ANASTACIA

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

D3.6

Final Security Enforcement Enablers Report

This deliverable presents the final results of ANASTACIA Task 3.3, which aims to manage the Security Enforcement Enablers requirements established in high-level terms through the ANASTACIA architectural components and their integration.

Distribution level	PU
Contractual date	30.09.2019 [M33]
Delivery date	
WP / Task	WP3 / T3.3
WP Leader	THALES SIX GTS France
Authors	Dallal BELABED (THALES SIX GTS FRANCE) Bruno VIDALENC (THALES SIX GTS FRANCE) Alejandro Molina Zarca (UMU)
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.

Table of Contents

Index of figures

Index of tables

PUBLIC SUMMARY

This task is about the Security Enforcement manager, which is a component of the Security Orchestrator plan in the ANASTACIA architecture. In this deliverable, we present the results of the “Final Security Enforcement Enablers Report”, which tackles the Task 3.3 within WP3. More precisely, this deliverable focuses on the development and the deployment of the core enablers for the security enforcement manager.

1 INTRODUCTION

This section will introduce this document by enumerating its aims, references, revision history and the different acronyms which were used.

1.1 AIMS OF THE DOCUMENT

This document is the sixth WP3 report. More precisely, this deliverable tackles the Task 3.3 within WP3. This task is about the Security Enforcement manager, which is a component of the Security Orchestrator plan in the ANASTACIA architecture [1].
This deliverable “Final Virtual Resources Manager Report” is the continuity of the D3.3 “Initial Security Enforcement Enablers”. The D3.6 goes step further focusing on the algorithms and methods for an efficient management of the resources, while the D3.3 was focused on the development of core enablers for the deployment and implementation phase of the security enforcement manager.

During the first period, a first definition of "Security Enablers Manager" was done at the early stage of the project, where the focus was about differencing the goals of the Security Enforcement Enablers regarding of the security orchestrator goals. Thus, the term "Security Enablers Manager" was changed to “Security Enablers Provider” to avoid any confusion where module's features are confirmed and published in the deliverable of Anastacia architecture D1.3. Besides; the module will be in charge of providing the list of available security enablers, according to the security capabilities, and relevant plugins for the refinement of MSPL security in low-data configuration. Moreover, the definition of the strategies related to the security enablers selection will be done in the Security Orchestrator module and will be driven by T3.3. To this aim, a specific subcomponent in the Security Orchestrator has been introduced called “Security Resource Planning” and this second period was focused on this subcomponent that was largely extended.

Now, the definition of the task 3.3 is stable, relevant plugins for the refinement of MSPL security in low-data configuration was defined. The definition of the subcomponent in the Security Orchestrator introduced was defined. The definition of the algorithms that will be used by the Security Orchestrator subcomponent was defined. Finally, a version of the Security Enabler Provider and the Security Resource Planning was developed and shared with the Anastacia partners.

This document is structured as follow: Section 2 provides an overview of the states of the art, the section 2 discusses the progress beyond the state of the art, the section 4 provides an overview of the Anastacia architecture, contextualizing the Security Enforcement Enablers in the ANASTACIA framework. Section 5 defines the refinement of the Security Enforcement Enablers components and its interaction with the other Anastacia components, Section 6, describes the implementation of the Security Enforcement Enablers components. 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 deliverable D3.1 – Initial Security Enforcement Manager Report.

- ANASTACIA deliverable D3.2 – Privacy Risk Modelling and Contingency – Initial Report.
- ANASTACIA deliverable D6.2 – Initial use cases implementation and tests Report.
- ANASTACIA deliverable D3.3 – Initial Security Enforcement Enablers.

1.3 Revision History

Version	Date	Author	Description
0.1	17.07.2019	Bruno VIDALENC (TCS)	Table of contents
0.2	24.07.2019	Dallal BELABED (TCS)	Revision of the Table of contents
0.3	14.08.2019	Bruno VIDALENC (TCS)	Extension of the resource planning module
0.4	28.08.2019	Dallal Belabed (TCS) Bruno VIDALENC (TCS)	Initial full draft of deliverable
0.5	26.09.2019	Alejandro Molina Zarca (UMU)	Security Enablers implementation Figure 7 contribution
0.6	30.09.2019	Alejandro Molina Zarca (UMU)	Reviewed the document
0.7	30.09.2019	Miloud Bagaa (AALTO)	Reviewed the document
1	30.09.2018	Dallal Belabed (TCS) Bruno VIDALENC (TCS)	Internal revision of the document

1.4 Acronyms and Definitions

Acronym	Meaning
MSPL	Medium-level Security Policy Language
HSPL	High-level Security Policy Language
IoT	Internet of Things
MANO	Management and Orchestration
NFV	Network Function Virtualization
SDN	Software Defined Networking
MEC	Mobile Edge Computing

M2L	Medium to Low
OVS	Open Virtual Switch
SO	Security Orchestrator
DPI	Deep Packet Inspection
IDS	Intrusion Detection System
SEP	Security Enforcement Plane
DDoS	Distributed Denial of Service attack

2 State Of The Art

Previously, network services were designed as an ordered set of physically wired hardware devices that handled traffic for security or optimization purposes. With the virtualization of network functions (NFV), the middle boxes are increasingly based on software and are installed above compatible standard equipment's with virtualization, enabling reduced costs and flexibility network. Nevertheless, with this new paradigm, new challenges have emerged. Indeed, the set of service functions, often chained to provide complex services, are completely separate from physical topology and virtual service functions (vSFs) are more ephemeral and dynamic in nature. Directing traffic across these dispersed virtual entities without compromising end-user sessions and quality of service (QoS) is therefore a complex challenge. Although Internet Service Providers (ISPs) rely critically on the core boxes for security and policy compliance, most existing NFV management solutions rely on a logically centralized omnipotent entity, typically called orchestrator. Such centralized approaches, in that they require a global view of the entire network to be able to chain services, introduce control responsiveness and resiliency issues (for example, a single point of failure). In addition, this can be quite expensive for operators as it requires the deployment of a new management and control infrastructure. In addition, the control part, which is supposed to change the network configuration to implement the Orchestrator's decisions, tends to be poorly interoperable with legacy devices and is therefore difficult to deploy incrementally. We believe that it is not necessary to centralize each orchestration decision for the sequence of service functions. Although some long-term high-level decisions need to remain centralized, we do not believe that the establishment of a service chain is part of it. We argue that the availability of the service would benefit from a distributed and dynamic design. With this in mind, we propose to improve the network routing layer to make it compatible with services, within the resource planning subcomponent at the security orchestrator level. In particular, it is possible to use any Interior Gateway Protocol (IGP), anycast addressing, and any service chain encapsulation to build a distributed command plan that considers services. We propose a modular architecture that we call NFV-Router (NFV-R). It does not require complex elements and remains interoperable with legacy devices. The architecture, implementation and evaluation will be presented in section 6.2.2.

3 DISCUSSION ON PROGRESS BEYOND THE STATE OF THE ART

Existing management and orchestration (MANO) solutions rely on a central orchestrator to create virtual links between these service functions. They apply detailed transmission rules on network devices or rely on the waypoint routing paradigm to properly direct traffic into a set of service functions. This centralized approach tends to be poorly interoperable with existing networks and to introduce control responsiveness and resiliency issues. Above all, they rely on a single point of failure of architecture: the orchestrator.

We propose to distribute MANO by including in each router/POP/datacenter (for the sake of generality, we call it NFV-Router or NFV-R) its own MANO part to communicates with its peers and to ensure end-to-end chaining via virtual service functions. (vSF) (See the figure below). This MANO becomes more robust to failure

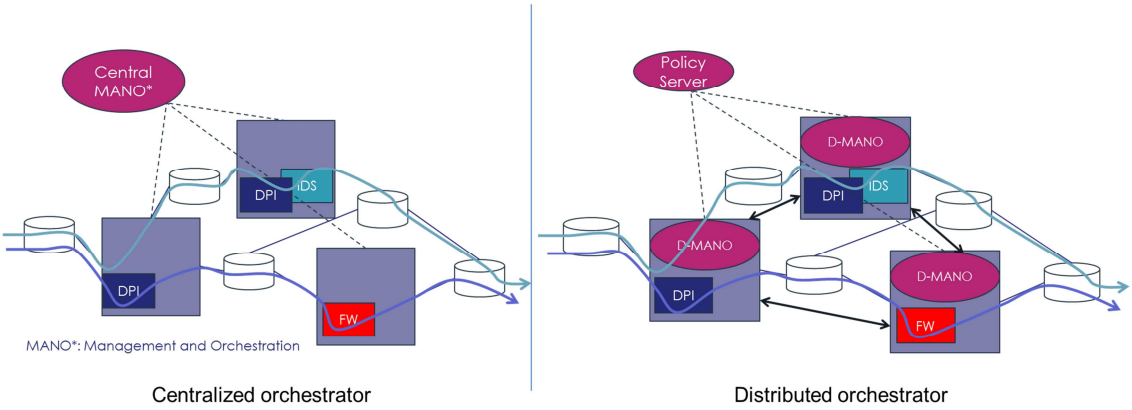


Figure 1: Distributed MANO approach

In this section, we explain how to augment the network routing layer to enable chaining of distributed service features. Any IGP network can be directly used to convey the location, type, and necessary information associated with a virtual appliance and to create an augmented network view. Based on this improved topology, any routing scheme can be used to direct traffic through service functions, i.e., to chain services. Such an approach can take full advantage of the field-proven scalability and robustness of the IGP. An IGP allows gateways (usually routers) to exchange routing information. This routing information is then used by each gateway to construct an IGP network view and route the network layer protocols. We propose to add the notion of service in such a view. We call it the topology of the service plan. It consists of two types of nodes: NFV routers (NFV-R), equivalent to IGP gateway nodes, and virtual service functions (vSFs), which are a new type that we introduce. The NFV-R are physical devices that run not only the protocol IGP but also hosts vSF. NFV-Rs can be traditional IP routers with VNF hosting capabilities, points of presence, or even data centers. vSFs correspond to a virtual service function, also called virtual network function, instances. They can provide different services depending on their type: intrusion detection system (IDS), firewall, NAT, stream coding, etc. These instances are hosted by the NFV-R, which allows them to advertise directly on the IGP, the functions they can provide. We also propose to take advantage of anycast addressing to include vSF in the service plan topology. All instances of vSF providing the same service are advertised by the NFV-Rs hosting them, with the same IP address but with their own vSF cost. Thus, in the service plan topology, a service type is represented by a vSF node, while an instance is represented by a link (see *Figure 1Figure 1: Distributed MANO approach*). This approach has many advantages.

First, it reduces the number of vSF routes advertised on the IGP. Secondly, a service chain can be explicitly and unambiguously described as an ordered sequence of waypoints to be reached (the anycast addresses) and relies on the network routing layer to choose the vSF instances and the path to use. Finally, NFV-Rs must simply advertise a vSF to make it immediately available for new incoming streams without further configuration. Nevertheless, it is known that anycast routing has drawbacks: packets belonging to the same stream can be routed to different instances of vSF if the best path changes, which would break the affinity of the vSF stream. To avoid this problem, we are improving vSF advertisements with a forwarding address: a dedicated unicast address on the NFV-R acting as a vSF proxy (for example, a router loopback interface). We also leverage on the flow routing to steer all packet from the same flow to the same vSF sequence. To do this, the NFV-R records the initial routing decision when the first packet of a stream is sent to the vSF address. *Figure 1* illustrates with a toy example the proposed approach. Left part shows the network topology consisting of NFV-R. Each vSF instance of a given type is advertised on the network with the same anycast address. In particular, the two firewall instances (FW) announce the same address (@IP fw). Flows must be processed here by a single chain with: IDS and FW. The first stream is routed through the IDS instance and then through the higher FW instance. Indeed, in this example, this vSF instance is at one hop from the NFV-R that hosts the IDS instance. NFV-Rs that host the used vSF advertise their neighbors with the new load or any other relevant information. When the second stream arrives, the instance below is prefered, resulting in a balancing of the load between the FW instances. As the route from the first stream has been recorded, it continues to be driven to the upper instance FW even though the best path has changed.

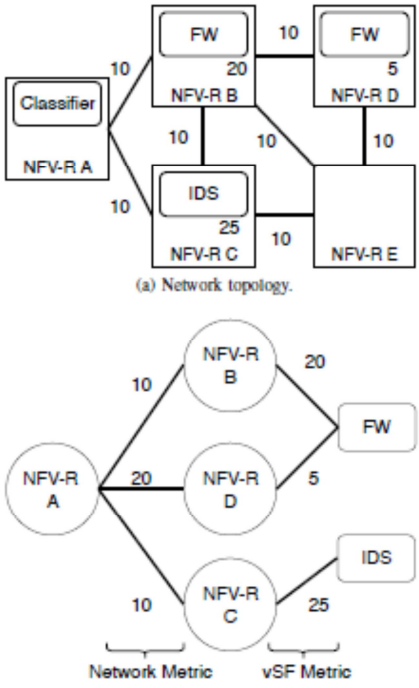


Figure 2: NFV-R D

Each NFV-R builds its service plan topology (example at node A in Figure 2) with network costs and vSF costs to calculate the next hop(s). Since the same address is advertised but no adjacency is established between the vSF (the two firewall instances in our example), flows that use a link to achieve a function of service must use the same link to exit. However, this link is only virtual; it is the representation of the vSF instance in the IGP because it runs directly on the NFV-R in reality. The IGP improvement makes has the advantage of using what is already done at the level of the network layer routing. Anycast addressing exploit IGP information sharing to create the augmented topology. Based on this topology, a routing decision maps the vSF type to the next appropriate NFV-R (s) based on the metric of

the network and instances. Finally, the protocol IGP offers us a robust IP connectivity between NFV-R to steer the flow through the right set of instances. Note that the IGP prevents flow remapping in case of link failure. Indeed, once the IGP has converged, the connectivity to the NFV-R is restored without any modification of the cached routing decisions. In addition, NFV-Rs can be deployed progressively in areas where they coexist with conventional routers. Indeed, the services being announced in the form of IP addresses, the classic routers will announce them to their neighbors. Based on this raw IP topology, an NFV-R is able to reconstruct the topology of the service plan. In our approach, a lightweight central management node is responsible for setting up high level policies on the NFV-R. As with any IGP, these rules are common to all nodes. They make it possible to control decision-making at each NFV-R. These policies include flow classification rules, which map traffic to the necessary chain of services. They also concern routing decisions since all NFV-Rs must share the same routing objectives. Based on the service plan topology, the NFV-R can use any path calculation algorithm (for example, the shortest path first) to choose the instance of the next vSF in the chain used by the flow. In addition, high-level strategies can also set the method of IGP calculation for vSF, indicating the data to be used and function allowing for the costs conversion. Our approach can rely on network encapsulation to transmit the information needed to route the flows across the associated service chain. This information can be used to make a routing decision at the source or at each NFV-R processing flow (hop by hop). This header shall include all or part of the service chain identified at classification stage at the ingress, and the next service step of that chain, and finnaly an flow identifier for caching the routing decision. In the example, the NFV-R that hosts the IDS instance must have a way of knowing that a packet belonging to a specific flow has been assigned to the IDS + FW service chain, that the next service to apply is FW , and which of the FW instances it should actually traverse. In the rest of this section we focus on jump-by-hop routing, although source routing is also applicable, and take OSPF as an example for the IGP protocol. Figure 2 illustrates how a service plan topology is constructed from a network topology. In this approach, each NFV-R calculates the shortest path to each vSF type in terms of network cost /vSF and maps each type of vSF to the associated NFV-R transmission address. This mapping can be easily calculated by running the Dijkstra algorithm. The cost that each NFV-R associates with the advertised prefix, ie the IGP cost to reach such an address, represents the state of the vSF and can be based on a multitude of parameters (for example, its available capacity, its load, etc.). The metrics used (link and service) must be of the same order of magnitude and they must be additive to ensure loop-less convergence, even with multiple constraints. By using OSPF, the NFV-Rs, as conventional routers, are already calculating a routing table to obtain aggregated network costs for NFV-Rs. Thus, the algorithm for finding the route to the next instance of vSF should run only on a depth 2 graph and add a very small computational overhead to the routing system.

4 SECURITY ENFORCEMENT ENABLERS IN THE ANASTACIA ARCHITECTURE

4.1 SECURITY ENABLER PROVIDER MODULE

The Security Enabler Provider is a component of the Security Orchestration Plane, as defined in the Anastacia architecture. This component is able to identify the security enablers which can provide specific security capabilities, to meet the security policies requirements. Moreover, when the Security Resource Planning, a sub-component of the security orchestrator, defined here after, selects the security enabler, the Security Enabler Provider is also responsible for providing the corresponding plugin.

The following tables describe the Security Enabler Provider and their interfaces:

Table 1: Security Enabler Provider description

Security Enablers Provider	
Function	The Security Enabler Provider is able to identify the list of security enablers which can provide the specific security capabilities to meet the security policies requirements. Besides, this component will be endowed with an interface for delivering security M2Lplugins (Medium2Lower), which, in turn, will allow translating policies from MSPL to Low-level configurations.
Subcomponent	(Not applicable)
Sources	Security Policy Interpreter / Security orchestrator
Consumers	Security Policy Interpreter / Security orchestrator
ANASTACIA activities involved	Security Policy Set-up Security Orchestration
Available assets	A first implementation of this component with different Security M2Lplugins (Medium2Lower), that allows translating from policies specified in MSPL to low level configuration, have already be implemented and integrated.

Interfaces focused on the security enablers management for policy refinement and policy translation are:

- SEPSEI (Table 2. Interpreter -> Security Enabler Provider Security Enabler (SEPSEI))
- SEPPI (
- Table 3. Interpreter -> Security Enabler Provider (SEPPI))

Table 2. Interpreter -> Security Enabler Provider Security Enabler (SEPSEI)

Security Enabler Provider Security Enabler Interface (SEPSEI)	
Description	The interface allows requesting the required security enablers for specific capabilities.

Component providing the interface	Security Enabler Provider	
	Input Data	List of capabilities
	Output Data	List of candidate security enablers
Consumer components	Policy Interpreter, Security Orchestrator	
Pre-conditions	There must be a correspondence between capabilities and security enablers.	
Post-conditions	(Not applicable)	
ANASTACIA activities involved	Security Policy Set-up Security Orchestration	

Table 3. Interpreter -> Security Enabler Provider (SEPPI)

Security Enabler Provider Plugin Interface (SEPPI)		
Description	The interface allows requesting the plugin that translates the MSPL file to low-level configuration.	
Component providing the interface	Security Enabler Provider	
	Input Data	Enabler name
	Output Data	Enabler translator plugin
Consumer components	Policy Interpreter	
Pre-conditions	There must be a correspondence between the security control name and the code location in the Security Enabler Provider.	
Post-conditions	(Not applicable)	
ANASTACIA activities involved	Security Policy Set-up Security Orchestration	

4.2 SECURITY RESOURCE PLANNING MODULE

The security resource planning uses the list of the selected enablers returned to the security orchestrator by the Security Enabler Provider to decide the more adequate enabler(s) among the list to be used to enforce the security. This selection is done through an Integer Linear Programming (*ILP*) model. The aim of the model is to select the best service (Virtual Network Function (VNF)) among the list of enablers selected previously by the selected Security Enabler Provider, in order cope with a security attack and that minimize

the maximum load nodes (CPU, RAM, bandwidth) of the topology, provided by the system model. Indeed, the system information will provide relevant data about the whole infrastructure, server capacity (CPU, RAM, etc), and VNF flavours (CPU, RAM, etc). On the other hand, the Security Enablers information will provide the data regarding the available Security Enablers capable to enforce specific capabilities.
The different VNFs are considered as a set of enablers, each enabler is characterized by its type and its resources. The security resource planning requests from the system model all the capacity information regarding the infrastructure and the VNFs Flavours. The set of topology nodes is also characterized by its type and its resources. The goal of the model is minimizing the maximum load nodes to improve provider cost revenue (provider energy efficiency goal). Furthermore, we assume that:

- Multiple services (VNFs) can be allocated on the same node,
- A VNF service cannot be split on multiple nodes.
- Each node can host multiple services.

The security Resource Planning Module is implemented as an autonomous plugin that receives a list of the enablers and the topology information from the System Model, based on that information we run the ILP implemented using IBM *CPLEX* Optimizer engine that gives the selected security enablers that cope with the security attack and the nodes where this security enablers have to be installed.

5 REFINEMENT SECURITY ENFORCEMENT ENABLERS

5.1 SECURITY ENABLER PROVIDER INTERACTIONS

5.1.1 Interactions with Policy Interpreter for refining security policies

This section shows the main interactions for a policy-based deployment between the Policy Interpreter and the Security Enabler Provider. Specifically, two different interactions have been contemplated. The first one will provide to the Policy Interpreter a list of security enabler candidates from the main identified capabilities. The second one will provide to the Policy Interpreter the specific Security Enabler Plugin in order to perform the policy translation. This policy translation process was defined in Anastacia D3.1 0, and also published in journal paper [1].

5.1.1.1 List of Security Enabler candidates

In Figure 3, the main interactions among the Policy Interpreter and the Security Enabler Provider are shown. In particular, a list of Security Enabler candidates retrieval process is shown, which will be needed to enforce the identified capabilities.

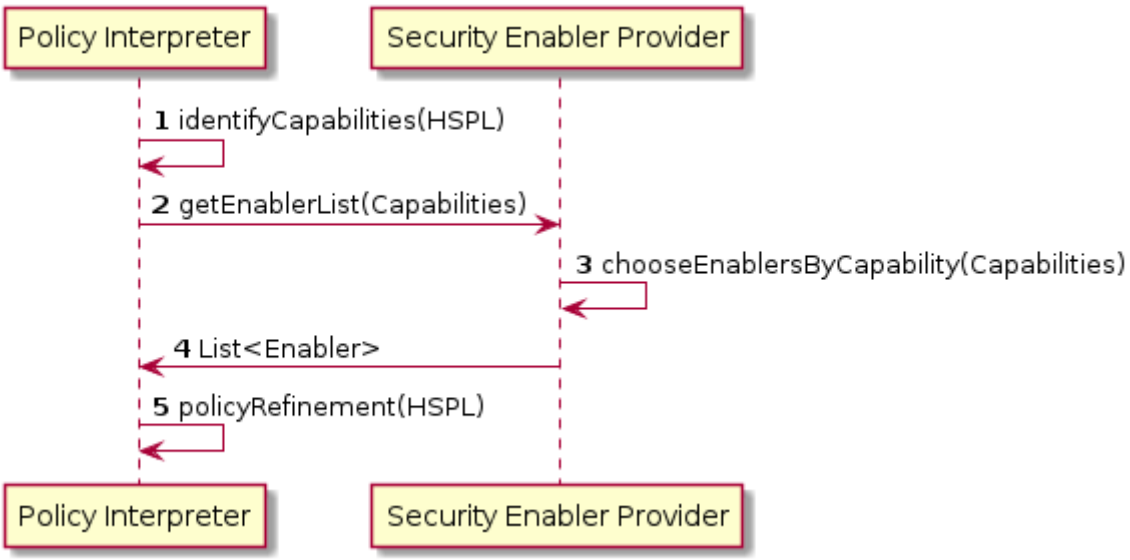


Figure 3: Security Enabler Candidates (Interactions with Policy Interpreter)

The process is compounded by the following steps:

1. Once the Policy Interpreter has received the High-level Security Policy, it identifies the required capabilities, which in turn will be needed by the system in order to enforce the security policy.
2. The Policy Interpreter requests a list containing the Security Enablers that are able to implement the identified capabilities.
3. The Security Enabler Provider performs a matching among the received capabilities and the existent Security Enablers (e.g. For filtering capability, it could return a SDN controller enabler, or a vFirewall enabler).
4. The Policy Interpreter receives the list of Security Enabler candidates. If the list contains at least one Security Enabler, the process can continue.
5. The Policy Interpreter performs the policy refinement process.

5.1.1.2 Get the Security Enabler Plugin

In Figure 4 the main interactions among the Policy Interpreter and the Security Enabler Provider can be observed, where the specific Security Enabler Plugin is retrieved containing the implementation regarding how the Medium-Level Security Policy must be translated into specific Security Enabler configuration.

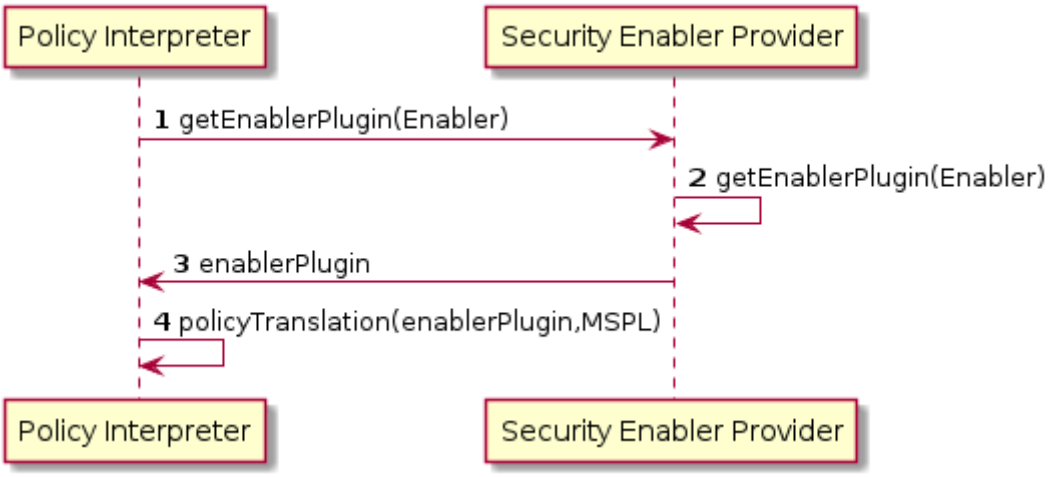


Figure 4: Security Enabler Plugin (Interactions with Policy Interpreter)

The process is compounded by the following steps:

1. The Policy Interpreter requests the specific Security Enabler Plugin to the Security Enabler Provider, indicating a plugin identifier.
2. The Security Enabler Provider obtains the plugin package for the specific plugin identifier.
3. The Policy Interpreter receives the plugin package.
4. The Policy Interpreter executes the plugin in order to translate the MSPL policy in low-level specific configuration for the specified Security Enabler.

5.1.2 Interactions with the Security Orchestrator

5.1.2.1 List of Security Enabler candidates

The main interactions between the Security Orchestrator and the Security Enabler Provider are shown. In particular, a list of Security Enabler candidates retrieval process is shown, which will be needed to enforce the identified capabilities Figure 5.

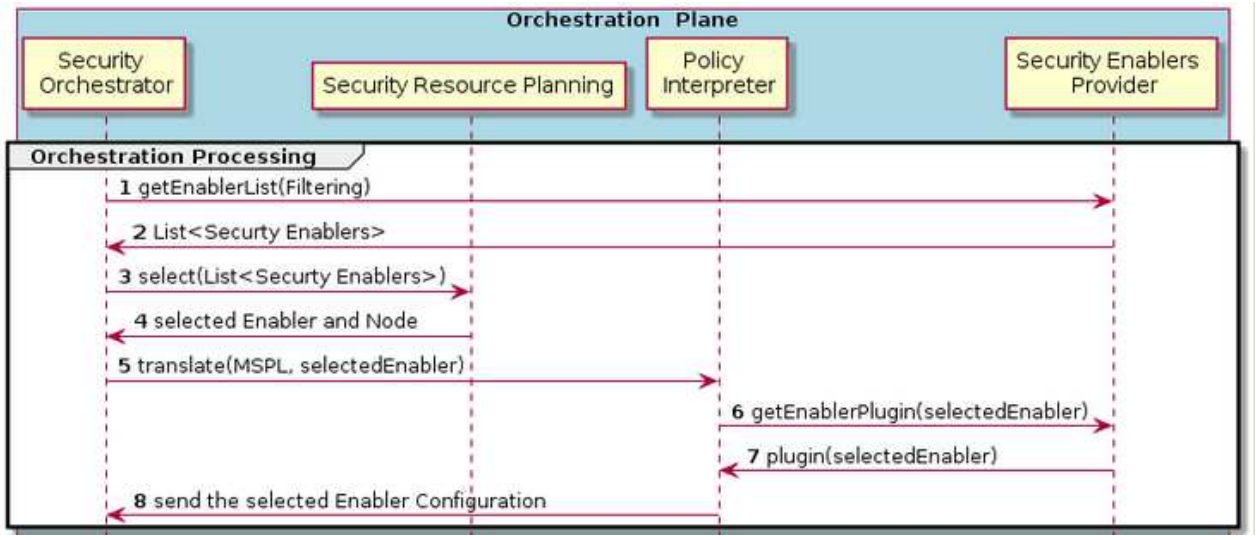


Figure 5: Security Enabler Candidates (Interactions with the Security Orchestrator)

The aforementioned process is compounded by the following steps:

1. The Security Orchestrator requests from the Security Enabler Provider the list of Security Enablers candidates for a security capability.
2. The list of Security Enablers candidates is provided to the Security Orchestrator.
3. The list of Security Enablers candidates is provided to the Security Resource Planning.
4. The Security Resource Planning gives to the Security Orchestrator the selected enabler and the selected node.
5. The Policy Interpreter performs the policy translation.
6. The Policy Interpreter requests the plugin for the selected enabler.
7. The Security Enabler Provider gives the accurate Plugin
8. The Policy Interpreter sends the selected enabler configuration to the Security Orchestrator.

5.2 Security Resource Planning module Interactions

The security Resource planning is a component of the security orchestrator and it only interacts with the Security orchestrator.

5.2.1 Interactions with the Security Orchestrator

This section shows the main interactions between the Security Resource Planning and the Security Orchestrator. In this case the workflow starts once the Security Orchestrator receives the list of the enabler candidates from the Security Enabler Provider. The list is sent to the Security Resource Planning, which makes the decision regarding which one will be the best Security Enabler able to enforce the Medium-level Security Policy. The selected enabler will be sent to the Security Orchestrator, which will be sent to the Policy Interpreter and then to the Security Enabler Provider to get the specific Security Enabler configuration for the selected Security Enabler through the Policy Interpreter.

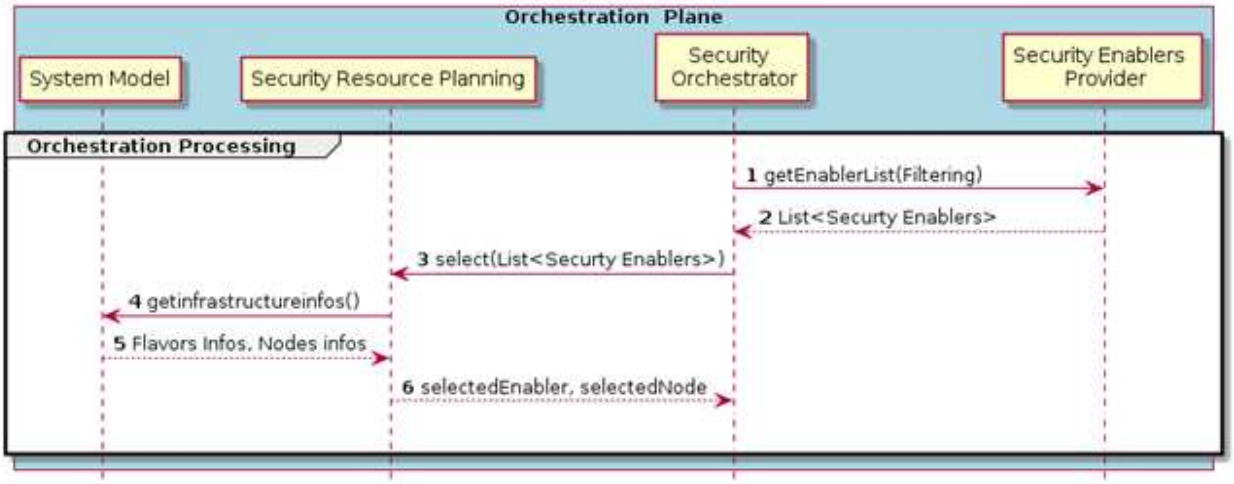


Figure 6: Security Resource Planning Module Interactions with the Security Orchestrator

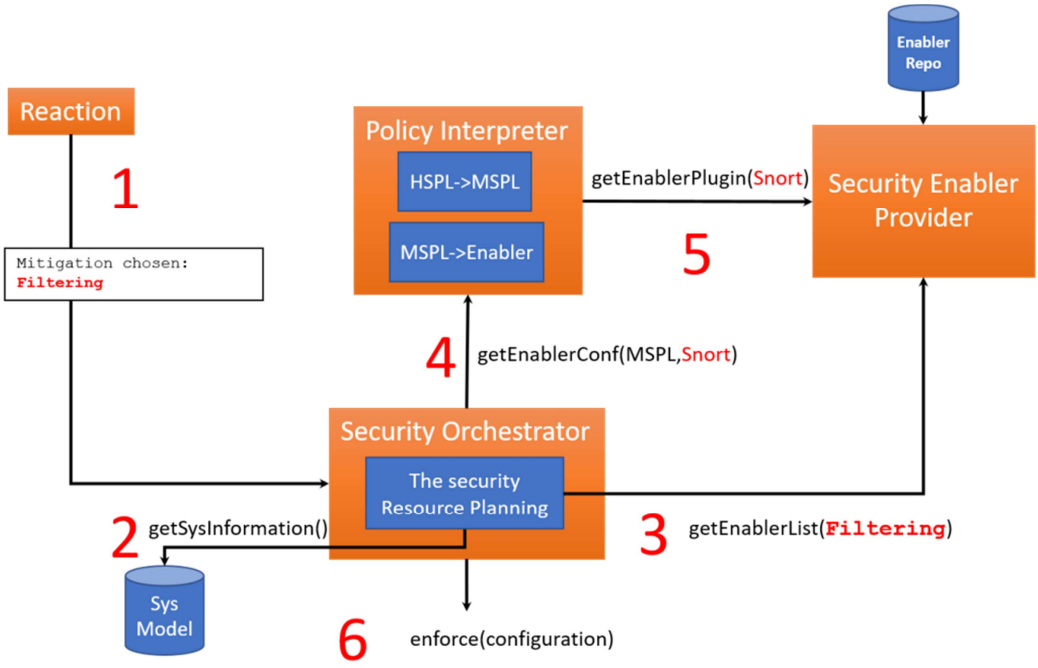
The aforementioned process is compounded by the following steps, Figure 6:

1. The Security Orchestrator requests from the Security Enabler Provider the list of Security Enablers candidates for a security capability.
2. The list of Security Enablers candidates is provided to the Security Orchestrator.
3. The Security Orchestrator provides to the Security Resource Planning the list of Security Enabler candidates provided by the Security Enabler Provider.
4. The Security Resource Planning requests the system model to get the VNF flavours and the infrastructure information.
5. The Security Resource Planning uses the list of Security Enabler candidates and the information given by the system model, in order to decide what kind of Security Enabler should be used to enforce the MSPL policy.
6. The Security Resource Planning gives to the Security Orchestrator the selected enabler and the selected node.

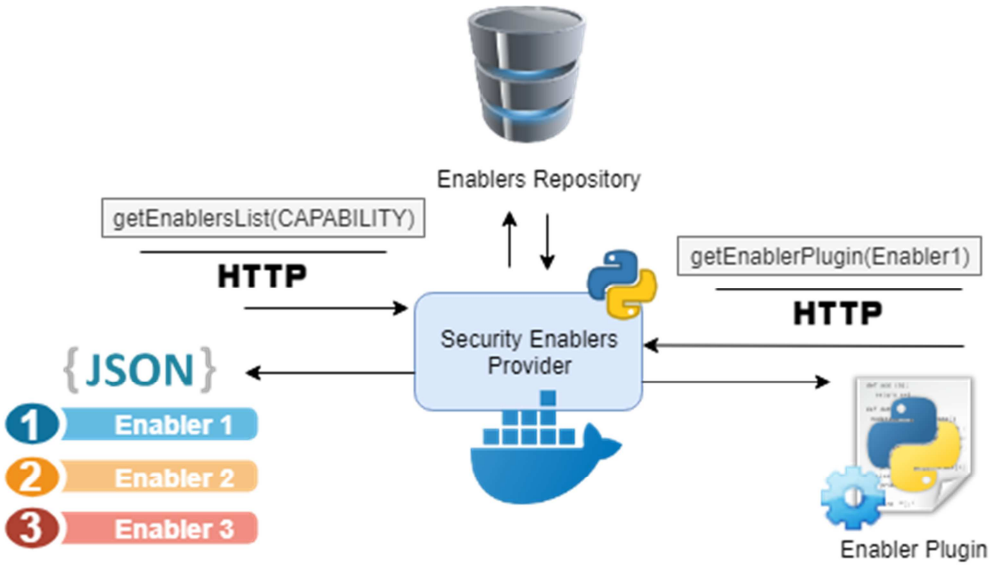
6 SECURITY ENFORCEMENT ENABLERS IMPLEMENTATION

6.1 SECURITY ENABLER PROVIDER

The Security Enabler provider has two main roles; it provides the list of the available enablers and it provides a plugin that translates the policies from MSPL to Low-level configurations. The first role is implemented as a piece of software that from specific capabilities given as an input it will provide the more accurate enablers. The second role is also implemented as piece of software capable to translate MSPL policies into specific configuration/tasks rules according to a concrete security enabler. The Figure 7 shows an example of Security Enforcement Enablers functionalities.



(a) Security Enforcement global interactions



(b) Security Enforcement Enablers Implementation

Figure 7: Security Enforcement Enablers Implementation

6.1.1 List of the security enablers

The plugin implementation that provides the list of the available enablers is done as follows. When, the security enabler provider received the list of capabilities, it must ask the Enabler repository and parse an XML file that contains the list of the available enablers. Each enabler that matches with the capabilities is added to the list. Finally, the list is returned to the security orchestrator, where the security resource planning will use it to decide the more adequate enabler(s) among the list to be used to enforce the security.

The Figure 7 shows an example of the security enabler provider, under a security attack, the reaction component has chosen as a mitigation strategy the filtering capability, Figure 7 (1). This capability will be sent to the security enabler provider "*getEnablerList(Filtering)*", Figure 7 (3). The security enabler provider will ask the Enabler repository to get the list of available enablers as "snort, iptables, firewall ….". The previous list will be used by the security resource planning, and based on the information given by "getSysInformation()" step , to make the proper enabler selection.

We have implemented this functionality using python language and we have implemented an API for requesting the list of enablers, the Figure 8: Security Enabler Provider (list of the security enabler requests) shows the API. The plugins are being implemented from scratch in python in ANASTACIA. The plugin consists on a piece of software. The main goal of this plugin approach is to provide a plugin repository where each security capability is linked with the list of the available enablers.

```
tai@Annecy:~/Documents/anastacia/implem/security-orchestrator/SecurityEnablerProvider_RessourcePlanning$ curl -v  195.148.125.10
0:8001/get_plugins?capabilities='Filtering_L3','Traffic_Divert'
*   Trying 195.148.125.100...
* Connected to 195.148.125.100 (195.148.125.100) port 8001 (#0)
> GET /get_plugins?capabilities=Filtering_L3,Traffic_Divert HTTP/1.1
> Host: 195.148.125.100:8001
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: gunicorn/19.7.1
< Date: Wed, 11 Jul 2018 11:21:47 GMT
< Connection: close
< content-length: 95
< content-type: application/json; charset=UTF-8
<
* Closing connection 0
{"candidate security enablers": ["onos", "snort", "onos nb", "iptables"]}
```

Figure 8: Security Enabler Provider (list of the security enabler requests)

6.1.2 Security Enabler

When the security resource planning selects the enabler, the orchestrator will send the selected enabler identification with an MSPL file that encloses the appropriate configuration. In the example the chosen enabler is "*Snort*" Figure 7: Security Enforcement Enablers Implementation (4). When the security enabler provider receives the request, it will provide the specific plugin for the selected security enabler, Figure 7: Security Enforcement Enablers Implementation (5), and, after the policy interpreter has performed the translation using the plugin, the orchestrator will receive the final configuration that will be enforced by launching/configuring the enabler, Figure 7: Security Enforcement Enablers Implementation (6).

Regarding the security enabler plugins, the implementation inherits and extends the concept from the SECURED project. Nonetheless, the plugins are being implemented from scratch in python in ANASTACIA to comply with the new requirements, new kind of technologies and enablers, and to be able to cope with our new MSPL policies. The plugin consists on a piece of software which will define well-known callable methods which will take in charge to perform the translation among the MSPL policy and the specific configuration or task. The main goal of this plugin approach is to provide a plugin repository where the developers can contribute with their own plugins and solutions. Actually, it is envisaged the same policy could be enforced using different plugins. For instance, the same filtering policy can be enforced through SDN using different plugins for each SDN controllers, or by using a virtual router by configuring IPTABLES as firewall. Inside of ANASTACIA scope, most of the following plugins have been already developed in order to be able to enforce the main identified security policies, an example of the snort plugin is given in 9.1.

At the time of writing this document they have been implemented different plugins in order to validate different capabilities and security policies. Specifically, the following security enabler plugins have been implemented:

1. **SDN ONOS Northbound Flow Plugin:** It translates MSPL network policies to ONOS northbound API configurations in order to apply different flows to the SDN switches.
2. **SDN ONOS Northbound Intent Plugin:** It translates MSPL network policies to ONOS northbound API configurations in order to apply different intents to the SDN network.
3. **SDN ODL Plugin:** It translates MSPL network policies in Opendaylight northbound API rules in order to apply different flow rules to the SDN switches.
4. **IPTABLES Plugin:** It translates MSPL network policies in IPTABLES rules. These rules can be enforced by using NETCONF.
5. **Open Virtual Switch Plugin:** It translates MSPL network policies in OVS rules according to the specified parameters.
6. **MMT Probe Plugin:** It translates MSPL monitoring policies into MMT Probe configurations.
7. **UTRC Agent Plugin:** It translates MSPL monitoring behavioural policies into specific configurations for the UTRC agent.
8. **CPABE Proxy Plugin:** It translates MSPL attribute-based privacy policies into specific configuration for a CPABE proxy.
9. **SNORT Plugin:** It will translate the MSPL policy onto a SNORT monitoring rules.
10. **DTLS/TLS Proxy Plugin:** It translates MSPL channel protection policies into specific configurations for DTLS/TLS secure connections.
11. **Cooja Plugin:** It translates MSPL iot-honeynet policies into Cooja specific configurations in order to generate an IoT virtual environment.
12. **vAAA XACML Plugin:** It translates MSPL authorization policies into the required configuration for the AAA infrastructure.
13. **IoT Control Plugin:** It translates the operational MSPL IoT security policies to the specific IoT Controller northbound rules.
14. **IPTABLES Plugin:** It is able to translate the MSPL policy onto iptables rules.

Due to the extensible and modular implementation, the framework allows the administrators to implement and deploy easily new security enabler plugins as well as to incorporate them to the system.

Moreover, as the module has been developed as an independent software an API was implement an example of the API is shown in Figure 9.

```
ubuntu@atles:~$ curl -v 10.0.0.57:8001/get_plugin?name='iptables'
*   Trying 10.0.0.57...
* Connected to 10.0.0.57 (10.0.0.57) port 8001 (#0)
> GET /get_plugin?name=iptables HTTP/1.1
> Host: 10.0.0.57:8001
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: gunicorn/19.7.1
< Date: Wed, 18 Jul 2018 15:14:20 GMT
< Connection: close
< content-length: 6618
< content-type: application/json; charset=UTF-8
<
{"plugin_file_name": "mspl_iptables.py", "plugin": "# # -*- coding: utf-8 -*-\n\"\"\nThis python module implements a MSPL->IPT
ABLES plugin inside the ANASTACIA European Project,\nextending the MSPL language defined in secured project.\nHow to use:\n\tpyt
hon3 mspl_iptables.py [MSPL_FILE.xml]\n\n\"\"\n__author__ = \"Alejandro Molina Zarca\"\n__copyright__ = \"Copyright 2018, ANAS
TACIA H2020\"\n__credits__ = [\"Antonio Skarmeta\", \"Jorge Bernal Bernab\u00e9\", \"Alejandro Molina Zarca\"]\n__license__ = \"
```

Figure 9: Security Enabler API

6.2 SECURITY RESOURCE PLANNING MODULE

We have defined and implemented a first version of the security resource planning that was resented in the deliverable D3.3. Besides, during this period we have conducted a research work that extend the resource planning module to include a dynamic Service Function Chain (SFC) requests placement that aim to reduce the routing overhead in case of an attack happen, i.e., if we add a security VM we will not need to reconfigure the SDN network rules to include the new service, which will balance the load. The first part of this section will present the resource capacity planning 6.2.1 already introduced in D3.3 and the second part will focus on the service function chain requests placement 6.2.2.

6.2.1 Resource capacity planning

We provide in the following a definition of the Security Resource Planning module enabled by SDN and NFV paradigms. Our model studies the service placement, where each service represents a VNF that will cope with a security attack. The aim of our model is to select the best service (VNF) among the VNF catalogue that cope with a security attack and that minimize the maximum load nodes (CPU, RAM, bandwidth) of the topology.

6.2.1.1 Problem statement

The different VNFs are considered as a set service S of a type t characterized by its resources R. The set of topology nodes is N, each node n is characterized by its type t characterized by its resources R.

- The goal of our model is minimizing the maximum load nodes to improve provider cost revenue (provider energy efficiency goal).

Subject to:

- Node integrity and capacity constraints.

Furthermore, we assume that:

- Multiple services (VNFs) can be allocated on the same node, but a service (VNF) cannot split on multiple nodes.
- Each node can host multiple services.

6.2.1.2 Mathematical formulation

sets	
N	set of nodes, MEC servers, Provider Nodes and Cloud.
R	resource types (CPU, RAM, storage).
S	set of services, where a service represent a VM.
T	set of the different type of services.
Indices	
$s = 1, 2, \ldots S$	index associated to the requested service.
$n = 1, 2, \ldots N$	index associated to the node.
Parameters	
α_{st}	equal to 1 if service $s \in S$ has a type t.
β_{nt}	equal to 1 if node $n \in N$ has a type t.
ϕ_{sr}	resource requested by service s.
Φ_{nr}	resource of node n.
variables	
U	maximum load nodes.
χ_s	equal to 1 if the service s is active.
θ_{sn}	equal to 1 if service $s \in S$ is assigned to node n.

Figure 10: Mathematical notations

The Figure 10 reports the mathematical notations used in the following Mixed Integer Linear Programming (MILP) formulation of the problem. The goal of model is to minimize the maximum latency and the number of active nodes.

$$\min \quad U \tag{1}$$

s.t.

$$\sum_{s \in S} \chi_s = 1 \tag{2}$$

$$\sum_{n \in N} \beta_{nt} \theta_{sn} = \alpha_{st} \chi_s \ \forall s \in S \tag{3}$$

$$\sum_{s \in S} \phi_{sr} \theta_{sn} \leq \Phi_{nr} \, U \ \forall r \in R \ \forall n \in N \tag{4}$$

The constraint (2) ensures that at least service s has been selected , (4) ensures that if a service s of type t is assigned to node n of type t, then the service s has to be assigned to only one node and the selected node has to be of service t. Moreover, the equality (3) ensures that if service is assigned to a node the node is active, and if no service is assigned to a node then the node is not active.
The goal of our model is to minimize the maximum latency and the number of active nodes. Constrained by a set of the integrity constraints, which ensures that at least one service s has been selected, that if a service s of type t is assigned to node n of type t, then the service s has to be assigned to only one node and the selected node has to be of service t.

The goal is also constrained by a set of the capacity constraints, which ensures that a service s is assigned to a node n only if there are available residual computing resources.

The security Resource Planning Module is implemented as an autonomous plugin that receives a list of the enablers and the topology information from the System Model, based on that information we run the ILP implemented using IBM CPLEX Optimizer engine that gives the selected security enablers that cope with the security attack and the nodes where this security enablers have to be installed.

6.2.2 Service Function Chain (SFC) requests placement

6.2.2.1 Architecture

As shown in the figure below, a NFV-R is composed of an IP router providing underlying connectivity, a connector, connecting the router to different instances of vSF, vSFs themselves, providing the services, and a distributed management and orchestration manager (D -MANO), and allowing the local autonomous management of the node.

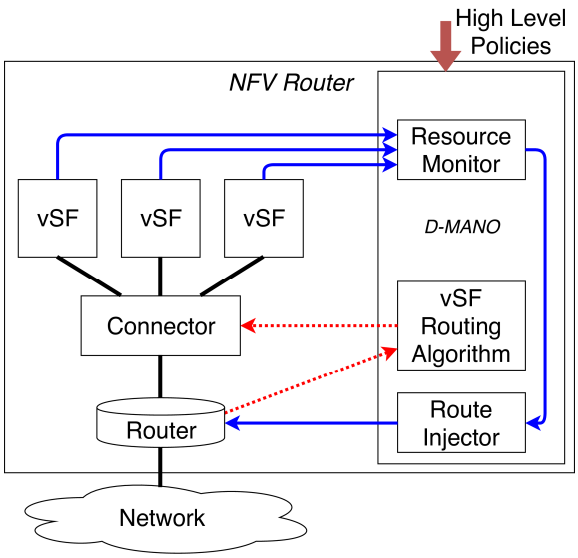


Figure 11: NFV router architecture

Router: The router provides underlying connectivity and participates in the IGP network. It exposes a control interface used by the D-MANO to inject or delete anycast vSF addresses, announcing the services available on the node and the associated costs. This control interface is also used for the IGP topology to build the topology of the service plan.

Connector: The connector distributes traffic to vSFs. It forwards incoming packets to the intended vSF instance, based on the encapsulation header. Once packets processed, the vSF returns them to the connector, which applies a decision of transfer to the next vSF depending on the topology of the service. These transfer decisions are cached in the connector, indexed by a hash, calculated using flow information, thus ensuring flow affinity. The connector also exposes a control interface, used by the D-MANO, to populate the service- oriented routing table and the mapping between the service function and the vSF instance unicast address. This information is used by the connector to enforce the chaining

decisions for outbound traffic and to balance the load locally between vSF instances providing the same service (same prefix).

vSF: vSF instances process the packet flow based on the service provided. Once a packet has been processed, the vSF instance updates the chaining header to point to the next service. Each instance is controlled and monitored by the D-MANO.

Distributed MANO: The D-MANO controls and manages the other components of the NFV-R. It is configured with high-level strategies, which guide its autonomous orchestration decisions. It has two essential control functions. The first is to monitor vSF instances, evaluating vSF costs, and then inject them into the IGP using the router. The second function is to obtain IGP information from the router to build the service plan topology, calculate the service-oriented routing table, and insert it into the connector.

6.2.2.2 Implementation

We have implemented the proposed solution, and present the technical choices we made for each component of the architecture described in the previous section.

Encapsulation header: Our implementation uses the Network Service (NSH) header to handle traffic across different services (RFC8300). Our choice is motivated by the fact that NSH is an IETF standard explicitly designed for service chaining and is widely used in many open-source frameworks (eg, OPNFV, Opendaylight, ONOS, fd.io). In NSH, the Service Path Identifier (SPI) field uniquely identifies a set of abstract service functions (that is, the service function chain), while the service index (SI) points to the next function to which the packet is to be delivered. NSH also provides extensible metadata fields that we use to transmit the hash value used to uniquely identify a flow along its chain. This hash value is calculated at the classification step with the 5-tuple of the original packet.

IGP: We use the Open Shortest Path First (OSPF) protocol because it is widely used and easily scalable, thanks to the opaque Link State Advertisements (LSA) technology. Opaque LSAs are used to share information about vSF instances and links. Although opaque LSAs flooding increase the control traffic load, this does not affect OSPF stability because they do not trigger a new computation of the shortest path. We use opaque LSAs to transmit 3 information: the anycast address of an instance of Vsf, the associated vSF cost, and the NSH endpoint IP address (of the next connector). We chose to use a simple vSF metric: the remaining processing capacity of the vSF instance.

Service Dependent Path Calculation Algorithm: We chose to use the WCMP (Weighted Cost MultiPath) to compute the routing table that takes into account the services. It is particularly suitable for our anycast approach, as it helps to balance traffic based on vSF cost. We use network link costs and vSF costs to weight the paths to an anycast vSF address. WCMP combines network and vSF costs to assign weight to different instances of vSF. This weight is the probability that a new flow will be sent to a given vSF instance. Since the vSF cost is periodically updated, WCMP adapts to the load by distributing the traffic on the low-loaded instances (lower cost, therefore higher WCMP weight).

Router: We build our NFV-R on Linux and use network namespaces to isolate the components. We use FRRouting, a suite of open source IP routing protocols, to implement our OSPF router. In particular, we use FRRouting's proposed OSPF API to reflect the Link-State Database (LSDB) in D-MANO and to inject opaque vSF LSAs.

Connector: We implemented connector logic in P4, a programming language for the data plane. P4 was chosen for several reasons: it can support any header (eg NSH) and it is dynamic, allowing us to cache the routing decision in order to manage the flow affinity. Our P4 code is executed on the simple switch target. Its Command Line Interface is exposed to the D-MANO to configure the switch and populate the WCMP table at runtime.

vSF: The vSFs are implemented in the form of simple processes (using scapy) that analyze incoming packets, decrement their IT NSH field and return them to the connector. Since the initial implementation focused on the different components of the proposed approach, we deliberately chose to use simplistic vSFs. The Python psutil library allows us to monitor the resources used by vSF processes.

D-MANO: The D-MANO has been implemented in Python. First, it queries the use of local vSF instance resources to generate the associated costs. The costs are then advertised on the network with opaque vSF LSAs. Secondly, the D-MANO receives vSF announces from its LSDB. With these data, he creates a service view. Based on this topology, it calculates the WCMP weights and updates them on the connector.

6.2.2.3 Evaluation

We use three ISP topologies that were previously used in [6]. They are summarized in the following table. The first one was generated synthetically, while the other two were deduced in the Rocketfuel.

Topology	Nodes	Edges	Demands	Type
rf1755	87	372	7527	Rocketfuel
rf3967	79	294	6160	Rocketfuel
synth50	50	276	2449	Synthetic

We use the weights provided with the dataset to configure the IGP link costs. We consider all nodes as NFV-R capable of hosting vSF instances. The dataset also contains request matrices that we use for service chain requests. For each experiment, we randomly select 5% of global requests to build our service requests (input, output, and throughput). We run two types of scenarios. In first scenario, only one type of vSF is present on the network. There are 10 examples. In second scenario, all requests must be directed to type 1 vSF, then to type 2 vSF. There are 5 instances of each type. In both scenarios, the vSF instances are placed on the nodes that have the greatest centrality between them, that is, the nodes traversed by the largest number of shortest IGP paths.

We have deployed NFV-R emulated topologies on the Grid'5000 testbed. Grid'5000 is a versatile, large-scale tesbed that provides access to a large amount of resources (12,000 core processor) across sites and interconnected over a 10 GB/s WAN network. This test bench is highly reconfigurable, making it an excellent experimental research tool. In our experiments, we use Distem, a network emulation tool, to deploy NFV-R LXC on servers without display system. For each NFV-R, Distem uses Linux control groups to allocate 4 vCPUs to each NFV-R and connects NFV-R to VXLAN tunnels to emulate topology links. We conduct our experiments on a cluster of 48 nodes with the following host characteristics: Intel Xeon E5-2630L v4 (Broadwell, 1.80 GHz, 2 CPU/node, 10 cores/CPU), 10 GB Ethernet interface. Traffic is generated at the granularity of a UDP flow. We set the flow arrival rate, the duration and the size of the packets. The packet rate of each flow is then adapted to match the bandwidth of the request. Once the steady state is reached (i.e., when a first period of flow duration has elapsed), each request generates the dataset input bandwidth with k UDP flow, k corresponding to the duration of the flow divided by the arrival rate, we set at 50 s and 2 flow/s respectively.

We compare different update periods for LSA vSFs and discuss the tradeoff between network traffic control overheads induced by these LSAs and network dynamics. Figure 12 and Figure 13 show the additional

overheads generated by NFV-R with one and two types of vSF, respectively. At a time, there are at all 10 instances in the network. Logically, when the frequency is low, overhead costs are low. We can observe that it is slightly higher with two types of vSF than with one. Because routers must propagate separate LSAs. The control overhead must be discussed with the dynamics of path traffic because a low period of LSA leads to a less accurate view of the network at the router level. Figure 14 shows traffic distribution when there is a type of vSF on the largest topology. We can see that the load is well distributed on instances, also taking into account OSPF weights. When the LSA period increases, the distribution of traffic tends to be less stable but still fair. Figure 15 presents the results when there are two types of vSF. The load doubles on instances because there are five of each type. It can also be observed that the spread slightly increases with the increase of the LSA period. Finally we summarize in*Table 4*and *Table 5* the results of the experiment for the other two topologies with one and two vSF types. They are consistent with the previous comments.

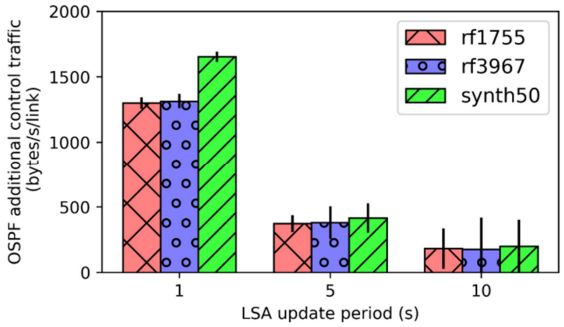


Figure 12: OSPF overhead for 1 vSF chains

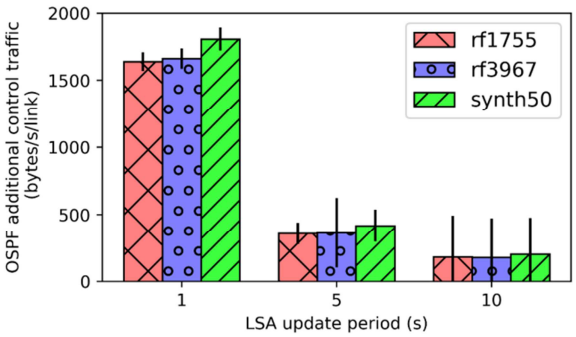


Figure 13: OSPF overhead for 2 vSF chains

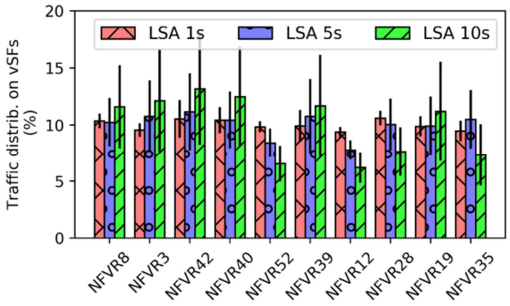


Figure 14: Traffic distribution over vSF for 1 vSF chains on RFI755 topology

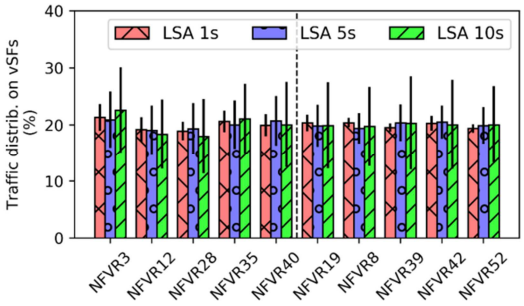


Figure 15: Traffic distribution over vSF for 2 vSF chains on RFI755 topology, type 2 on the right, type 1 on the left

Topology		RF3967			SYNTH50		
LSA Period		1s	5s	10s	1s	5s	10s
Worst NFV-R	Mean	12.20	12.25	11.25	10.33	10.36	11.05
	Std	0.27	0.78	1.68	1.80	3.57	6.01
Best NFV-R	Mean	7.70	8.38	8.60	9.80	9.62	7.46
	Std	0.59	1.55	2.21	1.74	3.15	4.56

Table 4: Traffic distribution (%) for 1 vSF chains.

Topology		RF3967			SYNTH50		
LSA Period		1s	5s	10s	1s	5s	10s
Worst NFV-R (Type 1 vSF)	Mean	22.34	20.99	21.09	23.92	27.08	31.38
	Std	2.04	2.75	4.61	2.84	6.30	10.50
Worst NFV-R (Type 2 vSF)	Mean	25.71	25.44	25.21	20.10	21.42	20.72
	Std	0.27	0.58	0.78	2.38	5.84	9.90
Best NFV-R (Type 1 vSF)	Mean	18.82	18.91	18.10	18.84	17.52	14.50
	Std	1.48	2.37	2.97	2.67	3.41	5.13
Best NFV-R (Type 2 vSF)	Mean	16.46	16.91	17.30	19.92	19.31	19.21
	Std	0.70	2.08	2.06	2.32	4.77	8.92

Table 5 Traffic distribution (%) for 2 vSF chains.

In this contribution, we proposed to increase the network routing layer. We are relying on the robustness and scalability of IGP, to steer traffic in service chains. We presented our solution, describe the architecture of our augmented node (called NFV-Router) and builds proof of concept, which can be progressively deployed on current networks. We evaluated our proposal on ISP topologies and demand matrices. In our large-scale emulation, the results show that with a small control traffic, NFV Routers drive traffic smoothly through the service chain.

7 CONCLUSIONS

This document provides a overview of the results of ANASTACIA Task 3.3, which aims to manage the Security Enforcement Enablers requirements established in high-level terms through the ANASTACIA architectural components and their integration.
First, the document discussed the progress of the network service management beyond the state of the art. Then, it thoroughly describes the inner core functionalities of the Security Enabler Provider and its different interactions with other components, and it also describes the Security Resource Planning component a sub component of the Security Orchestrator and its interaction with the other components. Finally, it presents the architecture and the implementation of security resource planning.

8 REFERENCES

[1] D. Belabed, M. Bouet, Rivera, P. Sobonsk, A. Molina Zarca ANASTACIA deliverable D3.3 – Initial Security Enforcement Enablers

[2] A Wion, M Bouet, L Iannone, V Conan, Let there be Chaining: How to Augment your IGP to Chain your Services, IFIP Networking Conference (IFIP Networking), 2019.

[3] D. R. (. T. T. I. F. (. D. B. (. C. C. (. A. S. J. B. A. M. J. O. (. R. M. P. (. A. E.-D. M. (. S. Ruben Trapero (ATOS), "D1.3 Initial Architectural Design".
AM Zarca, JB Bernabé, AS, K Yacine, y Belabed, S Bianchi "Initial Security Enforcement Manager Report".2018. H2020 Anastacia EU project deliverable D3.1. .

[4] Yacine Khettab, Ivan Farris, Diego Rivera, Rafael Marín Pérez, Jorge Bernal, Alejandro Molina, Dallal Belabed, Alie El-Din Mady, Piotr Sobonski, Deepak Mehta, ANASTACIA deliverable D3.2 – Privacy Risk Modelling and Contingency – Initial Report.

[5] T. Taleb, Y. Khettab, A. Laghrissi, D. Rivera, R.Marín Pérez, J. Bernal, A. Molina, D.Belabed, A.Mady, P.Sobonski, D. Mehta (UTRC), ANASTACIA deliverable D6.2 – Initial use cases implementation and tests Report.

9 ATTACHMENTS

9.1 EXAMPLE OF SNORT PLUGIN

```
1.  # # -*- coding: utf-8 -*-
2.  """
3.  This python module implements a MSPL->SNORT plugin inside the ANASTACIA European Project,
4.  extending the MSPL language defined in secured project.
5.  How to use:
6.     python3 mspl_snort.py [MSPL_FILE.xml]
7.
8.  """
9.  __author__ = "Dallal Belabed"
10. __copyright__ = "Copyright 2018, ANASTACIA H2020"
11. __credits__ = ["Dallal Belabed", "Alejandro Molina Zarca"]
12. __license__ = "GPL"
13. __version__ = "0.0.2"
14. __maintainer__ = "Dallal Belabed"
15. __email__ = "dallal.belabed@thalesgroup.com"
16. __status__ = "Development"
17.
18.
19. import sys,os
20. import json
21. import logging
22. logging.basicConfig(level=logging.INFO)
23. logger = logging.getLogger(__name__)
24. # Insert the parent path into the sys.path in order to import the mspl
25. parentPath = os.path.abspath("..")
26. if parentPath not in sys.path:
27.    sys.path.insert(0, parentPath)
28. import mspl
29.
30. """
31.    Output example:
32.    {"output":{alert icmp any any -
    > any any (msg:"ICMP_DoS ATTACK 1"; detection_filter: track by_src, count 50, seconds 10; sid:100001; rev:001;) \n
33. alert icmp any any -
    > any any (msg:"ICMP_DoS ATTACK 2"; detection_filter: track by_src, count 100, seconds 10; sid:100002; rev:001;)"}}
34. """
35.
36. class ITResourceType(mspl.ITResourceType):
37.
38.    def get_configuration(self):
39.       'Translate the ITResource element to SNORT configuration'
40.       capability_name = self.configuration.capability[0].Name
41.       logger.info("get_configuration capability_name...")
42.       if not capability_name in mspl.CapabilityType.itervalues():
43.          raise ValueError('The capability {} does not exist, please try with the current supported capabilities'.format(capabilit
    y_name))
44.       # In filtering case, call translation for filtering Action and filtering conf condition
45.       try:
46.          capability_configuration_function = getattr(self, "get_{}_configuration".format(capability_name))
47.       except AttributeError as e:
48.          print("Currently, {} is not implemented.".format(capability_name))
49.          print(e)
50.          sys.exit(1)
51.
```

```
52.         action, capability_enabler_conf = capability_configuration_function()
53.         # Build the enabler configuration, since the action is mandatory is not neccesary to check it
54.         enabler_conf = "{} {}".format(action, capability_enabler_conf if capability_enabler_conf else "")
55.         return enabler_conf
56.
57.     def get_Snort_configuration(self):
58.         'Returns the configuration for Filtering_L4 capability'
59.         # For filtering it is only needed the first configuration rule
60.         try:
61.             configuration_rule = self.configuration.configurationRule[0]
62.         except:
63.             print("Filtering_L4 capability requires at least a configurationRule element")
64.             sys.exit(1)
65.         action = configuration_rule.configurationRuleAction.get_configuration()
66.         filtering_enabler_conf = configuration_rule.configurationCondition.get_configuration()
67.
68.         return (action,filtering_enabler_conf)
69.
70.
71.
72. class FilteringAction(mspl.FilteringAction):
73.     def get_configuration(self):
74.         'Returns the filtering action for SNORT'
75.         FILTERING_ACTION = {"Alert": "alert", "Log":"log", "Pass":"pass"}
76.         return FILTERING_ACTION.get(self.FilteringActionType,"DROP")
77.
78. class FilteringConfigurationCondition(mspl.FilteringConfigurationCondition):
79.     def get_configuration(self):
80.         'Returns the traslated filtering SNORT parameters'
81.         filtering_enabler_conf = ""
82.         packet_filter_condition = self.packetFilterCondition
83.         State_ful_Condition = self.statefulCondition
84.         time_Condition = self.timeCondition
85.         if not packet_filter_condition:
86.             return None
87.         if not State_ful_Condition:
88.             return None
89.         if not time_Condition:
90.             return None
91.
92.         filtering_enabler_conf+=" {}".format(packet_filter_condition.ProtocolType) if packet_filter_condition.ProtocolType el
    se ""
93.         filtering_enabler_conf+=" {}".format(packet_filter_condition.SourceAddress) if packet_filter_condition.SourceAddress
     else ""
94.         filtering_enabler_conf+=" -
    > {} ".format(packet_filter_condition.SourcePort) if packet_filter_condition.SourcePort else ""
95.         filtering_enabler_conf+=" {} ".format(packet_filter_condition.DestinationAddress) if packet_filter_condition.Destinati
    onAddress else ""
96.         filtering_enabler_conf+=" {}".format(packet_filter_condition.DestinationPort) if packet_filter_condition.DestinationP
    ort else ""
97.
98.
99.         filtering_enabler_conf+=" (msg:\"{}\"".format(State_ful_Condition.Message) if State_ful_Condition.Message else ""
100.        filtering_enabler_conf+="; detection_filter:{}".format(State_ful_Condition.Detection_filter) if State_ful_Condition.Det
    ection_filter else ""
101.        filtering_enabler_conf+=", count {}".format(State_ful_Condition.Count) if State_ful_Condition.Count else ""
102.        filtering_enabler_conf+=", seconds {}".format(time_Condition.Time) if time_Condition.Time else ""
103.        filtering_enabler_conf+="; sid:{}".format(State_ful_Condition.Sid) if State_ful_Condition.Sid else ""
104.        filtering_enabler_conf+="; rev:{}".format(State_ful_Condition.Rev) if State_ful_Condition.Rev else ""
105.        filtering_enabler_conf+=";)"
106.
107.
108.        return filtering_enabler_conf if filtering_enabler_conf else None
```

```
109.
110. class M2LPlugin:
111.     'SNORT Medium to low plugin implementation'
112.
113.     def get_configuration(self,mspl_source):
114.         'Return the SNORT configuration from the mspl_source'
115.         # Customize the pyxb generated classes with our own behavior
116.         global mspl
117.         mspl.ITResourceType._SetSupersedingClass(ITResourceType)
118.         mspl.FilteringAction._SetSupersedingClass(FilteringAction)
119.         mspl.FilteringConfigurationCondition._SetSupersedingClass(FilteringConfigurationCondition)
120.         #Load the xml
121.         it_resource = mspl.CreateFromDocument(mspl_source)
122.         # Start the parser process using the xml root element
123.         return it_resource.get_configuration()
124.
125. if __name__ == "__main__":
126.
127.     #output file
128.     outputfile = "local.rules";
129.     source = open(outputfile, "w")
130.     # Pretty print
131.     separator = "*"
132.     separator_group = separator*5
133.     title = "SNORT CONFIGURATION"
134.     print("\n{}{}{}".format(separator_group,title,separator_group))
135.
136.     for i in range (1,len(sys.argv)):
137.         print (sys.argv[i])
138.         xml_file = sys.argv[i]
139.         # Instantiate the plugin
140.         m2lplugin = M2LPlugin()
141.         logger.info("Reading mspl file...")
142.         xml_source = open(xml_file).read()
143.         logger.info("Translating mspl to SNORT...")
144.         enabler_configuration = m2lplugin.get_configuration(xml_source)
145.         print(enabler_configuration)
146.         source.write(enabler_configuration+ "\n")
147.     print("{}{}{}\n".format(separator_group,separator*len(title),separator_group))
148.     print ("Snort file path: " + os.path.abspath(outputfile))
149.
150.     source.close()
```

9.2 EXAMPLE OF SECURITY ENABLER FOR BMS.4 USE CASE

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
<ITResource
   xmlns="http://modeliosoft/xsddesigner/a22bd60b-ee3d-425c-8618-beb6a854051a/ITResource.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://modeliosoft/xsddesigner/a22bd60b-ee3d-425c-8618-
beb6a854051a/ITResource.xsd ANASTACIA_MSPL_XML_Schema.xsd">
   <configuration xsi:type='RuleSetConfiguration'>
      <capability>
         <Name>IoT_control</Name>
      </capability>
      <configurationRule>
         <configurationRuleAction xsi:type='PowerMgmtAction' >
          <PowerMgmtActionType>OFF</PowerMgmtActionType>
         </configurationRuleAction>
         <configurationCondition xsi:type='FilteringConfigurationCondition'>
            <isCNF>false</isCNF>
            <packetFilterCondition>
               <DestinationAddress>2001:720:1710:4::2</DestinationAddress>
               <DestinationPort>5683</DestinationPort>
            </packetFilterCondition>
            <applicationLayerCondition xsi:type='IoTApplicationLayerCondition'>
              <URL>.power_off</URL>
             <method>PUT</method>
            </applicationLayerCondition>
         </configurationCondition>
         <externalData xsi:type='Priority'>
            <value>0</value>
         </externalData>
         <Name>Rule0</Name>
         <isCNF>false</isCNF>
      </configurationRule>
      <resolutionStrategy xsi:type='FMR'/>
      <Name>MSPL_b22c6384-ed08-487b-a3ca-ce2e557ca434</Name>
   </configuration>
</ITResource>
```

10 USE CASES DESCRIPTION FOR THE SECURITY ENFORCEMENT ENABLERS

In the section we give a description for each use case defined in the deliverable D6.2 the role of these two components; the Security Enabler Provider and the Security Resource Planning:

10.1 BMS.2: INSIDER ATTACK ON THE FIRE SUPPRESSION SYSTEM

The main objective of this use-case is to evaluate ANASTACIA framework for protecting the system from an insider attack and avoid any damage to the building assets. In this use-case, the attacker exploits the building operations workstation to request the activation of fire alert system managed by an IoT device. To cope with this attack, the **Cooja simulator through a Cooja VNF agent** will be used as main Security Enabler in order to replicate the current IoT affected environment, isolating the attacker into a virtual environment allowing the system administrator to evaluate the risk and impact of the unauthorized attempts generated by the attacker.

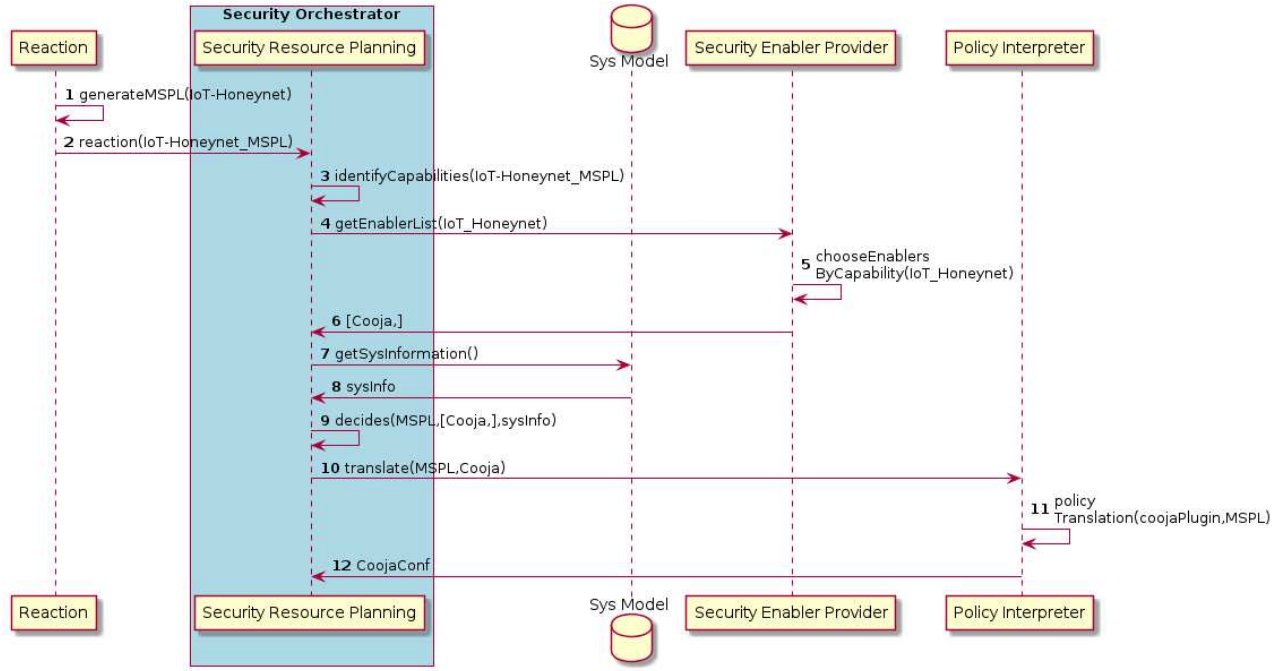


Figure 16: IoT-Honeynet policy deployment

Figure 16: IoT-Honeynet policy deployment shows the main workflow for the policy IoT-Honeynet policy deployment for this use case:

1. The Reaction module generates an IoT-Honeynet MSPL policy.
2. The Reaction module sends the MSPL policy to the Security Orchestrator.
3. The Security Resource Planning in the Security Orchestrator identifies IoT_Honeynet as the main capability for the MSPL.
4. The Security Resource Planning request the list of the Security Enabler candidates to the Security Enabler Provider, indicating IoT_Honeynet as the main identified capability.

5. The Security Enabler Provider performs a capability matching, generating the list of Security Enabler candidates.
6. The Security Enabler Provider returns the list including **Cooja** as main candidate.
7. The Security Resource Planning request the system information to the system model database.
8. The Security Resource Planning retrieves the requested information.
9. The Security Resource Planning makes decides to use Cooja as Security Enabler taken into account the status of the network and the available resources.
10. The Security Resource Planning request a policy translation specifying Cooja as Security Enabler.
11. The Policy Interpreter performs the policy translation, generating the Cooja Security Enabler configuration.
12. The Security Resource Planning receives the Cooja configuration and then deliver it to the Security Orchestrator which will be in charge to enforce the Security Enabler configuration.

Regarding the Cooja Security Enabler configuration, the Cooja simulation environment allows generating custom environments, since the IoT device specification up to more advanced network configurations like the transmitting range, or the success ratio.

```
<simconf>
 ...
 <simulation>
  <title>IoT-Honeynet-MSPL_b22c6384-ed08-487b-a3ca-ce2e557ca434</title>
  <radiomedium>
    ...
   org.contikios.cooja.radiomediums.UDGM
   <transmitting_range>15.0</transmitting_range>
   <interference_range>15.0</interference_range>
   <success_ratio_tx>1.0</success_ratio_tx>
   <success_ratio_rx>1.0</success_ratio_rx>
  </radiomedium>
  ...
  <motetype>
   se.sics.cooja.mspmote.WismoteMoteType
   <identifier>1</identifier>
   <description>CoAP Server</description>
   <source>[SOURCE_DIR]/coap-server.c</source>
   <commands>make coap-server.wismote TARGET=wismote</commands>
   <firmware>[FIRMWARE_DIR]/coap-server.wismote</firmware>
   ...
   <moteinterface>org.contikios.cooja.interfaces.Position</moteinterface>
   <moteinterface>org.contikios.cooja.mspmote.interfaces.MspMoteID</moteinterface>
   ...
  </motetype>
  <mote>
   <breakpoints />
   <interface_config>
    se.sics.cooja.interfaces.Position
    <x>33.260163187353555</x>
    <y>30.643217359962595</y>
    <z>0.0</z>
   </interface_config>
   <interface_config>
    se.sics.cooja.mspmote.interfaces.MspMoteID
    <id>1</id>
   </interface_config>
   <motetype_identifier>1</motetype_identifier>
  </mote> ...
</simconf>
```

Figure 17: Cooja configuration example

Figure 17: Cooja configuration example shows a Cooja configuration example, which is defining a mote type for an IoT device that implements CoAP, as well as some IoT device attribute like the physical location and the unique identifier. Once the Security Orchestrator receives the configurations it will enforce the configuration over the architecture.

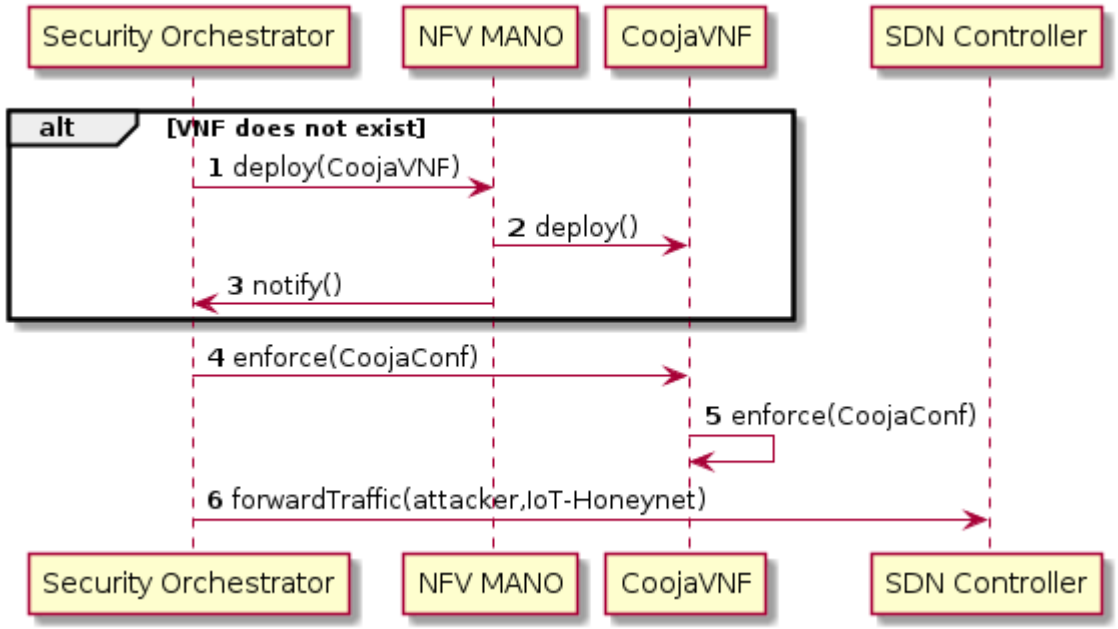


Figure 18: Cooja deployment

Figure 18: Cooja deployment shows the IoT-Honeynet deployment process:

1. If the Cooja VNF which implements the Security Enabler is not already running, the Security Orchestrator will request the Cooja deployment to the NFV MANO.
2. The NFV MANO then deploys the Cooja VNF which contains the Cooja agent.
3. The NFV MANO notifies the Security Orchestrator when the VNF is correctly deployed.
4. The Security Orchestrator requests the Cooja configuration enforcement.
5. The Cooja agent in the starts the simulation using the specific Cooja configuration.
6. The Security Orchestrator request to the SDN Controller a traffic forwarding in order to redirect the traffic from/to the attacker to the virtual environment.

10.2 MEC.3: DoS/DDoS ATTACKS USING SMART CAMERAS AND IoT DEVICES

The main objective of the MEC.3 use case is to validate ANASTACIA system against malicious IoT attackers that will attempt to denial of service through cameras system. In this use case a group of hackers gets the IP address of IoTs and cameras and use if for DDoS attack. Thus, this section identifies the different steps of test-cases to implement and evaluate UC_MEC.3 on Mobile (Multi-access) Edge Computing testbed. The following section will describe the role of Security Enabler Provider and the Security Resource planning modules in the use case. In this case, the process is compounded by the following steps:

1. The Reaction sends a MSPL file with the selected capability to Security Orchestrator.
2. The security orchestrator requests the list of security enablers from the security enabler provider.
3. The security enabler provider returns the list of available enablers to the security orchestrator.
4. The security orchestrator requests from the system the enabler flavours from OpenStack and the list of nodes that can host the security enablers.

5. The security resource Planning selects the security enabler and send it to the security orchestrator that will send it plus the MSPL file received from the reaction module to the policy interpreter.
6. The policy interpreter requests the security Enabler plugin from the security enabler provider.
7. The security enabler provider provides the appropriate plugin to the policy interpreter.
8. The policy interpreter performs the translation and returns the security enabler configuration to the Security Orchestrator.
9. The security orchestrator based on the enabler configuration will enforce the configuration.

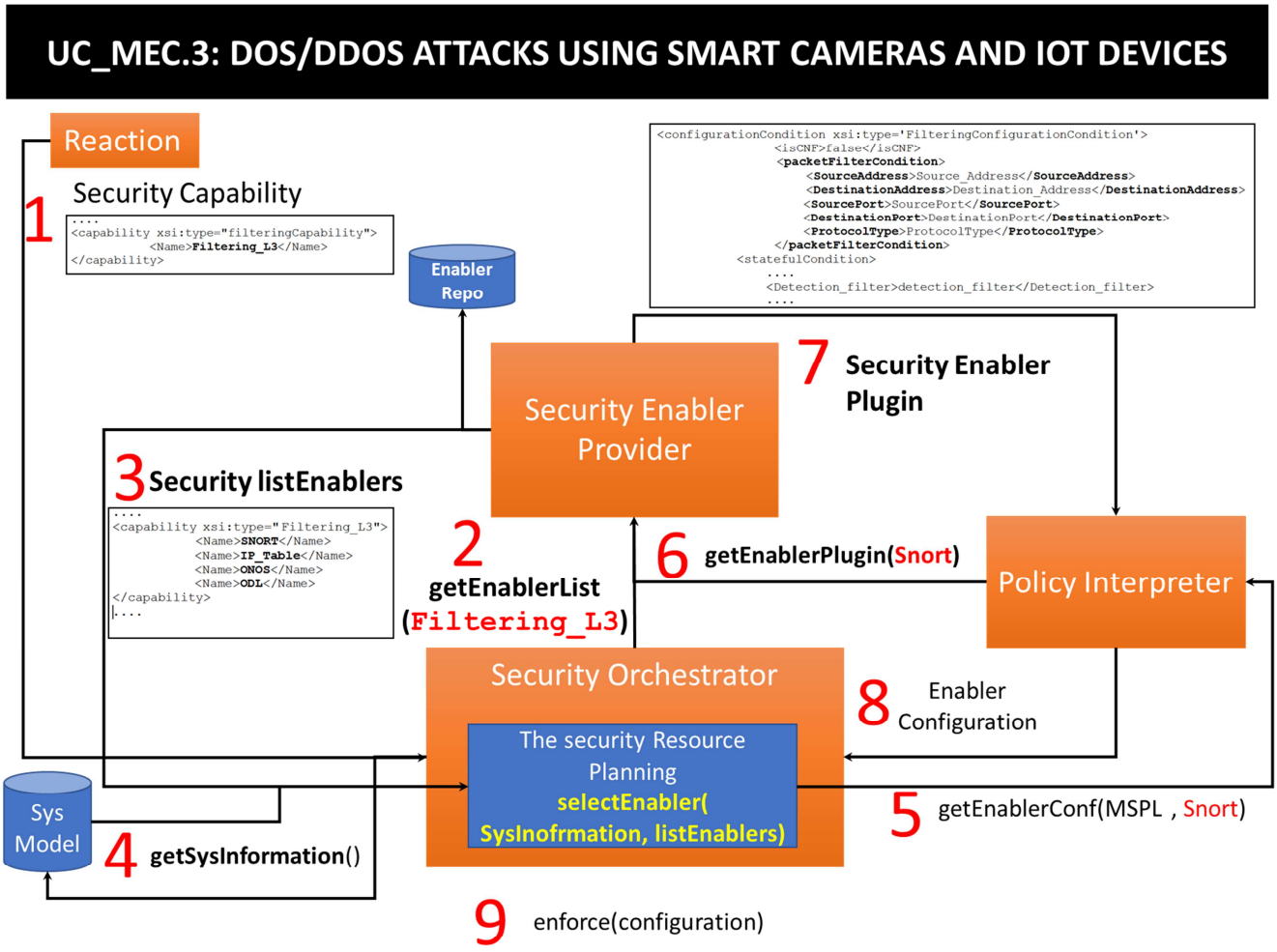


Figure 19 : Use case MEC.3: DOS/DDOS attacks using smart camera and IoT devices

Once the reaction component chooses the mitigation strategy to apply, in this use case the "filtering layer three" capability is chosen. This capability will be sent to the Security Orchestrator Figure 19 step 1 in form of an MSPL file, then the security orchestrator will request the list of security enablers that have the previous filtering capability, "*getEnablerList(Filtering)*" Figure 19 (step 2). As soon as, the security enabler provider receives the request it will query the list of enablers from the Enabler Repository and returns the list of available enablers; "IPtables, Different controllers such as (ONOS, Open DayLight), programmable switches as OpenVSwitch, Network Intrusion Detection & Prevention System as SNORT, etc." to the security orchestrator Figure 19 (step 3), which will transmit it to the Security resource planning. The security orchestrator based on the list of enablers will request from the system model the enabler flavours from Openstack and the list of nodes that can host the enablers Figure 19 (step 4). The security resource Planning will execute an integer linear programming (ILP) based on the system model information and the list of security enablers, and it will select the enabler based on different criteria as the infrastructure capacities. The selected enabler will be sent by the resource planning to the security orchestrator that will send it plus the MSPL file received from the reaction module to the policy interpreter Figure 19 (5). The

policy interpreter will request the security Enabler plugin from the security enabler provider Figure 19 (6). The security enabler provider will provide the appropriate plugin to the policy interpreter Figure 19 (7). The policy interpreter performs the translation and returns the security enabler configuration to the Security Orchestrator Figure 19 (8). Finally, the security orchestrator based on the enabler configuration will enforce the configuration Figure 19 (9).

10.3 BMS.3: REMOTE ATTACK ON THE BUILDING ENERGY MICROGRID

The BMS.3 describes the attack on a building energy microgrid system. In particular, this scenario copes with an SQL Injection attack on a SCADA server of the building. To this end, two principal security enablers will be selected by the Security Enabler Provider in this scenario:

- MMT-Probe and MMT-Security (as DPI engine)
- ONOS (as enabler to enforce the reaction to the attack)

In the following a description of the two MMT enablers is given:
MMT-Probe and MMT-Security: MMT-Probe is a DPI-based tool that allows testing security properties in live network traffic. Using the DPI technique, MMT-Probe us capable of capturing the traffic of the monitored link, read each packet and extract information from it. The set of data extracted from the packet (principally related with the status of the communication) are then fed to the MMT-Security module, in order to test security properties. In addition, MMT-Probe is also capable of aggregating the extracted information in order to generate statistics reports regarding the detected connection on the network. All the extracted and the statistics reports are supplied to other monitoring components in the ANASTACIA framework. In the case of the BMS.3 use case, an already-deployed and -configured instance of MMT will be used as a DPI monitoring module in order to detect attacks in the security enforcement plane. Considering this, the Security Orchestrator does not have to interact with this instance(in forms of configurations) to focus the analysis of the network, since the MMT software will actively monitor the incoming link of the attack target.
The aforementioned utilities are supplied in form of a virtual image that is ready to be deployed in the monitored network.
Any instance of the MMT-Probe can be configured to match the analysis requirements of the particular scenario. This is done by adjusting the configuration options that are provided to the new MMT instance. For further reference regarding how to configure MMT-Probe, please refer to D3.2.

10.4 BMS.4: CASCADE ATTACK ON A MEGATALL BUILDING

The BMS.4 use case describes a scenario in which the adversary is attacking a mega-tall building with an IoT infrastructure. The attacker manipulates the temperature sensors' values to very high levels which will trigger the fire and evacuation alarms on targeted floors, causing physical damage to the building infrastructure. ANASTACIA reacting to attack will use relevant MSPL policies to counter thread. ONOS will use policies to enforce adherence to specific network flows in SEP while Security Orchestrator will instruct the IoT controller to turn off/deactivate the IoT device.

1. In this scenario security enablers selected by the SEP are following:
    - IoT Controller – will enable/disable IoT temperature sensor,
    - ONOS – will control SDN traffic flow to IoT device,
2. The enabler configuration will be defined in this fashion:

IoT Controller (example listed in 0):

- Sensor identifier (IP address/port) that will be required to identify sensor in IoT Network.
- Action type: enable – turn on/activate IoT sensor, disable – turn off/deactivate IoT sensor,

SDN flow control (ONOS):

- After selecting the enabler by the security resource planning, using the relevant MSPL policy, the Security Orchestrator instructs the SDN controller to drop the malicious traffic, by enforcing the configuration given by the Policy Interpreter through the Security Enabler Provider
- This malicious traffic is identified using the IP source (The adversary) and IP destination (The target IoT device),
- Action type: drop the malicious traffic at the edge OVS (Open Virtual Switch) in order to avoid spreading it over the network.