 7 concludes this deliverable.

1.2 APPLICABLE AND REFERENCE DOCUMENTS

This document refers to the following documents:

- ANASTACIA project deliverable D1.3 – Initial Architecture Design.
- ANASTACIA Grant Agreement N°731558 – Annex I (Part A) – Description of Action.
- ANASTACIA Consortium Agreement v1.0 – December 6th 2016.
- ANASTACIA deliverable D1.1 – Holistic Security Context Analysis.
- ANASTACIA deliverable D1.2 – User-centred Requirement Initial Analysis.
- ANASTACIA deliverable D2.1 – Policy-based Definition and Policy for Orchestration, initial report.
- ANASTACIA deliverable D2.5 – Policy-based Definition and Policy for Orchestration, final report.

1.3 Revision History

Version	Date	Author	Description
0.1	04.07.2019	Alejandro Molina Zarca (UMU)	First table of contents and first contribution
0.2	22.07.2019	Alejandro Molina Zarca, Jorge Bernal, Antonio Skarmeta (UMU), Miloud Bagaa (AALTO)	Table of contents update
0.3	23.07.2019	Alejandro Molina Zarca (UMU)	Sota and key innovation
0.4	24.07.2019	Alejandro Molina Zarca (UMU)	Section 4
0.5	25.07.2019	Alejandro Molina Zarca (UMU)	Section 5
0.6	26.07.2019	Alejandro Molina Zarca (UMU)	Section 6
0.7	05.09.2019	Alejandro Molina Zarca (UMU)	Sota and conclusions review
0.8	06.09.2019	Alejandro Molina Zarca, Jorge Bernal, Antonio Skarmeta (UMU)	Internal review
0.9	26.09.2019	Miloud Bagaa (AALTO) and Mohammed Boukhalfa	Internal review

1.4 Acronyms and Definitions

Acronym	Meaning
MSPL	Medium-level Security Policy Language
HSPL	High-level Security Policy Language
PDP	Policy Decision Point
PEP	Policy Enforcement Point
SEC	Security Enforcement Manager

CIM	Common Information Model
SPL	Security Policy Language (SPL)
SDL	System Description Language
NSF	Network Security Functions
BMS	Building Management Systems
CPS	Cyber Physical System
CRUD	Create, Read, Update, and Delete
DSPS	Dynamic Security and Privacy Seal
IoT	Internet of Things
MANO	Management and Orchestration
MEC	Mobile (Multi-access) Edge Computing
NFV	Network Function Virtualization
SDN	Software Defined Networking
PSA	Personal Security Application
M2L	Medium to Low

2 State Of The Art

This section presents a state of the art for policy refinement and translation processes, as well as conflict detection and main related technologies.

Common Information Model (CIM) [1] is the main DMTF standard which provides a common definition of management-related information independent of any specification. The model defines concepts for authorization, authentication, delegation, filtering, and obligation policies. However, for an information model to be useful, it has to be mapped into some specification and for this purpose, CIM models are not suitable by themselves, due to the huge number of classes that composes it. xCIM High-level Security Policy Language (SPL) defined in [2], allows to the administrator the definition of security policies using a friendly language, near to the spoken English. It also has an internal format which is a language for formal modelling and low-level abstraction that is oriented to developers. On the other hand, xCIM System Description Language (SDL) is a sub-model that represents the medium level abstraction representation for system description. Meanwhile, xCIM Security Policy Language (SPL) is a sub-model of CIM that represents the medium/low level abstraction representation for security policies. Both in scope of POSITIF [3] and DESEREC [4] European projects. To process the policies, xCIM provides tools such a Policy Console and a Policy Translation Service that allow the definition and refinement of high-level rules. In other words, the translation from the high-level specification to low-level rules specified by a language based CIM-Policy Information Model (i.e. xCIM-SPL or internal format). These kinds of tools reduce the errors and permit additional checks. Due to the lack of information provided by the natural human concepts, the authors use templates to fill the required information, generating finally the final xCIM-SPL result. The tasks Transform and Complete showed in the figure, are deployed by XML Style Sheets (XSL) transformation because all documents (i.e. templates, xCIM-SPL definitions and SPL definition) are represented by XML.

By extending concepts and functionality from xCIM-SPL also present security policies at two levels of abstraction which must be refined/translated before they can be enforced. High-level Security Policy Language (HSPL) and the Medium-level Security Policy Language (MSPL) are two abstractions defined within the European SECURED [5] project to specify security policies based on the capability concept. A capability is the ability to provide a specific security functionality by a security enabler or component. HSPL is though for coarse-graned policies, allowing to define general policies to non-technical users, being independent on the underlaying technologies. On the other hand, MSPL allows to specify information close to the implementation, but still technology independent. Thus, HSPL/MSPL extend and improve the idea exposed on xCIM-SPL/SDL of two levels of device-independent languages, a lower dependent one and the use of capabilities. Regarding the refinement process, the first step consists on identifying the required capabilities of the HSPL policy. Once identified the capabilities, it is necessary to identify the Personal Security Application (PSA). The PSA can be defined as a hardware or software component able to enforce the identified capabilities. If there is not available any component implementing the required functionalities the process will return a non-enforzable analysis, otherwise, it will be performed a translation of an HSPL policy into MSPL policies, also including a service graph indicating which PSA could take care of which MSPL policy. Since MSPL is still device-independent, it must be translated into a specific security configuration for a specific Personal Security Application (PSA). In this case, a coordinator requests a M2L translation to a M2L service in order to translate a MSPL policy to a PSA specific configuration, indicating the PSA id. To support a wide set of low-level security controls, the translation is designed to be multi-device (e.g. netfilter/iptables or PF for a stateful firewall). The proposed approach uses a M2L (Medium-to-Low) plugin repository which contains a set of plugins that implements the translation between MSPL and the specified configuration. The plugins then, can be loaded by the M2L service in order to get the PSA configuration depending on the PSA id specified.

In [9] authors look at a general policy-based architecture in order to simplify several technologies in the context of IP networks. The solution can be considered as an adaptation of the IETF policy framework for

network provisioning, focused on a policy management tool. In the same way as previous cases, this policy management tool supports different levels of abstraction, i.e., business level and technical level, and it is composed by several components. It includes: *i)* the user interface which consists on a command line and graphic tools; *ii)* the resource discovery in order to determine the topology of the network; *iii)* the policy transformation logic component which is responsible to translate the business-level policies into technology-level policies as well as verify the policies are consistent, correct and feasible. Finally, the policy distributor which basically writes the technology-level policies to a repository. In a general way, authors also describe different types of policy representations from if-else semantics to policy schemas. They also provide generic high-level examples regarding conflict resolution as well as policy translations by using eXtensible Stylesheet Language Transformations (XSLT).

A less generic approach can be found at [10] where authors focus on management and translation of filtering security policies. They present a set of techniques in order to perform rules insertion, modification and removal, automatic discovery for rules conflicts, as well as filtering policies translation. Regarding the policy representation, they present a policy tree which represents the filtering rules, starting from the network protocol and ending with the specific action (protocol-src_ip-src_port-dst_ip-dst_port-action). For policy anomalies, authors focus on shadowed, correlation, generalization and redundancy and the basic idea for anomalies discovery is based on to determine if any two rules are in the same path of the policy tree. Regarding policy translation, the proposed solution translates the rules into policy tree paths in order to aggregate common branches and optimize future operations. This is, common values in different rules will follow the same path of the policy tree from the root, so here the policy translation can be seen as to find the best field ordering that provides maximum aggregation of a set of related rules.

Since policy-based network management seems to share the same philosophy about using security policies at different levels of abstraction, [11] claims that there is a lack of tools supporting that strategy, so they provide an ontology to represent the domain knowledge and then perform reasoning to create the network-level security controls. The main objective is to derive configurations for security controls based on ACL and secure channel mechanisms from a fixed set of business policies by using an automatic approach and interacting with the administrator when required. Authors then distinguish between two different kind of controls, these are, OS-level and application-level. In order to test the framework, authors also provide seven high-level security policy instances. These policy translation process then identifies the users and the involved devices and generates the device configurations for each of them.

In general, policy-based frameworks must be enriched with policies analysis in order to detect different types of anomalies depending on the policy domain, and there are several efforts in the literature covering in different ways this topic, from simple or specific analysis up to ontologies and taxonomies definition. In this regard, [14] contributes to the development of a specific IPSec policy management. They defined a high-level security requirement which can be used not only for generating IPSec configurations but also can be used as criteria to detect conflicts. In this domain, authors identify conflicts if the set of IPSec security policies together do not satisfy the security requirements. In a more generic approach, [13] authors analyse the types of overlap which may occur between policies as well as some possible approaches to the prevention. They classify conflicts in conflict of modalities and conflict of goals, and they identify different kind of conflicts and they provide an analysis of when each conflict occurs in the system. Following a similar approach, [14] provides a taxonomy of semantic conflicts and analyses the main features of each of them, also providing modelling for certain realistic scenarios. Our work is based on this effort and it extends the conflicts, also adding dependencies to the analysis.

3 Discussion on Progress Beyond the State of the Art

The deliverable D2.5 showed the beyond the state of art regarding the extended security policy features and the new ones. The implementation of these security models and the security enforcement management process allow us to deploy complex IoT scenarios and validate the new security policy models. Since several previous works are focused on networking policies, the first use cases we contemplated were in that direction. In this way [19] showed the first integration of ANASTACIA components and it provided a comparison between the performance of networking policy enforcement through different security enablers like ONOS and ODL SDN controller but also comparing this new approach with a more traditional one like the enforcement in a virtual router by using NETCONF. Beyond networking security policies, current implementation takes into account different kind of policy models for different purposes. Figure 1 shows the main concept proposed in one of the ANASTACIA results. In this case, the implementation allows us to establish **proactively** networking, authentication and authorization security policies, as well as to distribute crypto keys as part of the process for an IoT domain. The implementation also allows to specify security policies in **reactive** way which in this case will allow the DTLS traffic from the IoT device to the vProxy once the IoT device has been properly authenticated and authorised to put a specific resource in its IoT Broker.

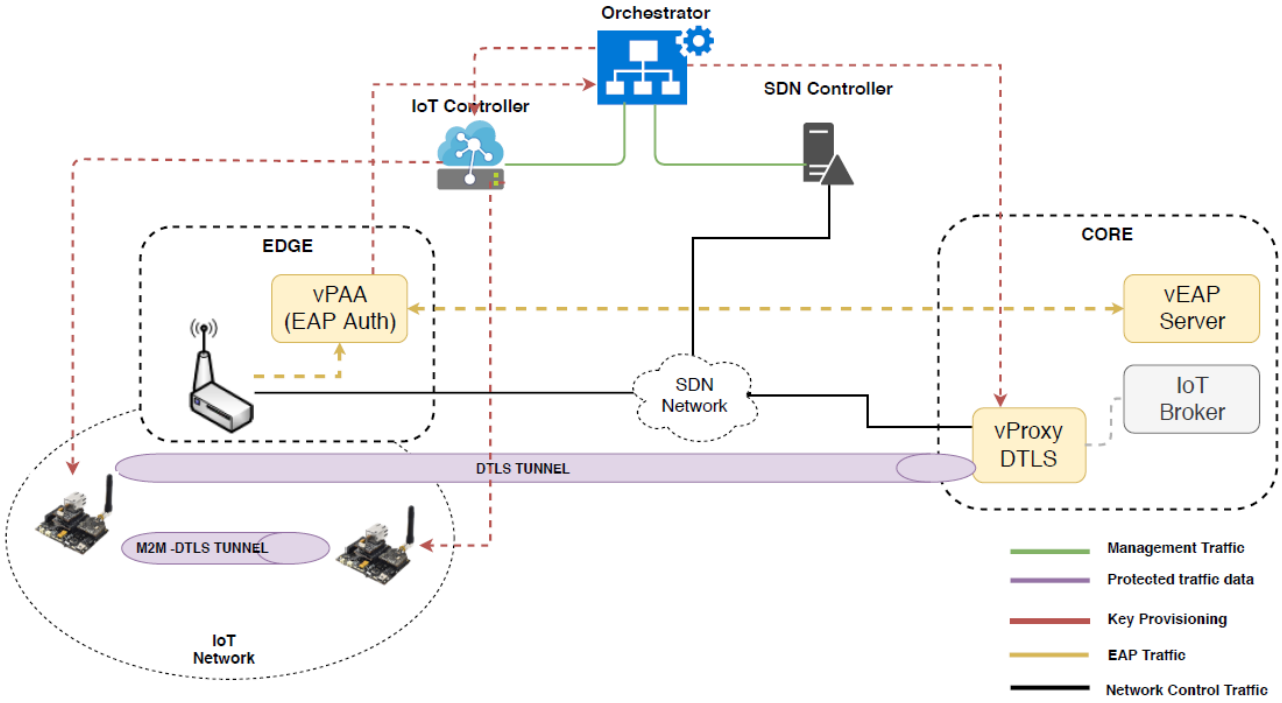

Figure 1: vAAA DTLS application

The previous approach shows networking reactive capabilities as result of authentication and authorization events, in [20] we presented the deployment of a use case where IoT devices in a building have been compromised and they provide abnormal behaviours. Then, IoT management policies are deployed as a reaction of the misbehaviour. The implementation then allows us to translate the reactive security policies as well as to enforce them through the northbound API of the IoT Controller. Considering this scenario, in the paper, we showed different performance metrics as result of the implementation. These are only examples of the ANASTACIA evolution and its application in different scenarios, current implementation allows to define HSPL Orchestration Policies (HSPL-OP) and MSPL Orchestration Policies (MSPL-OP) in proactive and reactive way as well as to enforce them taking into account conflicts, dependencies and different security enablers depending on the nature of the security policy.

4 SECURITY ENFORCEMENT MANAGEMENT DESIGN

The ANASTACIA architecture was already presented in D1.3. It has been evolved and the details about component modifications and their interactions will be presented in D1.5. This section is intended to be focused in those planes and components who provides the security enforcement management.

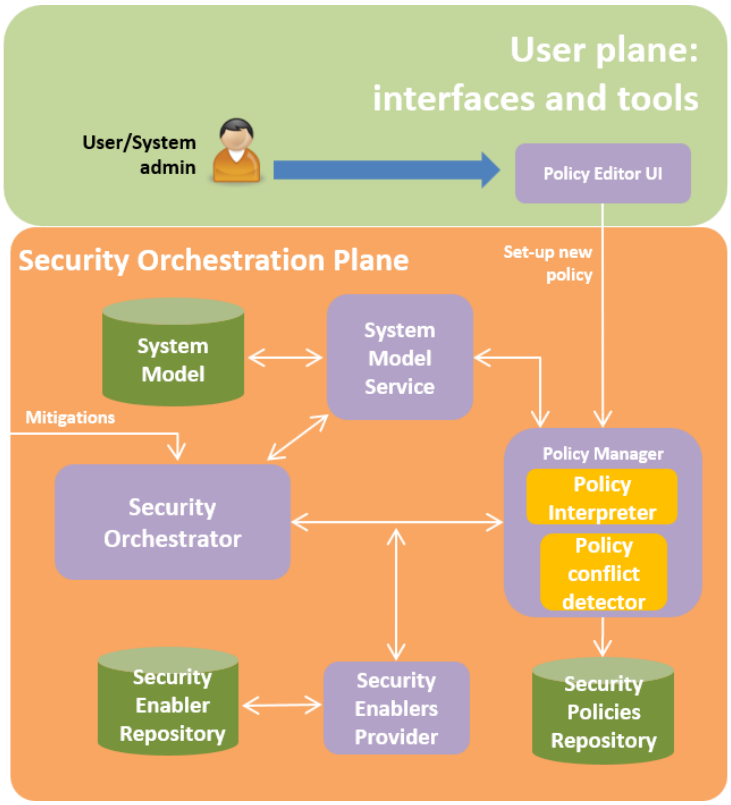

Figure 2: ANASTACIA architecture

Figure 2 shows the a high-level overview of some components which participate in the security enforcement management within the User Plane and the Security Orchestrator Plane of the ANASTACIA framework. The involved planes and components are:

- The **User Plane** provides interfaces, applications and tools that help system administrators to manage the IoT platform through the ANASTACIA framework. For instance, it is possible to configure pro-active security policies in order to enforce some initial requirements by using the **Policy Editor UI**. Other components like DSPS UI and Alerting and reaction dashboard have been omitted from the picture for this section.
- The **Security Orchestrator Plane** is in charge to perform policy refinement, translation and conflict detection processes by using the **Policy Interpreter** and the **Policy Conflict Detector** in order to obtain valid final configurations according on the requirements defined by the administrators or by the mitigation service. This plane is also is in charge to decide the best place to enforce the security policies and enforce them by using the different components of the **Security Orchestrator**, as well as to manage the system model, this is, all the information gathered regarding the underlaying infrastructure. Security Orchestrator and System Model will be detailed on deliverable D3.5.

4.1 MAIN COMPONENTS

This section (extended from D3.1) provides a more detailed description by the components involved on the security enforcement management process. Specifically, they are the Policy Editor Tool (Table 1), the Policy Interpreter (Table 2), the Policy Conflict Detector (Table 3), the Security Enabler Provider (Table 4), the System Model Service (Table 5) and the Security Orchestrator (Table 6). Following it is provided an overview of functionalities, subcomponents, sources and consumers, the activities they are involved and the previously available assets.

Table 1: Policy Editor UI description

Policy Editor UI	
Function	The Policy Editor Tool allows administrators to model orchestration security policies in the High-level Security Policy Language (HSPL) policies extended for the ANASTACIA scope. It also notifies if the new policies will generate conflict or dependencies in the system.
Subcomponent	-
Sources	User/Admin
Consumers	Policy Interpreter and Conflict Detector
ANASTACIA activities involved	Security Policy Set-up
Available assets	UMU Web front-end service implementation

Table 2. Policy Interpreter description

Policy Interpreter	
Function	The Policy Interpreter refines orchestration High-level Security Policy Language (HSPL) policies into orchestration Medium-level Security Policy Language (MSPL) policies. It also translates orchestration MSPL policies into Enablers/VNFs configuration or tasks.
Subcomponent	High to Medium Service (HSPL to MSPL) Medium to Lower Service (MSPL to specific configurations/tasks)
Sources	Policy Editor Tool Orchestrator Security Enabler Provider System Model Service
Consumers	Orchestrator Policy Conflict Detector

ANASTACIA activities involved	Security Policy Set-up Security Orchestration
Available assets	UMU policy Interpreter implementation

Table 3. Conflict detector description

Policy Conflict Detector	
Function	Policy Conflict detector allows to detect conflicts and dependencies in orchestration Medium-level Security Policies like same behaviour conflict, priority dependency conflict, duties conflict, event dependency, managers conflict and override conflict.
Subcomponent	-
Sources	Policy Editor Tool Policy Interpreter Orchestrator System Model Service
Consumers	Policy Editor Tool Policy Interpreter Orchestrator
ANASTACIA activities involved	Security Policy Set-up Security Orchestration
Available assets	UMU policy conflict detector service implementation

Table 4. Security Enabler Provider description

Security Enablers Provider	
Function	The Security Enabler Provider is able to identify the list of security enablers which provide specific security capabilities to meet the security policies requirements. Besides, this component will be endowed with an interface for delivering security M2Lplugins which will be used for the Policy Interpreter in order to perform the M2L translation process.
Subcomponent	-
Sources	Policy Interpreter Security Orchestrator
Consumers	Policy Interpreter

	Security Orchestrator
ANASTACIA activities involved	Security Policy Set-up Security Orchestration
Available assets	UMU/THALES security enabler provider service implementation

Table 5. System Model Service description

System Model Service	
Function	The System Model Service provides all the information regarding the architecture, network topology and current services.
Subcomponent	-
Sources	All components are able to enrich the system model
Consumers	All components are able to request system model information.
ANASTACIA activities involved	Security Policy Set-up Security Orchestration Monitoring Reaction
Available assets	UMU/AALTO system model definition AALTO system model service implementation

Table 6. Security Orchestrator description

Security Orchestrator	
Function	The ANASTACIA Security Orchestrator oversees orchestrating the security enablers according to the defined security policies. To this aim, it is involved in the selection of the best security enablers accounting for their security capabilities, the available resources in the underlying infrastructure, and the policies requirements.
Subcomponent	Subcomponents will be properly detailed in deliverable D3.5
Sources	Interpreter Security Enablers Provider Mitigation Action Service
Consumers	Security Enforcement Plane (Control and Management Domain components) Policy Interpreter Policy Conflict Detector

	Policy Repository
ANASTACIA activities involved	Security Policy Set-up Security Orchestration
Available assets	AALTO security orchestrator implementation

4.1 MAIN INTERFACES

This section describes the main interfaces for components involved in the security enforcement management. In the same way that the previous section, interfaces are exposed in a set of tables which shows the name of the interface, a short description, the component which provides the interface, input and outputs, pre and post conditions and the activities where the interface is involved. Security Orchestrator and System Model Service interfaces has been omitted since they will be explained in deliverable D3.5.

Table 7. Policy Editor User Interface

Policy Editor User Interface		
Description	The Policy Editor User Interface allows defining orchestration HSPL policies by providing different parameters.	
Component providing the interface	Policy Editor Tool	
	Input data	Orchestration HSPL policies including: Priority, Action, Object, Subject, Target, Purpose, Resource and dependencies.
	Output Data	Orchestration HSPL policy Orchestration MSPL policy and conflict detection and dependencies notification.
Consumer components	User/Admin	
Pre-conditions	System Model must contain the high-level information in order to allow user select high-level terms, e.g., IoT-device-1 Policy Interpreter and Policy Conflict Detector must be up and running in order to perform the policy refinement and conflict detection. Security Policies repository must be deployed in order to maintain a registry of policy status. Security Enabler Provider must be deployed in order to verify if there is any enabler capable to enforce the policy requirements (capability).	
Post-conditions	-	
ANASTACIA	Security Policy Set-up	

activities involved	

Table 8. Policy Interpreter H2M Interface

High to Medium interface (H2MI)		
Description	The interface allows to request the policy refinement process from orchestration High-level Security Policy language (HSPL) policies to orchestration Medium-level Security Policy language (MSPL) policies.	
Component providing the interface	Policy Interpreter	
	Input data	Orchestration HSPL policy
	Output Data	Orchestration MSPL policy
Consumer components	Policy Editor Tool/User/Admin	
Pre-conditions	Orchestration HSPL policy has been previously defined. Security Enabler Provider must be deployed in order to verify if there is any enabler capable to enforce the policy requirements (capability). System Model service must be deployed in order to refine high-level terms, e.g., `device:address`. Security Policies repository has been deployed in order to maintain a registry of policy status.	
Post-conditions	-	
ANASTACIA activities involved	Security Policy Set-up	

Table 9. Policy Interpreter M2L

Medium to Lower interface (M2LI)		
Description	The interface allows to request a policy refinement from orchestration Medium-level Security Policy Language (MSPL) policies to specific enabler configurations.	
Component providing the interface	Policy Interpreter	
	Input Data	Orchestration MSPL policy
	Output Data	Security Enabler configurations
Consumer components	Policy Editor Tool Security Orchestrator	
Pre-conditions	Orchestration MSPL policy has been previously defined.	

	System Model service must be deployed in order to security orchestrator is able to decide the best security enabler. Security Enabler Provider has been deployed in order to security orchestrator is able to obtain the best security enabler plugin. Security Policies repository has been deployed in order to maintain a registry of policy status.
Post-conditions	-
ANASTACIA activities involved	Security Policy Set-up Security Orchestration

Table 10. Policy Conflict Detector Interface

Medium Conflict Detection Interface (MCDTI)		
Description	The interface allows to request a medium-level policy conflict and dependencies detection.	
Component providing the interface	Conflict Detector	
	Input Data	Orchestration MSPL policy
	Output Data	Orchestration MSPL policy conflicts and dependencies.
Consumer components	Policy Editor Tool Security Orchestrator	
Pre-conditions	Orchestration MSPL policy has been previously defined. System Model service must be deployed in order to retrieve information about the current deployments. Security Policies repository has been deployed in order to retrieve information about the current security policies.	
Post-conditions	-	
ANASTACIA activities involved	Security Policy Set-up Security Orchestration	

Table 11. Policy Repository Interface

Policy Repository Interface	
Description	The interface allows to store in the policy repository the correspondence among orchestration HSPL and MSPL policies, as well as MSPL with security enabler configurations and the current enforcement status. It also allows retrieving policy templates as well as the stored security policies information.

Component providing the interface	Policy Repository	
	Input Data	HSPL, MSPL | MSPL, Conf | MSPL, status
	Output Data	Acknowledgement | security policies information
Consumer components	Policy Interpreter Security Orchestrator	
Pre-conditions	HSPL, MSPL | MSPL, Conf | MSPL, status must be properly defined	
Post-conditions	-	
ANASTACIA activities involved	Security Policy Set-up Security Orchestration	

Table 12. Security Enabler Provider Interface

Security Enabler Provider Interface		
Description	The interface allows requesting the available security enablers capable to enforce the orchestration MSPL policy. The interface also allows to request a specific plugin for a security enabler.	
Component providing the interface	Security Enabler Provider	
	Input Data	List of capabilities | security enabler ID
	Output Data	List of candidate security enablers | Security enabler plugin
Consumer components	Policy Interpreter Security Orchestrator	
Pre-conditions	Security Enabler Provider has been properly configured with a correspondence between capabilities and security enablers.	
Post-conditions	-	
ANASTACIA activities involved	Security Policy Set-up Security Orchestration	

5 SECURITY ENFORCEMENT MANAGEMENT PROCESSES

The security enforcement management is composed by different processes in order to manage and enforce the security policies defined by the administrator or by the Mitigation Action Service (MAS). If the orchestration security policies are defined at high-level (by the admin) it is required a policy refinement process in order to transform the high-level security policies into medium-level security policies. If the orchestration security policies are provided at medium-level (by the MAS or by the administrator) it is necessary to perform a policy translation process in order to translate the orchestration MSPL policies into final security enabler configurations. Since the new security policies could generate conflicts or dependencies intra and inter policies it is also required to perform a policy conflict and dependencies detection. Once has been determined that the security policies are suitable to be deployed, there is an orchestration process in order to enforce the security policies along the security enforcement plane. This last process has been omitted in this section since it will be detailed in deliverable D3.5.

5.1 POLICY REFINEMENT AND TRANSLATION PROCESSES

ANASTACIA framework extends and improves the HSPL and MSPL languages proposed in SECURED-FP7 European project [6].The first results of this extension and improvements were defined in ANASTACIA deliverable D2.1 [7]. That work was evolved up to its current status which was provided in ANASTACIA deliverable D2.5 [8]. The present section shows the final version of the policy refinement and policy translation processes.

5.1.1 HSPL to MSPL Refinement

In the **Policy set-up activity,** the security administrator is able to define orchestration security policies at two different levels, these are, High-level Security Policy Language (HSPL) and Medium-level Security Policy Language (MSPL) according to the level of abstraction that the administrator prefers. If the administrator decides to model HSPL orchestration policies, he/she can do it through the policy editor UI. Figure 3 shows the refinement process once the security administrator has defined the high-level orchestration security policy. The process is composed by the following points:

1. The security administrator defines an orchestration HSPL policy by using the Policy Editor UI in the Policy Editor Tool. For simplicity in the figure it is named as HSPL-OP (**H**igh-level **S**ecurity **P**olicy **L**anguage **O**rchestration **P**olicy).
2. The Policy Editor Tool requests the HSPL-OP refinement to the Policy interpreter.

Steps 3 to 12 are performed for each HSPL in the HSPL-OP.

3. The Policy Interpreter identifies the main capability that will be necessary in order to enforce the security policy in the system.
4. The Policy Interpreter sends the main identified capabilities to the Security Enablers Provider.
5. The Security Enablers Provider request to the Security Enablers Repository those security enablers which implement the required capabilities.
6. The repository returns the requested security enablers.
7. The Policy Interpreter receives the list of the security enablers candidates.
8. The Policy Interpreter verifies there is at least one security enabler per security policy in the policy for orchestration able to enforce the required capability.
9. If there is not a security enabler able to enforce the required capabilities the Policy Interpreter will notify the error.
10. Otherwise, the Policy Interpreter retrieves system model information in order to perform the policy refinement, e.g., ip addresses, ports, protocols…

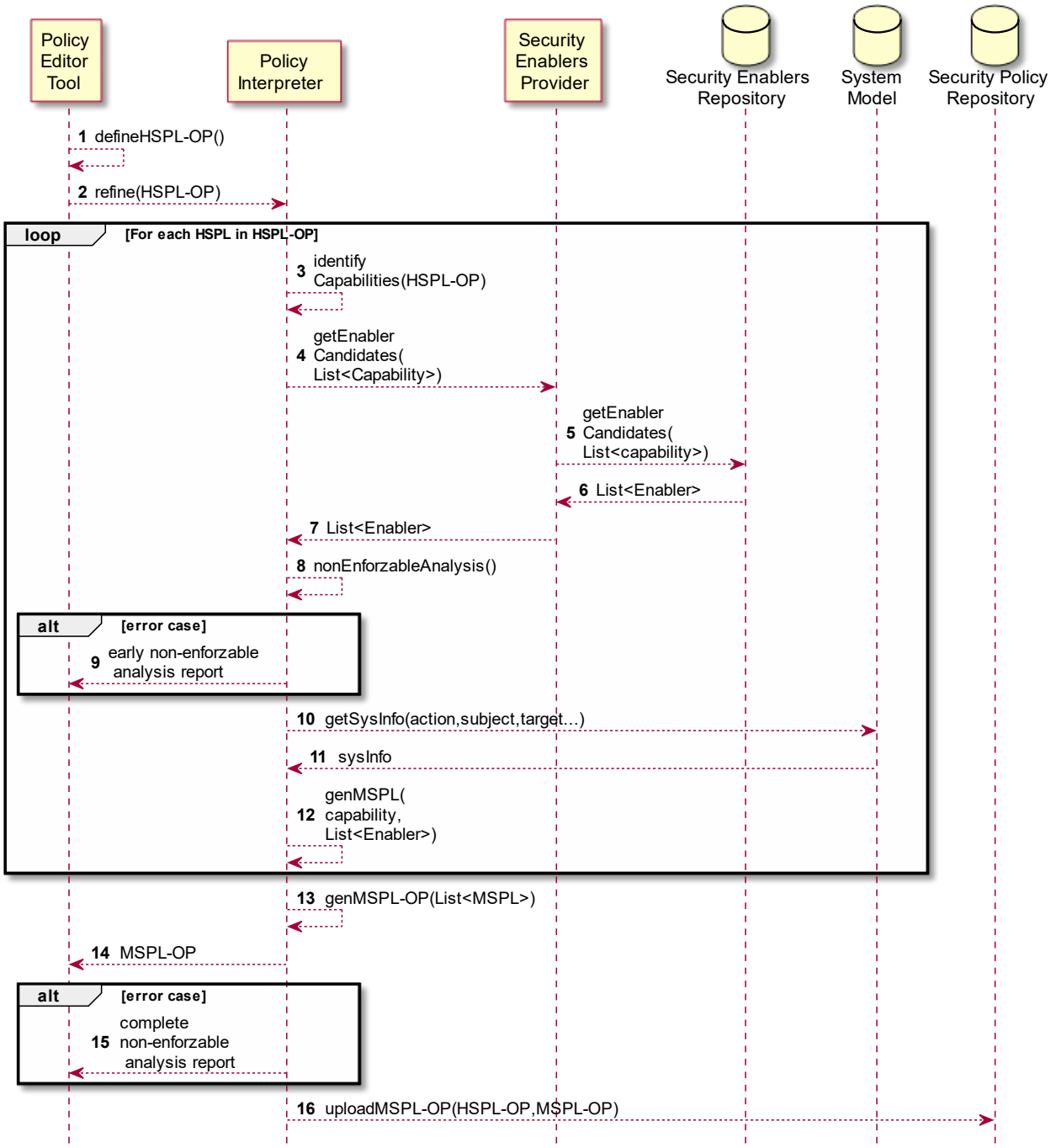

Figure 3: H2M Process

11. The Policy Interpreter receives the system model information.
12. The Policy Interpreter generates a MSPL policy by using the system model information, also adding the HSPL dependencies to the MSPL.
13. The Policy interpreter generates a MSPL-OP by gathering the MSPL list.
14. If there are no errors in the refinement process Policy Editor Tool receives the MSPL-OP.
15. Otherwise Policy Interpreter will notify the refinement errors.
16. HSPLOP and the correspondent MSPL-OP are uploaded to the Policy Repository in order to register the refinement result.

5.1.2 MSPL to Low-level Enforcement

Once an MSPL orchestration policy has been obtained, it must be translated into a set of final configurations for specific security enablers. Figure 4 shows the process in order to translate and enforce orchestration MSPL security policies without taking into account the conflict detection which will be explained in next section. The process is composed by the following points:

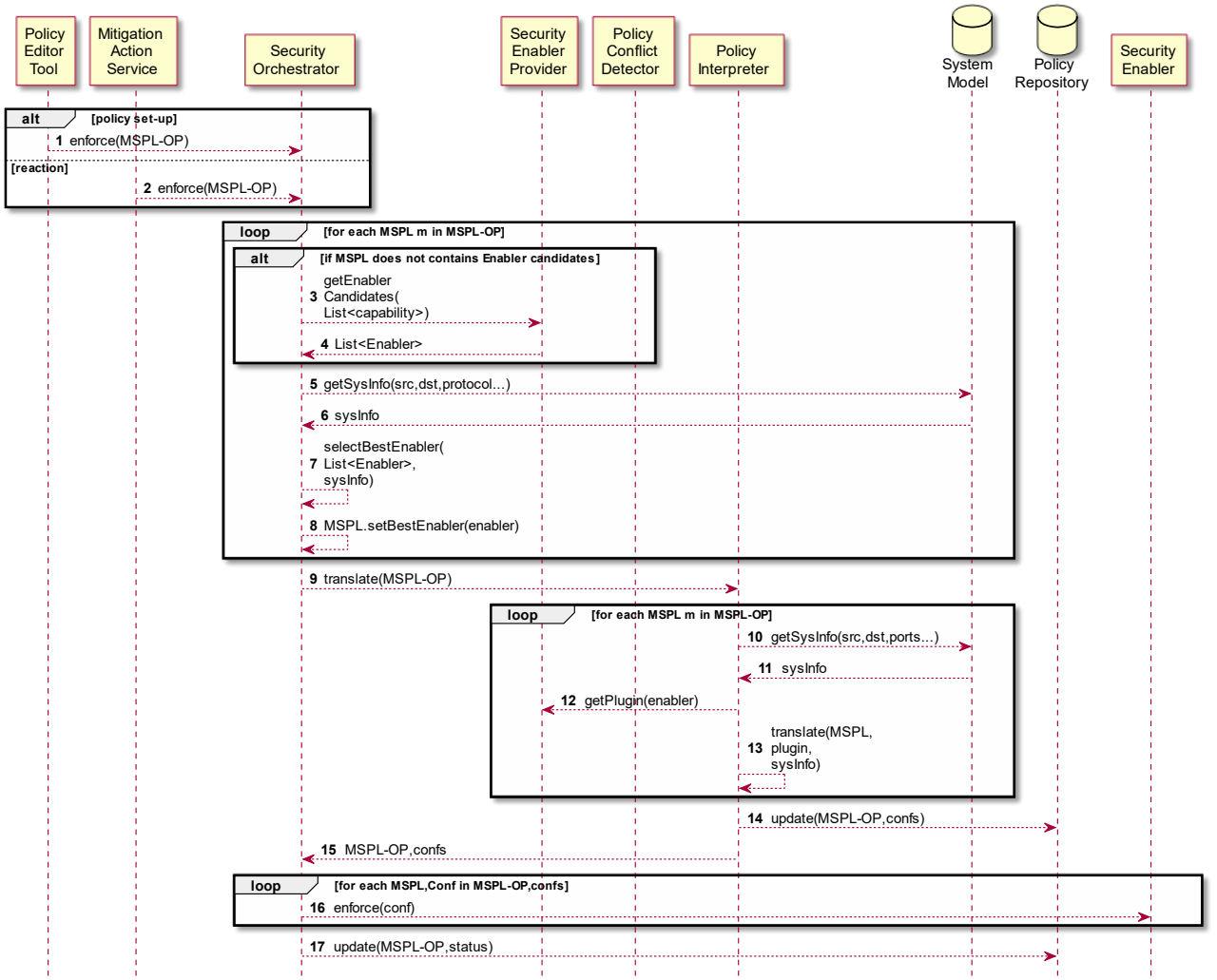

Figure 4: M2L process

1. If the MSPL-OP is generated as part of the Policy set-up activity, it will be provided by the Policy Editor Tool.
2. If it is generated as part of a reaction process it will be provided by the Mitigation Action Service.

Steps 3 to 8 are performed for each MSPL in the MSPL-OP.

3. If the MSPL does not contain security enabler candidates, the Security Orchestrator will request them to the Security Enablers Provider.
4. The Security Enablers Provider provides the security enablers which could enforce the security policy.
5. The Security Orchestrator request to the System Model the available information of the underlying technologies for the entities involved in the security policy.
6. The Security Orchestrator receives the requested information.
7. The Security Orchestrator uses the system model information and the available security enabler candidates to decide which one will be the best security enabler in order to enforce the security policy.

8. The Security Orchestrator updates the MSPL by replacing the security enabler candidates by the selected one.
9. The Security Orchestrator requests the MSPL-OP translation to the Policy Interpreter.

Steps 10 to 13 are performed for each MSPL in the MSPL-OP.

10. Policy Interpreter requests system model information in order to perform the translations.
11. Policy Interpreter receives the system model information.
12. Policy Interpreter retrieves the specific plugin for the selected security enabler.
13. Policy Interpreter performs the policy translation by using the specific security enabler plugin.
14. Policy Interpreter upload the correspondence between the MSPL-OP and the generated configurations.
15. Policy Interpreter returns the final configurations as well as the conflicts and dependencies to the Security Orchestrator.

Steps 16 is performed for each MSPL in the MSPL-OP.

16. MSPL is enforced according on their priority.
17. Security Orchestrator updates the policy status into the Policy Repository.

5.2 Policy Conflict and Dependencies detection Process

Conflict detection and dependencies enforcement process extends the MSPL to low-level enforcement process by adding policy conflict and dependencies analysis both inter and intra policies level.

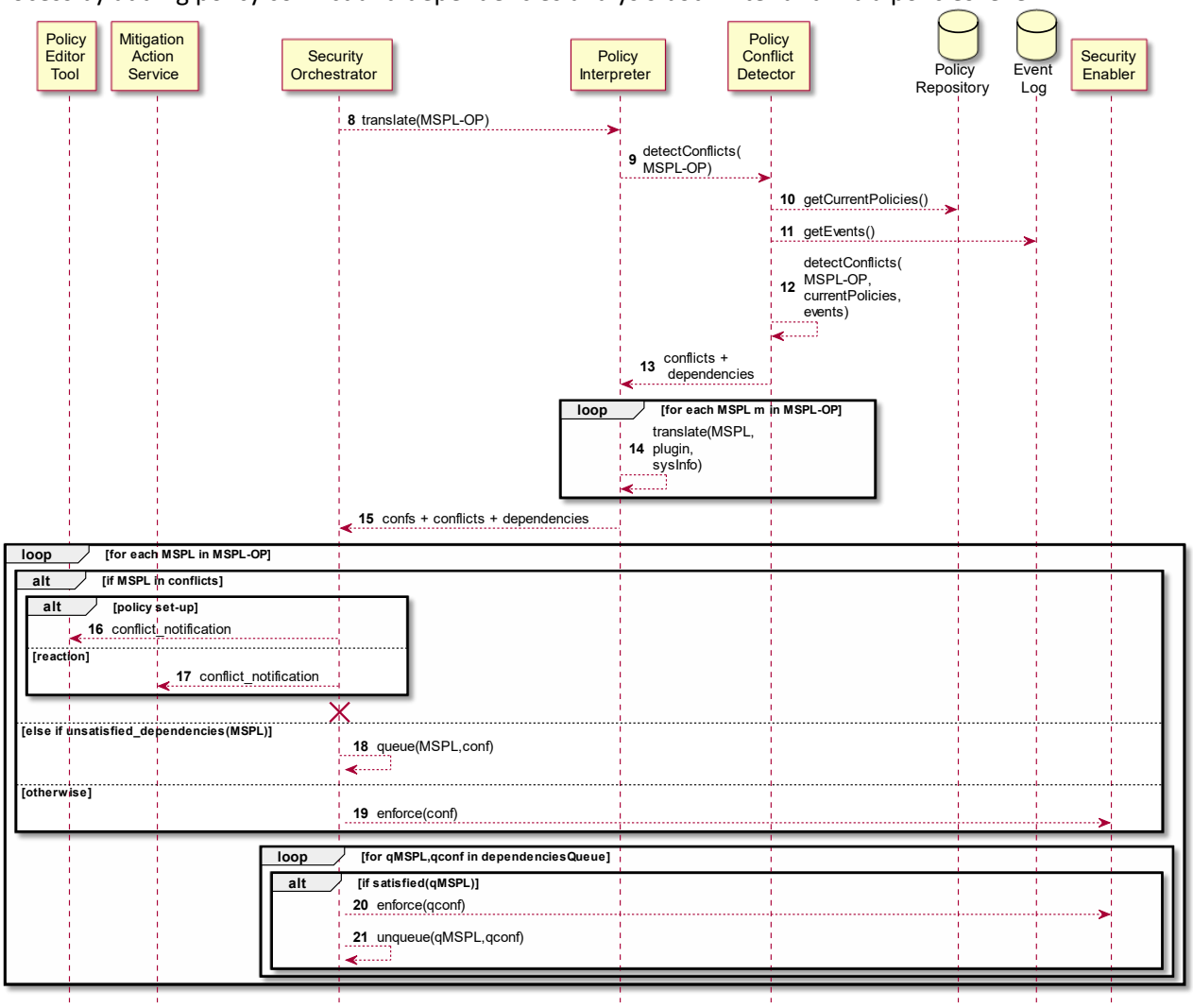

Figure 5: Conflict detection and dependencies enforcement process

Figure 5 starts at point 8 since points 1 to 7 are the same than in the precious case. Points 9 onward are explained below:

9. The Policy Interpreter requests the policy conflict and dependencies detection to the Policy Conflict Detector.
10. Policy Conflict Detector retrieves from the Policy Repository the security policies that are already deployed in the system.
11. Policy Conflict Detector retrieves from the Event Log (log database) the events related to the security policies.
12. Policy Conflict Detector performs the conflict and dependency detection by analysing different kind of conflicts inter and intra security policies.
13. Policy Conflict Detector returns policy conflicts and dependencies.
14. Policy Interpreter performs the MSPL-OP translation. This point has been simplified in this figure for simplicity.
15. Policy Interpreter returns the correspondence between MSPL-OP and configurations as well as its conflicts and dependencies.

Steps 16 to 24 are performed for each MSPL in the MSPL-OP.

16. If the MSPL policy presents some kind of conflict and the enforcement process started as part of the policy set-up process the Security Orchestrator notifies the issue to the Policy Editor Tool.
17. If the MSPL policy presents some kind of conflict and the enforcement process started as part of the reaction process the Security Orchestrator notifies the issue to the Mitigation Action Service.
18. If the MSPL policy presents an unsatisfied dependency it must be queued.
19. Otherwise the MSPL is enforced according on their priority.
20. The MSPL queue is verified
21. If the queued MSPL dependencies have been satisfied it is enforced.

Once the different processes involved in the security enforcement management have been explained, netx section provides a more detailed information regarding the implementation of the components and the interfaces. Regarding the Security Orchestrator implementation, this document provides a brief summary since it will be described in the deliverable D3.5.

6 Final Policy-Based Security Enforcement Management Implementation

This section provides details about the implementation of those components involved in the policy-based security enforcement management. They are the Policy Editor Tool, the Policy Interpreter, the Policy Conflict and Dependencies Detector, the Security Enablers Provider and the Security Orchestrator.

6.1 Policy Editor Tool Implementation

In order to ease the policy set-up activity, it has been developed from scratch the Policy Editor Tool which provides the Policy Editor UI. The Editor Tool has been developed in Django Python framework and the service has been dockerised to boost a quick the deployment process, and increase the flexibility and scalability. Figure 6 shows the interactions and technologies in order to allow the administrator to model HSPL-OP.

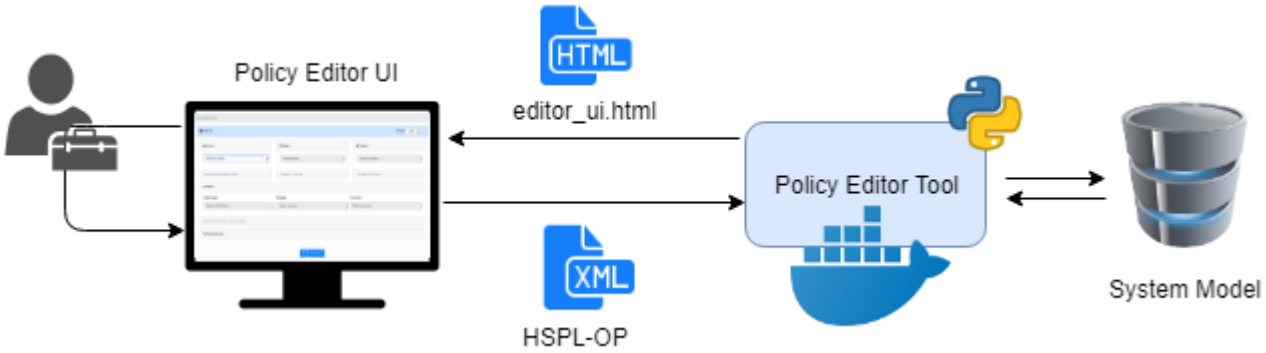

Figure 6: Policy Editor Tool HSPL-OP Modeling

The friendly GUI has been developed as a Django web application. In this way, the administrator is able to model HSPL-OP by filling the basic fields of high-level security policies. Figure 7 shows the HSPL-OP form. This form allows to compose different HSPL policies in a HSPL-OP. Each HSPL policy is composed by the *action* to be performed over an *object* for a specific *subject*, *target*, *purpose* and *resource*. The content of these fields is retrieved dynamically from the system model according on the values selected.

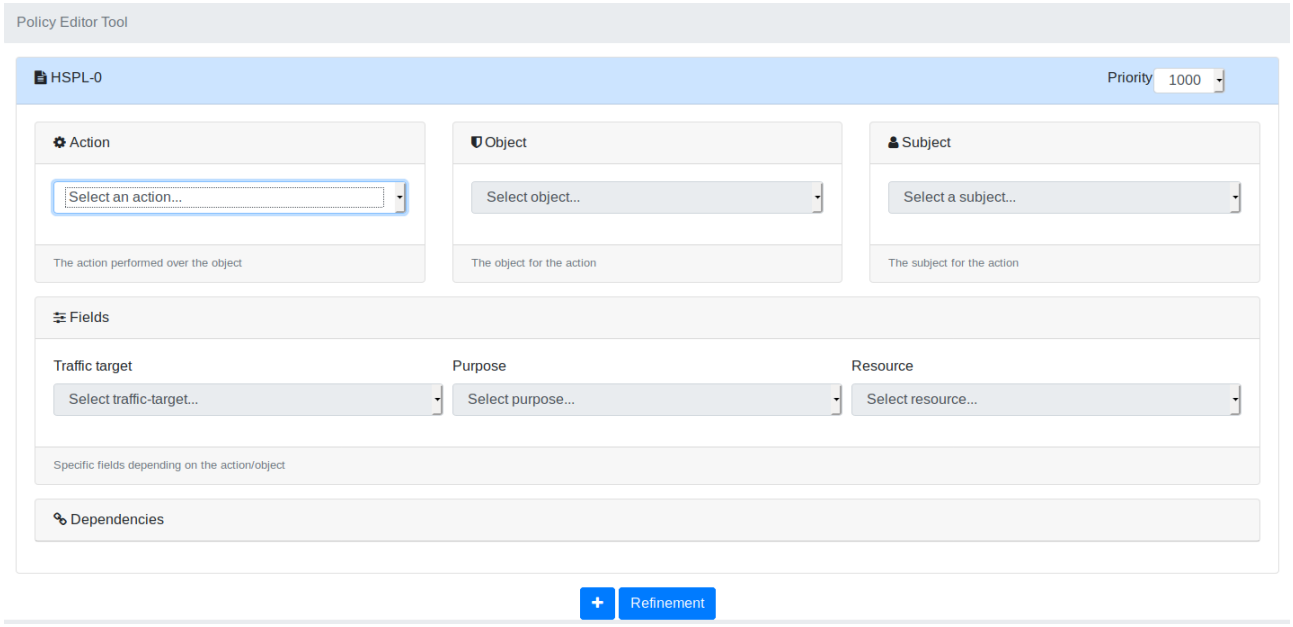

Figure 7: HSPL-OP Editor UI

New HSPL policies can be added by clicking the "+" blue button. In the same way, dependencies can be added by opening the dependencies section. Figure 8 shows the dependencies section which allows the administrator to introduce new dependencies by including the dependency type and the target of the dependency.

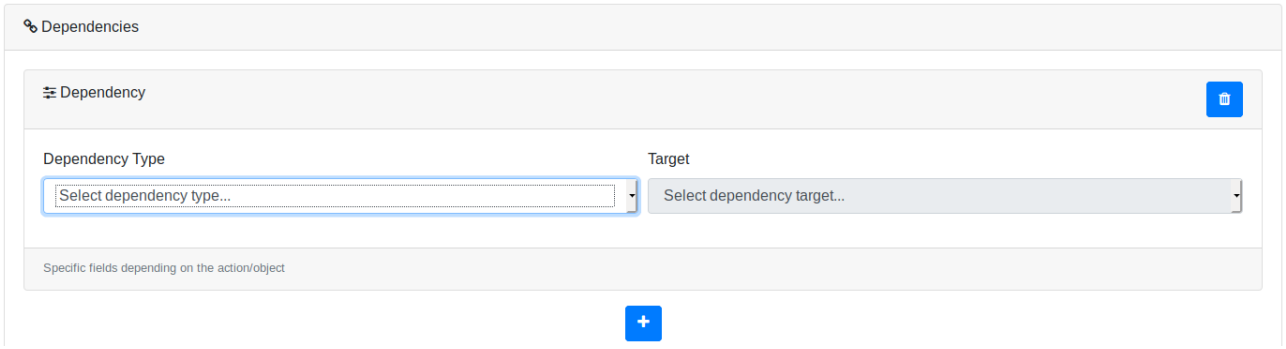

Figure 8: HSPL-OP Editor UI dependencies

Once the administrator has built the HSPL-OP, he can send the obtained results to the Policy Editor Tool by clicking in the "Refinement" button. This button will request the policy refinement by sending the XML HSPL-OP file to the Policy Editor Tool, which in its turn will request the refinement process to the Policy Interpreter. Figure 9 shows the interactions between the administrator and the Policy Editor UI and Tool once the first has requested the HSPL-OP refinement.

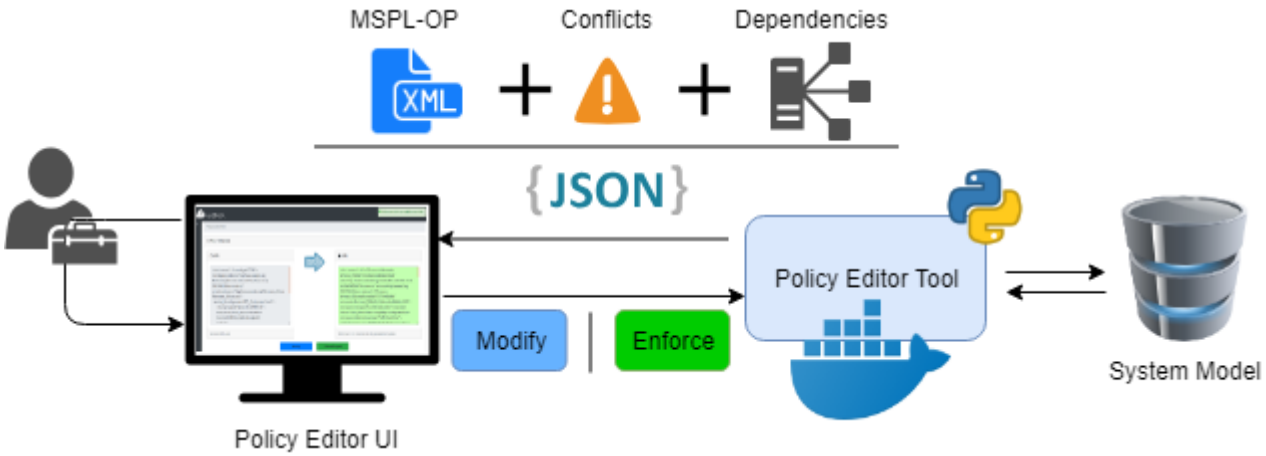

Figure 9: Policy Editor Tool HSPL-OP Refinement

As it is shown in the figure, once the policy refinement has been performed the Policy Editor Tool also requests a policy conflict detection in order to also notify possible dependencies and conflicts. In this way the Policy Editor Tool returns a JSON file that includes not only the XML MSPL-OP refinement but also two JSON lists with the conflicts and dependencies which has been computed by the Policy Conflict and Dependencies Detector. If there are no conflicts and dependencies detected the Policy Editor UI will show the refinement as successful. Figure 10 shows an example of a successful refinement process.

Figure 10: Policy Editor UI Successful Refinement

Otherwise, the Policy Editor UI will notify to the administrator the kind of dependencies or conflicts and the security policies that are generating this behaviour. Figure 11 shows two different pictures. The first one corresponds to a dependencies detection whereas the second one corresponds to a conflict detection.

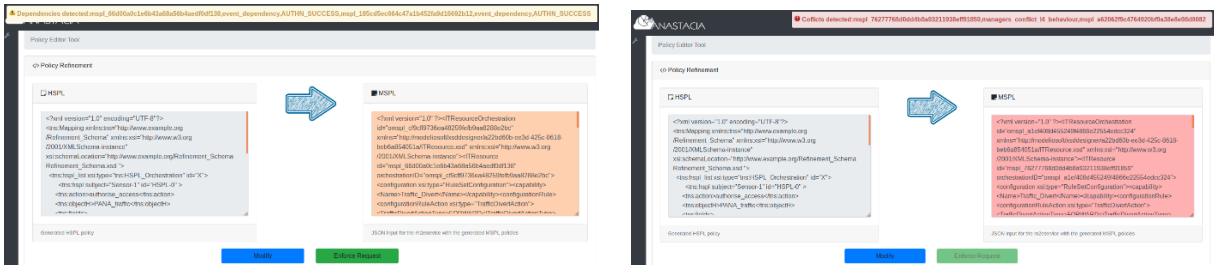

Figure 11: Dependency and Conflict detection

The main differences between them, apart from the colour of the notification, is that in the second case the "Enforcement" button is disabled. This occurs due a pendency could be solved dynamically by the Security Orchestrator at enforcement time, but the administrator must ensure that the new security policies he/she models will not generate conflicts between them and between the ones that are already deployed in the system.

6.2 POLICY INTERPRETER IMPLEMENTATION

Policy interpreter services have been implemented from scratch by using Falcon framework. Falcon is a Python REST API framework which provides fast, reliable, extensible and compatible development properties. Since we have two different levels of security policies, we have developed two different services in order to provide the H2M policy refinement and the M2L policy translation.

6.2.1 H2MService

The service who provides the High-level to Medium-level security policies refinement is denominated *H2MService*. This service has been implemented as a class inside the Falcon framework and it is able to process POST requests to the endpoint */h2mservice*. Other methods have been overridden in order to provide an exception indicating that only POST method is allowed.

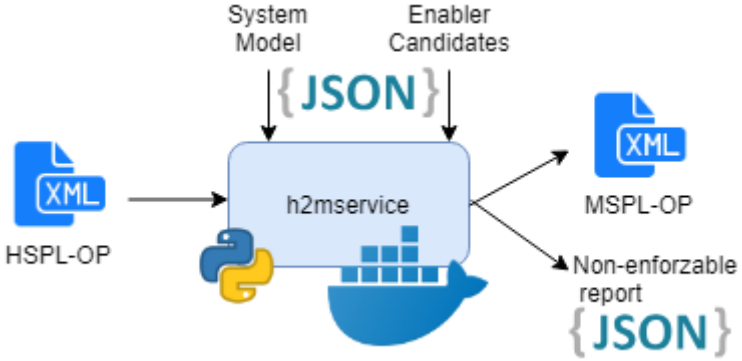

Figure 12: h2mService implementation

Figure 12 shows the main technologies involved in the implementation of the h2mservice. The python service is dockerised in the same way of the policy editor tool in order to obtain the same benefits. Once it receives the POST request, it verifies that the body contains the XML HSPL-OP and then it creates a *H2MRefiner* object and it invokes the *get_mspl* method in order to perform the policy refinement. To parser the XML HSPL-OP it has been used the pyxb tool. This tool allows to generate python classes for each XML element defined in an XML schema which in this case corresponds with the HSPL schema defined in the deliverable D2.5. By using this tool, the refinement module extends the behaviours of the classes generated by pyxb in order to add refinement methods like the *get_mspl* one. Taking this into account the *get_mspl* method in the H2MRefiner object only needs to parse the XML HSPL-OP and invoke the *get_mspl*

method of the HSPL-OP object. This method generates a new random ID for the HSPL-OP in case it has not been already provided, then it builds a skeleton of an empty MSPL-OP which also generates a random ID. For each HSPL in the HSPL-OP it is also verified that a HSPL ID has been provided, otherwise a new one is generated and the HSPL orchestration id is filled with the one of the HSPL-OP. Then *to_mspl* method implemented in the HSPL object is invoked. This method is the one in charge to refine HSPL policies in one or more MSPL policies depending on the implementation criteria. In current implementation some security policies will generate two MSPL security policies to represent the bi-directional condition. Once the HSPL refinement method is invoked it verifies that action and object defined in the HSPL policy are valid, this is, they are previously defined in the schema. The process then retrieves system model information (e.g., ip address, port, available channel protection) for all the high-level values involved in the HSPL policy. In order to determine the capability, it performs a match between the action and the object against a capability mapping, for instance, the action "*authorise_access*" and the object "*AllTraffic*" will generate a *Traffic_Divert* capability. Once determined the involved capabilities, the security enablers candidates are requested. If there are not security enabler candidates available, the process will finish at this point and a refinement error will be generated. Otherwise, a method *get_X_Y_mspl* will be invoked where X corresponds to the action and Y corresponds to the kind of object, e.g. *get_authorise_access_traffic_mspl*. In this way we are able to process independently each combination of actions and objects, but we can also to abstract common methods in order to be used for different combinations. Independently of the combination, these kinds of methods follow a common structure. First, an empty base skeleton for the MSPL is defined by creating the main MSPL objects (the MSPL python classes have been also generated by using the pyxb tool), also replicating the HSPL dependencies into MSPL dependencies. Once the base MSPL has been generated, the method performs the specific refinement by using specific data previously retrieved from the system model, filling then the actions and conditions. For example, in order to authorise access to some kind of traffic, the traffic divert action is set to *"FORWARD"* , the forward action fills fields like destination address and interface of the HSPL traffic target and the forward condition establish as source address and source interface those retrieved for the subject, as well as destination port and protocol those retrieved for the object. For example, if we model that SensorA is authorized to access authentication traffic against the authentication agent, the traffic from SensorA IP with authentication traffic destination port and authentication traffic protocol must be redirected against the authentication agent IP address. When the HSPL policy has been refined, the result is included in the MSPL-OP. Finally, when the whole HSPL-OP has been translated in a MSPL-OP which contains all the refinements it is stored in the Policy Repository and it is returned to the main method of the *H2MService* who returns a HTTP 200 OK which includes the resultant XML MSPL-OP.

6.2.2 M2LService

The service who provides the Medium-level to Low-level security policies translation is denominated M2L*Service*. This service has been implemented as a class inside the Falcon framework and it is able to process POST requests to the endpoint */m2lservice*. Other methods have been overridden in order to provide an exception indicating that only POST method is allowed.

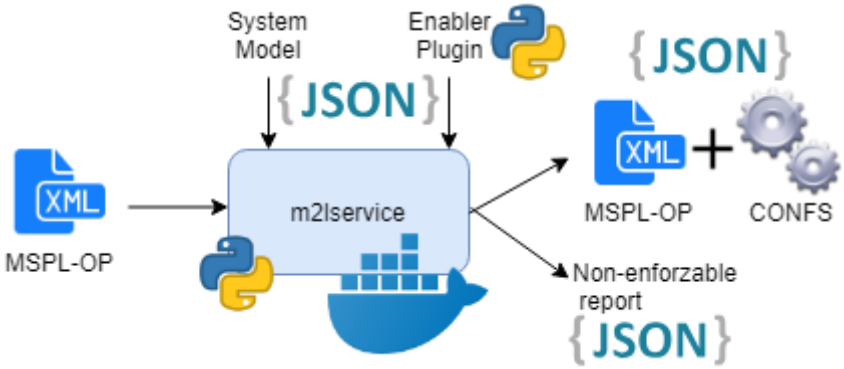

Figure 13: m2lservice implementation

Figure 13 shows the main technologies involved in the implementation of the m2lservice. The python service is dockerised in the same way of the h2mservice in order to obtain the same benefits. Once it receives the MSPL-OP by a POST request the service requests a dependencies and conflict detection to the conflict detection service which will be explained later (it has been omitted of the figure for simplicity). Conflict and dependences are then stored, and the policy translation begins. In this case a *MSPLTranslation* object is created and the *translate* method is invoked by passing as parameter the MSPL-OP. In the same way of the refinement module, the translator module also implements a customisation of the main MSPL schema elements in order to provide translation methods. This is, one the MSPL-OP has been loaded as python object we can directly invoke the translate method. This method loops over each MSPL policy in the MSPL-OP and invokes the translated method for each of them. The MSPL translate method gets the security enabler name from the MSPL and downloads the translator plugin from the Security Enabler Provider and stores it in the temporal *m2l_plugins* folder. Once the plugin has been downloaded, it is imported dynamically to the python code as a *M2LPlugin* object. Each security enabler plugin implements the M2LPlugin class which contains the *get_configuration* method. This method receives a MSPL policy and returns the specific configuration for the specific security enabler who is implemented by the plugin. The translation process is similar to the refinement one. The main MSPL element, *ITResourceType*, is customised in order to extend the default behaviour of the pyxb tool in order to add the *get_configuration* method. Since the same enabler could enforce different capabilities, this method identifies the capability of the received MSPL policy, and it invokes the *get_Z_configuration* where Z is the capability. If the identified capability is not implemented the process will return a non-eforzable notification. Otherwise the method is invoked and depends on the capability the MSPL translation is addressed in different way. This is, for the DTLS_protocol capability it is required to set as data protection action the DTLS security parameters whereas for power management is enough to specify the power management action. This different behaviour depending on the capability is implemented by adding and customising the *get_configuration* method for each element involved in the capability (capability, actions and conditions). Finally, when each MSPL has been translated, the result is provided in JSON format.


```
{
   {
     "translations": {
        "omspl_translation": {
          "mspl_id": "mspl_9f1a88b4fc67421b98de270d5a63d35f",
          "mspl": "<ITResourceOrchestration>…</ITResourceOrchestration>”,
          "mspl_translations": [{
            "mspl_id": "mspl_9f1a88b4fc67421b98de270d5a63d35a",
            "mspl": "<ITResource>…</ITResource >”,
            "enabler": "onos_nb",
            "enabler_conf": "{\"priority\": 60000,\"tableId\": 0,…}”,
          }, {
            . . .
          }],
        }
   }
}
```


Figure 14: m2lservice output example

Figure 14 shows an example of output for the m2lservice. As it is shown, the result is composed by the MSPL-OP ID and its plain text as well as the ID, plain text, enabler and enabler configuration for each MSPL part of the MSPL-OP.

6.3 Policy Conflicts and Dependencies Detector Implementation

The service who provides the Medium-level conflict and dependencies detection is denominated *MCDTService*. This service has been implemented as a class inside the Falcon framework and it is able to process POST requests to the endpoint */mcdtservice*. Other methods have been overridden in order to provide an exception indicating that only POST method is allowed.

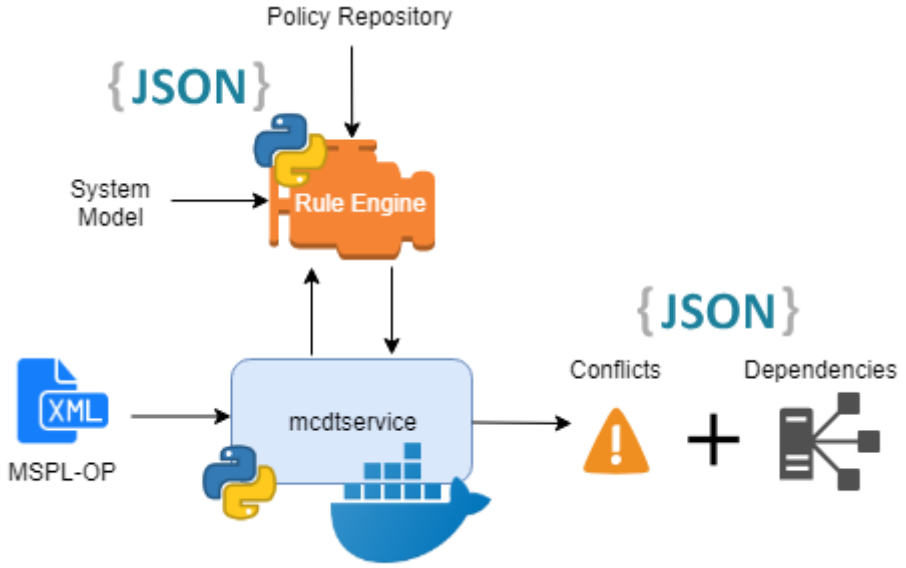

Figure 15: mcdtservice implementation

Figure 15 shows the main technologies involved in the implementation of the mcdtservice. The python service is dockerised in the same way of the m2lservice in order to obtain the same benefits. Once it receives the MSPL-OP by the POST method the MSPL-OP is loaded into a python object in the same way as the previous services, then a *MSPLConflictDetector* object which implements the *detect* method. This method receives a MSPL-OP object and uses it as source of the rule engine. As rule engine it is used Pyke. Pyke introduces a form of logic programming inspired in Prolog to the Python community by providing a knowledge-based inference engine.

```
verify_same_filtering_l4_behaviour
 foreach
  mspls.mspl($mspl1, $status1)
  mspls.mspl($mspl2, $status2)
  check $mspl1.id != $mspl2.id
  check $mspl1.configuration.capability.Name ==
    $mspl2.configuration.capability.Name == "Filtering_L4"
  check $mspl1.configuration.configurationRule.configurationRuleAction.FilteringActionType ==
    $mspl2.configuration.configurationRule.configurationRuleAction.FilteringActionType
  check
$mspl1.configuration.configurationRule.configurationCondition.packetFilterCondition.SourceAddress
  ==
$mspl2.configuration.configurationRule.configurationCondition.packetFilterCondition.SourceAddress
  check
$mspl1.configuration.configurationRule.configurationCondition.packetFilterCondition.DestinationAd
dress
  ==
$mspl2.configuration.configurationRule.configurationCondition.packetFilterCondition.DestinationAd
dress
  check
$mspl1.configuration.configurationRule.configurationCondition.packetFilterCondition.SourcePort ==
$mspl2.configuration.configurationRule.configurationCondition.packetFilterCondition.SourcePort
  check
$mspl1.configuration.configurationRule.configurationCondition.packetFilterCondition.DestinationPo
rt
 ==
$mspl2.configuration.configurationRule.configurationCondition.packetFilterCondition.DestinationPo
rt
  check
$mspl1.configuration.configurationRule.configurationCondition.packetFilterCondition.Interface ==
$mspl2.configuration.configurationRule.configurationCondition.packetFilterCondition.Interface
  check
$mspl1.configuration.configurationRule.configurationCondition.packetFilterCondition.ProtocolType
==
$mspl2.configuration.configurationRule.configurationCondition.packetFilterCondition.ProtocolType

assert
  mspls.mspl_conflict($mspl1,same_behaviour_filtering_l4_conflict,$mspl2)}
```

Figure 16: MSPL Pyke rule example

Figure 16 shows an example of rule engine in order to detect if two MSPL policies shares the same kind of behaviour for the Filtering_L4 capability. Currently the knowledge rule base is composed by the following examples:

- Redundancy ID conflicts
- Redundancy conflict by behaviour (Filtering L4 example)
- Contradiction or Managers conflict (Filtering L4 and Traffic divert examples)
- Conflict of duties (DTLS and Traffic inspection example)
- Conflict of priority
- Override behaviour conflict
- Dependencies (policy and event examples)

These kinds of rules are previously compiled for the Pyke rule engine which generates python code. In this way, when the service receives the MSPL-OP object, it initializes the engine and it retrieves the current MSPL policies that have been already enforced in the system, then it establishes these security policies as the base of knowledge. Finally, it loops over the MSPL-OP and inserts as new MSPL fact each MSPLs contained in the MSPL-OP. When a conflict or dependency is detected it is asserted as new fact in the *mspl_conflict* or *mspl_dependencies* knowledge respectively. This knowledge is the one returned in JSON format.

```
{
  "mspl_conflicts": [
   ["mspl_9f1a88b4fc67421b98de270d5a63d36b", "priority_dependency_conflict",
      "mspl_9f1a88b4fc67421b98de270d5a63d36a"]
  ],
  "mspl_dependencies": [
   ["mspl_9f1a88b4fc67421b98de270d5a63d36b", "policy_dependency",
      "mspl_9f1a88b4fc67421b98de270d5a63d36a"]
  ]
}
```

Figure 17: mcdtservice output example

Figure 17 shows an example of conflict and dependency detection. The JSON provides a list of three element tuples composed by the ID of the first MSPL involved in the issue, the kind of conflict or dependency and the ID of the second MSPL involved.

6.4 Security Orchestrator Implementation

This section has been extracted from deliverable D3.1 just with the aim to summarise the implementation concepts of the security orchestration since the details will be explained in the deliverable D3.5.
The security orchestrator is responsible for providing on-demand security policy enforcement on the IoT domain. This task is performed by taking in charge the transformation of the relevant security policies provided by the security policy interpreter into specific enabler configuration. It also monitors and supervises the underlying infrastructure for any potential flaws.

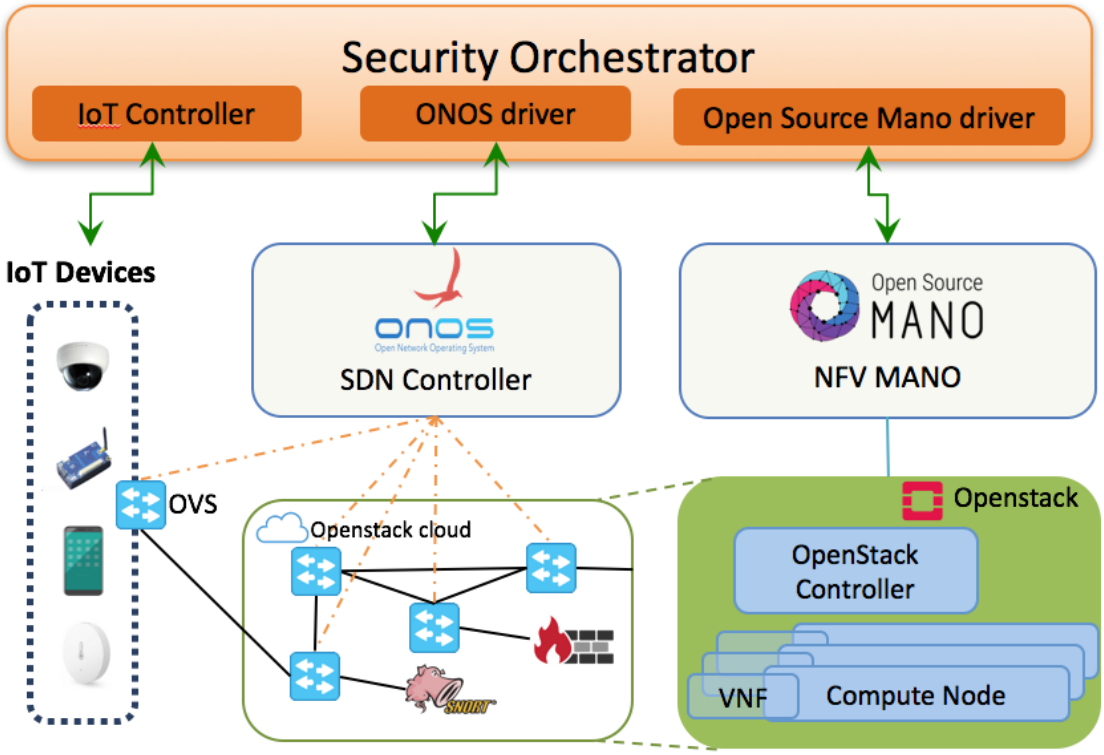

Figure 18: Security Orchestrator Implementation Architecture

To this aim, the security orchestrator interacts with three key components:

- **The IoT controller:** Used to enforce IoT-specific mitigation actions, such as IoT devices access control, authentication and power on/off. This interaction is done through Rest-API to send queries to the IoT controller depending on the security policy provided by the MSPL file.
- **The NFV MANO:** An ETSI-defined framework designed for managing and orchestrating resources in the cloud. It is used by the security orchestrator to create and configure a wide range of security enablers. It has three main functioning blocks:
  - NFV Orchestrator: Manages the registration of Network Services (NS) and Virtual Network Function (VNF) packages, lifecycle of different network services and the resources allocation requests.
  - VNF Manager: Configures and monitors each VNF after its instantiation.
  - Virtualized Infrastructure Manager (VIM): Interacts with the compute, network and storage resources (clouds) in order to provision relevant VNFs.
- **The SDN controller:** is accountable for managing network resources and enabling the programmability of the underlying network. The SDN orchestration is done through the ONOS driver. This driver has been developed in order to automate the SDN management using one or multiple ONOS SDN controllers. It controls multiple Open Virtual Switches (OVS) in order to enable the following functionalities:
  - Traffic forwarding (steering) to VNFs.
  - Traffic mirroring to different VNFs.
  - Traffic dropping.
  - Bandwidth limitation.

The combined usage of these components enables the security orchestrator to enforce the relevant security policies either through direct actions such as: traffic dropping and IoT devices power on/off, or more complex actions when it comes to VNFs:

- **Provisioning:** Creating the appropriate VNF on a chosen VIM (According to the VNF application graph) such as: Intrusion Detection Systems (IDS) and Firewalls…

- **VNF Configuration:** Using the MSPL to low level translation, the security orchestrator pushes the specific configuration of each VNF (IDS rules, Firewall configuration...)
- **Networking Setup:** Injecting the relevant SDN flow rules to manage the traffic to be analysed, for example: mirroring the traffic to a monitoring agent or steering the traffic through a firewall.

6.5 SECURITY ENABLERS PROVIDER IMPLEMENTATION

The security enabler provider has been implemented in python from scratch and it provides two main functionalities, these are, to provide a list of available security which are able to enforce the capability or capabilities received as parameter and to provide the specific plugin which implements the translation from the MSPL policy to the specific configuration of the security enabler. The implementation details of this module have been omitted in this document since they are properly explained in the section 6 of the deliverable D3.6.

7 CONCLUSIONS

This document provides the final report about the policy refinement, translation and conflict and dependencies detection processes being devised and implemented in WP3, and concretely in Task 3.1. In this regard, the document has described the Policy Interpreter component, the policy editor tool as well as the security enablers provider of the Anastacia framework, as main components in charge of performing those tasks. The report delves into the state of art regarding the policy refinement techniques, technologies as well as policy-based frameworks and conflict detection approaches. The document defines the relationships and interfaces between the policy interpreter and the rest of components. Besides, it has been detailed the policy refinement process from high-level security policy language to the medium-level security policy language, as well as the translation process from the medium-level security policy language to the specific security enabler configurations, also including policy conflict and dependencies detection processes.

Once it has been illustrated at design level, it has been provided an explanation regarding the current implementation and integration of the main components and services involved on the policy enforcement, that is, the Policy Editor Tool, the Policy Interpreter, the Security Orchestrator and the Security Enabler Provider.

8 REFERENCES

[1] Common Information Model (CIM), DMTF Standard.

[2] Jorge Bernal Bernabe, Juan M. Marin Perez, Jose M. Alcaraz Calero, Jesus D. Jimenez Re, Felix J. Garcia Clemente, Gregorio Martinez Perez, Antonio F. Gomez Skarmeta, **"Security Policy Specification"**, Network and Traffic Engineering in Emerging Distributed Computing Applications, IGI Global, pp. 66-93, 2012.

[3] Policy-Based Security Tools and Framework (POSITIF), EU project, FP6, IST-2002-002314

[4] Dependable Security by Enhanced Reconfigurability (DESEREC), IST-2004-026600, EU project, Framework Programme 6

[5] SECURED EU FP7 project, deliverable D4.1: **Policy specification**.

[6] SECURED EU FP7 project, deliverable D4.2: **Policy transformation and optimization techniques**.

[7] ANASTACIA D2.1: Policy-based definition and policy for orchestration – Initial report

[8] ANASTACIA D2.5: Policy-based definition and policy for orchestration – Final report

[9] Simplifying Network Administration Using Policy-Based Management, Dinesh C. Verma, IBM Thomas J Watson Research Center

[10] Management and Translation of Filtering Security Policies, Ehab S. Al-Shaer and Hazem H. Hamed Multimedia Networking Research Laboratory School of Computer Science, Telecommunications and Information Systems DePaul University, Chicago, USA

[11] Ontology-based Security Policy Translation, Cataldo Basile, Antonio Lioy, Salvatore Scozzi, and Marco Vallini, Politecnico di Torino.

[12] IPSec/VPN Security Policy: Correctness, Conflict Detection and Resolution, Zhi Fu, S. Felix Wu Computer Science Department, North Carolina State University. He Huang, Kung Loh Nortel Networks. Fengmin Gong, Ilia Baldine, Chong Xu Advance Networking Research MCNC

[13] Policy Conflict Analysis in Distributed System Management, Jonathan D. Moffett, Department of Computer Science University of York, UK. Morris S. Sloman Department of Computing Imperial College of Science Technology and Medicine University of London, UK.

[14] Detection of semantic conflicts in ontology and rule-based information systems, Jose M. Alcaraz Calero, Juan M. Marín Pérez, Jorge Bernal Bernabé, Felix J. Garcia Clemente, Gregorio Martínez Pérez, Antonio F. Gómez Skarmeta.

[15] Diego Lopez et al. **I2NSF Framework for Interface to Network Security Functions**. RFC 8329, IETF. Feb 2017. I2NSF Working Group

[16] L. Xia et al. **Information Model of NSFs Capabilities**. Internet-Draft, IETF. December 2017.

[17]I. Farris et al., "Towards provisioning of SDN/NFV-based security enablers for integrated protection of IoT systems," 2017 IEEE Conference on Standards for Communications and Networking (CSCN), Helsinki, 2017, pp. 169-174.doi: 10.1109/CSCN.2017.8088617

[18] S. Ziegler, A. Skarmeta, J. Bernal, E. E. Kim and S. Bianchi, "ANASTACIA: Advanced networked agents for security and trust assessment in CPS IoT architectures," 2017 Global Internet of Things Summit (GIoTS), Geneva, 2017, pp. 1-6.doi: 10.1109/GIOTS.2017.8016285

[19] Molina Zarca, A., Bernal Bernabe, J., Farris, I., Khettab, Y., Taleb, T., & Skarmeta, A. (2018). Enhancing IoT security through network softwarization and virtual security appliances. International Journal of Network Management, e2038. doi:10.1002/nem.2038

[20]Molina Zarca, A., Garcia-Carrillo, D., Bernal Bernabe, J., Ortiz, J., Marin-Perez, R., & Skarmeta, A. (2019). Enabling Virtual AAA Management in SDN-Based IoT Networks †. Sensors, 19(2), 295. doi:10.3390/s19020295