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

D3.3

Initial Security Enforcement Enablers Report

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

Distribution level	PU
Contractual date	31.07.2018 [M19]
Delivery date	06.08.2018 [M20]
WP / Task	WP3 / T3.3
WP Leader	THALES SIX GTS France
Authors	D. Belabed, M. Bouet (THALES SIX GTS FRANCE); D. Rivera (Montimage) ; P. Sobonski (UTRC); A. 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

Task 3.3 is about the Security Enforcement manager, which is a component of the Security Orchestrator plan in the ANASTACIA architecture [1].

In this deliverable, we present the first results of “Initial Security Enforcement Enablers”, which tackles the Task 3.3 within WP3.

More precisely, this deliverable focuses on the development of the core enablers for the deployment of 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 third WP3 report. This “Initial Security Enforcement Enablers” report focuses on the development of core enablers for the deployment and implementation phase of the security enforcement manager. 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].

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 [1]. Besides; the module will provide 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”.

Now, the definition of the Task 3.3 is stable, relevant plugins for the refinement of MSPL security in low-data configuration were 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 first 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 Anastacia architecture, contextualizing the Security Enforcement Enablers in the ANASTACIA framework.
- Section 3 defines the refinement of the Security Enforcement Enablers components and its interaction with the other Anastacia components,
- Section 4, describes the first implementation of the Security Enforcement Enablers components.
- Section 5 gives a description of the role of the Security Enforcement Enablers components for each use case defined in the deliverable D6.2.
- Finally, section 6 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.

1.3 Revision History

Version	Date	Author	Description
0.1	17.05.2018	Dallal Belabed (TCS)	Table of contents
0.2	06.06.2018	Dallal Belabed (TCS) Diego Rivera (Montimage)	MEC.3 and BMS.2 use cases description regarding the security enforcement enablers
0.3	11.06.2018	Piotr Sobonski (UTRC)	BMS.4 use case description content
0.4	12.06.2018	Alejandro Molina Zarca (UMU)	Interactions with Policy Interpreter
0.5	20.06.2018	Dallal Belabed (TCS) Mathieu Bouet (TCS)	Security Enforcement Enablers in the ANASTACIA Architecture description and Integration of the contributions of the different partners.
0.6	15.07.2018	Dallal Belabed (TCS) Mathieu Bouet (TCS)	Revision of Security Resource Planning Module and Integration of the First Security Enforcement Enablers Implementation
0.7	19.07.2018	Dallal Belabed (TCS) Mathieu Bouet (TCS)	Initial full draft of deliverable
0.8	27.07.2018	Dallal Belabed (TCS)	Internal revision of the document
0.9	31.07.2018	Alejandro Molina Zarca(UMU), Piotr Sobonski (UTRC)	Revised version
1.0	31.07.2018	Dallal Belabed (TCS)	Final version released, ready to be delivered

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 Security Enforcement Enablers in the ANASTACIA Architecture

2.1 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.
	Security Enabler Provider

Component providing the interface	**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	

2.2 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.

3 REFINEMENT SECURITY ENFORCEMENT ENABLERS

3.1 SECURITY ENABLER PROVIDER INTERACTIONS

3.1.1 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 [5], and also published in journal paper [1] (formerly introduced in conference paper [4]).

3.1.1.1 List of Security Enabler candidates

In Figure 1, 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.

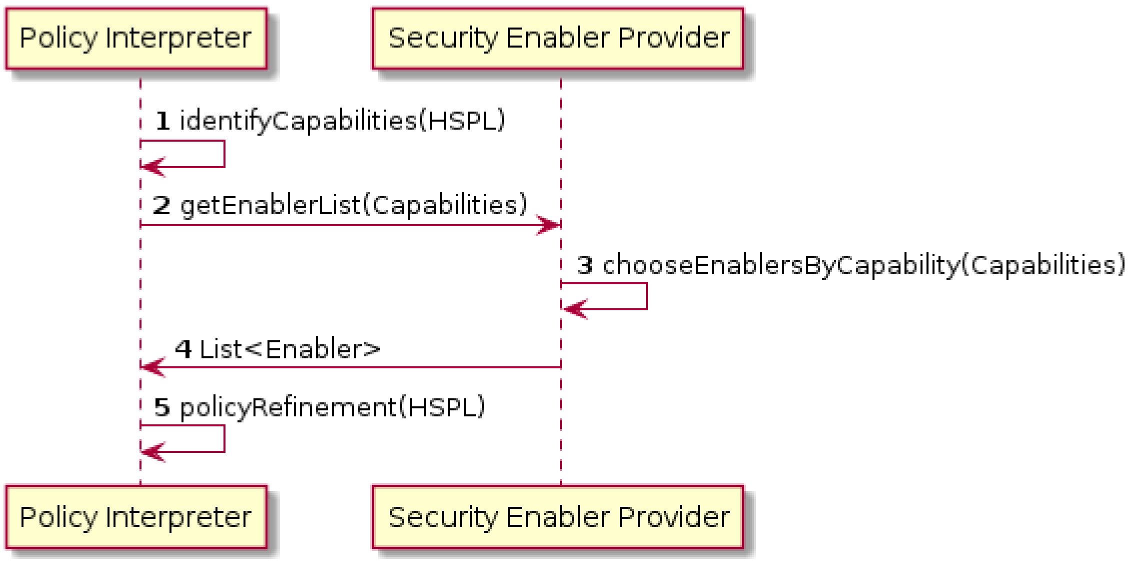

Policy Interpreter
Security Enabler Provider
1 identifyCapabilities(HSPL)
2 getEnablerList(Capabilities)
3 chooseEnablersByCapability(Capabilities)
4 List<Enabler>
5 policyRefinement(HSPL)
Policy Interpreter
Security Enabler Provider

Figure 1: 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.

3.1.1.2 Get the Security Enabler Plugin

In **Errore. L'origine riferimento non è stata trovata.** 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.

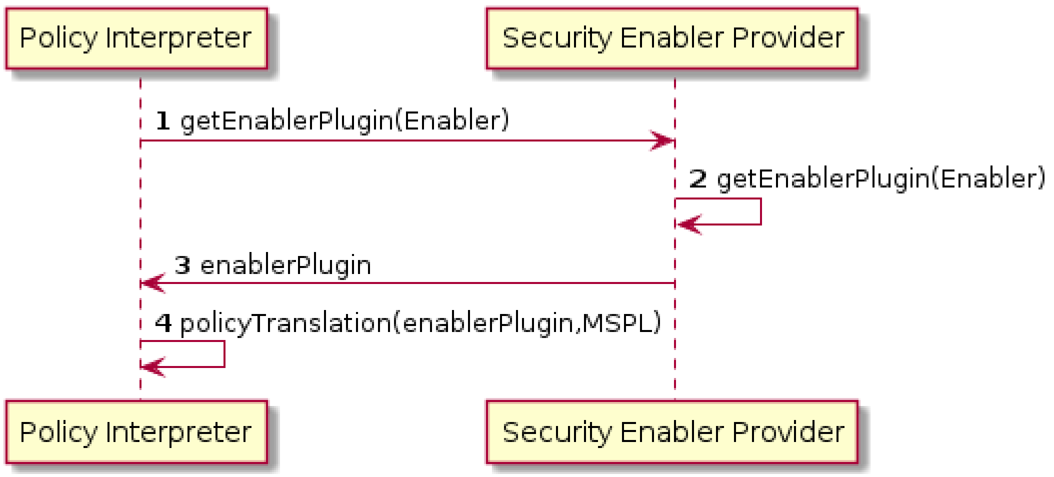

Policy Interpreter
Security Enabler Provider
1 getEnablerPlugin(Enabler)
2 getEnablerPlugin(Enabler)
3 enablerPlugin
4 policyTranslation(enablerPlugin,MSPL)
Policy Interpreter
Security Enabler Provider

Figure 2: Security Enabler Plugin (Interactions with Policy Interpreter)

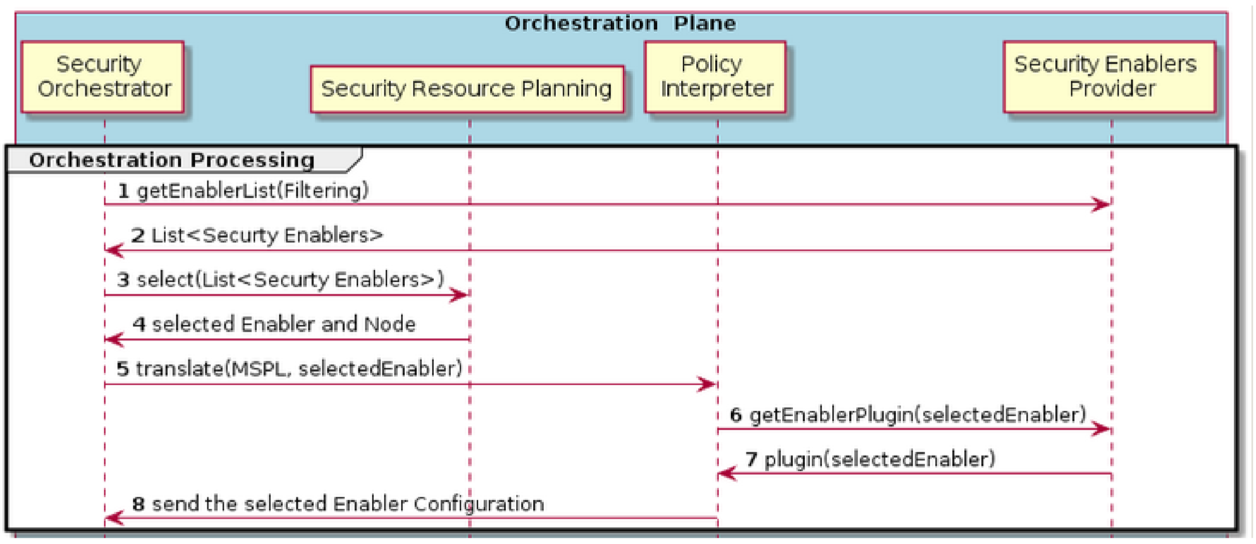
The process is compounded by the following steps:

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.

3.1.2 Interactions with the Security Orchestrator

3.1.2.1 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 **Errore. L'origine riferimento non è stata trovata.**.



Orchestration Plane
Security Orchestrator
Security Resource Planning
Policy Interpreter
Security Enablers Provider
Orchestration Processing
1 getEnablerList(Filtering)
2 List<Securty Enablers>
3 select(List<Securty Enablers>)
4 selected Enabler and Node
5 translate(MSPL, selectedEnabler)
6 getEnablerPlugin(selectedEnabler)
7 plugin(selectedEnabler)
8 send the selected Enabler Configuration

Figure 3: 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.

3.2 Security Resource Planning module Interactions

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

3.2.1 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.

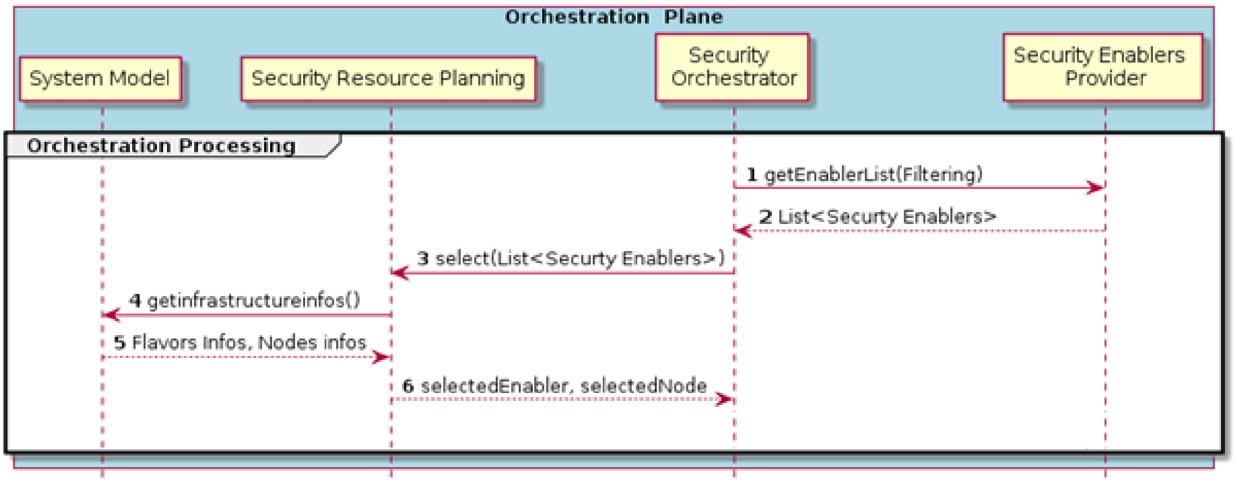

Orchestration Plane
System Model
Security Resource Planning
Security Orchestrator
Security Enablers Provider
Orchestration Processing
1 getEnablerList(Filtering)
2 List<Securty Enablers>
3 select(List<Securty Enablers>)
4 getinfrastructureinfos()
5 Flavors Infos, Nodes infos
6 selectedEnabler, selectedNode

Figure 4: Security Resource Planning Module Interactions with the Security Orchestrator

The aforementioned process is compounded by the following steps, Figure 4:

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.

4 First Security Enforcement Enablers Implementation

4.1 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 5 shows an example of Security Enforcement Enablers functionalities.

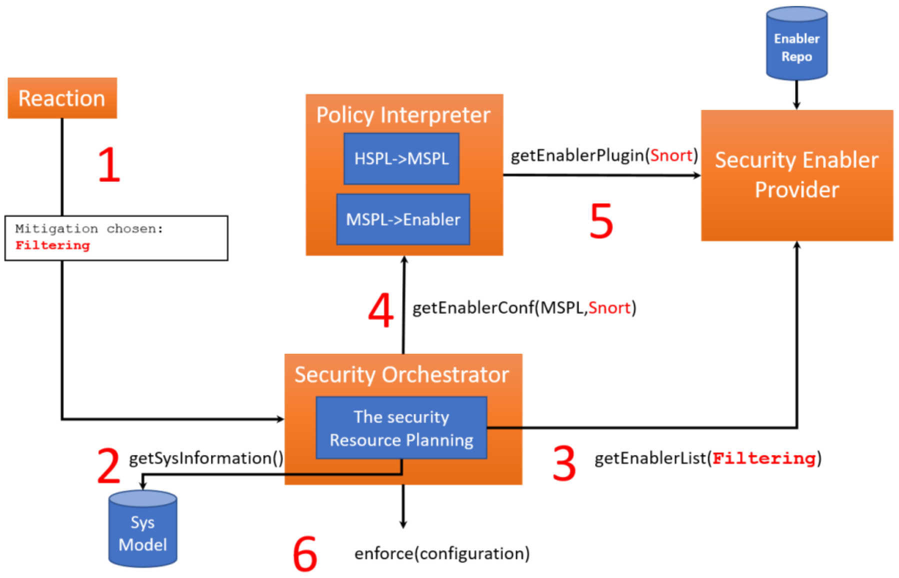


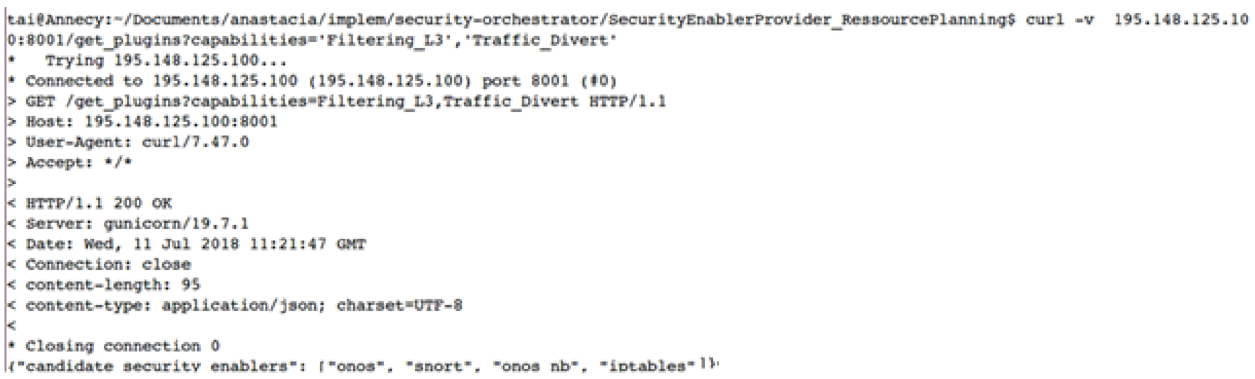
Figure 5: Security Enforcement Enablers Implementation

4.1.1 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 5 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 5 (1). This capability will be sent to the security enabler provider "*getEnablerList(Filtering)*", Figure 5 (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 6: 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 6: Security Enabler Provider (list of the security enabler requests)

4.1.2 Security Enabler

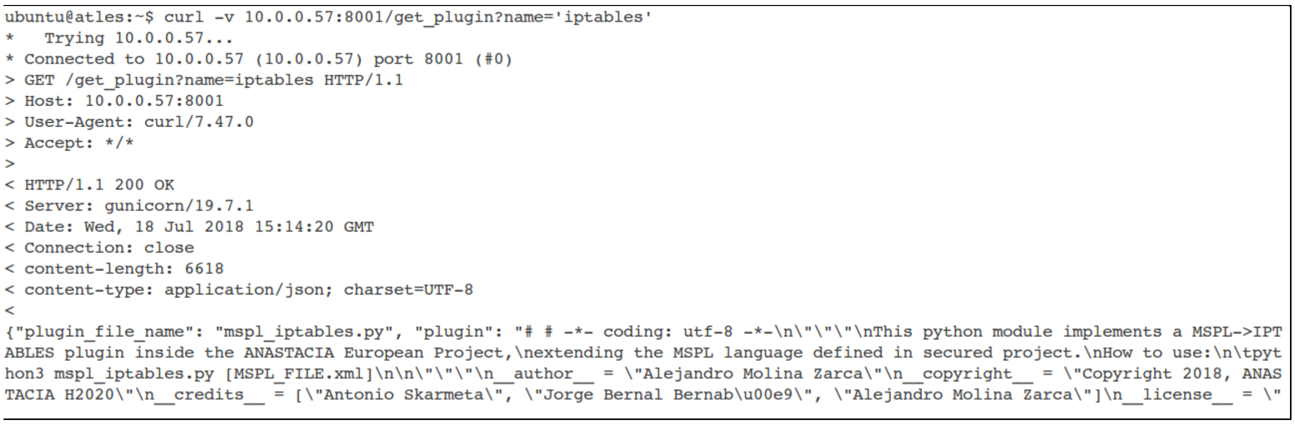
When the security resource planning selects the enabler, the orchestrator will send the selected enabler identification with an MSPL file that encloses the appropriate configuration. In the example the chosen enabler is "*Snort*" Figure 5: Security Enforcement Enablers Implementation (4). When the security enabler provider receives the request, it will provide the specific plugin for the selected security enabler, Figure 5: 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 5: 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 8.1:

1. **SDN ONOS Plugin:** It will take in charge the MSPL policy translation onto a set of rules understandable by the ONOS controller.
2. **SDN ODL Plugin:** It will take in charge the MSPL policy translation onto a set of rules understandable by the OpenDayLight controller.
3. **IPTABLES Plugin:** It will be able to translate the MSPL policy onto iptables rules.
4. **OpenWRT Plugin:** It will be in charge to generate OpenWRT router configuration from an MSPL policy.
5. **Quagga Plugin:** It will be in charge to generate Quagga router configuration from an MSPL policy.
6. **Open Virtual Switch Plugin:** It will generate OVS rules according to the specified policy.
7. **PfSense Plugin:** It will take in charge to generate the PfSense firewall configuration.
8. **SNORT Plugin:** It will translate the MSPL policy onto a SNORT monitoring rules.
9. **DTLS/TLS Proxy Plugin:** It will obtain the proxy configuration for DTLS/TLS secure connections.
10. **VPN Plugin:** It will generate the OpenVPN configuration from the secure policies parameters.
11. **Kippo Plugin:** It will perform the translation from MSPL to Kippo honeypot.
12. **Cooja Plugin:** It will translate the physical IoT system model provided to a Cooja virtual one.

13. **vAAA XACML Plugin:** It will generate the required configuration for the AAA infrastructure from the MSPL policy.
14. **Data Aggregator Proxy Plugin:** It will translate the security policy to a configuration of a data aggregator proxy in order to provide pseudonimity.
15. **IoT Control Plugin:** This plugin will be in charge of translating the operational IoT security policies to the specific IoT Controller rules.

Moreover, an as the module has been developed as an independent software an API was implement an example of the API is shown in Figure 7.



```
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 7: Security Enabler API

4.2 Security Resource Planning module

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.

4.2.1 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).

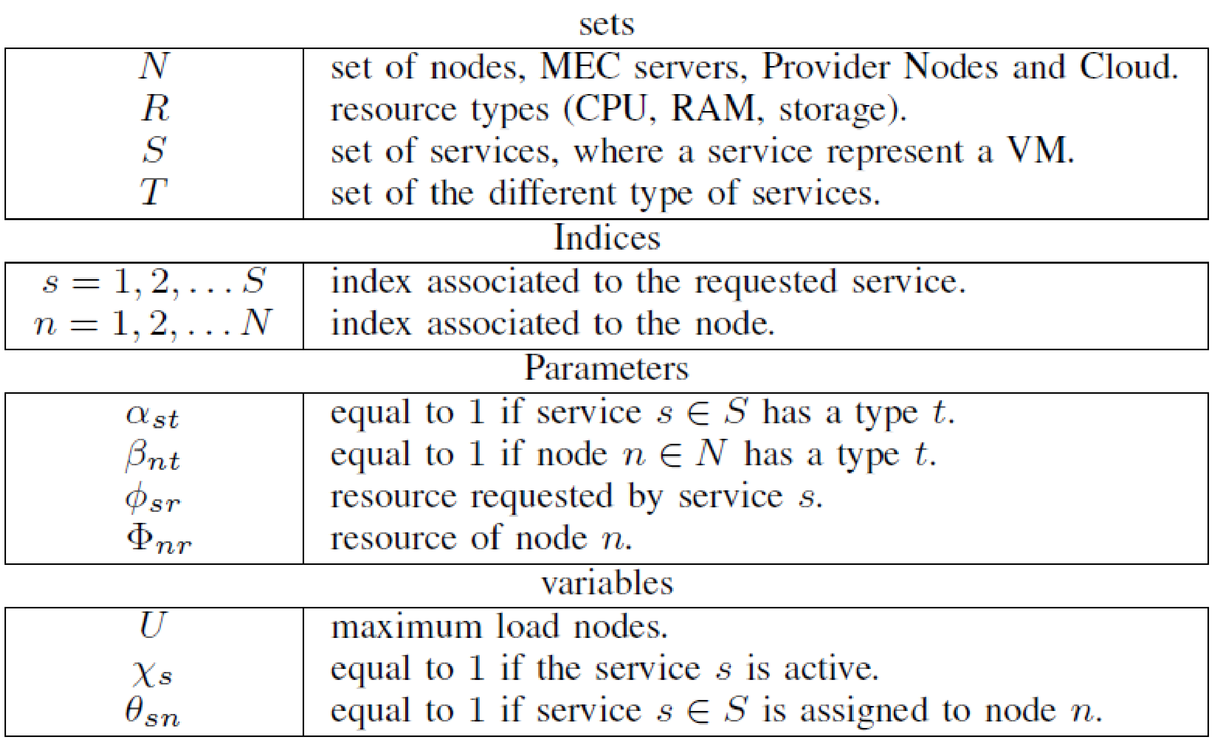
Subject to:

- 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.

4.2.2 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
$s = 1, 2, \dots S$	index associated to the requested service.
$n = 1, 2, \dots N$	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 8: Mathematical notations

The Figure 8 reports the mathematical notations used in the following Mixed Integer Linear Programming (MILP) formulation of the problem.

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.

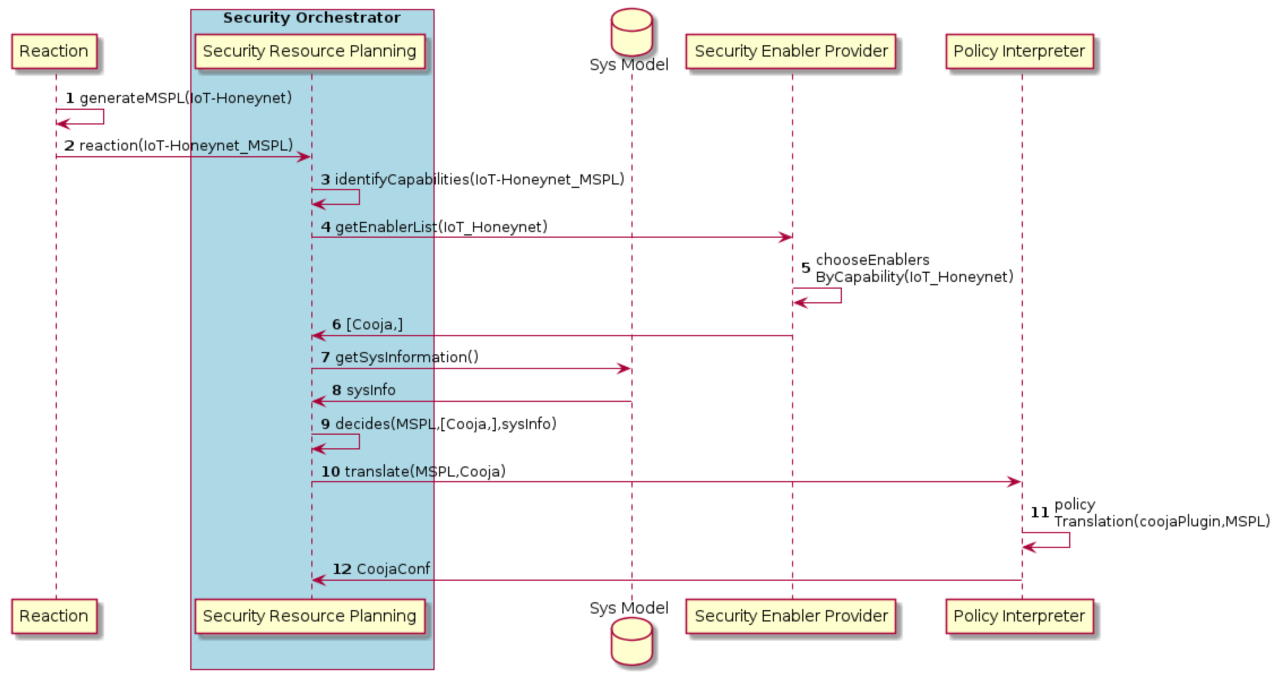
The security Resource Planning Module is implemented as an autonomous plugin that receives a list of the enablers and the topology information from the System Model, based on that information we run the ILP implemented using IBM CPLEX Optimizer engine that gives the selected security enablers that cope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

5 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:

5.1 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.



Security Orchestrator
Reaction
Security Resource Planning
Sys Model
Security Enabler Provider
Policy Interpreter
1 generateMSPL(IoT-Honeynet)
2 reaction(IoT-Honeynet_MSPL)
3 identifyCapabilities(IoT-Honeynet_MSPL)
4 getEnablerList(IoT_Honeynet)
5 chooseEnablers ByCapability(IoT_Honeynet)
6 [Cooja,]
7 getSysInformation()
8 sysInfo
9 decides(MSPL,[Cooja,],sysInfo)
10 translate(MSPL,Cooja)
11 policy Translation(coojaPlugin,MSPL)
12 CoojaConf

Figure 9: IoT-Honeynet policy deployment

Figure 9: 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 10: Cooja configuration example

Figure 10: 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.

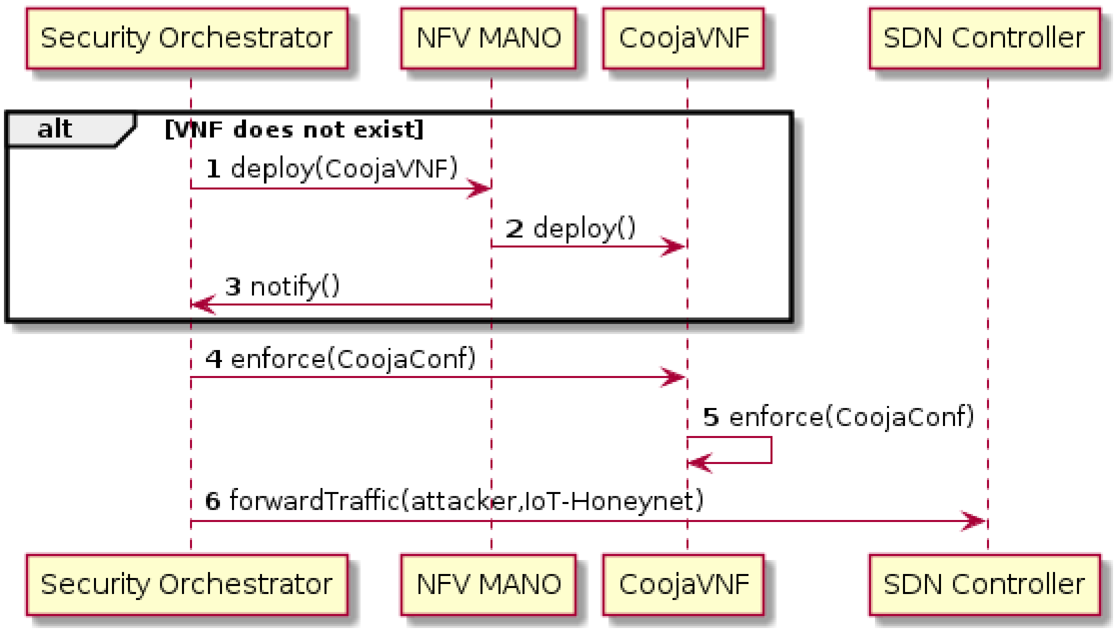


Figure 11: Cooja deployment

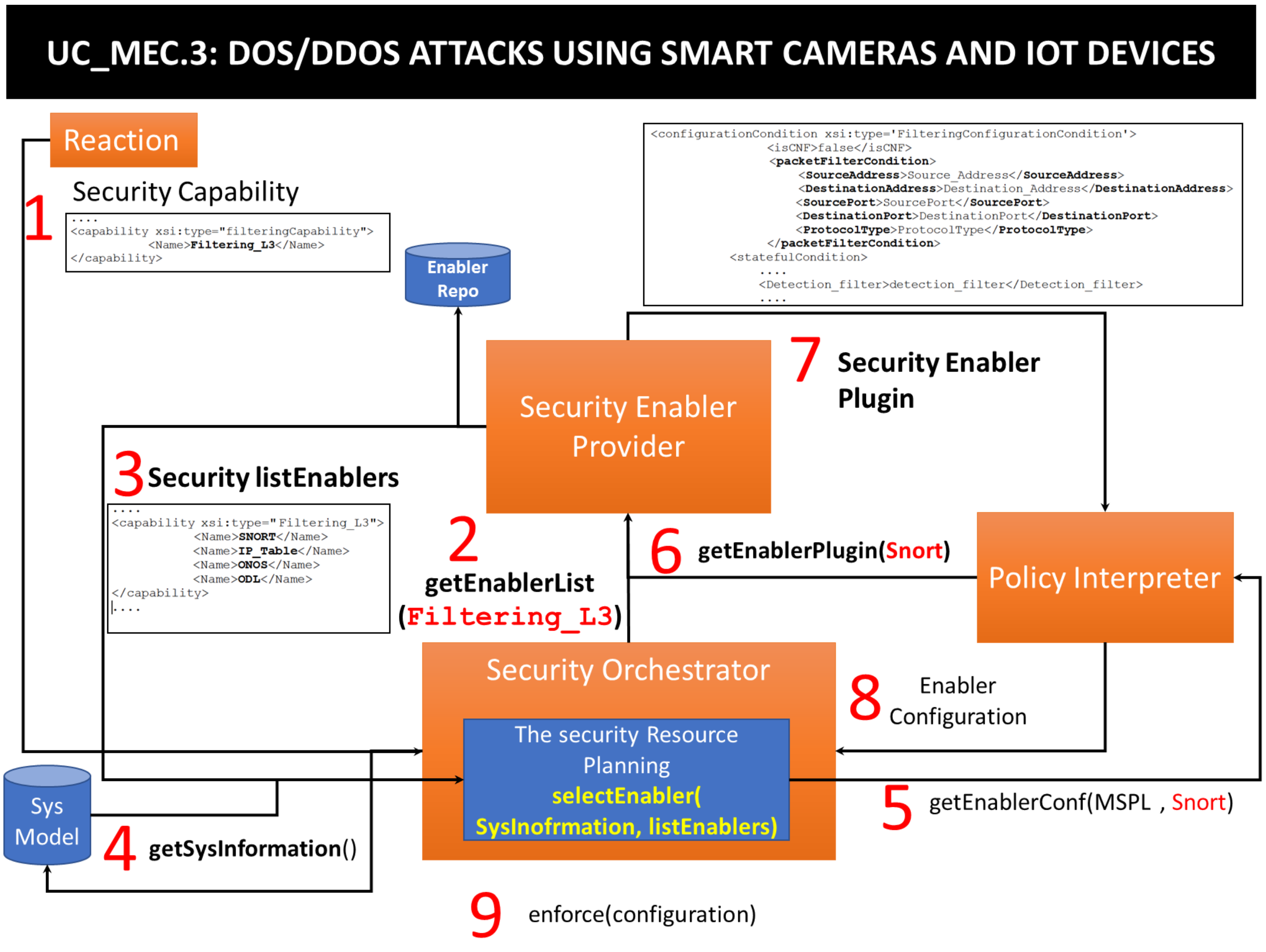
Figure 11: Cooja deployment shows the IoT-Honeynet deployment process:

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.

5.2 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.



UC_MEC.3: DOS/DDOS ATTACKS USING SMART CAMERAS AND IOT DEVICES
Reaction
1 Security Capability
Enabler Repo
7 Security Enabler Plugin
Security Enabler Provider
3 Security listEnablers
2 getEnablerList (Filtering_L3)
6 getEnablerPlugin(Snort)
Policy Interpreter
Security Orchestrator
The security Resource Planning
selectEnabler(SysInofrmation, listEnablers)
8 Enabler Configuration
Sys Model
4 getSysInformation()
5 getEnablerConf(MSPL , Snort)
9 enforce(configuration)

```
....
<capability xsi:type="filteringCapability">
            <Name>Filtering_L3</Name>
</capability>
```

```
<configurationCondition xsi:type='FilteringConfigurationCondition'>
                <isCNF>false</isCNF>
                <packetFilterCondition>
                    <SourceAddress>Source_Address</SourceAddress>
                    <DestinationAddress>Destination_Address</DestinationAddress>
                   <SourcePort>SourcePort</SourcePort>
                   <DestinationPort>DestinationPort</DestinationPort>
                   <ProtocolType>ProtocolType</ProtocolType>
                </packetFilterCondition>
           <statefulCondition>
               ....
               <Detection_filter>detection_filter</Detection_filter>
               ....
```

```
....
<capability xsi:type="Filtering_L3">
            <Name>SNORT</Name>
            <Name>IP_Table</Name>
            <Name>ONOS</Name>
            <Name>ODL</Name>
</capability>
....
```

Figure 12 : 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 12 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 12 (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 12 (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 12 (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 12 (5). The policy interpreter will request the security Enabler plugin from the security enabler provider Figure 12 (6). The security enabler provider will provide the appropriate plugin to the policy interpreter Figure 12 (7). The policy interpreter performs the translation and returns the security enabler configuration to the Security Orchestrator Figure 12 (8). Finally, the security orchestrator based on the enabler configuration will enforce the configuration Figure 12 (9).

5.3 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.

5.4 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.

6 CONCLUSIONS

This document provides a first overview of the first results of ANASTACIA Task 3.3, which aims to manage the first stage of Security Enforcement Enablers requirements established in high-level terms through the ANASTACIA architectural components and their integration.

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. Then we have presented a first implementation of these components. Finally, we have discussed the different the security enforcement capabilities regarding the different defined use cases.

7 REFERENCES

[1] 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”.

[2] Sébastien Ziegler, Antonio Skarmeta, Jorge Bernal, Eunsook Eunah Kim, Stefano Bianchi, **“ANASTACIA, Advanced Networked Agents for Security and Trust Assessment in CPS IoT Architectures”**. The 1st 2017 GLOBAL IoT SUMMIT, 2nd Workshop on User-centric security, privacy and data governance in smart cities (USP4SC). June 2017.

[3] Alejandro Molina Zarca, Jorge Bernal Bernabe, Ivan Farris, Yacine Khettab, Tarik Taleb, Antonio Skarmeta, **“Enhancing IoT Security through Network Softwarization and Virtual Security Appliances**”. International Journal of Network Management, 2018.

[4] I. Farris, J. B. Bernabe, N. Toumi, D. Garcia-Carrillo, T. Taleb, A. Skarmeta, B. Sahlin. **“Towards Provisioning of SDN/NFV-based Security Enablers for Integrated Protection of IoT Systems”**. IEEE Conference on Standards for Communications and Networking (CSCN-2017)

[5] AM Zarca, JB Bernabé, AS, K Yacine, B Dallal, S Bianchi “**Initial Security Enforcement Manager Report”**.2018. H2020 Anastacia EU project deliverable D3.1.

8 ATTACHMENTS

8.1 EXAMPLE OF SNORT PLUGIN

```
# # -*- coding: utf-8 -*-
"""
This python module implements a MSPL->SNORT plugin inside the ANASTACIA European Project,
extending the MSPL language defined in secured project.
How to use:
    python3 mspl_snort.py [MSPL_FILE.xml]

"""
__author__ = "Dallal Belabed"
__copyright__ = "Copyright 2018, ANASTACIA H2020"
__credits__ = ["Dallal Belabed", "Alejandro Molina Zarca"]
__license__ = "GPL"
__version__ = "0.0.2"
__maintainer__ = "Dallal Belabed"
__email__ = "dallal.belabed@thalesgroup.com"
__status__ = "Development"


import sys,os
import json
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
# Insert the parent path into the sys.path in order to import the mspl
parentPath = os.path.abspath("..")
if parentPath not in sys.path:
    sys.path.insert(0, parentPath)
import mspl

"""
    Output example:
    {"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
alert icmp any any -
> any any (msg:"ICMP_DoS ATTACK 2"; detection_filter: track by_src, count 100, seconds 10; sid:100002; rev:001;)"}}
"""
```

```

class ITResourceType(mspl.ITResourceType):

    def get_configuration(self):
        'Translate the ITResource element to SNORT configuration'
        capability_name = self.configuration.capability[0].Name
        logger.info("get_configuration capability_name...")
        if not capability_name in mspl.CapabilityType.itervalues():
            raise ValueError('The capability {} does not exist, please try with the current supported capabilities'.format(capability_name))
        # In filtering case, call translation for filtering Action and filtering conf condition
        try:
            capability_configuration_function = getattr(self, "get_{}_configuration".format(capability_name))
        except AttributeError as e:
            print("Currently, {} is not implemented.".format(capability_name))
            print(e)
            sys.exit(1)

        action, capability_enabler_conf = capability_configuration_function()
        # Build the enabler configuration, since the action is mandatory is not neccesary to check it
        enabler_conf = "{} {}".format(action, capability_enabler_conf if capability_enabler_conf else "")
        return enabler_conf

    def get_Snort_configuration(self):
        'Returns the configuration for Filtering_L4 capability'
        # For filtering it is only needed the first configuration rule
        try:
            configuration_rule = self.configuration.configurationRule[0]
        except:
            print("Filtering_L4 capability requires at least a configurationRule element")
            sys.exit(1)
        action = configuration_rule.configurationRuleAction.get_configuration()
        filtering_enabler_conf = configuration_rule.configurationCondition.get_configuration()

        return (action,filtering_enabler_conf)



class FilteringAction(mspl.FilteringAction):
    def get_configuration(self):
        'Returns the filtering action for SNORT'
        FILTERING_ACTION = {"Alert": "alert", "Log":"log", "Pass":"pass"}
```

```python
        return FILTERING_ACTION.get(self.FilteringActionType,"DROP")

class FilteringConfigurationCondition(mspl.FilteringConfigurationCondition):
    def get_configuration(self):
        'Returns the traslated filtering SNORT parameters'
        filtering_enabler_conf = ""
        packet_filter_condition = self.packetFilterCondition
        State_ful_Condition = self.statefulCondition
        time_Condition = self.timeCondition
        if not packet_filter_condition:
            return None
        if not State_ful_Condition:
            return None
        if not time_Condition:
            return None

        filtering_enabler_conf+=" {}".format(packet_filter_condition.ProtocolType) if packet_filter_condition.ProtocolType else ""
        filtering_enabler_conf+=" {}".format(packet_filter_condition.SourceAddress) if packet_filter_condition.SourceAddress else ""
        filtering_enabler_conf+=" -> {} ".format(packet_filter_condition.SourcePort) if packet_filter_condition.SourcePort else ""
        filtering_enabler_conf+=" {} ".format(packet_filter_condition.DestinationAddress) if packet_filter_condition.DestinationAddress else ""
        filtering_enabler_conf+=" {}".format(packet_filter_condition.DestinationPort) if packet_filter_condition.DestinationPort else ""


        filtering_enabler_conf+=" (msg:\"{}\"".format(State_ful_Condition.Message) if State_ful_Condition.Message else ""
        filtering_enabler_conf+="; detection_filter:{}".format(State_ful_Condition.Detection_filter) if State_ful_Condition.Detection_filter else ""
        filtering_enabler_conf+=", count {}".format(State_ful_Condition.Count) if State_ful_Condition.Count else ""
        filtering_enabler_conf+=", seconds {}".format(time_Condition.Time) if time_Condition.Time else ""
        filtering_enabler_conf+="; sid:{}".format(State_ful_Condition.Sid) if State_ful_Condition.Sid else ""
        filtering_enabler_conf+="; rev:{}".format(State_ful_Condition.Rev) if State_ful_Condition.Rev else ""
        filtering_enabler_conf+=";)"


        return filtering_enabler_conf if filtering_enabler_conf else None

class M2LPlugin:
    'SNORT Medium to low plugin implementation'

```

```
  def get_configuration(self,mspl_source):
    'Return the SNORT configuration from the mspl_source'
    # Customize the pyxb generated classes with our own behavior
    global mspl
    mspl.ITResourceType._SetSupersedingClass(ITResourceType)
    mspl.FilteringAction._SetSupersedingClass(FilteringAction)
    mspl.FilteringConfigurationCondition._SetSupersedingClass(FilteringConfigurationCondition)
    #Load the xml
    it_resource = mspl.CreateFromDocument(mspl_source)
    # Start the parser process using the xml root element
    return it_resource.get_configuration()

if __name__ == "__main__":

  #output file
  outputfile = "local.rules";
  source = open(outputfile, "w")
  # Pretty print
  separator = "*"
  separator_group = separator*5
  title = "SNORT CONFIGURATION"
  print("\n{}{}{}".format(separator_group,title,separator_group))

  for i in range (1,len(sys.argv)):
    print (sys.argv[i])
    xml_file = sys.argv[i]
    # Instantiate the plugin
    m2lplugin = M2LPlugin()
    logger.info("Reading mspl file...")
    xml_source = open(xml_file).read()
    logger.info("Translating mspl to SNORT...")
    enabler_configuration = m2lplugin.get_configuration(xml_source)
    print(enabler_configuration)
    source.write(enabler_configuration+ "\n")
  print("{}{}{}\n".format(separator_group,separator*len(title),separator_group))
  print ("Snort file path: " + os.path.abspath(outputfile))

  source.close()
```

8.2 Example of Security Enabler for BMS.4 use case

```xml
<?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>
```