Secure Provisioning of Cloud Services
based on SLA Management

SPECS Project - Deliverable 4.3.2

Implementation of the enforcement SLA components - Intermediary

Version no. 1.1
18 February 2016

The activities reported in this deliverable are partially supported
by the European Community's Seventh Framework Programme under grant agreement no. 610795.

Deliverable information

Deliverable no.:	D4.3.2
Deliverable title:	Implementation of the enforcement SLA components – Intermediary
Deliverable nature:	Prototype
Dissemination level:	Public
Contractual delivery:	18 February 2016
Actual delivery date:	18 February 2016
Author(s):	Jolanda Modic (XLAB), Miha Stopar (XLAB)
Contributors:	Damjan Murn (XLAB), Aljaž Košir (XLAB), Massimiliano Rak (CeRICT), Silviu Panica (IeAT), Giancarlo Capone (CeRICT)
Reviewers:	Dana Petcu (IeAT), Valentina Casola (CeRICT)
Task contributing to the deliverable:	T4.3
Total number of pages:	118

Executive summary

This document presents the intermediary implementation of the Enforcement module. The demonstrated prototypes follow the implementation plan introduced in D4.3.1, and are based on requirements and design presented in D4.1.2 and D4.2.2, respectively, user stories and validation scenarios defined in T5.1, and even some new requirements from stakeholders.

With developed prototypes the entire refined enforcement flow is demonstrated in detail. First prototypes of security mechanisms (presented in D4.3.1) are improved and additionally prototypes of few other mechanisms are presented as well.

As described in deliverables of task T4.1 and T4.2, the Enforcement module comprises a set of main components (Planning, Implementation integrated with the Broker and a Chef Server, Diagnosis, Remediation Decision System) and a set of security mechanisms (WebPool, E2EE with DBB, SVA, TLS, DoS, and AAA).

In this document, we present the status of development activities related to task T4.3, and show how to install and use the prototype components and mechanisms developed under this task. All main Enforcement components and the majority of security mechanisms are already available, and links and descriptions of Bitbucket repositories are provided. All prototypes cover a very large number of requirements. However, since in WP4 there is a dedicated task for validation and testing of the Enforcement module, results are discussed in deliverable D4.5.2. These include coverage of validation scenarios, coverage of requirements, and unit tests.

Table of contents

Index of figures

Index of tables

1. Introduction

Developing tools for automatic and dynamic management of the security SLA life-cycle is a challenging task. One has to consider not only which security features to offer so that the resulting SLAs are implementable from the developer's and also CSP's perspective, but also how to offer them so that the offers are understandable to all End-users (EUs). Furthermore, one has to consider not only the aspects of negotiation, but also steps and details of renegotiation if needed (after unsuccessful remediation of SLA violations) or requested (at any point by an EU or a CSP). All negotiation and renegotiation steps have to be defined in accordance to deployment possibilities and constraints. And the other way around, the SLA implementation process has to be designed in a way to enable automatic acquisition of resources and deployment of services as specified in a signed SLA. In SPECS, the Enforcement module covers these SLA implementation aspects with two components, namely Planning and Implementation. They take as an input a signed SLA, and prepare all configuration details so that the SLA can be implemented, and actually acquire all resources and deploy all services according to the implementation plan.

Another crucial functionality offered by the Enforcement module is the one focused on detecting deviations from agreed upon security settings and remediating them in an automatic and secure way. For each security metric chosen by the EU in the SLA negotiation phase to be enforced by some security mechanism, in the SLA implementation phase the Enforcement module has to identify measurements with which the monitoring system can evaluate the validity of SLOs related to that metric. And for each deviation from what is expected, in the SLA remediation phase the Enforcement module has to determine actions to be taken in order to prevent or recover from the detected SLA violation. In the SLA remediation phase, the Enforcement module has to not only predict and manage SLA violations, but also determine which events to handle first to avoid causing more damage. In SPECS, the Enforcement module orchestrates SLA remediation phase with two components, namely Diagnosis and RDS. The first one analyses suspicious events detected by the monitoring core, and the other one prepares a remediation plan according to the performed analysis.

In Figure 1 the position and the role of the Enforcement module is presented. As mentioned above, the implementation phase is conducted after a successful negotiation according to a signed SLA, and remediation phase is activated after the monitoring core notifies about a possible attack or a failure. Considering that not every detected suspicious event actually presents an SLA violation (remediation → monitoring), that remediation plan is actually executed by the Implementation component (remediation → implementation), and that unsuccessful remediation can end with a renegotiation (remediation → renegotiation), this brings us to the full SLA life-cycle in SPECS.

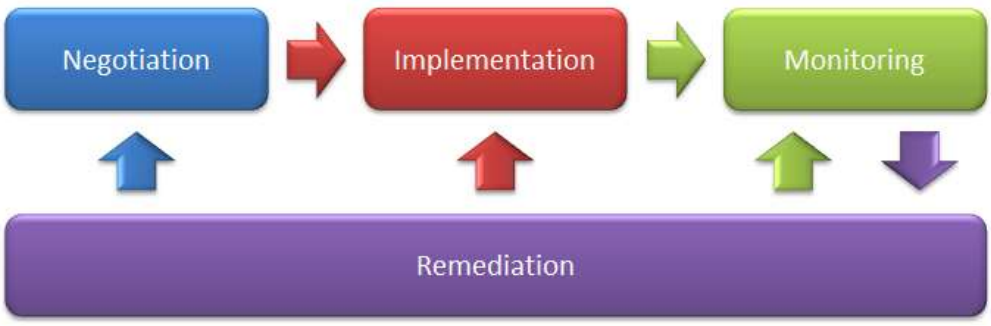

Figure 1. SLA phases

This document presents the second version of the Enforcement prototype. If the first prototypes (described in D4.3.1) demonstrated the basic enforcement flow and the use of three elementary security mechanisms (Broker, WebPool, and SVA), this next versions are extended with all enforcement functionalities (actual refinements are detailed in Section 3), including an innovative approach to SLA implementation and SLA remediation phases. Additional prototypes presented in this deliverable have been developed according to the refinement and enhancement of the Enforcement design. Moreover, security mechanisms presented in first year are revamped according to the feedback from other tasks. Also, additional security mechanisms are included (DBB, E2EE, and TLS), as anticipated in D4.3.1.

Improvements of the enforcement process were due to the feedback received from other design, implementation, and testing activities. Additional input came from refinements within tasks T5.2. If in the first year the requirement was only to implement a mechanism offering end-2-end encryption, in the second year a set of requirements grew and resulted in the need to also offer a mechanism implementing secure storage with backup and some additional security features discussed later in this document (see Section 4).

The last version of the software prototype (which will be presented in D4.3.3) will implement all designed security mechanisms presented in D4.2.2 to support the coverage of all user stories defined in WP5 and almost all associated requirements gathered and discussed in D4.1.2.

Note that all testing activities have been conducted in task T4.5 and reported in deliverable D4.5.2.

The document is structured as follows. After presenting the inputs and outputs of the project which are considered in this deliverable and its relation with other deliverables (Section 2), the focus turns to the Enforcement core.

Section 3 presents a detailed flow orchestrated by each main Enforcement component, and provides brief installation and usage guides for each of them. The status of development activities is also presented, and the coverage of the associated requirements is discussed.

Section 4 aims at demonstrating security mechanisms included in the prototype. For each security mechanism a set of security controls and security metrics enforced by the mechanism is presented. Refinements of the design are described, and implementation and remediation details are specified. As for the main Enforcement components, installation and usage guides are also included for each security mechanism.

The document is concluded with a short synthesis of the current development status and a brief preview of the last prototype described in D4.3.3 at the end of the project.

Note that all details about Credential Service and Security Tokens mechanisms are provided in deliverables of dedicated task T4.4, and the Auditing component is discussed in deliverables D1.4.1 and D1.4.2.

2. Relationship with other deliverables

The Enforcement prototype developed in the second year of the project and described in this document demonstrates the entire enforcement flow, from planning and implementation to diagnosis and remediation.
The SLA implementation phase has been refined considering inputs from

- WP1 (final architecture of the Platform, design of interfaces),
- WP2 (negotiation and renegotiation flows, initial implementation), and
- WP3 (design and implementation of monitoring components).

The SLA remediation phase has been developed considering inputs from

- WP1 (Platform's design, definition of interfaces) and
- WP3 (monitoring process).

Of course, the main input came from WP4 design and implementation activities carried out in the first year of the project (initial prototype, Enforcement design) and from the validation and testing task. Refinements of validation scenarios conducted in WP5 also provided valuable feedback.

The current document served and will serve as an input for the activities conducted in parallel and for the remaining implementation activities in WP2, WP3, and development, testing, and integration activities in WP1, WP4, and WP5. The definition of security metrics might provide an input for standardisation activities in WP6.

All mentioned relationships are detailed in Figure 2.

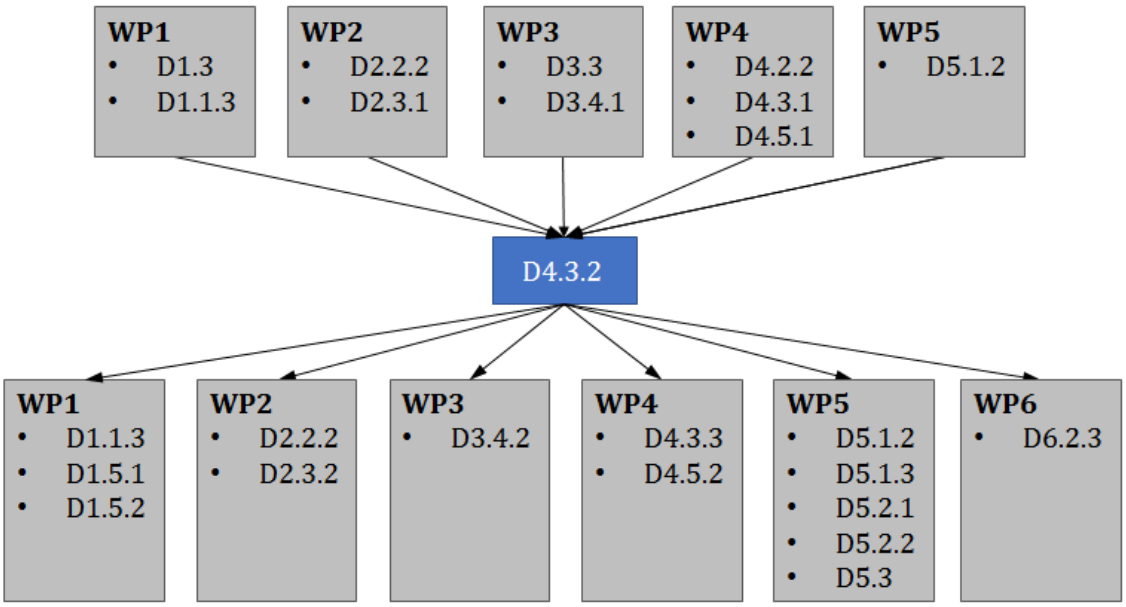

Figure 2. Relationship with other deliverables

3. Main Enforcement components

The Enforcement module is a system that oversees two phases of the SLA life cycle, namely SLA implementation and SLA remediation. As described in deliverable D4.2.2, the orchestration of all implementation and remediation activities is conducted by four main Enforcement components (see Figure 3).

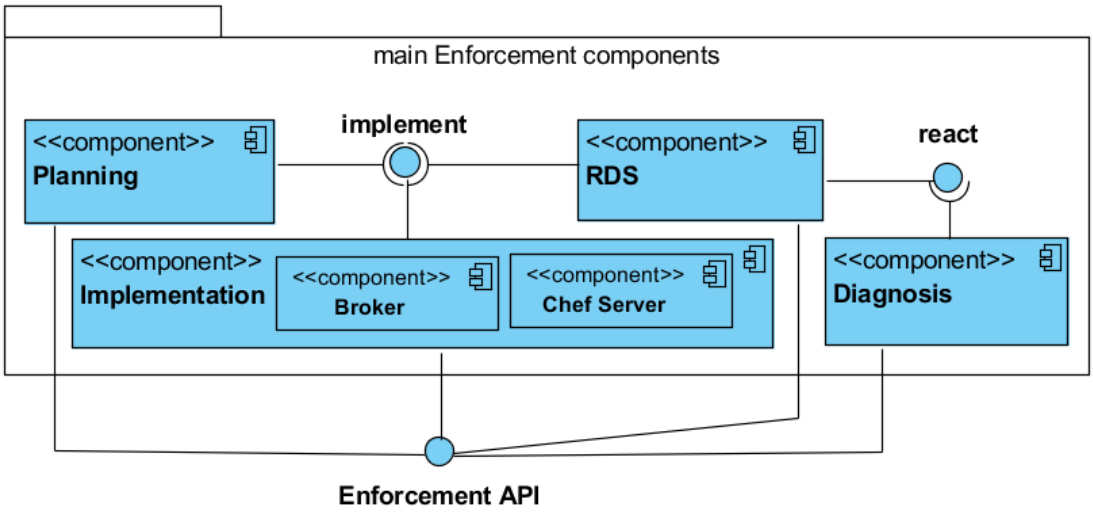

Figure 3. Enforcement module's high level architecture (main Enforcement components)

The SLA implementation phase takes place after the initial SLA signature (see Figure 4) and after renegotiation or termination of the SLA (see Figure 5). The process in all cases is driven by two main Enforcement components. The Planning component

- generates supply chains for EU's security requirements (in SLA negotiation phase),
- builds implementation plan for a signed SLA, and
- prepares a reaction plan for a renegotiated or terminated SLA.

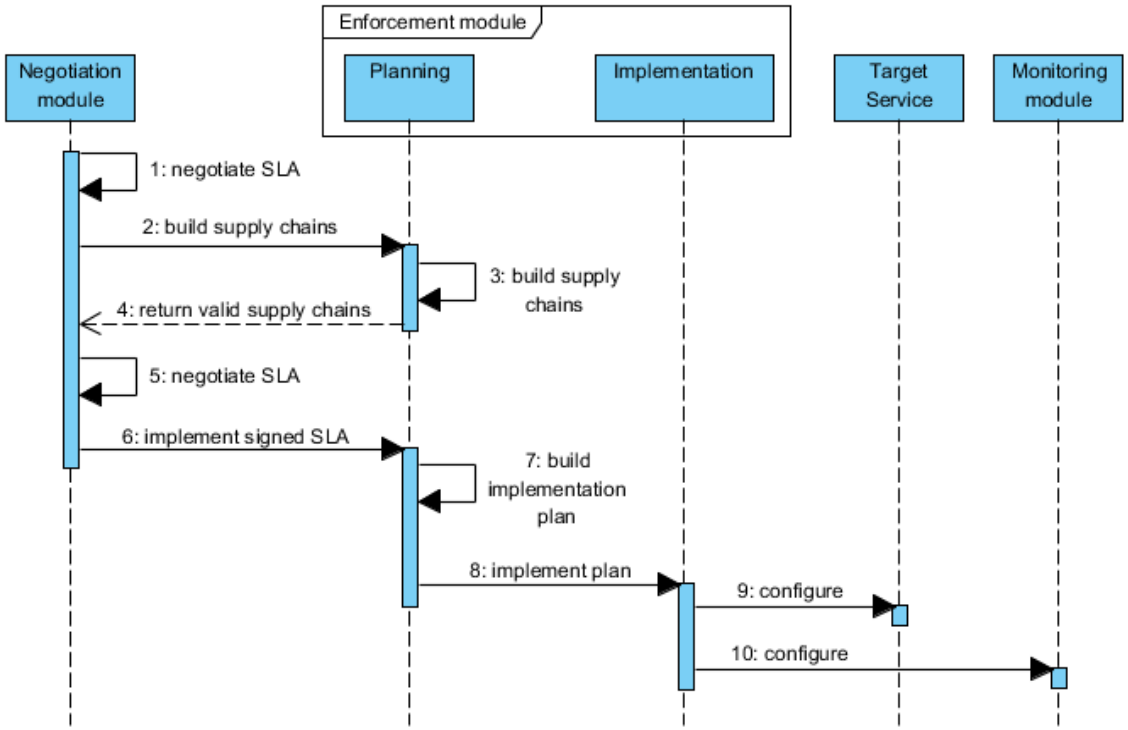

Figure 4. SLA implementation phase (initial implementation of an SLA)

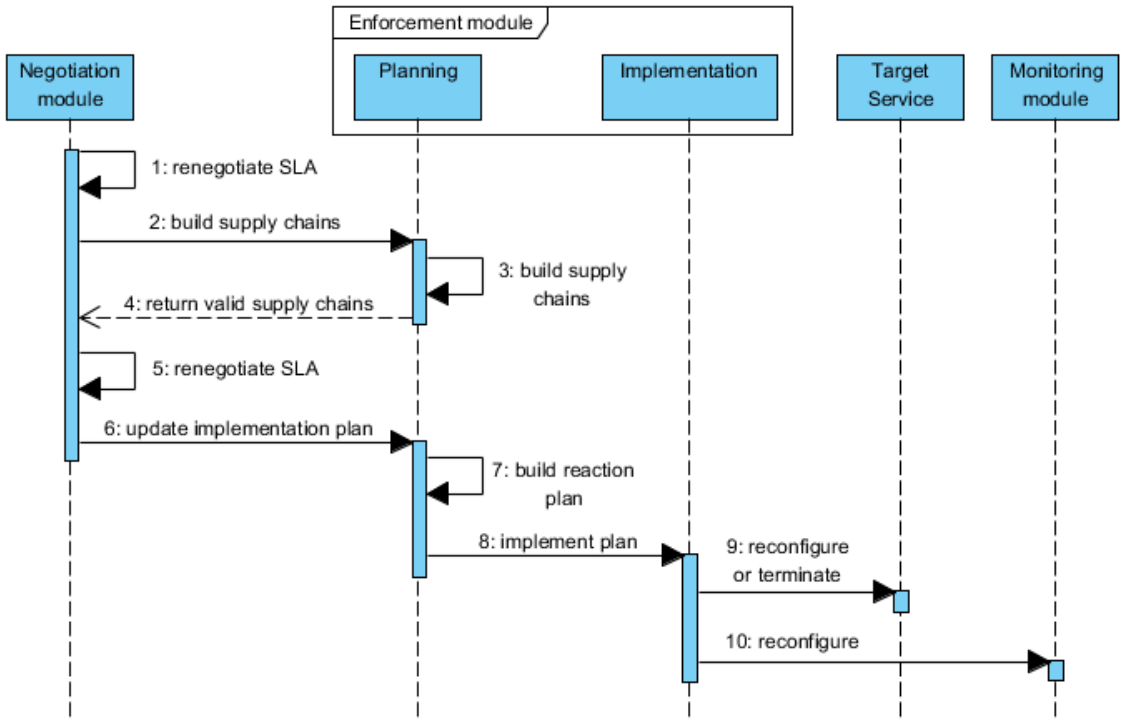

Figure 5. SLA implementation phase (after renegotiation or termination)

The implementation component acquires resources and deploys and configures (or reconfigures or terminates) security mechanisms and configures (or reconfigures) monitoring components according to the implementation or reaction plan. As will be discussed in Section 3.6, in Y2 the Implementation component is integrated with the Broker and a Chef Server.

The SLA remediation phase (presented in Figure 6) is orchestrated by the Diagnosis component which classifies and analyses detected monitoring events, and by the RDS component which prepares remediation plans according to results of the diagnosis process. The remediation plan is later executed by the Implementation component.

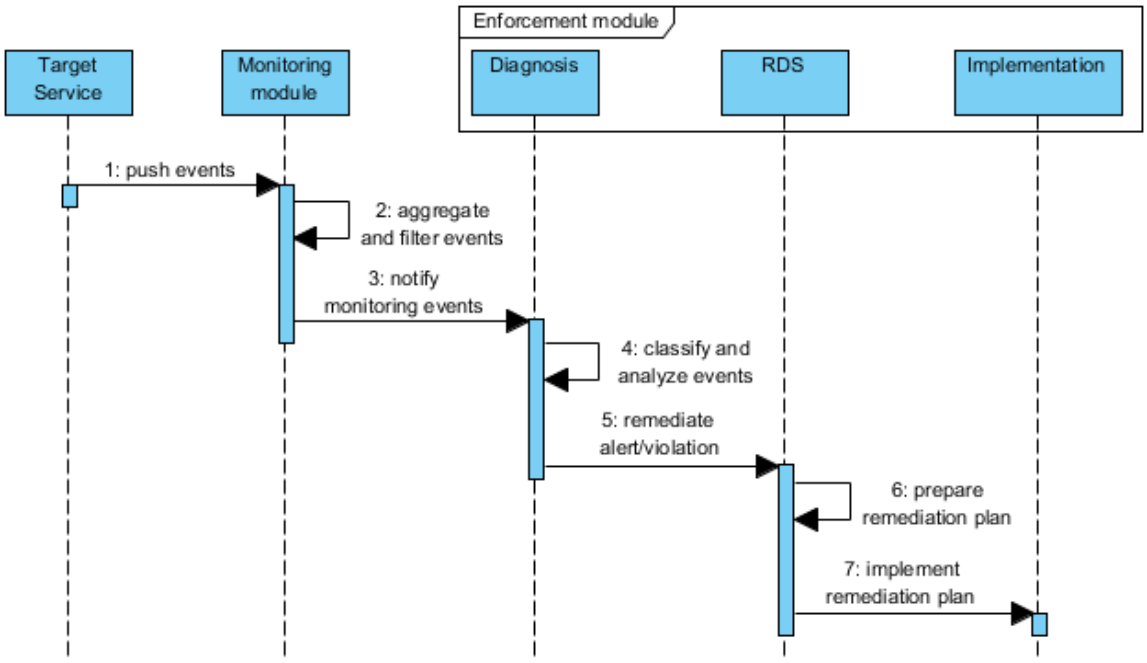

Figure 6. SLA remediation phase

Auditing component does not have a functional role in the enforcement flow, but is an essential component for all logging activities. As mentioned at the beginning of this section, further implementation details are available in D1.4.1 and D1.4.2.

The architecture of the module is roughly the same as it was reported in D4.2.2 (apart from integrating Implementation component with the Broker and a Chef Server). What really evolved from year one is the enforcement process. Refinements are outlined in Table 1 below.

Further details of the internal processes of the entire enforcement flow are described in subsections dedicated to main Enforcement components. For details of interactions among components/modules (from the interface perspective) see deliverable D1.3.

Component	Year 1	Year 2
Planning	• Validates supply chains. • Builds implementation plan according to a signed SLA.	• Builds valid supply chains. [*Generation and validation of supply chains is merged into one step.*] • Builds implementation plan according to a signed SLA and associated supply chain. • Builds a reaction plan to reconfigure target services after SLA renegotiation and SLA termination. [*Added after refinement of the renegotiation process.*] • Updates the Monitoring Policy. [*Moved from the Implementation component due to refinement of the SLA implementation phase.*]
Implementation	• Executes implementation plan. • Updates the Monitoring Policy.	• Executes implementation plan. • Executes remediation plan to reconfigure target services during SLA remediation. • Executes reaction plan to reconfigure target services after SLA renegotiation or SLA termination. [*Added due to refinements of remediation and renegotiation processes.*]
Diagnosis	• Classifies, analyses, and prioritizes monitoring events. • Determines root causes of monitoring events.	• Classifies, analyses, and prioritizes monitoring events. [*Root cause analysis has been moved to the RDS component during the refinement of the remediation process.*]
RDS	• Searches for redressing techniques.	• Determines root causes of monitoring events. [*Moved from the Diagnosis in the refinement of the SLA remediation phase.*] • Searches for redressing techniques. • Builds remediation plan. [*Added during refinements of the SLA remediation phase.*]

Table 1. Refinements of the enforcement process

In the next subsection we provide the current status of development.

3.1. Status of development activities

In Table 2 we present coverage of requirements associated to Enforcement module by main Enforcement components.

Requirements for main Enforcement components	SPECS software components				
	Planning	Implementation	Diagnosis	RDS	Broker
ENF_PLAN_R1-R12	X				
ENF_PLAN_R8-R9[1]	X			X	
ENF_IMPL_R1-R9		X			
ENF_IMPL_R10[2]	X				
ENF_DIAG_R1-R18			X		
ENF_REM_R1-R11				X	
SLA_NEG_R30-R31				X	
ENF_BROKER_R1-R5					X

Table 2. SPECS Enforcement components and related requirements

There are 61 requirements for the Enforcement module, related to the main Enforcement components (requirements associated to security mechanisms are discussed in Section 4).

The current implementation of the Planning component covers 11 out of 13 requirements associated to the component. Remaining two, namely *ENF_PLAN_R8* and *ENF_PLAN_R9*, will be completely cover with the final prototype. Note that both requirements are related to building a reaction and migration plan to recover from an alert or a violation. And this step is performed by the current prototype. But since reaction plan is also needed after renegotiation and after termination of an SLA (which will be implemented after M24), we will consider *ENF_PLAN_R8* and *ENF_PLAN_R9* covered only at the end of the project.

The Implementation component already covers almost all associated requirements (8 out of 9). Remaining one, namely *ENF_IMPL_R9*, related to implementation of the reaction plan will be completely cover at M30, since it also covers SLA implementation phase after SLA renegotiation and termination.

Similarly, the development of the Diagnosis component at M24 covers almost all related requirements (17 out of 18). The remaining one, namely *ENF_DIAG_R7* (related to expressing violations in terms of KPI rules), will possibly be cover with the final version of the component.

The current prototypes for the RDS component and the Broker cover all associated requirements.

To summarize, with the current prototypes of the core Enforcement components 57 of all elicited core Enforcement requirements are implemented. The current development status is summarized in Table 3. Note that the details related to the Enforcement API are reported in D1.3.

In the last 6 months of the project, remaining effort will mainly be spent on developing planning and implementation activities related to the steps after SLA renegotiation and after SLA termination. We also expect some improvements for the Diagnosis and RDS. The final results will be reported in D4.3.3 at M30.

[1] In Y1 these two requirements were covered only by the Planning component. In the refinement of the enforcement process we split responsibilities between the Planning and the RDS component.
[2] This requirement was initially covered by the Implementation component. During developement stage this task was reassigned to the Planning component.

Module	Artifacts under development	Status
Enforcement module	component:Planning	Available
	component:Implementation	Available
	component:Diagnosis	Available
	component:RDS	Available
	component:Broker	Available

Table 3. Enforcement module implementation status

The first prototypes of all Enforcement core components are available on the project's Bitbucket repositories:

- The Planning component is available at [1].
- The Implementation component is available at [2].
- The Diagnosis component is available at [3].
- The RDS component is available at [4].
- The Broker is available at [30].

The following subsections provide all design, development, installation, and usage details for core Enforcement components.

3.2. Planning component

As mentioned in the introduction of Section 3 and reported in D4.2.2, the Planning component is involved in the SLA negotiation and SLA implementation phase. The initial planning process is described in D4.2.2. In the following we provide with a detailed description of the final version.

3.2.1. Overview

During negotiation[3], the Planning component generates valid supply chains according to EU's security requirements (see Figure 7, where ***fX[sY]*** refers to step ***Y*** in figure ***X***). The Supply Chain Manager invokes the Planning component to build supply chains ***f7[s1]*** by passing a list of CSPs, list of SLOs, and a list of mechanisms able to implement SLOs. The Planning first builds resource combinations ***f7[s2]***, which means extracting for each provider a list of zones, virtual machine (VM) types, and maximum acquirable number of VMs per CSP per zone. Note that in the supply chain generation process {CSP_1, $zone_1$, $VMtype_1$}, {CSP_1, $zone_1$, $VMtype_2$}, and {CSP_1, $zone_2$, $VMtype_1$} are treated as three different providers.

For each reported security mechanism its metadata is retrieved ***f7[s3-4]*** from the SLA Platform. Each security mechanism consists of a set of components that are able to enforce and monitor metrics associated to it. Mechanism's metadata includes information about its components as well as all related configuration requirements and constraints (e.g., firewall rules, resource consumption, dependencies and incompatibilities among components). For example, WebPool mechanism, described in Section 4.2, comprises a load balancer (HAProxy) and two web servers (Apache and Nginx). WebPool's constraints report that for each deployment of the mechanism we need exactly one balancer and at least one web server (the actual number of needed web servers is determined by the level of redundancy requested by the EU), that at most one web server can be deployed on each acquired VM, and that EU's required level of diversity (i.e., number of web server types) determines the number of different deployed web servers. The metadata for the WebPool mechanism can be seen in

[3] For details of negotiation process see D2.2.2.

Appendix 4.

The Planning component then selects all mechanisms' components needed to implement the SLA ***f7[s5]***. In the WebPool case, if the EU requested level of redundancy (i.e., number of web server replicas) to be 3, and level of diversity to be 1, we need one balancer and three web servers of the same type, i.e., one instance of HAProxy and three instances of either Apache or Nginx. When all components are chosen for all mechanisms to be implemented, the Planning extracts constraints from mechanisms' metadata ***f7[s6]***. For the example at hand (where we choose HAProxy and Apache) the constraints would include:

- HAProxy cannot be allocated to a VM together with Apache.
- The number of instances of component HAProxy must be equal to 1.
- The total number of instances of component Apache must be greater or equal to 3.
- The minimum number of VMs must be greater or equal to 4 (minimum number of VMs is determined by the level of redundancy).

When the Planning prepares the list of constraints for all mechanisms and components to be deployed, it has to find the optimal allocation of components over (a minimal possible number of) acquired resources. This means determining the number of needed resources and allocation of chosen components over all acquired resources, considering all implementation and configuration constraints (as reported in metadata, e.g., load consumption). All constraints can be expressed in terms or linear equations ***f7[s7]***. Thus the described planning problem can be modelled and solved ***f7[s8]*** in terms of an Integer Linear Program (ILP)[4]. The solution of the planning problem is then translated into a supply chain format ***f7[s9]***.

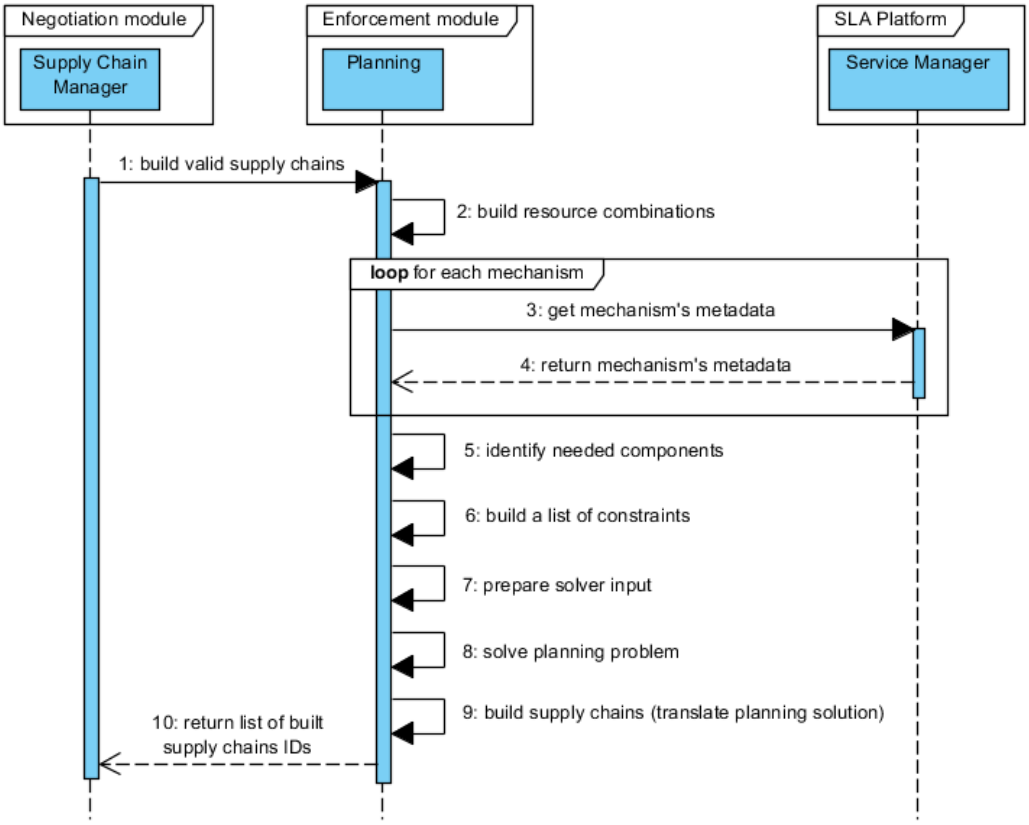

Figure 7. Generation of supply chains

[4] For solving the planning problem we use an open-source Java library Joptimizer [18].

For the example above, the supply chain (which reports a provider, number of needed resources, and allocation of components to deploy over those resources) is:

- Provider: {CSP, zone, VM type}, for example {aws-ec2, us-east-1, c1.medium}[5].
- Number of needed VMs: 4
- Allocation: VM_1: {HAProxy}, VM_2: {Apache}, VM_3: {Apache}, VM_4: {Apache}

The precise and detailed formulation of the planning problem (steps ***f7[s6-9])*** with examples is described in Appendix 1.

The list of IDs of all generated supply chains for the EU's security requirements is returned to the Negotiation module ***f7[s10]***.

Note that since we consider all constraints related to deployment (dependabilities, incompatibilities, limitations) and all constraints related to CSPs' offers (VM type for each provider/zone determines which and how many components can be deployed on which VM), all generated supply chains are implementable. So no further validation process is needed. If the Planning component cannot build a single valid supply chain, the EU is notified about the unfeasible set of security requirements and is asked to start a new negotiation cycle.

When the EU signs an SLA, the next step for the Planning component is to prepare an implementation plan according to the SLA and the associated supply chain. The following description of the process of building the implementation plan follows the sequence diagram on Figure 8 below.

The SPECS Application triggers the Planning component to prepare the implementation plan ***f8[s1]*** by passing it the ID of the signed SLA. Since the SLA has been signed, we can track all activities related to it, thus we log the start of the process by logging the activation of the Planning component labelling it with the ID of the SLA ***f8[s2]***. Logging details are provided in description of the Auditing component in deliverables D1.4.1 and D1.4.2.

The Planning retrieves the signed SLA ***f8[s3-4]*** and parses it ***f8[s5]*** to extract SLOs, and then retrieves the associated supply chain ***f8[s6]***. Note that in the SLA negotiation phase each supply chain built for the set of EU's security requirements implied one SLA Offer. After the SLA signature all rejected supply chains (linked to rejected SLA Offers) were deleted so that each signed SLA is only accompanied with one supply chain.

The set of security mechanisms is extracted from the supply chain and for each mechanism all of the associated implementation details are retrieved from the SLA Platform ***f8[s7-8]***. An example of configuration details for the WebPool mechanism is provided at the end of the document (see Appendix 4).

Mechanism's configuration details are prepared by the developer of a mechanism. For each mechanism a list of enforceable and a list of monitorable metrics has to be provided. Note that some mechanisms can enforce a metric by providing the infrastructure to assure a certain level of security (e.g., providing web server replicas with WebPool) or by providing specific configurations and functionalities on the existing resources (e.g., encrypting stored data with E2EE mechanism). But some mechanisms are only able to monitor validity of SLOs (e.g.,

[5] For details about the CSP, zones, and instance types, see **Error! Reference source not found.**.

Nmap[6] is only able to monitor certain TLS metrics) by observing the state of the system.

As will be later discussed in the description of the diagnosis process (in Section 3.4), each metric is associated to a set of measurements with which we can detect alerts and violations. And for each measurement a detectable monitoring event is defined. For the entire set of monitoring events the developer also specifies a remediation flow which consists of a list of remediation actions. And since all installations and configurations are performed automatically with Chef[7] recipes, the developer also has to prepare a list of recipes associated to each remediation action. Mechanism's metadata part consists of a set of components that implement the mechanism, enriched with all configuration details and implementation constraints.

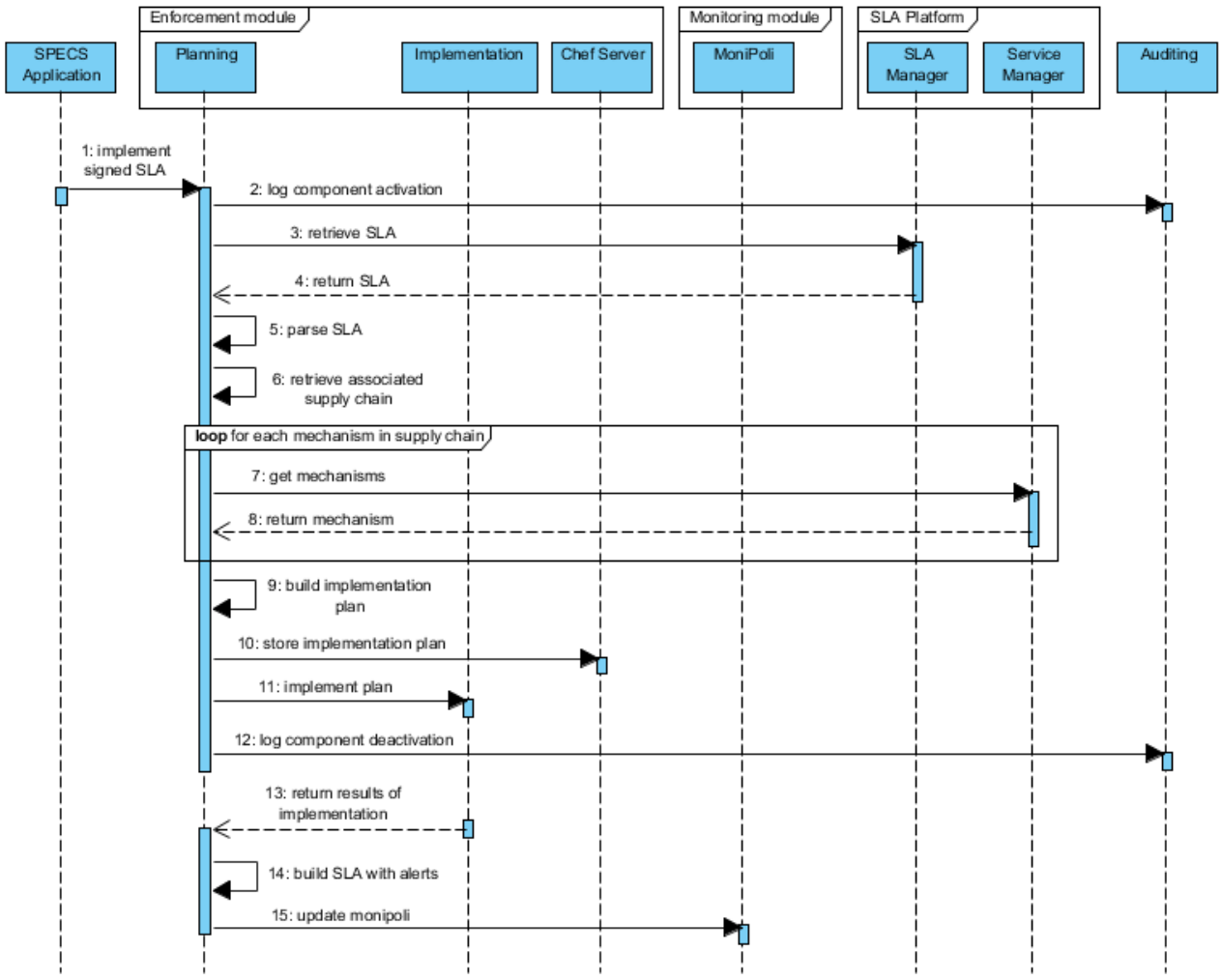

Figure 8. Building implementation plan

The Planning then builds the implementation plan ***f8[s9]***, which includes:

- IDs of associated SLA and supply chain.
- Information about the CSP (provider ID, zone, VM type).
- IP address and the port number of the monitoring core.

[6] For details see D3.4.1.

[7] The use of Chef for automatic implementation and configuration of resources in SPECS has been discussed in D4.2.2.

- A list of pools and for each of them a list of VMs to acquire. For each VM a set of components to be deployed is included with all configuration details (firewall rules, Chef recipes with which the listed components are installed and configured, etc.).
- A list of SLOs is added with which the Implementation component determines configuration details in Chef recipes.
- A list of measurements is included with which the validity of SLOs in the SLA is evaluated. The list of measurements also includes information related to alert and violation thresholds needed for step ***f8[s14]***, and a list of detectable monitoring events.

The built implementation plan is stored in the Chef server ***f8[s10]*** and Implementation component is invoked to execute the plan ***f8[s11]***. Deactivation of the Planning component and thus finalization of the planning process is logged ***f8[s12]***.

After the plan has been successfully implemented and the Implementation component returns the result of the acquisition and deployment process ***f8[s13]***, the Planning prepares a list of violation and alert thresholds in the XML format ***f8[s14]***. With such an "SLA with alerts", the Monitoring Policy (MoniPoli) is reconfigured ***f8[s15]*** to detect suspicious monitoring events that could result in alerts or violations of the implemented SLA.

In the first year of the project two policies were anticipated. The MoniPoli was defined for violation thresholds and the Enforcement Policy was assumed for alert thresholds. During refinement and optimisation stages of the development activities in WP3 and WP4, the MoniPoli was developed in such a way that it manages both, rules for violations and rules for alerts. For details of the MoniPoli see D3.3 and associated prototype deliverables.

An example of the implementation plan and the associated SLA with alerts is presented in Appendix 2.

The EU has the opportunity to renegotiate an SLA or request its termination before the expiration date. As we will explain in Section 3.5, renegotiation or termination might be necessary after an unsuccessful remediation of an alert or a violation. In either case, the Planning component has to prepare the so called reaction plan.

The process of building a reaction plan is depicted in Figure 9 below. The following are details of each step of the planning process after renegotiation.

As described in D2.2.2, during the renegotiation process the EU signs a new SLA which is accompanied with a new supply chain. The SPECS Application invokes the Planning to prepare a reaction plan (and update the existing implementation plan for the initial SLA) with the ID of the new SLA ***f9[s1]***. The Planning first logs its activation ***f9[s2]***, and then retrieves the new SLA ***f9[s3-4]***, the associated new supply chain ***f9[s5]***, and the initial implementation plan ***f9[s6-7]***. The Planning compares the initial deployment setting with the new SLA and the new supply chain ***f9[s8]***. According to needed reconfigurations a reaction plan is built ***f9[s9]***. A reaction plan is basically a list of fake violations that require immediate remediation actions. For example, if the new supply chain reports more resources than are included in the initial implementation plan, the Planning prepares a list of violations of type "VM is unresponsive". Such a violation requires an acquisition of a new VM. Similarly, if the EU renegotiated new security capability that requires deployment of a new mechanism on existing resources, the Planning prepares a violation of type "mechanism's component is unavailable" which requires installation of the missing component. Or if some SLO in the new SLA reports higher value of a

metric as was initially negotiated, the Planning has to prepare a violation of type "configuration issue" which requires reconfiguration of installed components (reconfiguration is executed with new metric values).

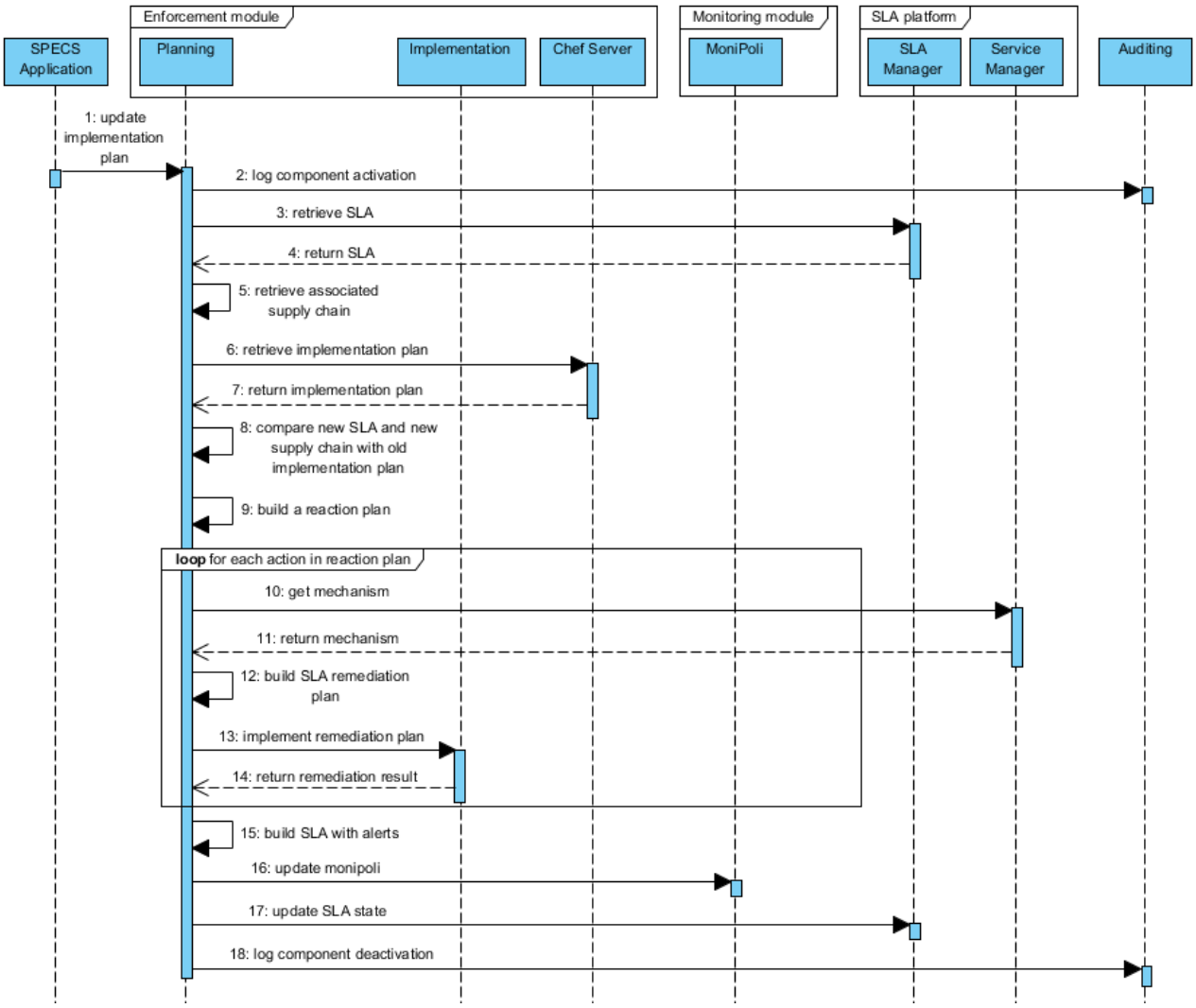

Figure 9. Building reaction plan

For all mechanisms involved in the list of fake violations the Planning retrieves mechanisms' deployment details from the SLA Platform ***f9[s10-11]***. When remediation plans for all prepared violations are extracted, the SLA remediation plan is built and implemented ***f9[s12-14]***. Implementation details will be provided in Section 3.3.

After the implementation of the SLA remediation plan (i.e., after the implementation of the reaction plan), the MoniPoli is updated ***f9[s15-16]***, the state of the SLA is updated to *Observed* ***f9[s17]***, and completion of the planning process is logged ***f9[s18]***.

More implementation details for the planning process after renegotiation will be provided at M30 in D4.3.3.

In case where the EU wants to terminate the SLA, the process is the same as described in Figure 9, except that we skip steps 3-5 and 8. For SLA termination the Planning component only needs the implementation plan to determine which resources and which services and components to terminate.

In the following subsections we provide with description of repository, and present installation and usage guides for the current Planning prototype.

3.2.1. Repository

The Planning component is implemented as a Maven-based Java project with two modules: planning-core and planning-api. It is designed using the Spring framework [40]. The source code can be found on the project's Bitbucket repository at [1].

3.2.2. Description and design

As mentioned in the previous section, the Planning component consists of two modules: planning-core and planning-api. The planning-core module contains the implementation of all the functionality supported by the Planning component with the corresponding Java API, while the planning-api module provides RESTful API wrapper around the Java API of the planning-core. The planning-core module is packaged as a Java library (JAR file), the planning-api module is packaged as a Java web archive (WAR file) which depends on the planning-core library. The persistence layer is based on the Spring Data framework which is integrated with the MongoDB database using the Spring Data MongoDB project.

3.2.3. Installation

The source code for the Enforcement Planning component can be found on project's Bitbucket repository at [1].

Prerequisites:

- Java web container
- MongoDB
- Java 7
- SPECS dependencies: SPECS Utility Data Model (available at [37])

The project can be built from source code using Apache Maven 3 tool. First clone the project from the Bitbucket repository using a Git client:

```
git clone git@bitbucket.org:specs-team/specs-core-enforcement-
planning.git
```

then go into the specs-core-enforcement-planning directory and run:

```
mvn package
```

The project is packaged as a web application archive file with the name planning-api.war which has to be deployed to a Java web container. For example, to deploy the application to Apache Tomcat, just copy the war file to the Tomcat webapps directory:

```
cp planning-api/target/planning-api.war /var/lib/tomcat7/webapps/
```

The application configuration is located in the file planning.properties in the Java properties format. The file contains the following configuration properties:

```
sla-manager-api.address=https://localhost/sla-manager-
api/sla_manager_rest_api
service-manager-api.address=https://localhost/service-manager-api/cloud-
sla
implementation-api.address=https://localhost/implementation-api
auditing-api.address=https://localhost/auditing
monipoli-api.address=https://localhost/monipoli
mongodb.host=localhost
mongodb.port=27017
mongodb.database=enforcement-planning
```

Make the necessary changes and restart the web container for changes to take effect. The Planning API should now be available at https://<host>:<port>/planning-api.

3.2.4. Usage

The Planning component provides REST API which is fully described in the deliverable D1.3.

The following tables provide a brief summary of offered resources and methods involved in the generation of supply chains and preparation of implementation plans orchestrated by the Planning component.

Resources	**Supply Chain Activities**: A collection of Supply Chain Activities maintained by the Planning component. **Supply Chain Activity**: An object representing a set of information needed to build Supply Chains, and a list of identifiers of associated built Supply Chains. **Supply Chains**: A collection of Supply Chain objects maintained by the Planning component. **Supply Chain**: An object representing a custom set of cloud resources and a set of security mechanisms' components to be deployed over the custom set of cloud resources in order to implement an SLA.
Methods	`GET/sla-enforcement/sc-activities`
	`POST/sla-enforcement/sc-activities`
	`GET/sla-enforcement/sc-activities/{sca-id}`
	`GET/sla-enforcement/sc-activities/{sca-id}/status`
	`GET/sla-enforcement/sc-activities/{sca-id}/sc-list`
	`GET/sla-enforcement/supply-chains`
	`GET/sla-enforcement/supply-chains/{sc-id}`
	`DELETE/sla-enforcement/supply-chains/{sc-id}`

Table 4. API associated to generation of supply chains

Resources	**Planning Activities**: A collection of Planning Activities maintained by the Planning component. **Planning Activity**: An object representing all information about built Implementation Plans associated to an SLA. **Reconfigurations**: A collection of Reconfigurations maintained by the Planning component. **Reconfiguration**: An object representing required reconfiguration of an Implementation Plan.
Methods	`GET/sla-enforcement/plan-activities`
	`POST/sla-enforcement/plan-activities`
	`GET/sla-enforcement/plan-activities/{pa-id}`
	`PUT/sla-enforcement/plan-activities/{pa-id}`
	`GET/sla-enforcement/plan-activities/{pa-id}/status`
	`GET/sla-enforcement/plan-activities/{pa-id}/plansnum`
	`GET/sla-enforcement/plan-activities/{pa-id}/planlist`
	`GET/sla-enforcement/plan-activities/{pa-id}/active`
	`GET/sla-enforcement/reconfigs`
	`POST/sla-enforcement/reconfigs`

Table 5. API associated to generation of implementation plans

3.3. Implementation component

As described in the previous Section 3.2, all implementation activities in SPECS are carried out according to the implementation plan that is generated in the planning step of the SLA

implementation phase. The initial SLA implementation process is described in D4.2.2. In the following we provide with a detailed description of the final version.

3.3.1. Overview

Implementation component (integrated with the Broker mechanism and a Chef Server, discussed in Section 3.6) orchestrates acquisition of resources and deployment and configuration of security mechanisms responsible for enforcing and monitoring security features agreed in the SLA.

Sequence diagram in Figure 10 outlines all details of the implementation plan execution.

After the Planning component triggers implementation of the built plan ***f10[s1]***, the Implementation component logs the start of the process ***f10[s2]***. First the implementation plan is retrieved ***f10[s3-4]*** and parsed ***f10[s5]***. According to the implementation plan, all cloud resources are acquired and configured with Chef recipes ***f10[s6]***. After each recipe has ran successfully (and the log from the Chef Server is returned ***f10[s7]***), the implementation plan is updated (e.g., with IP addresses) ***f10[s8]*** and activated services are logged ***f10[s9]***. More details about acquisition and configuration of resources are provided in Section 3.6.

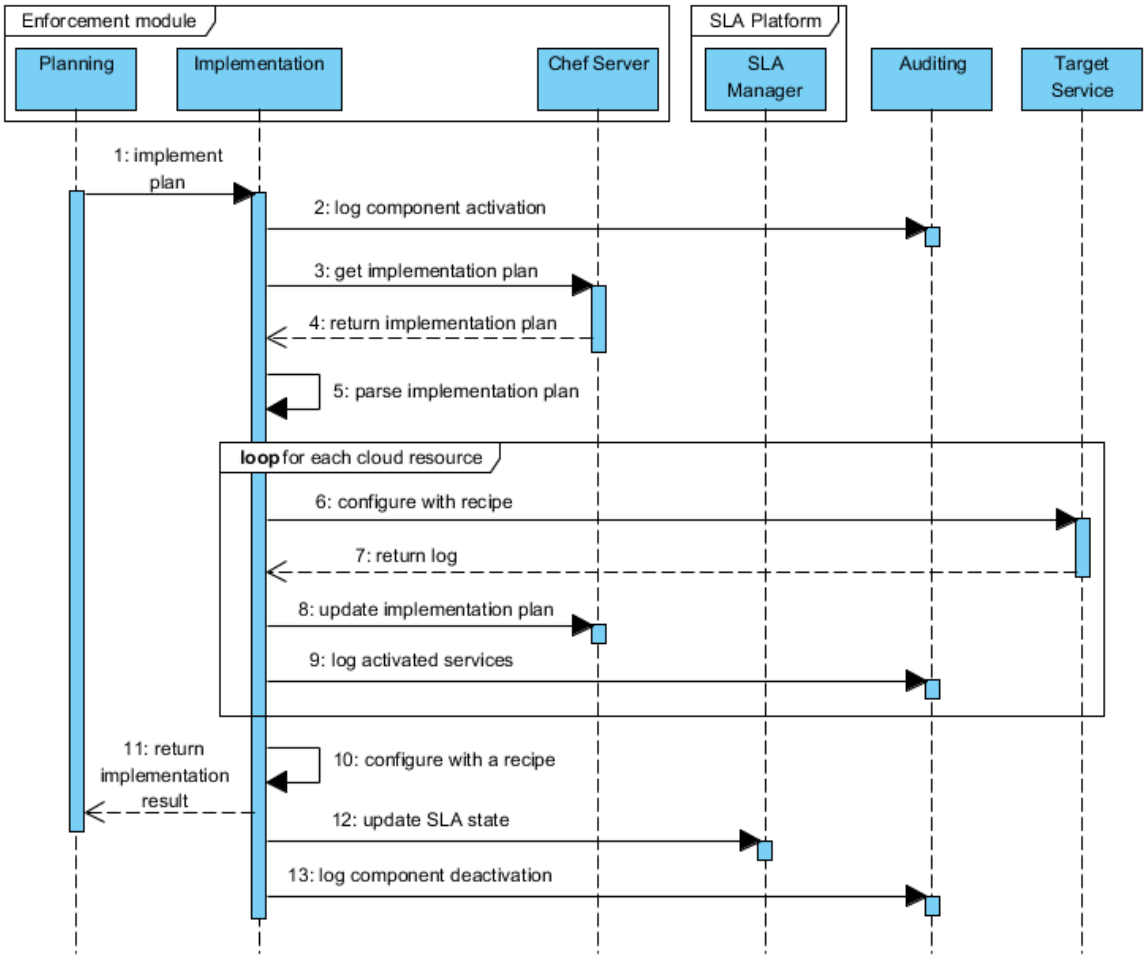

Figure 10. Executing implementation plan

After the cloud resources has been successfully configured, the internal SPECS components (e.g., monitoring aggregators, Chef Server) have to be adapted to manage newly implemented

SLA ***f10[s10]***. At the end of the implementation process, the result is reported to the Planning component (which updates the MoniPoli as discussed in Section 3.2) ***f10[s11]***, the SLA state is updated to *Observed* ***f10[s12]*** and the completion of the implementation process is logged ***f10[s13]***.

Once the signed SLA (i.e., the associated implementation plan) is successfully implemented, it enters the monitoring phase. And in case when monitoring adapters detect deviations from the agreed configurations and the Diagnosis component confirms that the SLA has been violated (or that the detected event implies a possible future SLA violation), the RDS component prepares a remediation plan (for details see Section 3.5). Generated remediation plan then has to be implemented in order to recover from the SLA violation or SLA alert. The process of implementing a remediation plan is depicted in Figure 11 and described below.

The RDS component triggers implementation of remediation plan ***f11[s1]*** and the Implementation component logs the start of the process ***f11[s2]***. The implementation plan associated to the alerted/violated SLA is retrieved ***f11[s3-4]*** and parsed ***f11[s5]***.

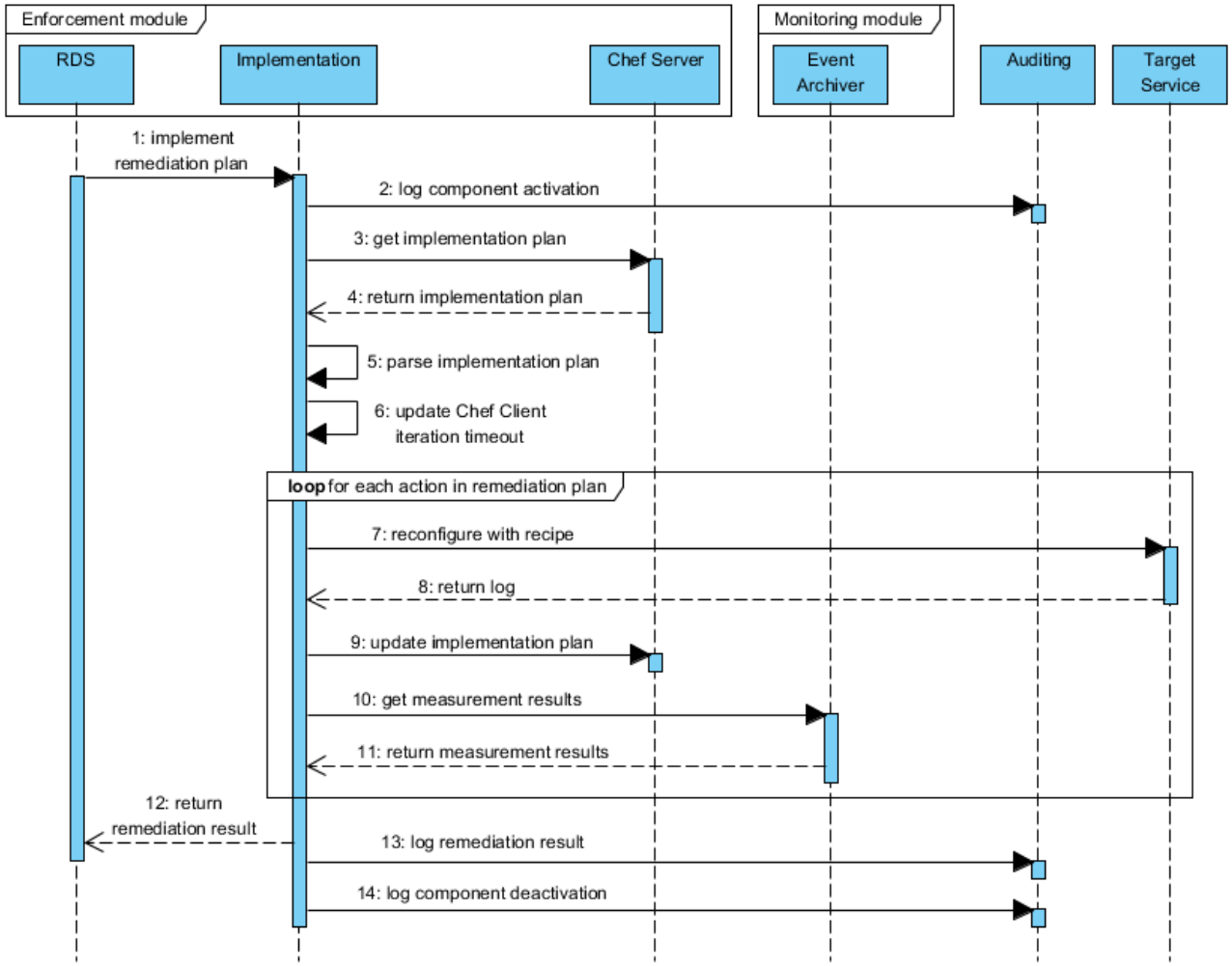

Figure 11. Implementing remediation plan

Each action in remediation plan (for details see Section 3.5) requires a

- reconfiguration of the target service and/or
- invocation of a measurement related to the reconfiguration on the target service.

For example, if the reconfiguration is related to restarting an unresponsive web server, the associated monitoring adapter is invoked to check availability of the restarted web server. But since measurement results cannot be obtained directly from the target service (all measurement results go from the monitoring adapter to the monitoring core and only then to the Enforcement module), the Chef Client timeout has to be adapted ***f11[s6]***.

Reconfigurations of target services are performed similarly as in the execution of the initial implementation plan. For each action in the remediation plan target service is reconfigured with a Chef recipe ***f11[s7]*** and after the Chef Server returns the log reporting successful run of the recipe ***f11[s8]***, the implementation plan is updated if needed (e.g., with new IP address) ***f11[s9]***. In order to determine whether the remediation action has eliminated the SLA alert/violation, the Implementation component queries the Event Archiver for a measurement result that has been invoked with the reconfiguration recipe ***f11[s10-11]***.

A sequence of remediation actions can either end with a successful elimination of the SLA alert/violation or we run out of actions to automatically remediate the alert/violation. The result of the remediation process is reported back to the RDS component ***f11[s12]*** which either updates the SLA state to *Observed* or notifies the EU that the automatic remediation has not been successful.

Implementation component logs the remediation result ***f11[s13]*** and the completion of the implementation process ***f11[s14]***.

As anticipated in Section 3.2, focused on the Planning component, the Implementation component is also in charge of implementing a remediation/reaction plan built after SLA renegotiation or after SLA termination.

Implementation of the remediation plan built by the RDS is invoked by the RDS and the invocation call is labelled with *Reconfigure*. After SLA renegotiation, the remediation/reaction plan is built by the Planning component, but the implementation process stays the same. The invocation call originating from the Planning component instead of RDS is labelled as *Reconfigure*. In case of SLA Termination, the invocation call originating from the Planning component is labelled as *Terminate*. In this case the implementation process is the same, except that steps ***f11[s6]*** and ***f11[s10-11]*** are skipped.

More implementation details for the implementation of remediation/reaction plan after SLA renegotiation or SLA termination will be provided at M30 in D4.3.3.

In the following subsections we provide with description of repository, and present installation and usage guides for the current Implementation prototype.

3.3.2. Repository

The Implementation component is implemented as a Maven-based Java project with two modules: implementation-core and implementation-api. It is designed using the Spring framework [40]. The source code can be found on the project's Bitbucket repository at [2].

3.3.3. Description and design

As mentioned in the previous section, the Implementation component consists of two modules: implementation-core and implementation-api. The implementation-core module

contains the implementation of all the functionality supported by the Implementation component with the corresponding Java API, while the implementation-api module provides RESTful API wrapper around the Java API of the implementation-core. The implementation-core module is packaged as a Java library (JAR file) and the implementation-api module is packaged as a Java web archive (WAR file) which depends on the implementation-core library. The persistence layer is based on the Spring Data framework which is integrated with the MongoDB database using the Spring Data MongoDB project.

3.3.4. Installation

The source code of the Enforcement Implementation component can be found on project's Bitbucket repository at [2].

Prerequisites:

- Java web container
- MongoDB
- Java 7
- Chef Server
- SPECS dependencies: SPECS Utility Data Model (available at [37])

The project can be built from source code using Apache Maven 3 tool. First clone the project from the Bitbucket repository using a Git client:

```
git clone git@bitbucket.org:specs-team/specs-core-enforcement-
implementation.git
```

then go into the specs-core-enforcement-implementation directory and run:

```
mvn package
```

The project is packaged as a web application archive file with the name implementation-api.war which has to be deployed to a Java web container. For example, to deploy the application to Apache Tomcat, just copy the war file to the Tomcat webapps directory:

```
cp implementation-api/target/implementation-api.war
/var/lib/tomcat7/webapps/
```

The application configuration is located in the file implementation.properties in the Java properties format. The file contains the following configuration properties:

```
sla-manager-api.address=https://localhost/sla-manager-
api/sla_manager_rest_api
planning-api.address=https://localhost/planning-api
event-archiver.address=https://localhost/event-archiver
auditing-api.address=https://localhost/auditing
monitoring-api.address=https://localhost/monitoring
mongodb.host=localhost
mongodb.port=27017
mongodb.database=enforcement-implementation
chef-server.organization
chef-server.organizationPK
chef-server.endpoint
chef-server.username
chef-server.password
```

Make the necessary changes and restart the web container for changes to take effect. The Implementation API should now be available at https://<host>:<port>/implementation-api.

3.3.5. Usage

The Implementation component provides REST API which is fully described in the deliverable D1.3.

The following table provides a brief summary of resources and methods related to the actual SLA implementation and implementation plans.

Resources	**Implementation Activities**: A collection of Implementation Activities maintained by the Implementation component. **Implementation Activity**: An object representing all information about the process of implementing an SLA. **Implementation Plans**: A collection of Implementation Plans maintained by the Chef Server. **Implementation Plan**: An object representing a detailed set of resources and their configurations required to implement an SLA.
Methods	`GET/sla-enforcement/impl-activities`
	`POST/sla-enforcement/impl-activities`
	`GET/sla-enforcement/impl-activities/{ia-id}`
	`GET/sla-enforcement/impl-activities/{ia-id}/status`

Table 6. API associated to implementation plans

3.4. Diagnosis component

During the SLA implementation phase, we deploy not only security mechanisms that enforce negotiated metrics but also components that are able to monitor them. Installed monitoring adapters continuously report about the status of the acquired resources and services running on top of them. Events are sent to the Monitoring module which aggregates, archives, and filters them according to the MoniPoli. Events that break at least one of the Monipoli rules are notified to the Diagnosis component.

The initial diagnosis process is described in D4.2.2. In the following we provide with a detailed description of the final version.

3.4.1. Overview

When an EU chooses a specific metric and sets a value to it, he/she basically sets a violation threshold for that metric which is then added to the MoniPoli in the form of a MoniPoli rule. For example, if an EU signs an SLA with an SLO *Vulnerability Scanning Frequency = 24h* (and for this metric we measure the age of the scanning report), the rule added to the MoniPoli is *report_age >24h*. This means that the MoniPoli will notify the Diagnosis each time the age of the scanning report is higher than 24h.

In order to prevent violations and introduce the so called alerts, we associate each metric with additional measurements and thus additional MoniPoli rules (alert threshold). For example, as seen in the example above, the metric *Vulnerability Scanning Frequency* is associated to the *report_age* measurement and this is its basic measurement. This means that each time this measurement deviates from what is expected, the associated SLA is violated. Each metric has one basic measurement, but in order to detect violations even before they occur, we introduce additional measurements. Additional measurements associated to the *Vulnerability Scanning Frequency* metric are related to availability of the vulnerability list and responsiveness of the vulnerability scanner (a list of published software vulnerabilities and a responsive vulnerability scanner are needed in order to perform vulnerability scan). If any of the

MoniPoli rules associated to the additional measurements is broken, an SLA alert is raised. This allows us to remediate the alert before an actual violation occurs.

In the SLA implementation phase all deployed monitoring adapters are configured so that all basic and all additional measurements associated to the entire set of SLOs/metrics in the SLA are continuously taken and sent to the monitoring core. For each security mechanism in Section 4 we report a list of metric that implement the mechanism and a list of associated basic and additional measurements.

The following are the details of the diagnosis process which is also presented with a sequence diagram in Figure 12 below.

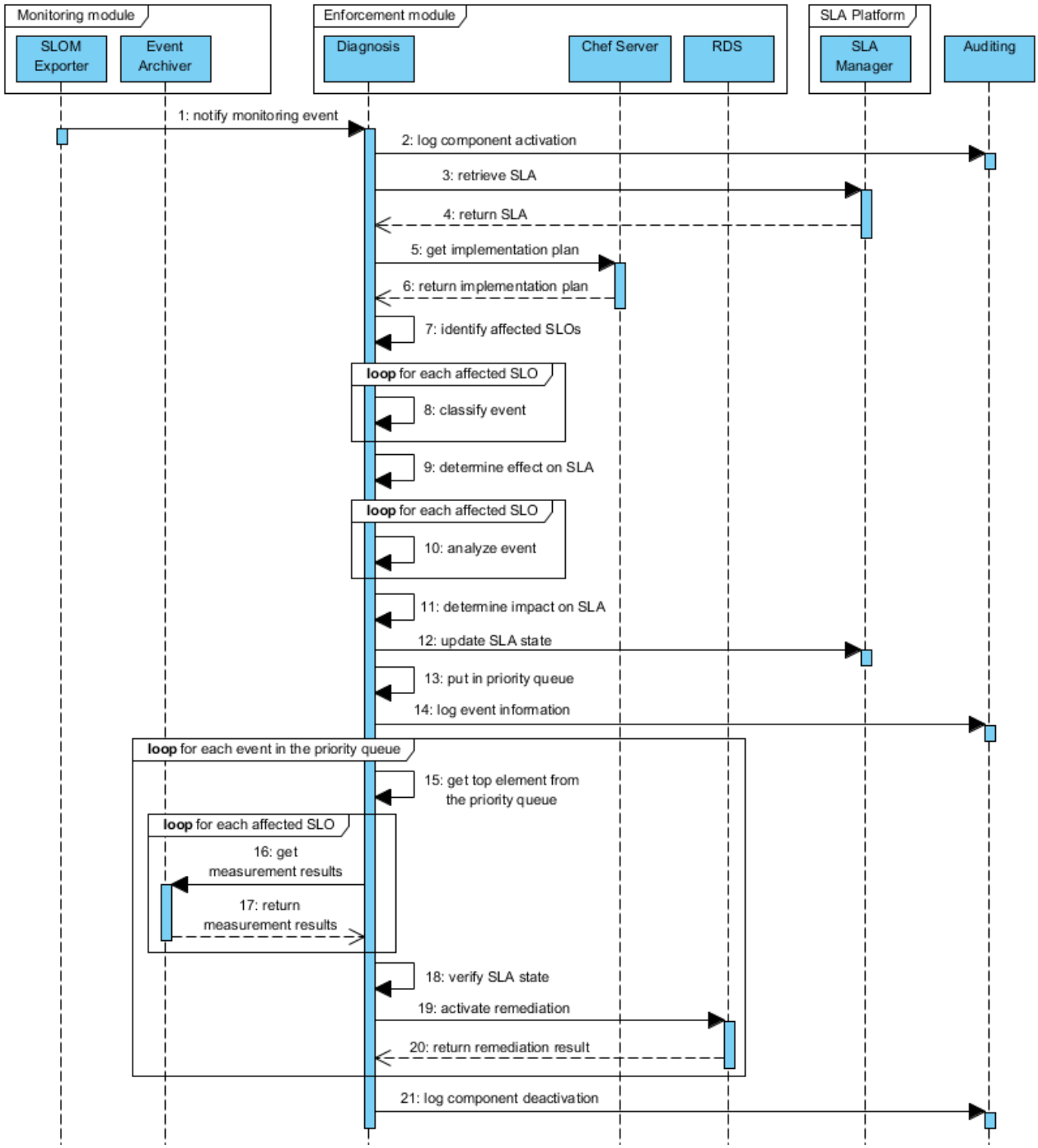

Figure 12. Diagnosis process

When the Diagnosis component is notified about a monitoring event ***f12[s1]***, the activation of the process is logged ***f12[s2]***. The notification includes the ID of the affected SLA, so the Diagnosis can retrieve it from the SLA Manager ***f12[s3-4]***. With the ID of the affected SLA the Diagnosis also queries the Chef Server for the associated implementation plan ***f12[s5-6]***. The affected SLOs are identified based on the measurement reported in the notification ***f12[s7]***.

For each affected SLO the Diagnosis performs classification ***f12[s8]***. This means that it has to determine whether the notified event represents an alert, a violation, or a false positive. This is done in two steps:

1. The value of the measurement reported in the notification is compared with the associated threshold specified in the implementation plan. If the reported value is below the threshold (or above the threshold, depends on the type of the measurement), the notified event is a false positive.
2. If the event is not a false positive, the Diagnosis checks if the measurement reported in the notification is a basic or an additional one. Deviations of basic measurements indicate SLO violations, deviations of additional measurements indicate SLO alerts.

Next, the effect on the entire SLA is determined ***f12[s9]***. If any of affected SLOs is labelled as violated, the SLA is violated. If all affected SLOs are labelled as alerted, the SLA is alerted. As already said, if none of the affected SLOs is either alerted or violated, the notified event represents a false positive.

When false positives are discarded, the notified event has to be analysed with respect to each affected SLO ***f12[s10]***. To determine the impact that the monitoring event has on each SLO, the Diagnosis has to numerically evaluate the risk level of an alert or the severity level of a violation. At this stage of the development, we introduce an innovative technique to model the risk/severity levels according to the event type and the importance levels assigned to the affected SLOs as shown in Table 7.

		Event type	
		Alert	Violation
		Risk level	Severity level
Importance level	Low	1	4
	Medium	2	5
	High	3	6

Table 7. Risk and severity levels of alerts and violations

More meaningful evaluation method would depend on the type of the measurement (e.g., boolean, numerical) associated to the notified monitoring event, the deviation of the expected value, and would take into account even dependencies among SLOs, costs and damages associated to occurrence of the event, and historical data. Considering that such a methodology for evaluating risk associated to an alert or a violation takes thorough experiments and thoughtful research, we might further develop it in the last phase of the project and formalize it at M30.

Considering that we maintain alerted/violated SLAs in a priority queue and we handle them according to the risk/severity level (SLAs with higher impact level first), this setting assures that all violations are handled before alerts, and that all SLOs with higher importance levels are remediated first. It may occur that the Diagnosis receives different notifications affecting one SLA at almost the same time. And to avoid performing different reconfigurations on a

target service at the same time, monitoring events have to be handled one by one and not in parallel. Hence the need for the priority queue.

The final impact that the notified monitoring event has on the SLA, is calculated as the maximum risk/severity level of all affected SLOs ***f12[s11]***.

When the event has been classified and analysed, and the SLA impact level has been determined, the state of the SLA is updated to Alerted/Violated ***f12[s12]***, the SLA is put in the priority queue (alerted/violated SLAs with the highest impact levels are put at the top and are remediated first) ***f12[s13]***, and all information related to the notified event is logged ***f12[s14]***.

Note that all SLAs with the same risk/severity level are placed in the priority queue according to the time of the event occurrence (SLAs for which the notified monitoring event occurred first are put on top of the list).

Before each alerted/violated SLA is pushed to the RDS component to find the best suited proactive/reactive solution, the Diagnosis component has to verify if the conditions of the alert/violation still persist. It may happen, for example, that during the time the alerted/violated SLA is in priority queue, the alert either escalates to a violation (in this case the Diagnosis received a new notification) or diminishes (in this case the SLA is simply removed from the priority queue and labelled as *Observed*). In case where a violation no longer persists, the SLA's state is changed to *Observed*, but the EU is still notified in order to have an opportunity to claim penalties.

The Diagnosis takes the top element from the priority queue ***f12[s15]***. In order to verify the state of an alerted/violated SLA ***f12[s18]***, the Diagnosis queries monitoring results from the Event Archiver for the measurements related to the affected SLOs ***f12[s16-17]***. Each alerted/violated SLA is sent to the RDS component to perform remediation ***f12[s19]***. Afterwards, the RDS reports if the alert/violation has been successfully eliminated ***f12[s20]*** in order to manage the next alerted/violated SLA. When the priority queue is emptied, the deactivation of the Diagnosis component is logged ***f12[s21]***.

An example of the diagnosis process for a monitoring event related to the SVA mechanism is provided in Appendix 3. Installation and usage guides for the current Diagnosis prototype are provided in the next two subsections.

In the following subsections we provide with description of repository, and present installation and usage guides for the current Diagnosis prototype.

3.4.2. Repository

The Diagnosis component is implemented as a Maven-based Java project with two modules: diagnosis-core and diagnosis-api. It is designed using the Spring framework [40]. The source code can be found on the project's Bitbucket repository at [3].

3.4.3. Description and design

As mentioned in the previous section, the Diagnosis component consists of two modules: diagnosis-core and diagnosis-api. The diagnosis-core module contains the implementation of all the functionality supported by the Diagnosis component with the corresponding Java API, while the diagnosis-api module provides RESTful API wrapper around the Java API of the

diagnosis-core. The diagnosis-core module is packaged as a Java library (JAR file), the diagnosis-api module is packaged as a Java web archive (WAR file) which depends on the diagnosis-core library. The persistence layer is based on the Spring Data framework which is integrated with the MongoDB database using the Spring Data MongoDB project.

3.4.4. Installation

The source code for the Enforcement Diagnosis component can be found on project's Bitbucket repository at [3].

Prerequisites:

- Java web container
- MongoDB
- Java 7
- SPECS dependencies: SPECS Utility Data Model (available at [37])

The project can be built from source code using Apache Maven 3 tool. First clone the project from the Bitbucket repository using a Git client:

```
git clone git@bitbucket.org:specs-team/specs-core-enforcement-
diagnosis.git
```

then go into the specs-core-enforcement-diagnosis directory and run:

```
mvn package
```

The project is packaged as a web application archive file with the name diagnosis-api.war which has to be deployed to a Java web container. For example, to deploy the application to Apache Tomcat, just copy the war file to the Tomcat webapps directory:

```
cp diagnosis-api/target/diagnosis-api.war /var/lib/tomcat7/webapps/
```

The application configuration is located in the file diagnosis.properties in the Java properties format. The file contains the following configuration properties:

```
planning-api.address=https://localhost/planning-api
implementation-api.address=https://localhost/implementation-api
rds-api.address=https://localhost/rds-api
sla-manager-api.address=https://localhost/sla-manager-
api/sla_manager_rest_api
auditing-api.address=https://localhost/auditing
mongodb.host=localhost
mongodb.port=27017
mongodb.database=enforcement-diagnosis
```

Make the necessary changes and restart the web container for changes to take effect. The Diagnosis API should now be available at https://<host>:<port>/diagnosis-api.

3.4.5. Usage

The Diagnosis component provides REST API which is fully described in the deliverable D1.3.

The following table provides a brief summary of resources and methods related to the actual SLA implementation and implementation plans.

Resources	**Notifications**: A collection of Notifications maintained by the Diagnosis. **Notification**: A message related to a monitoring event. **Diagnosis Activities**: A collection of Diagnosis Activities maintained by the Diagnosis. **Diagnosis Activity**: An object representing a set of information associated to a Notification object. Information contains affected SLA ID and classification result.
Methods	`GET/sla-enforcement/notifications`
	`POST/sla-enforcement/notifications`
	`GET/sla-enforcement/notifications/{n-id}`
	`GET/sla-enforcement/diag-activities`
	`GET/sla-enforcement/diag-activities/{da-id}`
	`GET/sla-enforcement/diag-activities/{da-id}/status`
	`GET/sla-enforcement/diag-activities/{da-id}/sla-id`
	`GET/sla-enforcement/diag-activities/{da-id}/classification`

Table 8. API associated to diagnosis of notifications

3.5. Remediation Decision System component

When a monitoring event has been analysed, the second step of the remediation phase is to identify the proper proactive/corrective actions to mitigate the risk of having a violation or to recover from one.

The initial remediation process is described in D4.2.2. In the following we provide with a detailed description of the final version.

3.5.1. Overview

For each security mechanism, the developer is expected to provide not only implementation and configuration details for the mechanism, but also actions needed to automatically manage mitigation of or recovery from violations related to security metrics and controls guaranteed by the mechanism.

Note that the details discussed below are completely new aspects of remediation process and have not yet been presented in any of previous deliverables.

As already mentioned (see Sections 3.2 and 3.4), each security metric is mapped to one or more basic and additional measurements. With each of these measurements alert and violation thresholds are set. Thus for each measurement a monitoring event is defined (i.e., an event where the alert/violation threshold is not respected) and a remediation plan has to be set (a set of actions needed to recover from an alert/violation).

For example, let us take a security mechanism that is offered through two security metrics, namely *SM1* and *SM2*. For each of these metrics we assign one basic measurement and some additional measurements as shown in Table 9 (metrics are reported in columns and associated measurements and their properties are presented in rows). For the validity of the SLOs related to both metrics, the thresholds for mapped measurements are as shown in the last column of the table, where *SM1_value* and *SM2_value* represent values for metrics *SM1* and *SM2* set by the EU during negotiation phase. For example, security metric *SM1* is associated with the basic measurement *BasMSR1* which can have integer values representing hours. When the EU selects a desired value *SM1_value* for metrics *SM1,* the threshold for the associated basic measurement *BasMSR1* is set to *SM1_value* (i.e., *BasMSR1≤ SM1_value*). Furthermore, each measurement is associated with one detectable monitoring event as shown in Table 10. For example, whenever the Monitoring module detects a deviation of

measurement *BasMSR1* (i.e., whenever the Monitoring module detects *BasMSR1> SM1_value*), this occurrence represents a violation of the SLO related to metric *SM1*.

Measurement			Measurement kind	Metrics		Threshold
ID	Type	Unit		SM1	SM2	
BasMSR1	integer	hours	basic	✓		BasMSR1 ≤ SM1_value
BasMSR2	boolean	n/a			✓	BasMSR2 = SM2_value
AddMSR1	boolean	n/a	additional	✓	✓	AddMSR1 = yes
AddMSR2	integer	hours		✓		AddMSR2 < SM1_value/2

Table 9. Measurements defined for metrics SM1 and SM2.

Monitoring event		Affected metrics		Event type
ID	Condition	SM1	SM2	
E1	BasMSR1 > SM1_value	✓		violation
E2	BasMSR2 != SM2_value		✓	
E3	AddMSR1 = no	✓	✓	alert
E4	AddMSR2 ≥ SM1_value/2	✓		

Table 10. Monitoring events for metrics SM1 and SM2

As described in previous section (Section 3.4, describing diagnosis process), remediation phase starts when the Monitoring module detects a suspicious behaviour, i.e., whenever it detects that some measurement value deviated from the defined threshold. To remediate such occurrences, a detailed remediation plan has to be defined specifically for each measurement. A remediation plan comprises a set of remediation actions and a clear sequence in which they should be executed. A remediation action is composed of some monitoring activity (i.e., take some measurement) or an enforcement activity followed by a monitoring action (i.e., change some configuration and check if the reconfiguration was successful).

The next table outlines remediation plans defined for alerts and violations of SLOs related to metrics SM1 and SM2 from the example above. Tables should be read from the top to bottom. Notation Ax refers to remediation action x.

If any chain of remediation actions ends in state O (result row reports O), that means that alert/violation has been successfully remediated and alert/violation no longer persist. In this case, the alerted/violated SLA can be put back into *Observed* state.

If the chain of remediation actions ends with action N that means that alert/violation could not have been automatically resolved and the EU should be notified about the event and asked for further assistance (whether the affected SLA should be renegotiated or terminated). The EU is informed about the event itself and about the affected SLOs.

Event	E1				E2				E3		E4			
Step1	A1				A4				A5		A6			
	yes		no		yes		no		yes	no	yes		no	
Step2	A2		A3		A3		A5		O	N	A2			N
	yes	no	yes	no	yes	no	yes	no			yes	No		
Step3	O	A3	O	N	O	N	O	N			O	A3		
		yes no										yes	no	
Step4		O N										O	N	

Table 11. Remediation plan for alerts and violations related to metrics SM1 and SM2

As already mention in the description of implementation of a remediation plan in Section 3.3, each remediation action requires some reconfiguration and invocation of a measurement related to the executed reconfiguration. Thus each action can either result in success (the measurement value is as expected and indicates successful reconfiguration) which is in the remediation plan denoted as a *yes*, or it can result in a failure (the measurement value indicates that the executed reconfiguration has not succeeded) which in the table above is denoted as a *no*.

Monitoring events, associated remediation plans, and remediation actions for each security mechanism are provided in dedicated parts of Section 4. Chef recipes for each security mechanism can be found on a dedicated Bitbucket site [33].

A simplified flow diagram for remediation process described above is presented in Figure 13.

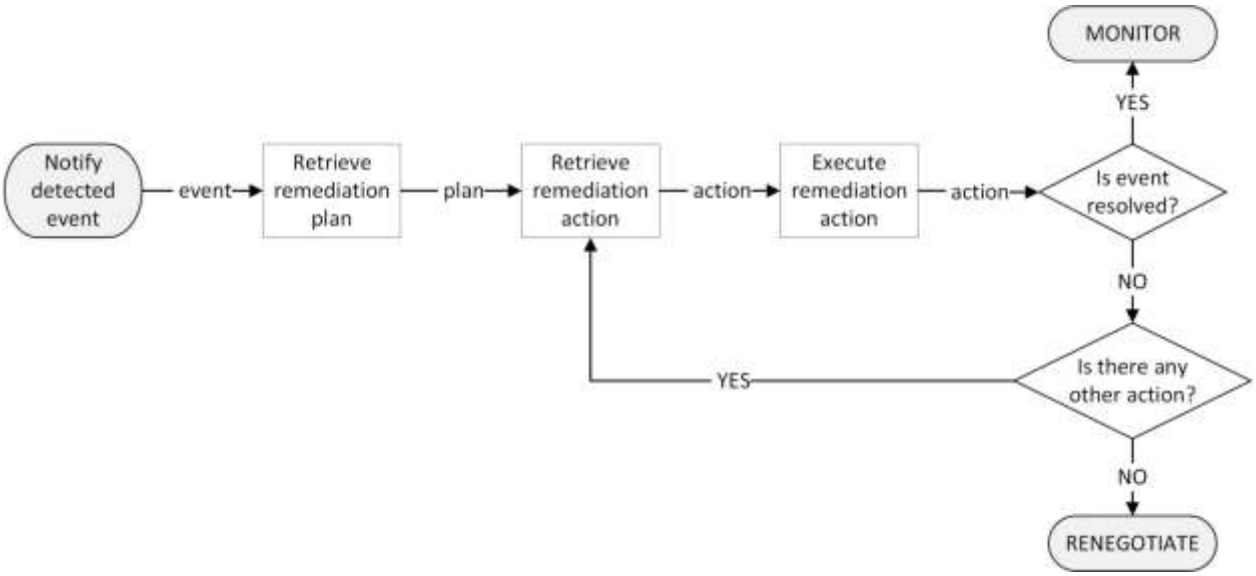

Figure 13. Remediation flow

The following description outlines the steps required to automatically mitigate from SLA alerts and violations (as shown in sequence diagram in Figure 14 below).

The Diagnosis component triggers remediation process by passing an alerted/violated SLA together with all information about the detected monitoring event to the RDS ***f14[s1]***. The RDS logs its activation ***f14[s2]*** and updates the state of the SLA to proactive redressing (in case of alerts) or remediating (in case of violations) ***f14[s3]***. The Diagnosis retrieves the implementation plan for the affected SLA ***f14[s4-5]***, and identifies affected capabilities ***f14[s6]***. Note that each SLO is mapped to one capability and this mapping is provided in the implementation plan.

For each affected capability a list of mechanisms able to implement it is retrieved from the Service Manager ***f14[s7-8]***. After the supply chain associated to the alerted/violated SLA is retrieved ***f14[s9-10]***, the RDS identifies affected mechanisms ***f14[s11]***. For each affected mechanism the RDS gets all mechanism related information from the Service Manager ***f14[s12-13]***, and extracts its remediation plan ***f14[s14]***. Finally, an SLA remediation plan is built ***f14[s15]*** which includes a sequence of actions required to mitigate the alert/violation. Note that mechanism's remediation plan combines actions for all possible monitoring events associated to the metrics the mechanism enforces and/or monitors. But the SLA remediation

plan only consists of remediation actions relevant for the detected monitoring event.

The RDS passes the SLA remediation plan to the Implementation component ***f14[s16]***. Execution of the remediation plan is discussed in Section 3.3.

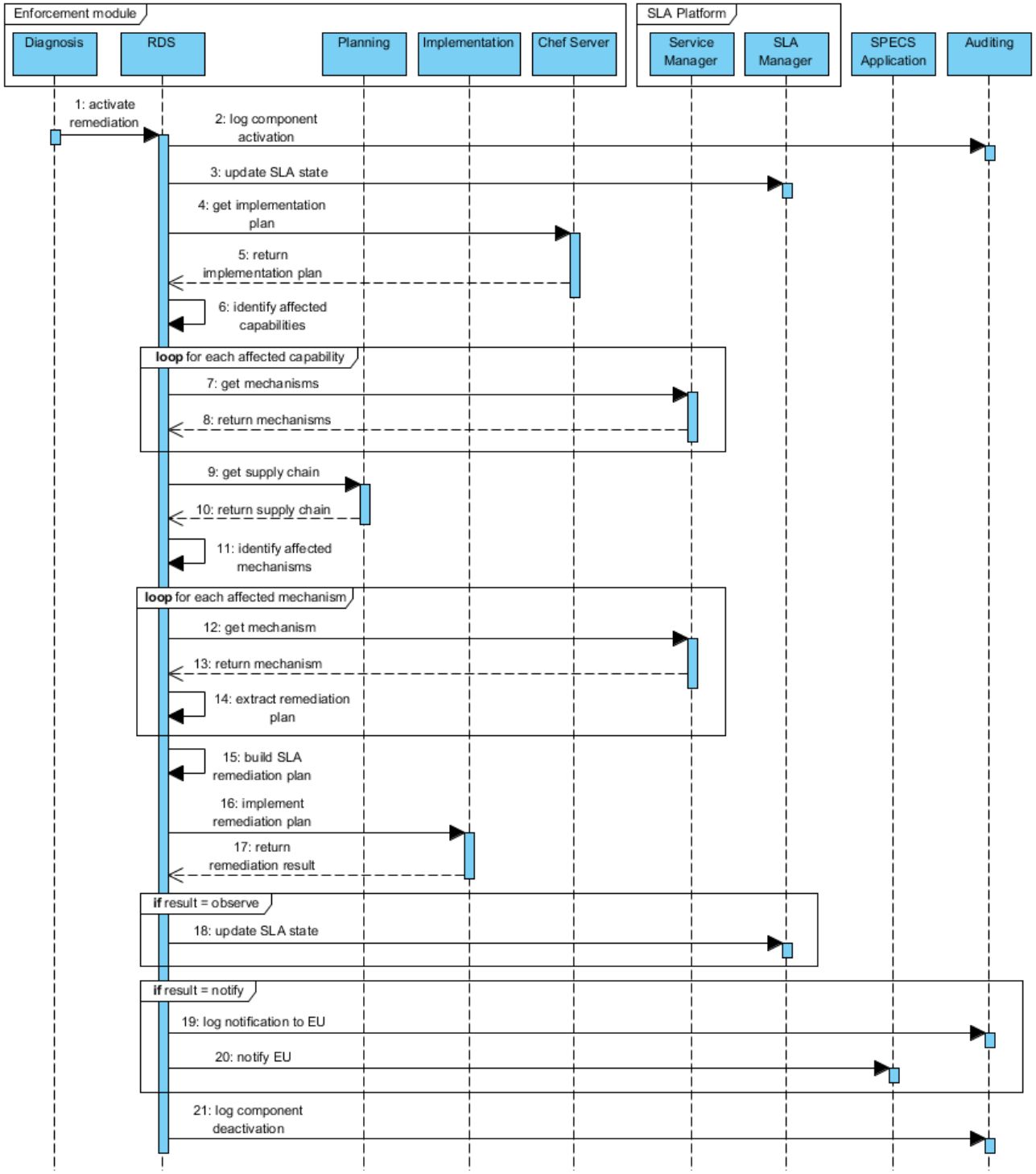

Figure 14. Remediation process

After all remediation actions reported in the SLA remediation plan has been executed, the Implementation component reports the result of the process ***f14[s17]***. As described above, the sequence of remediation actions can either end with a successfully mitigation of the SLA alert/violation, or we run out of actions to automatically remediate the alert/violation. Thus

the Implementation component either reports that the SLA alert/violation has been resolved, or it reports that remediation was unsuccessful which implies that the EU has to be involved (to either renegotiate the SLA, terminate the SLA, or accept the risks that the occurred event represents). If the alert/violation has been eliminated, the state of the SLA is updated to Observed ***f14[s18]***. Otherwise the EU is notified about the status of the SLA, the detected event, and about failed remediation actions ***f14[s20]***. Prior to that the notification sent to the EU is logged ***f14[s19]***. The remediation process ends with RDS logging its deactivation ***f14[s21]***.

In the following subsections we provide with description of repository, and present installation and usage guides for the current RDS prototype.

3.5.2. Repository

The Remediation component is implemented as a Maven-based Java project with two modules: remediation-core and remediation-api. It is designed using the Spring framework [40]. The source code can be found on the project's Bitbucket repository at [4].

3.5.3. Description and design

As mentioned in the previous section, the Remediation component consists of two modules: remediation-core and remediation-api. The remediation-core module contains the implementation of all the functionality supported by the Remediation component with the corresponding Java API, while the remediation-api module provides RESTful API wrapper around the Java API of the remediation-core. The remediation-core module is packaged as a Java library (JAR file), the remediation-api module is packaged as a Java web archive (WAR file) which depends on the remediation-core library. The persistence layer is based on the Spring Data framework which is integrated with the MongoDB database using the Spring Data MongoDB project.

3.5.4. Installation

The source code for the Enforcement Remediation Decision System component can be found on project's Bitbucket repository at [4].

Prerequisites:

- Java web container
- MongoDB
- Java 7
- SPECS dependencies: SPECS Utility Data Model (available at [37])

The project can be built from source code using Apache Maven 3 tool. First clone the project from the Bitbucket repository using a Git client:

```
git clone git@bitbucket.org:specs-team/specs-core-enforcement-rds.git
```

then go into the specs-core-enforcement-rds directory and run:

```
mvn package
```

The project is packaged as a web application archive file with the name rds-api.war which has to be deployed to a Java web container. For example, to deploy the application to Apache Tomcat, just copy the war file to the Tomcat webapps directory:

```
cp rds-api/target/rds-api.war /var/lib/tomcat7/webapps/
```

The application configuration is located in the file rds.properties in the Java properties format.

The file contains the following configuration properties:

```
planning-api.address=https://localhost/planning-api
implementation-api.address=https://localhost/implementation-api
sla-manager-api.address=https://localhost/sla-manager-
api/sla_manager_rest_api
auditing-api.address=https://localhost/auditing
mongodb.host=localhost
mongodb.port=27017
mongodb.database=enforcement-rds
```

Make the necessary changes and restart the web container for changes to take effect. The RDS API should now be available at https://<host>:<port>/rds-api.

3.5.5. Usage

The RDS component provides REST API which is fully described in the deliverable D1.3.
The following table provides a brief summary of resources and methods related to the SLA remediation orchestrated by the RDS component.

Resources	**Remediation Activities**: A collection of Remediation Activities maintained by the RDS component. **Remediation Activity**: An object representing all information related to an SLA remediation process. **Remediation Plans**: A collection of Remediation Plans maintained by the RDS component. **Remediation Plan**: An object representing a chain of actions required for an SLA remediation.
Methods	`GET/sla-enforcement/rem-plans`
	`GET/sla-enforcement/rem-plans/{rp-id}`
	`GET/sla-enforcement/rem-plans/{rp-id}/result`
	`GET/sla-enforcement/rem-activities`
	`POST/sla-enforcement/rem-activities`
	`GET/sla-enforcement/rem-activities/{ra-id}`
	`GET/sla-enforcement/rem-activities/{ra-id}/status`

Table 12. API associated to the SLA remediation

3.6. Broker mechanism and Chef Server

The Broker component is used to handle the whole process of acquiring, deploying, and configuring a new resource available on the CSP. In particular, this component allows to acquire a new virtual machine on Amazon or on Eucalyptus each machine is configurable by defining the location and the system requirements, i.e., the operating system to install, the CPU type and the Ram size; once the virtual machines have been acquired, or more in general, if one or more virtual machines are already available, it is possible to configure each of them with any software using the Chef configuration management tool [31].

Initial architecture reported two components, namely Resource Broker (for acquiring and configuring resources) and Broker Configuration Manager (to configure Resource Broker). In Y2 both components are merged into one. Moreover, current Broker implementation also integrates Chef Server. Thus the final architecture of the Broker mechanism is as follows:

- **Resource Broker (Broker)** acquires cloud resources.
- **Chef Server** configures cloud resources and services running on top of them.

As described in Section 3.2, the Planning component generates implementation plans that include the characteristics of the CSP and required configurations of each virtual machine in terms of both resources and security mechanisms that have to be installed on each of them.

As already described in D4.2.2, in SPECS the Chef is used for automatic configuration management and orchestration. All deployment, management, and configuration details reported in each implementation plan are handled in terms of Chef *cookbooks* and *recipes*. The following is the summary of the Chef architecture (also depicted in Figure 15):

- **Chef Server** is the centralized store for configuration data in the infrastructure. It stores and indexes cookbooks, environments, templates, metadata, files, and distribution policies. Chef Server is aware of all machines it manages, and in this way, Chef Server also acts as an inventory management system.
- **Chef Workstation** is the location from which *cookbooks* and *recipes* are authored, policy data (i.e., *recipes*, *cookbooks*) are defined, data is synchronized with the *chef-repo* (the location in which the *cookbooks*, *recipes*, etc., are stored) and data is uploaded to the Chef Server. For example, the Enforcement Implementation component is running on the workstation and uses a *knife* to assign recipes to the VMs. The initial set of *recipes* is to be prepared by the SPECS developers, while others might be defined and added individually by the SPECS owners.
- **Chef Nodes** contain a *chef-client* (i.e., an agent that runs locally on every *node* that is under management by Chef) and perform various automation tasks. The *nodes* use the chef-client to ask the Chef Server for configurations (*recipes*, *templates*), and the chef-client then does the configuration work on the *nodes*. In SPECS, each of the machines (physical or virtual machine) that are to be security-hardened will take a role of a Chef *node*. Communicating with a chef-client, the machine will receive a configuration (from Chef server) which is supposed to provide a desired security level.

A prerequisite is the presence of a machine that hosts a Chef Server: it can be either a custom installation (On-Premise) or the Hosted solution provided by Chef itself: in both cases, each Chef Server is identified by an IP address, a username and a password.
Once the Chef Server has been properly configured, it is necessary to create or update one or more cookbooks each of one describe the security mechanism you want to install. When this preliminary procedure has been completed, it is possible to use the broker component in all its functionalities. Please note that the implementation plan provided by the Planning component has to be stored into the Chef Server, so the Broker offers also this functionality.

Since the Broker component is used to acquire and configure resources from a CSP, it is been developed as a library, so anyone can import and use it, but in SPECS project it is been imported into the Implementation component, since this is the component that has to aim to acquire and configure resources. As said before, in order to use the Broker, it is necessary to first upload the cookbooks on the Chef Server, and then the implementation plan provided by the Planning component; so the Broker is able also to store and retrieve an implementation plan on the Chef Server.

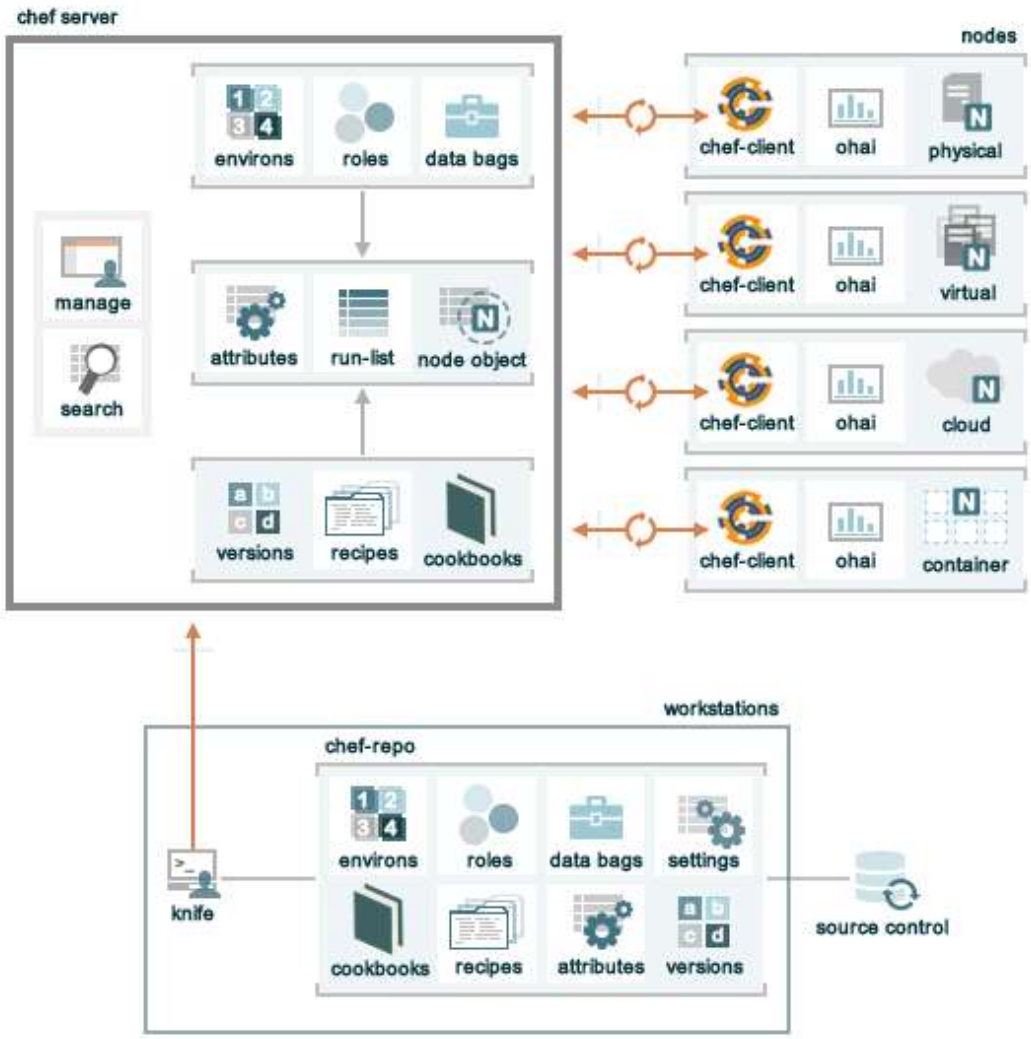

Figure 15. Chef architecture

In the following subsections we provide with description of repository, and present installation and usage guides for the current Broker prototype.

3.6.1. Repository

The component is actually made up of two sub-components, the Broker Configuration Manager and the Resource Broker: they are available on Bitbucket, in the same package, at [30].

3.6.2. Description and design

The main functionalities provided by the Broker component are: (i) to enable the access and usage of an external CSP, (ii) to acquire or delete a cluster of VMs on one of the enabled external CSPs and (iii) to execute scripts on a cluster of VMs.

As already discussed in D4.2.2, at the state of the art, a great number of solutions for brokering cloud resources exists. These solutions can be used in two different ways: as a closed application can be installed, configured and used or as a library that enables to easily develop a custom broker application.

In the context of SPECS, the Broker is used in many different scenarios and by other SPECS internal components. Therefore it should be easily adapted to the request that will depend on the different SPECS application that can be developed on top of the framework.

So, we have developed a new simple cloud application, based on jclouds. In order to provide these functionalities, the following components were customized:

- **Broker Configuration Manager:** manages Broker configurations.
- **Resource Broker:** acquires and manages external CSPs' resources.

3.6.3. Installation

The source code of the Broker mechanisms is available on the project's Bitbucket site [30].

Prerequisites:

- Git client
- Maven
- Java 7

To install the Broker, here are the general steps:

- clone the git repository;
- convert it into a Maven project;
- execute the 'maven install' command in order to execute tests and to generate the artifact;

In particular, if you're using Eclipse as IDE, here there is a detailed explanation of the necessary steps you have to use to install both projects:

- Import project from git as a "general project";
- right click on the project, click on "configure", then click on "Convert to Maven Project";
- right click on the project, click on "Run as", then click on "Maven install".

The Broker generates an artifact (a jar file) that can be used by any project that want to use the functionalities provided by the Broker itself: in order to use it, it is just necessary to add this jar file as a dependency in the pom file of the project that wants to use it.

3.6.4. Usage

Once the Broker has been properly configured as a dependency of the project, you can use all the functionalities it provides.

To better understand how to use the Broker, we can divide its functionalities into *acquisition* and *configuration* phases.

The java class that allows handling the acquisition phase is called `CloudServiceImpl` and is located into package `eu.specs.project.enforcement.broker`. The constructor of this component, as shown here, takes as argument a string representing the provider, the default username of the machine you want to acquire, and an instance of the `ProviderCredential` class that stores the information about the credentials useful to acquire resources from the provider:

```
public CloudServiceImpl (String provider, String defaultUser,
ProviderCredential providerCredential)
```

Once an instance of the `CloudServiceImpl` class has been created, you can call the method `createNodesInGroup`, with the following method:

```
public NodesInfo createNodesInGroup(String groupName, int
numberOfInstances, InstanceDescriptor descriptor, NodeCredential
nodeCredential, int... inboudports)  throws NoSuchElementException,
Exception)
```

The used parameters are:

- `groupName`: the "group" associated to the nodes you want to acquire;
- `numberOfInstances`: the number of machines you want to acquire;
- `descriptor`: an instance of the `InstanceDescriptor` class that represents the characteristics of each machine;
- `nodeCredentia`: an instance of the `NodeCredential` class that represents the credential used to configure the machine (public and private key so you can access the machine without user and password);
- `inboudports`: the ports that you want to enable on each machine (configured into the firewall provided by the CSP itself).

Once the resources have been acquired, you can execute a script on a node, using the following method:

```
public executeIstructionsOnNode(String user, ClusterNode node, String[]
istructions, String privateKey, boolean sudo, CloudServiceImpl compute)
```

The parameters are explained here:

- `user`: username of the OS user that has to execute the script (i.e. root)
- `node`: represents the node on witch you want to execute the script;
- `istructions`: represents the script(s) you want to execute;
- `privateKey`: the private key useful to access the remote machine;
- `sudo`: state if the script has to be run in sudo mode;
- `compute`: an instance of the class `CloudServiceImpl`.

At this point, it is possible to configure the software you want to install on each node. As stated before, it's necessary to upload the implementation plan on the Chef Server: in SPECS this procedure is done by the Planning component, but it can be called directly with the method `uploadDatabagItem` provided by the class `ChefServiceImpl`. In order to use this and all the methods provided by this class, you have to instantiate an object of this class calling its constructor whose sign is shown here:

```
public ChefServiceImpl (String organization, String organizationPK,
String chefServerEndpoint, String username, String passwordPK);
```

The parameters are:

- `organization`: represents the organization registered on the Chef Server;
- `organizationPK`: the private key used to access the Chef Server;
- `chefServerEndpoint`: the IP address of the chef server;
- `username`: the username of the user registered on Chef Server;
- `passwordPK`: the private key of the user registered on Chef Server;

Once you got an instance of `ChefServiceImpl` class, it's possible to use its methods.

In order to upload an implementation plan on it, you can use the following method:

```
public void uploadDatabagItem(String databagName,String
databagItemId,String databagItemValue);
```

The parameters are:

- `databagName`: the name of the databag you want to update;
- `databagItemId`: the identifier of the databag;
- `databagItemValue`: the databag itself, the implementation plan.

Once the databag has been properly uploaded on Chef Server, you can install the chef-client on each virtual machine acquired, by calling the following method:

```
public ChefNodeInfo bootstrapChef(String group, NodesInfo
nodes,CloudServiceImpl cloudservice,String attribute) {
```

The parameters are:

- `group`: the same group defined before;
- `nodes`: the list of nodes previously acquired;
- `cloudservice`: the instance of the `CloudServiceImpl` class;
- `attribute`: represents the Json whose value can be read by each recipe; it's a way to pass parameters to each recipe. Since our recipes needs the information located in the implementation plan that has to be uploaded on the Chef Server (as shown before), you can use the following structure:
  "String attribute=
  "{\"implementation_plan_id\":\""+*databagId*+"\"}";"
  where `databagId` has to be the identifier of an exsisting Databag on the chef server.

Once the bootstrap phase has finished, it's possible to execute recipes on each node, through the method:

```
public void  executeRecipesOnNode(ClusterNode node, List<String>
runList,String group,CloudService compute,NodeCredential nodecred);
```

The parameters are:

- `node`: an instance of the class `ClusterNode` that represents the node;
- `runList`: is a list of strings; each element is the name of the recipe you want to execute on that node;
- `group`: the same group defined before;
- `compute`: the instance of the `CloudServiceImpl` class;
- `nodecred`: the instance of the `NodeCredential` class that contains the credential of the node.

4. Security mechanisms

This section presents details on security mechanisms and their prototypes developed in year two. These mechanisms can be negotiable by any SPECS application and, in order to give a practical understanding of their usage, these mechanisms are here presented in the context of two user stories refined in D5.1.2, namely *Secure Storage* (STO) and *Secure Web Container* (WEB). Each of these user stories is implemented with a dedicated security service offered by SPECS. And for each of these services SPECS offers one mandatory security capability (implemented by negotiable security mechanisms) and a set of optional security capabilities according to the SLA. Mapping is shown in Table 13.

Security service	User story	Negotiable security mechanisms		
		Mandatory	Optional	
Secure Web Server	WEB	WebPool	SVA	TLS
Secure Storage	STO	DBB	E2EE	SVA

Table 13. Negotiable security mechanism offered by SPECS through different services

The Secure Web Server service is implemented with the WebPool mechanism (also denoted as Secure Web Server mechanism; see Section 4.2) which provides pools of web servers and assures resilience to security incidents through redundancy and diversity. Further security features can be guaranteed with Software Vulnerability Assessment tools (SVA) and TLS security mechanism, discussed in Sections 4.4 and 4.5, respectively.

The Secure Storage service is implemented with the DBB mechanism (Database and Backup mechanism; see Section 4.3) which provides storage and assures business continuity through backup. Further enhancements of security are possible with deployment of E2EE mechanism (End-2-end encryption mechanism; see Section 4.3) and SVA mechanism.

SVA mechanism offers evaluation of the security level of the system achieved through periodic vulnerability scans and reports about available updates and upgrades of vulnerable libraries on the system. TLS mechanism ensures communication privacy with a set of possible configurations for the TLS protocol (such as cryptographic strength, certificate pinning, HTTP to HTTPS redirection, etc.). E2EE mechanism offers end-2-end encryption to guarantee security and integrity of the stored data within the secure storage service.

Figure 16 below presents component diagram for Enforcement module's security mechanisms. Available security mechanisms are presented in the following subsections. Note that components of the Vertical Layer are discussed in deliverables of the task T1.4. In particular, Security Tokens and Credential Service mechanisms are discussed in deliverables D4.4.1 and D4.4.2, and the Auditing component and the User Manager component are described in D1.4.1 and D1.4.2. Finally, details about AAA and DoS Mitigation mechanisms will be reported in deliverable D4.3.3.

In the following dedicated subsections, each mechanism is described in detail. Considering the requirements and design have already been provided and discussed in D4.1.2 and D4.2.2, respectively, the focus in this document will be on one side on refinements of the architecture and functionalities due to the feedback received from the developers of mechanisms and from the developers of the SPECS flow (i.e., core processes, namely (re)negotiation, implementation, monitoring, and remediation), and on current and future implementation activities on the other side.

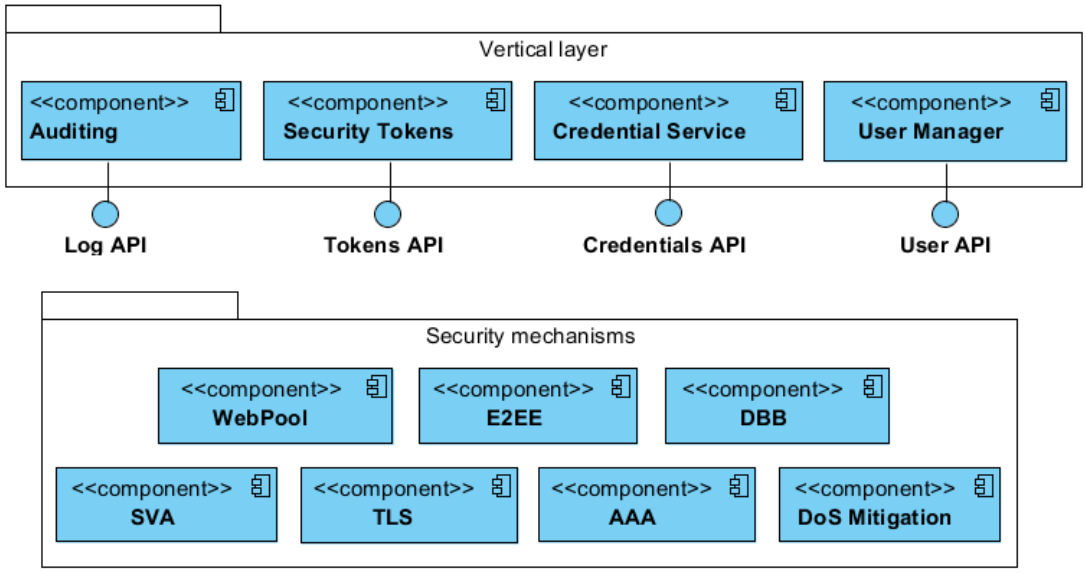

Figure 16. Enforcement module's high level architecture (security mechanisms)

As mentioned in the introduction of this document, feedback from stakeholders involvement activities conducted during the second year of the project has also been taken into consideration. Along with the strong interest for end-2-end encryption came the need for a mechanism offering secure storage equipped with the backup functionality. Table 14 presents new requirements for such a mechanism arising from T5.1 and T5.2.

REQ_ID	Requirement	Description
ENF_DBB_R1	Offer secure storage	The mechanism must be able to automatically offer secure storage in the cloud.
ENF_DBB_R2	Assure business continuity with backup	The mechanism must be able to guarantee business continuity with backup.

Table 14. New requirements for SPECS security mechanisms

For each security mechanism discussed in this document a list of associated security metrics, measurements, and security controls (from NIST [24] and CSA's CCM [25] frameworks) is presented. For each measurement associated to a metric, a set of detectable monitoring events, remediation actions, and remediation plans is defined. Brief installation and usage guides are also provided.

A summary of all security metrics associated to security mechanisms discussed in this deliverable is provided in Appendix 5.

Implementation of negotiable security mechanisms follows the initial implementation plan introduced in D4.3.1, but EU's requirements have also been taken into account (which is shown in the refinement of the initial implementation plan presented in D4.5.2 and D5.1.2). The majority of development efforts were focused on DBB, E2EE, and SVA mechanisms. In parallel, some improvements of WebPool mechanism were performed, and the initial prototype of the TLS mechanism has been developed.

The rest of negotiable mechanisms (AAA and DoS) designed in D4.2.2 will be introduced in the final iteration of this document.

Note that secure interaction mechanisms (Credential Service and Security Tokens) are discussed in the dedicated deliverable D4.4.2.

4.1. Status of development activities

In Table 15 we present coverage of requirements associated to security mechanisms discussed in this deliverable.

Requirements for security mechanisms	SPECS Security mechanisms				
	Secure Web Server mechanism (WebPool)	DBB mechanism (DBB)	Encryption mechanism (E2EE)	SVA Security mechanism (SVA)	TLS Security mechanism (TLS)
ENF_POOL_R1-R5	X				
ENF_TLS_R1-R5					X
ENF_SVA_R1-R4				X	
ENF_CRYPTO_R1-R4			X		
ENF_DBB_R1-R2		X			

Table 15. SPECS Security mechanisms and related requirements (already implemented)

There are 20 requirements related to security mechanisms discussed in this document, where 18 of them have already been covered with prototypes presented in this deliverable.

The prototypes of the E2EE, DBB, and TLS mechanisms cover all of the associated requirements. Future improvements will be focused on performance and robustness.

Current prototype of the WebPool implements 4 out of 5 of all associated requirements. The remaining one, *ENF_POOL_R5*, will possibly be covered with the final implementation of the mechanism.

SVA mechanism implements 3 out of 4 associated requirements. The remaining one, namely *ENF_SVA_R3* (related to automatic upgrades of vulnerable libraries and fixes of misconfigurations) will most likely remain uncovered due to the complexity of the problem. In the last six months of the project the goal is to integrate OpenVAS and Nikto scanners into the existing design to support all defined metrics and activities.

The current status of development activities is summarized in Table 16. In the last 6 months of the project remaining effort will mainly be spent on implementing the remaining uncovered requirements as discussed above, increasing performance and robustness, and developing the remaining two security mechanisms, namely AAA and DoS. The final results will be reported in D4.3.3 at M30.

Module	Artifacts under development	Status
Enforcement module	component:WebPool	Available
	component:DBB	Available
	component:E2EE	Available
	component:SVA	Available
	component:TLS	Available

Table 16. Enforcement module implementation status

The first prototypes of all mechanisms demonstrated in this deliverable are available on the project's Bitbucket repositories:

- The WebPool mechanism is available at [32].
- The E2EE and DBB mechanisms are available at [15], [16], [17], and [41].
- The SVA mechanism is available at [5], [6], [7], and [8].
- The TLS mechanism is available at [29].

All Chef recipes needed for deployment and configuration of security mechanisms and all configuration details for security mechanisms (mechanisms' metadata) are available at [19].

4.2. Secure Web Server mechanism

In this section, we present a description and implementation details for the mechanisms involved in the following validation scenarios defined and refined in T5.1 (see D5.1.2):

- *SWC-02 Secure_Web_Container_Brokering*
- *SWC-03 Secure_Web_Container_TLS_Enhanced*
- *SWC-04 Secure_Web_Container_SVA_Enhanced_Alert*
- *SWC-05 Secure_Web_Container_TLS_SVA_Enhanced_Violation*
- *SWC-06 Secure_Web_Container_TLS_Multitenancy*
- *SWC-07 Secure_Web_Container_Web_Pool_Replication_Enhanced_Alert*
- *SWC-08 Secure_Web_Container_Web_Pool_Replication_Enhanced_Violation*

4.2.1. Overview

Secure Web Server is a security mechanism whose aim is to guarantee the level of diversity and the level of redundancy defined into the SLA signed by the EU. The level of redundancy states the number of VMs it's necessary to acquire from the CSP, while the level of diversity defines how many different web containers have to be installed. It's important to note that the total number of VMs that have to be acquired is given by the level of diversity. Apart from that, we need additional VM that is configured to act as a load balancer/proxy to properly forward the HTTP traffic towards the different web container instances, depending on traffic features and on-going alerts/violation detections.

The level of diversity and the level of redundancy are both defined by the EU in the SLA negotiation phase, but the actual implementation details are settled by the Planning component during generation of supply chains.

4.2.1.1. Architecture

The architecture of the WebPool mechanism is composed of two components already introduced in D4.2.2:

- **Web Container Pool Manager** component cooperates with the Broker component in order to acquire and configure web servers. Serves as a load balancer and a monitoring component for all deployed web servers, and is implemented by HAProxy [34].
- **Pool Agent** is a web servers deployed on an acquired VM. Current implementation supports two different web servers, namely Nginx [35] and Apache [36].

Note that all the http requests that any user can send, have the HAProxy machine as destination: the load balancer, according to the balancing rules defined into its configuration file, forwards each incoming request to one of the virtual machines that host the web

container.

Because of the presence of the HAProxy component, it has been necessary to make the machines that hosts web containers able to store information about the session, since it is not guaranteed that the requests starting from the same client are handled by the same web container. The software used to handle the session is memcached.

4.2.1.2. Security metrics and controls

Security metrics associated to the WebPool mechanism are defined in the following two tables. For each metric we provide a description, possible values with units, default values, and actions that need to be taken in order to enforce the metric. These actions are periodically performed by the load balancer deployed on EU's target services. Note that setting a metric to its default value ensures the maximum possible level of security associated to that metric.

Name		Value	Default value	Unit
Level of redundancy (**LOR**)		int > 0	2	/
Description	This metric sets the minimum number (with respect to EU's requirements and technological constraints) of web server engines which are set-up and kept active throughout the service operation to increase the protection from attacks and vulnerabilities exploits. For example, *level_of_redundancy = 3*, ensures that there are at least three web servers running.			
Actions taken to enforce the metric	First, a required number web servers are installed and run. Periodically: 1. Check if all the web servers installed are properly running.			

Table 17. WebPool security metric LOR

Name		Value	Default value	Unit
Level of diversity (**LOD**)		int > 0	2	/
Description	This metric sets the number of different web server types available on target VMs. For example, for *level of diversity = 2*, SPECS ensures that there are at least two different types of web servers deployed and available.			
Actions taken to enforce the metric	First, a required number of different types of web servers are installed and run. Periodically: 1. Check if all different types of web servers installed are properly running.			

Table 18. WebPool security metric LOD

As described in Section 3, we associate each WebPool metric with a basic measurement and one or more additional measurements (with which the alert/violation thresholds are set and MoniPoli rules are built). The following two tables present all measurements together with MoniPoli rules associated to WebPool metrics.

Metric	Level of redundancy (LOR)
SLO	level_of_redundancy = N
Measurements	**MoniPoli rules**
number_of_servers	number_of_servers ≥ N

Table 19. Measurements and MoniPoli rules associated to WebPool metric LOR

Metric	Level of diversity (LOD)
SLO	level_of_diversity = N
Measurements	MoniPoli rules
diversity_level	diversity_level ≥ N

Table 20. Measurements and MoniPoli rules associated to WebPool metric LOD

The WebPool metrics defined above implement NIST and CCM security controls presented in the following table.

Control Family/Group	Control Name	Control ID	Security metric	
			LOR	LOD
NIST				
Contingency Planning	Alternate Storage Site	CP-6	✓	
	Alternate Processing Site	CP-7	✓	
	Information System Backup	CP-9	✓	
	Information System Recovery and Reconstruction	CP-10	✓	
System and Communications Protection	Denial of Service Protection	SC-5	✓	
	Architecture and Provisioning for Name/Address resolution Service	SC-22	✓	
	Session Authenticity	SC-23		✓
	Distributed Processing and Storage	SC-36	✓	
System and Services Acquisition	Allocation of Resources	SA-2	✓	
System and Information Integrity	Predictable Failure Prevention	SI-13	✓	
CCM				
Business Continuity Management & Operational Resilience	Business Continuity Planning	BCR-01	✓	✓

Table 21. Mapping of WebPool metrics to NIST and CCM security controls

4.2.1.3. Remediation

As discussed in Section 3.5, each measurement defines one monitoring event. Table 22 lists all possible monitoring events related to WebPool metrics that can be detected by the Monitoring module.

ID	Condition	Affected metrics	Event type
WP-E1	number_of_servers < LOR_value	LOR	violation
WP-E2	diversity_level < LOD_value	LOD	

Table 22. Monitoring events related to WebPool metrics

Table 23 presents actions needed to remediate WebPool alerts and violations.

ID	Description
WP-A1	Check if the number of responsive web servers is ≥ LOR_value.
WP-A2	Restart unresponsive web server and check if number of responsive web servers is ≥ LOR_value.
WP-A3	Replace unresponsive web server and check if number of responsive web servers is ≥ LOR_value.
WP-A4	Check if the number of responsive web server types is ≥ LOD_value.
WP-A5	Restart unresponsive web server and check if the number of responsive web server types is ≥ LOD_value.
WP-A6	Replace unresponsive web server and check if the number of responsive web server types is ≥ LOD_value.
WP-A7	Check if the number of responsive web servers is ≥ LOR_value.

Table 23. WebPool remediation actions

Figure 17 presents remediation plan for managing alerts and violations of WebPool metrics. For details on the structure of a remediation plan see Section 3.5.

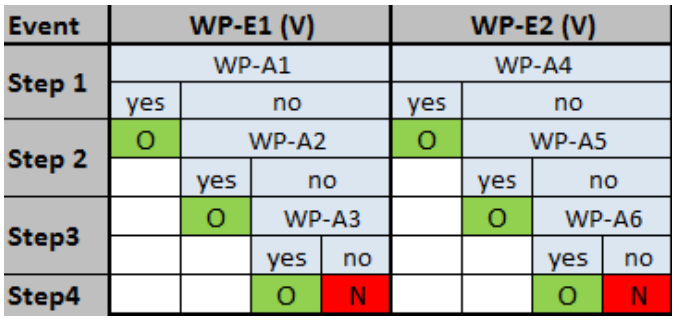

Figure 17. Remediation plans for monitoring events WP-E1 and WP-E2

All mechanism's implementation, configuration, and remediation details are available on project's Bitbucket site [19]. The source code for the mechanism is also available on a dedicated Bitbucket repository [32].

4.2.1.4. Development

With respect to the initial design of the mechanism (presented in D4.3.1), the main changes occurred with the refinements of remediation activities, and integration and performance related improvements. If during the integration the need to apply some changes arises, they will be applied by the end of the project and reported in D4.3.3.

As already mentioned earlier, the design of the mechanism integrates existing open-source tools (HAProxy, Apache, Nginx), but they were adopted to fit the project's and users' needs.

The following sections provide brief description of repository, and installation and usage guides.

4.2.2. Repository

The component is actually made up of two sub-components, the Web Container Pool Manager and the Pool Agent: they are available on Bitbucket, in the same package, at [32].

4.2.3. Description and design

This mechanism enables the configuration and acquisition of a secure web server, through the set-up of a pool of web container instances configured to ensure redundancy and diversity.

As already discussed in D4.2.2, the following components were proposed:

- **Web Container Pool Manager:** cooperates with the Broker component in order to acquire and configure different web containers belonging to a pool.
- **Pool Agent:** acts as a balancer/proxy towards the web containers belonging to a pool.

It also enables the interaction with the Monitoring module and the Enforcement RDS component in order to provide incident/vulnerabilities management capabilities.

This security mechanism adopts open source solutions to use the Broker and provide redundancy and diversity security requirements. In particular, this mechanism reuses and properly configures HAProxy (to implement forwarding capabilities) and a Memcached service (to offer a distributed memory object caching system).

4.2.4. Installation

Before explaining how to configure the whole security mechanism, it's important to note that once Apache and Nginx components have been installed on their virtual machines, the Ha-Proxy has to be configured with the IP addresses of the machines that host the web containers: this configuration in important since the HAProxy component has to forward all the incoming http requests to one of the machine hosting the web container that have to process the request itself.

Please note also that the target machines need the tcp ports 80 and 11211 to be opened.

In order to make the security mechanism easy to install, three tar.gz files have been developed (one for Apache, one for Nginx, and one for HAProxy). Each of these files contains all the necessary dependencies, so no other software needs to be installed or configured except the mechanisms itself.

It's possible to install the whole mechanism using the command line, or using the chef recipes available at the same link provided before.

As said before, the first components that have to be installed are those related to web containers, such as Nginx and Apache (with Memcached).

4.2.4.1. How to install Apache with Memcached

In order to install Apache, you need to extract the `apache2-php5-memcached.tar.gz` file (note that this file is located into the `file` folder at the same link provided before) in `/opt` folder.

```
tar -zxvf /tmp/apache2-php5-memcached.tar.gz
```

Copy the file `php.ini.erb` (note that this file is located into the `template` folder at the same link provided before) in the folder `/opt/php5/etc` and rename it as `php.ini`, edit it, setting the value of `session.save_path` property to `tcp://<ip_address>:11211`.

Export the environment variable `LD_LIBRARY_PATH` to `/opt/dependencies/memcache/libevent/lib64/`.

Enable memcached with the following command:

```
/opt/memcached/bin/memcached -u root -l 0.0.0.0 -p 11211 -M -m 64 -d
```

Now it's possible to make apache running, using the command:
`/opt/apache2/bin/apachectl start`

4.2.4.1. How to install Nginx

In order to install Nginx, you need to extract the `nginx_php.tar.gz` file (note that this file is located into the `file` folder at the same link provided before) in `/opt` folder.
`tar -zxvf /tmp/nginx_php.tar.gz`

Copy the file `php.ini.erb` (note that this file is located into the `template` folder at the same link provided before) in the folder `/opt/php/lib64` and rename it as `php.ini`.

Edit this file setting the value of `session.save_path` property to `tcp://<ip_address>:11211`.

Extract the `memcached.tar.gz` file (note that this file is located into the `file` folder at the same link provided before) in `/opt` folder.
`tar -zxvf /tmp/memcached.tar.gz`

Copy the file `hosts.erb` (note that this file is located into the template folder at the same link provided before) into folder `/etc`, rename it as `hosts` and edit it adding the following as last line of the file:
`<ip_address_of_the_current_machine> <node_identifier>`

Copy the file `index.php.nginx` (note that this file is located into the file folder at the same link provided before) into folder `/opt/nginx/html/`.

Export the environment variable `LD_LIBRARY_PATH` to `/opt/memcached_libs/libevent/lib64/`.

Enable memcached with the following command:
`/opt/memcached/bin/memcached -u root -l 0.0.0.0 -p 11211 -M -m 64 -d`

Create user, group with the following commands:

```
groupadd nginx
useradd nginx
usermod -a -G nginx nginx

groupadd www-data
useradd www-data
usermod -a -G www-data www-data
touch /opt/nginx_installed
```

Now it's possible to make nginx running, using the command:
`/opt/nginx/sbin/nginx`

Run the following command to start the PHP-FPM module:
`/opt/php/init.d.php-fpm start`

4.2.4.1. How to install HaProxy

In order to install HAProxy, you need to extract the `haproxy-hatop.tar.gz` file (note that this file is located into the `file` folder at the same link provided before) in `/opt` folder.

```
tar -zxvf /tmp/haproxy-hatop.tar.gz
```

Copy the file `haproxy_config.erb` (note that this file is located into the template folder at the same link provided before) in the folder `/opt/haproxy/configs/` and rename it as `haproxy.cfg`.

Edit this file, adding the at the end of it, one line like the following for each machine hosting a webcontainer:

```
<ip_address_of_machine_hosting_web_container> <node_identifier> :80 check
```

Copy the file `hosts.erb` (note that this file is located into the template folder at the same link provided before) into folder `/etc`, rename it as `hosts` and edit it adding the following as last line of the file:

```
<ip_address_of_the_current_machine> <node_identifier>
```

Execute HaProxy service running the following command:

```
/opt/haproxy-1.5.9/haproxy -f /opt/haproxy/configs/haproxy.cfg
```

Now it is necessary to download and to install Java 7; to download it, run the following command:

```
cd /opt
wget http://www.java.net/download/jdk7u80/archive/b05/binaries/jdk-7u80-ea-bin-b05-linux-x64-20_jan_2015.tar.gz
```

Extract it running th following command:

```
tar xvzf /tmp/java.tar.gz
```

Export the environment variable `JAVA` running the following commands:

```
export JAVA_HOME=/opt/jdk1.7.0_80/
export PATH=$PATH:$JAVA_HOME/bin/
```

Copy the file `webpool-adapter.jar` (note that this file is located into the `file` folder at the same link provided before) into folder `/opt`.

Run as root user:

```
java -jar webpool-adapter.jar <EventHubIP> <EventHubPort> <redundancy> <diversity> <slaID> <node_id> > webpool-adapter.log &
```

Let's explain the arguments:

- `<EventHubIP>` is the iIP address of the machine that hosts the Event Hub component;
- `<EventHubPort>` is the port number on which machine that hosts the Event Hub component is listening to;
- `<redundancy>` is the level of redundancy that has been measured;
- `<diversity>` is the level of diversity that has been measured;
- `<slaID>` is the identifier of the SLA that has been implemented;
- `<node_id>` is an unique identifier that represents each node;

Till now, it has been defined how to install each component manually, but it is possible also to install them using Chef recipes.
The first step is to build a Json file that represents the implementation plan (for an example of an implementation plan see Appendix 2). Once this file has been prepared, in order to install the recipes on each node, you need first to have a Chef Server installed and configured

properly, than you need Chef Workstation from which it is possible to execute the bootstrap of each target node. Please note that you need a number of available machines defined by the metric Level of Redundancy, plus one that is the machine hosting HAProxy component.

In order to install Apache component, you need to run the following command:

```
knife bootstrap <public_ip_address_of_the_node_that_will_hosts_apache> -x
<chef_user_name> -P <chef_user_password> --node-name <node_name> --run-
list 'recipe['WebPool:apache']' -j '{
"implementation_plan_id":"<identifier_of_implementantio_plan>"}'
```

In order to install Nginx component, you need to run the following command:

```
knife bootstrap <public_ip_address_of_the_node_that_will_hosts_nginx> -x
<chef_user_name> -P <chef_user_password> --node-name <node_name> --run-
list 'recipe['WebPool:nginx']' -j '{ '{
"implementation_plan_id":"<identifier_of_implementantio_plan>"}'}'
```

In order to install HAProxy component, you need to run the following command:

```
knife bootstrap <public_ip_address_of_the_node_that_will_hosts_haproxy >
-x <chef _user_name> -P <chef_user_password> --node-name <node_name> --
run-list 'recipe['WebPool:haproxy']' -j '{ '{ "implementation_plan_id":"<
identifier_of_implementantio_plan>"}'}'
```

Please note that the last recipe you have to execute is the HAProxy one.

4.2.5. Usage

In order to use this security mechanism, all we need to do is to open a browser and type the IP address of the machine on which we have installed HAProxy component: we will see the first page defined in Nginx web container and Apache web container, depending on the policy used by HAProxy to forward the requests to the web containers defined in its property file.

4.3. DBB and E2EE mechanisms

In this section, we present a description and implementation details for the mechanisms involved in the following validation scenarios defined in T5.1 (see D5.1.2):

- *SST-02 Secure_Storage_brokering_with_Client_Crypto*
- *SST-03 Secure_Storage_with_Defined_CSP*
- *SST-04 Secure_Web_Container_Client_Encryption_Replication_Alert*
- *SST-05 Secure_Web_Container_Client_Encryption_Replication_Violation*

4.3.1. Overview

Under the umbrella of Secure Storage service, SPECS offers two security capabilities enhancing the security of cloud storage solutions, namely *Database and Backup as-a-Service* and *End-2-End Encryption*, implemented with DBB and E2EE security mechanisms, respectively.

Both mechanisms could be completely separated, but since in the current prototype the E2EE is just an upgrade of the DBB mechanism, they are discussed in pair.

When storing data with CSPs, EUs usually have to accept the risk of security incidents and failures related to modifications and loss of stored data. More than that, EUs can never be sure that

- confidentiality (C) and integrity (I),
- *write-serializability* (WS), i.e., consistency among updates, and
- *read-freshness* (RF), i.e., requested data always being fresh as of the last update,

are always respected. And what is more important, even if EUs are aware of data modifications or loss of data, they cannot prove to third parties when the cloud is to blame for WS or RF violations. On the other hand, the cloud provider itself cannot disprove false accusations.

In order to offer to EUs secure storage solution with end-2-end encryption, and allow them to not only detect but also prove violations related to modification and loss of stored data, Secure Storage service in SPECS is offered with the DBB and E2EE security mechanisms which provide the following functionalities:

- E2EE:
  - Client-side encryption enforcing *confidentiality* and *integrity*.
- DBB:
  - Detection and proof of violations related to *write-serializability* (WS) and *read-freshness* (RF).
  - Backup of stored data.

Note that in order to acquire Secure Storage service through SPECS, the DBB mechanism is mandatory whereas E2EE is just optional.

With the backup service we ensure that in case of detected WS/RF violations, the database can either be restored from the backup or the REST API can be moved from the primary storage site to the backup. In this case the backup can replace the role of the primary target service. In this way we ensure that any corrupted or missing data (caused either by WS/RF violations or failures on the main database) can be to some extent retrieved or replaced. And by "to some extent" we mean that the database can be restored to the state of the last completed backup. Any data lost or corrupted between two backups cannot be recovered.

As described in [9], WS and RF are monitored and proved with so called *attestations*, which are signed messages that accompany each EU's request and each CSP's response. They bind the clients to the requests they make and the cloud to a certain state of the data. With every request (through the `get` or `put` interface), clients and cloud exchange attestations, i.e., every `get`/`put` request is associated with an attestation.

4.3.1.1. Architecture

DBB and E2EE mechanisms are implemented with the following components (the architecture is depicted in Figure 18):

- **DBB Client and E2EE Client plug-in** components operate directly on the EU's machine independently from the SPECS Platform (the EU downloads the tool from the web store once the SLA is signed). DBB Client provides a web interface for uploading and downloading files. If the E2EE functionality is requested, the E2EE Client plug-in provides client side encryption/decryption of the files being sent/received to/from the CSP through the DBB client component. An EU can use more than one Client component to access the data.

- **DBB Main Server** and **DBB Backup Server** are the main components (application server) deployed on primary storage site and backup, respectively. On the primary storage site, DBB Server oversees all configurations, handles all put/get requests, and orchestrates all associated operations (i.e., writes/reads the data, performs backups, sends EU's attestations to the Client). The DBB Server on the backup site is responsible for backups and restorations.
- **DBB Main DB** and **DBB Backup DB** are the database servers deployed on primary storage site and backup, respectively.
- **DBB Auditor** performs auditing, i.e., checks if sequences of put/get attestations form correct write/read chains, and checks if WS/RF violations were detected in time.
- **DBB Monitoring Adapter** monitors databases on both storage sites (i.e., checks if backups and restorations of backup are performed successfully, monitors availability of both servers, availability of Auditor, and availability of both DBs) and monitors certification status of the DBB Client version in web store.
- **E2EE Monitoring Adapter** monitors certification status of the Client code available at the Web store.

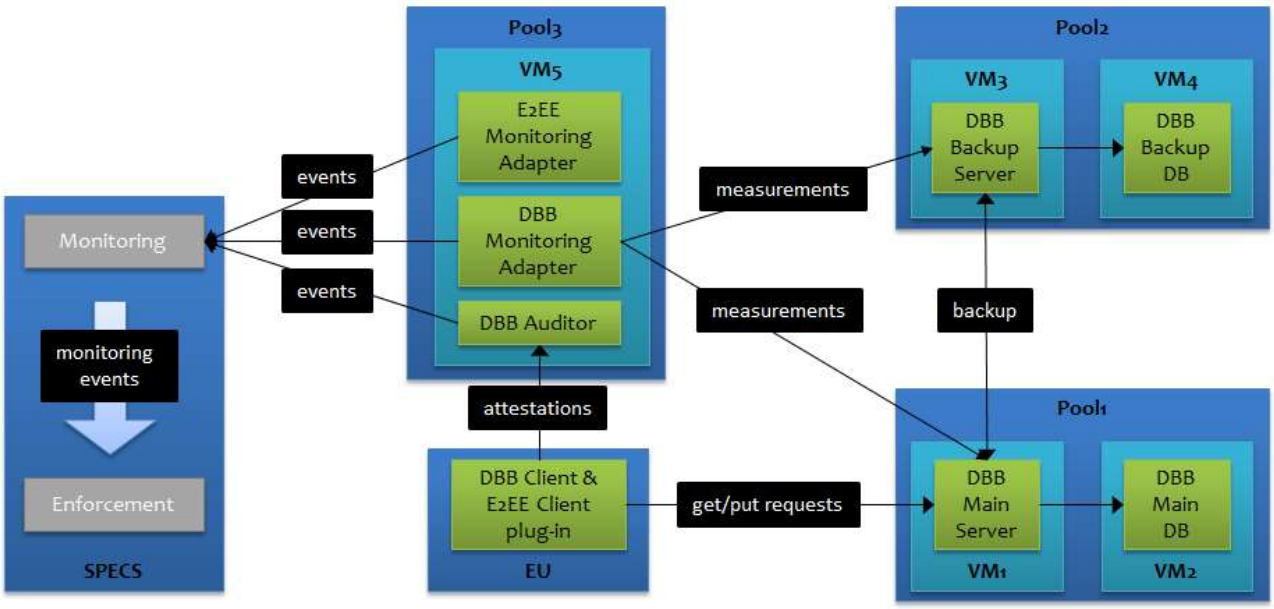

Figure 18. Architecture of the DBB and E2EE mechanisms

In the SLA implementation phase (discussed in Sections 3.2 and 3.3), Enforcement module deploys and configures all DBB and E2EE components on five different virtual machines (VMs) acquired with three different CSPs or at least in three different pools of VMs acquired with the same CSP, and provides the EU with the link to the DBB/E2EE Client component.

In order to separate main and backup DB, and to separate Auditor from both servers, we can use the so called pooling method, i.e., we make three different calls to a CSP with request for different VMs. This way we assure that all acquired VM pools are independent resources:

- $Pool_1$ contains two VMs, one for Main Server, one for Main DB.
- $Pool_2$ contains two VMs, one for Backup Server, one for Backup DB.
- $Pool_3$ contains one VM for Auditor and both monitoring adapters.

First VM (VM_1 in $Pool_1$) hosts the Main Server. The second VM (VM_2 in $Pool_1$) is used as the primary storage site and it hosts the Main DB. With every `put/get` request from an EU (actually from the Client), the Main Server sends to the EU a Cloud's attestation. The Client

forwards that attestation to the Auditor. The Auditor collects attestations and after each epoch checks if the CSP maintains the correct data. Attestations not only show if the CSP maintains the right data in the right way but also prevent the EU of falsely accusing the CSP of any corruption or loss of the EU's data.
Note, in order to guarantee integrity of the auditing process and ensure the highest possible level of security, the Auditor is physically separated from both storage sites (the Auditor is hosted on VM_5 in $Pool_3$).

If any WS/RF violations are detected by the Auditor, SPECS can automatically restore the database from the backup, set up a new database, or move the Main Server.

Note, backups are not performed in real-time, thus some data saved in the time between two backups cannot be restored in case of WS/RF violations or system failures. What SPECS can guarantee is that the main database can be restored to the state of the last backup.

The third and the fourth VMs (VM_3 and VM_4 in $Pool_2$) are used for data backup. One hosts the server which orchestrates the backup and the restoration process (Backup Server), and the other one hosts the backup database (Backup DB).

The last VM (VM_5 in $Pool_3$) hosts (besides the Auditor) a monitoring adapter (DBB Monitoring Adapter) which not also monitors responsiveness of both servers and both DBs, but can also check if backups and restorations are successful, and monitors the integrity of the backed up data (using, e.g., Proofs of data storage approach [10], [11], [12], [13], [14]). In case of failures or attacks, the occurrence is notified to the Monitoring module. The same VM also hosts the E2EE Monitoring Adapter which observes the certification status of the E2EE Client (i.e., checks if the web store maintains the latest version of the Client code which is certified).

Configurations of DBB and E2EE components (all except DBB/E2EE Clients) depend on EU's choice of security controls and security metrics.

During the SLA monitoring phase, both monitoring adapters and Auditor continuously send monitoring data to the Monitoring module which determines whether any of sent events indicate a possible alert/violation and should therefore be further analysed by the Enforcement module. Whenever the Enforcement module is notified about a possible DBB/E2EE alert/violation, the notified event has to be classified, analysed, and remediated (see Sections 3.4 and 3.5). All WS/RF violations are also notified to EUs.

Attestations are the core objects for the auditing process. As shown in Figure 19, each time the Client performs a `get` (the request contains data block ID and Client's `get attestation`), the Main Server returns the requested data and attaches the Cloud's `cloud get attestation`. Cloud's attestation is automatically forwarded to the Auditor by the Client.

As depicted in Figure 20, each time the Client performs a `put` (the request contains the data and Client's `put attestation`), the Main Server stores the data, returns the block ID, and attaches the Cloud's `cloud put attestation`. The Client automatically forwards a copy of the attestation to the Auditor.

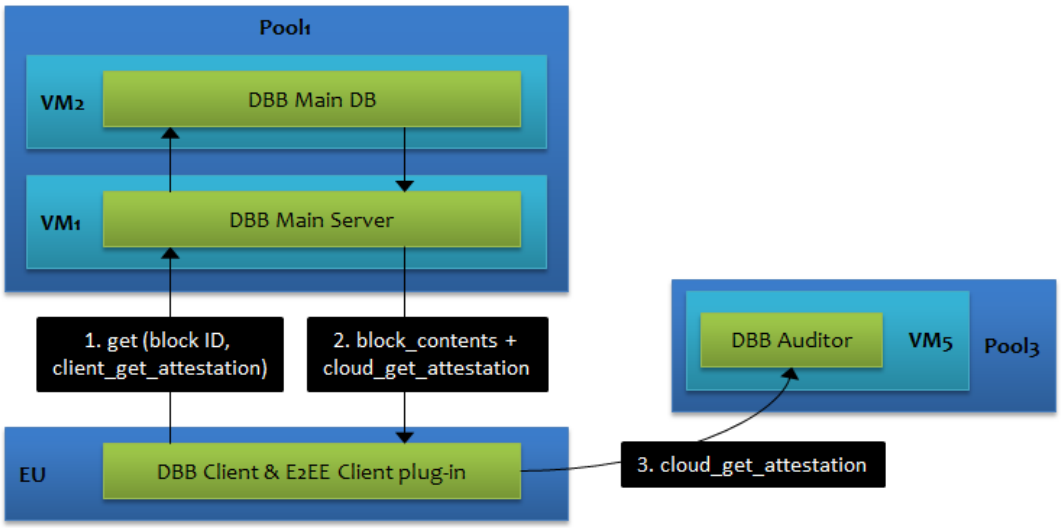

Figure 19. Client's `get` request actions

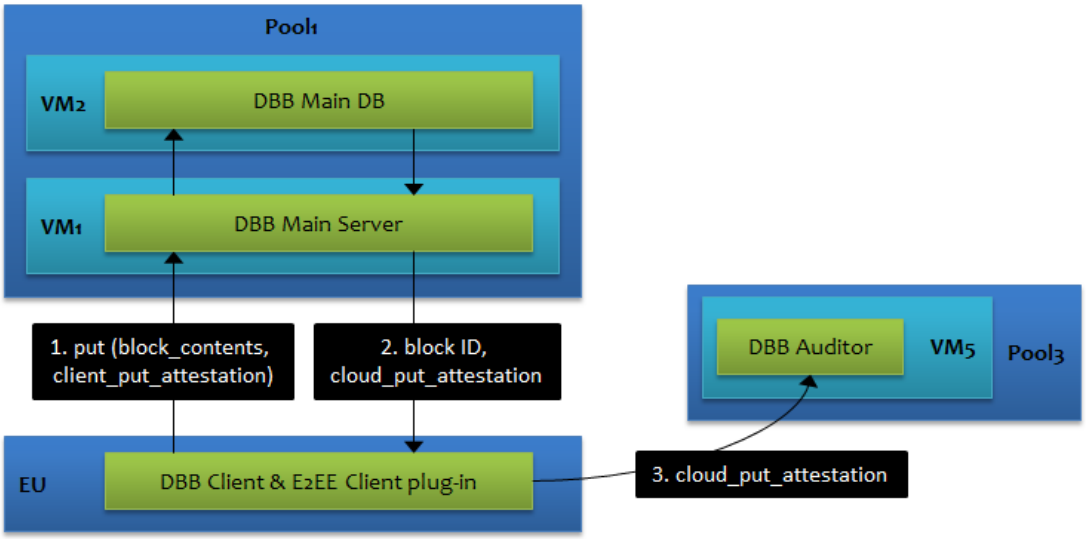

Figure 20. Client's `put` request actions

Each attestation is composed of different elements which are hashed and signed. For details on the structure and the contents of attestations, and on the way they are used to check for and prove I, WS and RF violations, see [9]. Note that the structure of the attestation also depends on the EU's security requirements (the structure of the attestation differs for WS and RF).

Auditor collects all attestations for an *epoch* (i.e., predefined fixed time period) and audits them only after each epoch finishes. Note that only EU sends CSP's attestations to the Auditor which checks them to verify WS and RF. The CSP stores EU's attestations for potential cases when the EU would trigger false accusations.

Main Server and Client save copies of all attestations for the current epoch. In case of failures on the Auditor's target service, SPECS can deploy a new resource with a new Auditor and provides it with all attestations for the current epoch. In this way, we continuously monitor Auditor's database and ensure that any WS/RF violations will be detected during auditing.

Attestations are (at this moment) not audited in real-time. Instead, time is divided into periods called *epochs*. During each epoch, attestations are collected, and after each epoch, SPECS performs auditing and checks chains of attestations to detect possible cloud's misbehaviour.

4.3.1.2. Detectable attacks and system failures

In the current prototype of the DBB mechanism, we are able to detect the following attacks and failures either related to I, WS, and RF, or to infrastructure outages.

Put ignore failure/attack. Write operations to the Main DB (hosted on $VM_2/Pool_1$) did not succeed or they were cancelled by a delete operation. In this case we have to acquire new VM inside the same pool ($VM_6/Pool_1$), set up a new Main DB on VM_6 by restoring the data from the backup, and redirect the Main Server to the new main database. Infrastructure before and after a successful remediation of a *put ignore attack/failure* is depicted in Figure 21 below.

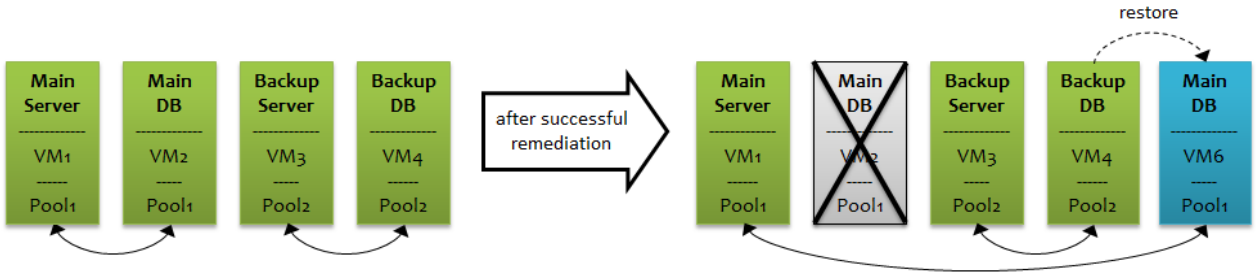

Figure 21. The infrastructure after a successful remediation of a *put ignore attack/failure*

Fork attack: the API (i.e., the Main Server) and the main database (Main DB) might have been modified and they cause inconsistent views to different clients (e.g., maintaining two copies of some data and placing some writes to one copy of the data and other writes on the other copy of the same data). In this case, we have to acquire a new pool with two VMs. We acquire one VM ($VM_6/Pool_3$) to set up a new main server. Because the main database (Main DB) might have been modified, we acquire one VM ($VM_7/Pool_3$), set up another main DB for a new main storage site, and perform a data restoration (of backup DB to the new DB). Note that moved API (the new server) is directed to the new DB (Main DB hosted on $VM_7/Pool_3$). Infrastructure before and after a successful remediation of a *fork attack* is depicted in Figure 22 below.

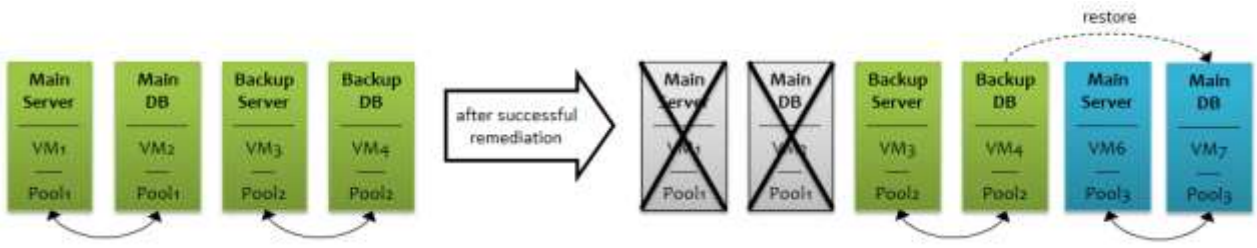

Figure 22. The infrastructure after a successful remediation of a *fork attack*

Stale file attack: an attacker caused that stale files have been returned to EUs. In this case we need to add a new VM (VM_6) to the $Pool_1$, set up a new Main Server there and direct it to the primary database (Main DB). Any missing or corrupted data in Main DB is restored from the backup. The state of the infrastructure before the attack and after a successful remediation of it is presented in Figure 23 below.

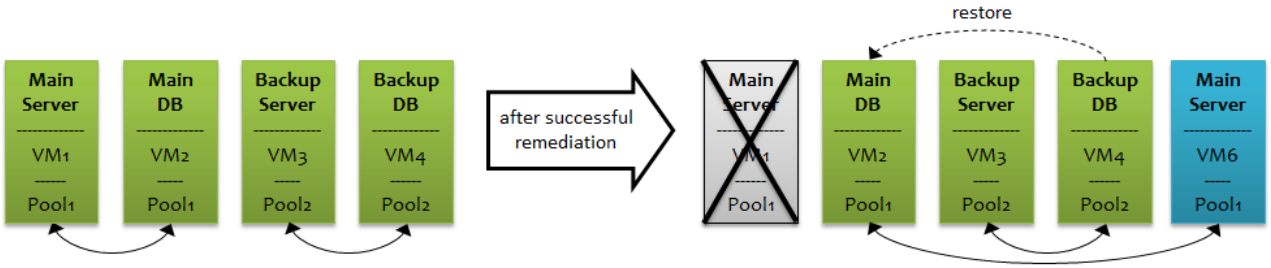

Figure 23. The infrastructure after a successful remediation of a *stale file attack*

Primary server failure: due to internal issues, the primary server is unresponsive. In this case we try to restart the server. If that fails, we acquire a new VM in the same pool ($VM_6/Pool_1$) and set up a new main server. Note, after a successful remediation, the new main server is directed to the Main DB (hosted on VM_2/CSP_1). Infrastructure before and after a successful remediation of a *primary server failure* is depicted in Figure 24 below.

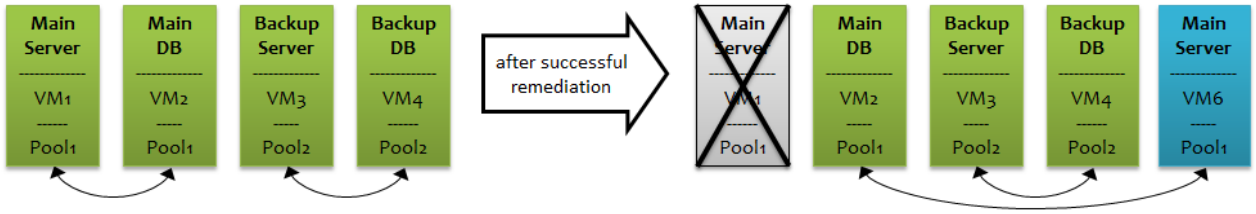

Figure 24. The infrastructure after a successful remediation of a *primary server failure*

Primary database server failure: due to internal issues, the main database server is unresponsive. In this case we try to restart the server. If that fails, we acquire new VM (VM_6 in $Pool_1$), set up a new Main DB and perform a restoration (of backup to a new main database). Infrastructure before and after a successful remediation of a *primary database server failure* is depicted in Figure 25 below.

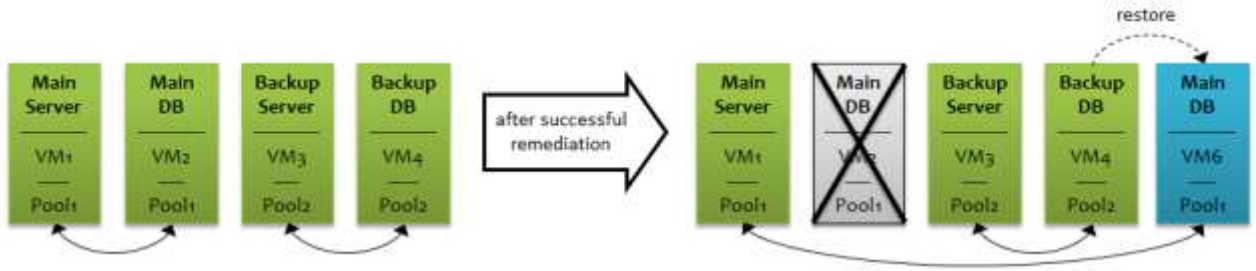

Figure 25. The infrastructure after a successful remediation of a *primary database server failure*

Backup server failure: due to internal issues, the backup server is unresponsive. In this case we try to restart the server. If that fails, we acquire new VM (VM_6 in $Pool_2$) and set up a new Backup Server. We immediately invoke a new backup process. Infrastructure before and after a successful remediation of a *backup server failure* is depicted in Figure 26 below.

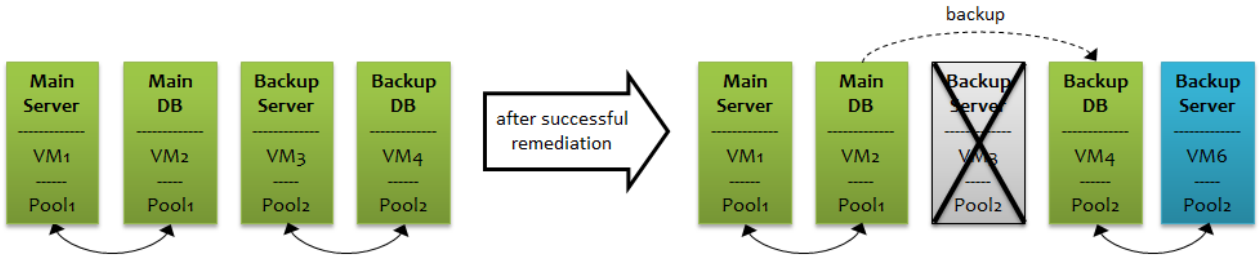

Figure 26. The infrastructure after a successful remediation of a *backup server failure*

Backup database server failure: due to internal issues, the backup database server is unresponsive. In this case we try to restart the server. If that fails, we acquire new VM (VM_6 in $Pool_2$), set up a new Backup DB and perform a backup (of original backup to a new backup). Infrastructure before and after a successful remediation of a *backup database server failure* is depicted in Figure 27 below.

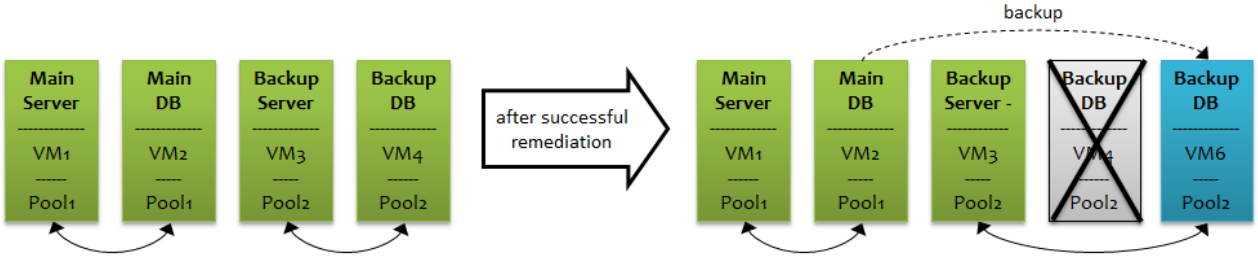

Figure 27. The infrastructure after a successful remediation of a *backup database server failure*

Auditor failure: due to internal issues, the Auditor (hosted on $VM_5/Pool_3$) may become unresponsive. In this case we try to restart the Auditor. If that fails, we acquire new VM in the same pool, deploy new Auditor, and copy all Client's and all Server's attestation for the current epoch from the Server and Client, respectively, and deploy a new DBB Monitoring Adapter. Note, when the Client places a request to the main Server (either a `put` or a `get` request), the server returns the new URL of the Auditor. Infrastructure before and after a successful remediation of a *backup database server failure* is depicted in Figure 28 below.

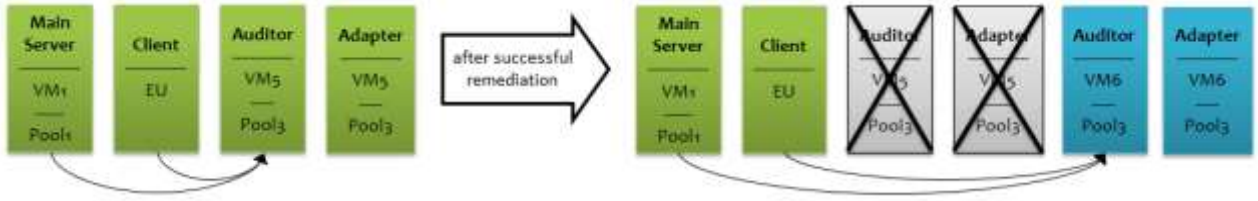

Figure 28. The infrastructure after a successful remediation of an *auditor failure*

4.3.1.3. Security metrics and controls

Security metrics associated to the DBB and E2EE mechanisms are defined in the following three tables. For each metric we provide a description, possible values with units, default values, and actions that need to be taken in order to enforce the metric. Note that setting a metric to its default value ensures the maximum possible level of security associated to that metric.

Name		Value	Default value	Unit
Write-serializability (**WS**)		yes	yes	n/a
Description	This metric ensures the EU that any WS violations to the stored data will be detected in a defined period of time (detection periods are less than 2**epoch*). In case of WS violations, the EU will be notified, and the system will be restored to the state of the last finished *epoch*.			
Actions taken to enforce the metric	1. With every `put` request from the Client the data is stored in the Server's DB. 2. The Server sends `cloud put attestation` to the Client which automatically forwards it to the Auditor. 3. After each *epoch*, the Auditor checks attestation chains. 4. If WS violation is detected, the EU is notified and the Server's DB is restored to the state of the last finished *epoch*.			

Table 24. DBB security metric WS

Name		Value	Default value	Unit
Read-freshness (**RF**)		yes	yes	n/a
Description	This metric ensures the EU that any RF violations to the stored data will be detected in a defined period of time (detection periods are less than 2**epoch*). In case of RF violations, the EU will be notified, and the system will be restored to the state of the last finished *epoch*.			
Actions taken to enforce the metric	1. With every `get` request from the Client the data is retrieved from the Server's DB and sent to the Client. 2. The Server sends `cloud get attestation` to the Client which automatically forwards it to the Auditor. 3. After each *epoch*, the Auditor checks attestation chains. 4. If RF violation is detected, the EU is notified and the Server's DB is restored to the state of the last finished *epoch*.			

Table 25. DBB security metric RF

Name		Value	Default value	Unit
Client-side encryption certification (**EC**)		yes	yes	n/a
Description	This metric ensures that the E2EE Client component available at the provided address is certified and thus grants the security of the encryption.			
Actions taken to enforce the metric	Before providing the EU with the link to the E2EE Client component, check if the version of the component is certified (i.e., check if the web store maintains the latest version of the Client). Then periodically: 1. Check certification status of the Client component.			

Table 26. E2EE security metric EC

Note that we cannot monitor components installed on the EU's private infrastructure (i.e., we cannot monitor the Client component) after they have been downloaded and installed. But in order to guarantee client-side encryption, we check (prior to installation, and continuously during the SLA monitoring phase) the version of the E2EE Client component available at the web store. For this purpose we introduced *Client-side Encryption certification* metric with which we can guarantee that the E2EE Client component available at the web store (if downloaded and installed as advised, and configurations are not changed) is certified (and thus ensures a defined level of cryptographic protection).

As described in Section 3, we associate each metric with a basic measurement and one or more additional measurements. The following tables present all measurements together with MoniPoli rules associated to DBB and E2EE metrics.

Metric	Write-serializability (WS)
SLO	write_serializability = yes

Measurements	MoniPoli rules
ws_time_since_detection	ws_time_since_detection < 2*epoch
ws_put_ignore	ws_put_ignore = no
ws_fork	ws_fork = no
primary_server_availability	primary_server_availability = yes
primary_db_server_availability	primary_db_server_availability = yes

backup_server_availability	backup_server_availability = yes
backup_db_server_availability	backup_db_server_availability = yes
auditor_availability	auditor_availability = yes

Table 27. Measurements and MoniPoli rules associated to DBB metric WS

Metric	Read-freshness (RF)
SLO	read-freshness = yes
Measurements	MoniPoli rules
rf_time_since_detection	rf_time_since_detection < 2*epoch
rf_stale	rf_stale = no
primary_server_availability	primary_server_availability = yes
primary_db_server_availability	primary_db_server_availability = yes
backup_server_availability	backup_server_availability = yes
backup_db_server_availability	backup_db_server_availability = yes
auditor_availability	auditor_availability = yes

Table 28. Measurements and MoniPoli rules associated to DBB metric RF

Metric	Client-side encryption certification (EC)
SLO	encryption_certification = yes
Measurements	MoniPoli rules
code_certification	code_certification = yes

Table 29. Measurements and MoniPoli rules associated to DBB metric EC

Measurements *ws_time_since_detection* and *rf_time_since_detection* are the basic measurements for the associated metrics and report the delay from the moment the violation occurred to the moment when it was detected.

As discussed above, the Auditor monitors occurrences of various attacks associated to write-serializability and read-freshness. For this purpose measurements *ws_put_ignore*, *ws_fork*, and *rf_stale* were introduced.

In order to enable secure storage service and thus assure validity of SLOs related to WS and RF, all servers and the Auditor have to be up and running. DBB Monitoring Adapter observes their availability and report results through all **_availability* measurements.
Encryption certification metric is validated with basic measurement *code_certification*.

The defined DBB and E2EE metrics implement NIST and CCM security controls presented in the following table.

Control Family/Group	Control Name	Control ID	Security metric		
			WS	RF	EC
NIST					
Contingency Planning	Contingency Plan | Resume all Missions / Business Functions	CP-2 (4)	✓	✓	
	Contingency Plan | Alternate Processing / Storage Site	CP-2 (6)	✓	✓	
	Alternate Storage Site | Separation from Primary Site	CP-6 (1)	✓	✓	
	Information System Backup	CP-9	✓	✓	
	Information System Backup | Redundant	CP-9 (6)	✓	✓	

	Secondary System				
System and Communications Protection	Cryptographic Key Establishment and Management	SC-12			✓
	Cryptographic Protection	SC-13			✓
System and Information Integrity	Software, Firmware, and Information Integrity	SI-7	✓	✓	
	Software, Firmware, and Information Integrity | Integrity Checks	SI-7 (1)	✓	✓	
	Software, Firmware, and Information Integrity | Automated Notifications of Integrity Violations	SI-7 (2)	✓	✓	
	Software, Firmware, and Information Integrity | Automated Response to Integrity Violations	SI-7 (5)	✓	✓	
CCM					
Infrastructure & Virtualization Security	Change Detection	IVS-02	✓		
Application & Interface Security	Data Integrity	AIS-03		✓	
Business Continuity Management & Operational Resilience	Business Continuity Planning	BCR-01	✓	✓	
	Policy	BCR-11	✓	✓	
Encryption & Key Management	Entitlement	EKM-01			✓
	Sensitive Data Protection	EKM-03			✓

Table 30. Mapping of DBB and E2EE metrics to NIST and CCM security controls

4.3.1.4. Remediation

As discussed in Section 3.5, each measurement defines one monitoring event. Table 31 below lists all possible monitoring events related to E2EE and DBB metrics that can be detected by the Monitoring module.

ID	Condition	Affected metrics		Event type
E2EE-E1	code_certification = no	EC		violation
DBB-E1	ws_time_since_detection ≥ 2*epoch	WS		
DBB-E2	rf_time_since_detection ≥ 2*epoch	RF		
DBB-E3	ws_put_ignore = yes	WS		alert
DBB-E4	ws_fork = yes	WS		
DBB-E5	rf_stale = yes	RF		
DBB-E6	primary_server_availability = no	WS	RF	
DBB-E7	primary_db_server_availability = no	WS	RF	
DBB-E8	backup_server_availability = no	WS	RF	
DBB-E9	backup_db_server_availability = no	WS	RF	
DBB-E10	auditor_availability = no	WS	RF	

Table 31. Monitoring events related to E2EE and DBB metrics

The next table reports remediation actions required to mitigate E2EE and DBB alerts and recover from SLA violations related to DBB and E2EE mechanisms.

ID	Description
E2EE-A1	Upload the latest version of the E2EE Client to the web store and check its availability.
DBB-A2	Acquire a new VM in the main server pool, set up a new main DB, connect it to main server, and check if the main DB is responsive.
DBB-A3	Perform restoration (of backup DB to main DB) and check if it is complete (availability of main DB).
DBB-A4	Acquire a new VM in a new pool, set up a new main server, and check if the main server is responsive.
DBB-A5	Acquire a new VM in the main server pool, set up a new main server, connect it to main DB, and check if the main server is responsive.
DBB-A6	Restart the primary server and check if it is available.
DBB-A7	Restart the primary DB server and check if the primary DB server is available.
DBB-A8	Restart the backup server and check if it the backup server available.
DBB-A9	Acquire a new VM in the backup server pool, set up a new backup server, connect it to backup DB, and check if the backup server is responsive.
DBB-A10	Perform backup (of original DB to backup DB) and check if it is complete (backup DB availability).
DBB-A11	Restart the backup DB server and check if the backup DB server is available.
DBB-A12	Acquire a new VM in the backup server pool, set up a new backup DB, connect it to backup server, and check if the backup DB is responsive.
DBB-A13	Restart auditor and check if it is available.

Table 32. E2EE and DBB remediation actions

The next two figures report remediation plans related to alerts and violations of DBB and E2EE metrics. For details on the structure of a remediation plan see Section 3.5.

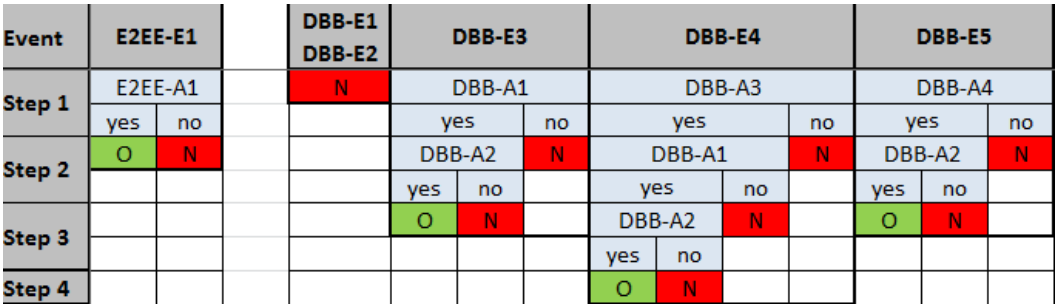

Figure 29. Remediation plans for monitoring events E2EE-E1, and DBB-E1 to DBB-E5

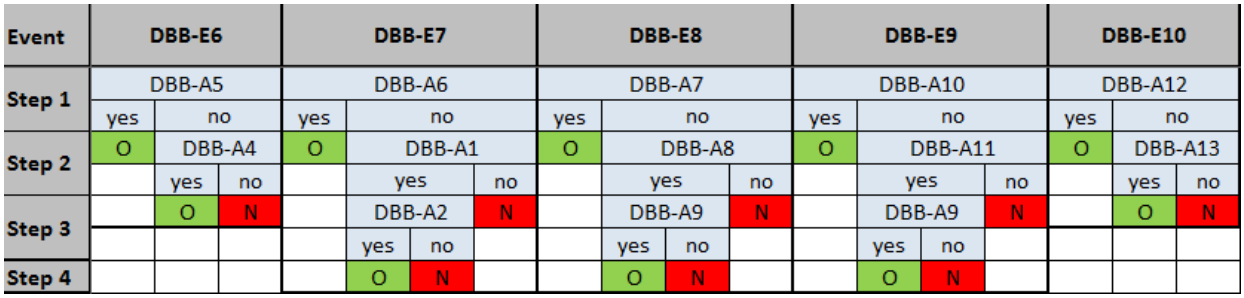

Figure 30. Remediation plans for monitoring events DBB-E6 to DBB-E10

All implementation and configuration details for both mechanisms are available on project's Bitbucket [19]. The code for all DBB and E2EE components is also available on mechanisms' Bitbucket repositories [15], [16], and [17].

Let's consider a simple example. An EU signs an SLA with a single SLO related to metric *Write-Serializability* (WS). This metric has an additional measurement related to availability of the

backup server. At one moment the DBB Monitoring Adapter detects unavailability of the backup server. Monitoring event is notified to the Diagnosis component. Diagnosis classifies event (DBB-E8), identifies affected SLOs, calculates the risk/severity level associated to the event, and forwards the alerted/violated SLA to the RDS component. According to the remediation plan in Figure 30, the first action taken is action DBB-A7. This means RDS invokes the Implementation component to restart the backup server. If the restarted server is now responsive, the alert is resolved, and the SLA's state is updated to *Observed*. If the server is still unavailable (e.g., VM hosting the backup server crashed), the RDS component executes action DBB-A8, i.e., invokes the Implementation component to acquire new VM in the backup server pool, to set up a new backup server, to connect it to the backup database, and check if the set up was successful. If the set up was successful, the RDS component invokes the Implementation component to perform a backup of the main database to the backup (action DBB-A9). If remediation of the described alert is successful, the state of the SLA is updated to *Observed*. If automatic remediation failed, the EU is notified about the occurrence (backup server is unavailable, set up of a new backup site was successful, backup process is failing) and about the SLOs this alert affects (SLO related to the metric WS). Once the alert is notified, the state of the SLA is updated to *Observed*.

4.3.1.5. Development

The development of DBB and E2EE mechanisms started in the second year of the project. Implementation activities started according to the design presented in D4.2.2, but since new requirements were identified, the design has evolved as presented above. Also, remediation aspects of the initial E2EE mechanism have been further analysed and defined.

The current prototype offers all functionalities reported in elicited requirements. Some improvements are still expected in the last year of the project, mainly focused on better performance.

Both DBB and E2EE mechanisms have been developed according user stories (defined in T5.1), requirements from EUs (elicited in T4.1), and internally developed design (performed in T4.2). DBB has been developed based on ideas from [9], and E2EE integrates Crypton [20], [21]. All other SPECS related functionalities (supporting detection of WS and RF violation, enabling associated measurements, and backing remediation actions) have been implemented from scratch.
The following sections report short overview of repositories, and provide with installation and usage guides.

4.3.2. Repository

E2EE mechanism consists of four main components: E2EE Server, E2EE Client, E2EE Auditor and E2EE Monitoring Adapter. The components can be found on project's Bitbucket repositories [15], [16], [17], and [41].

Chef recipes which SPECS Platform uses to start E2EE components are available in the specs-core-enforcement-repository [38].

DBB mechanism relies on PostgreSQL [23] and thus mostly provides scripts (recipes) which manage PostgreSQL. The recipes can be found in specs-core-enforcement-repository [39].

4.3.3. Description and design

E2EE server consists of two subprojects:

- Client: the code that gets compiled to the crypton.js which is then used in E2EE Client.
- Server: REST API which is used by crypton.js; functionality to store data into PostgreSQL.
- Test: integration tests for communication client:server.

E2EE Client does not have subprojects - it currently works as a web page using crypton.js that is compiled in E2EE Server for cryptographic operations. E2EE Client will be transformed into Chrome extension once finalized, but this require only minor modifications and the structure will not change.

E2EE Auditor consists of four packages:

- Attestations: the attestations format is defined here.
- Auditor: the core of Auditor - offering REST API which is receiving attestations; logic for detection of violations.
- Simulation: contains a simplified client and cloud storage REST API; all attestations received from cloud storage REST API are sent to the Auditor REST API.
- Tests: tests which send attestations (using simulation package) to the Auditor REST API and then analyse whether the detection of violations properly work (violations are simulated using simulation package).

E2EE Monitoring Adapter does not have any subpackages. It consist of three Python scripts:

- crypton.py: monitors the availability of E2EE servers and databases;
- auditor.py: monitors the availability of E2EE Auditor;
- component.py: parent class of cyrpton Auditor and Crypton;
- util.py: functionality for sending monitoring events to the Event Hub.

4.3.4. Installation

E2EE server is built on top of Crypton [20]. Crypton offers a REST API [21], uses Redis [22] for a session store (to handle huge amounts of requests) and PostgreSQL as the main data store. In SPECS we manage REST API + Redis and PostgreSQL as two separated components. PostgreSQL is actually provided by DBB mechanism to be able to offer database and backup functionality independently from encryption.

Prerequisites:

- Redis
- PostgreSQL
- Crypton
- Node.js

The Server can be installed on Ubuntu machine by following the steps below:

Installation of Node.js

```
curl -sL https://deb.nodesource.com/setup | bash -
sudo apt-get install nodejs
```

Installation of PostgreSQL

```
sudo apt-get install postgresql
```

Installation of Redis

```
wget http://download.redis.io/releases/redis-stable.tar.gz
tar xzf redis-stable.tar.gz
cd redis-stable
make && make install
```

Installation of DBB+E2EE Server

```
git clone https://bitbucket.org/specs-team/specs-mechanism-enforcement-e2ee-server.git
cd specs-mechanism-enforcement-e2ee-server/server
npm link
```

Run DBB+E2EE Server

```
cd specs-mechanism-enforcement-e2ee-server/server/bin
./cli.js run
```

The Client will be provided as a Chrome extension once finalized (performance issues which will be reported in T5.2 deliverable), however at the moment for the testing purposes it can be run as a web page as described below.

Install and run DBB+E2EE Client

```
git clone https://bitbucket.org/specs-team/specs-mechanism-enforcement-e2ee-client
cd specs-mechanism-enforcement-e2ee-client/web
python web.py
```

The web page should be now available at http://localhost:8080.

The DBB+E2EE Auditor can be installed and run as described below.

Install and run DBB+E2EE Auditor

```
git clone https://bitbucket.org/specs-team/specs-mechanism-monitoring-e2ee-auditor
cd specs-mechanism-monitoring-e2ee-auditor/auditor
python auditorapi.py
```

Installation DBB+E2EE Monitoring Adapter

```
hg clone https://bitbucket.org/specs-team/specs-mechanism-monitoring-e2ee-adapter
python monitor.py monitor ip port event_hub_url
```

4.3.5. Usage

Once the server and monitoring are running and the client is installed on the user's machine, we need to launch an application (E2EE Client) and connect it to the E2EE Server deployed by SPECS. The user can then register/login to the server via the form depicted below in Figure 31. Apart from credentials, the EU also has to provide URL for the Server (DBB+E2EE server URL) and Auditor (DBB+E2EE monitoring URL).

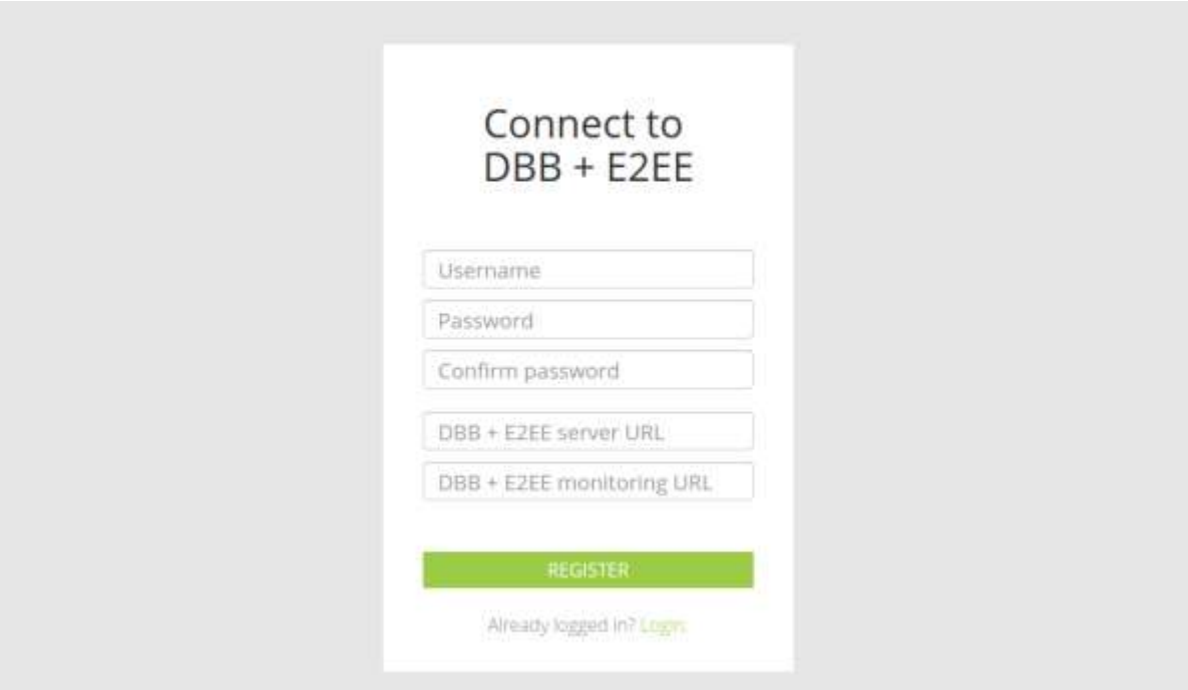

Figure 31. Login to DBB+E2EE server

Once logged in, a user can encrypt/decrypt the files and share the files with other users. See Figure 32.

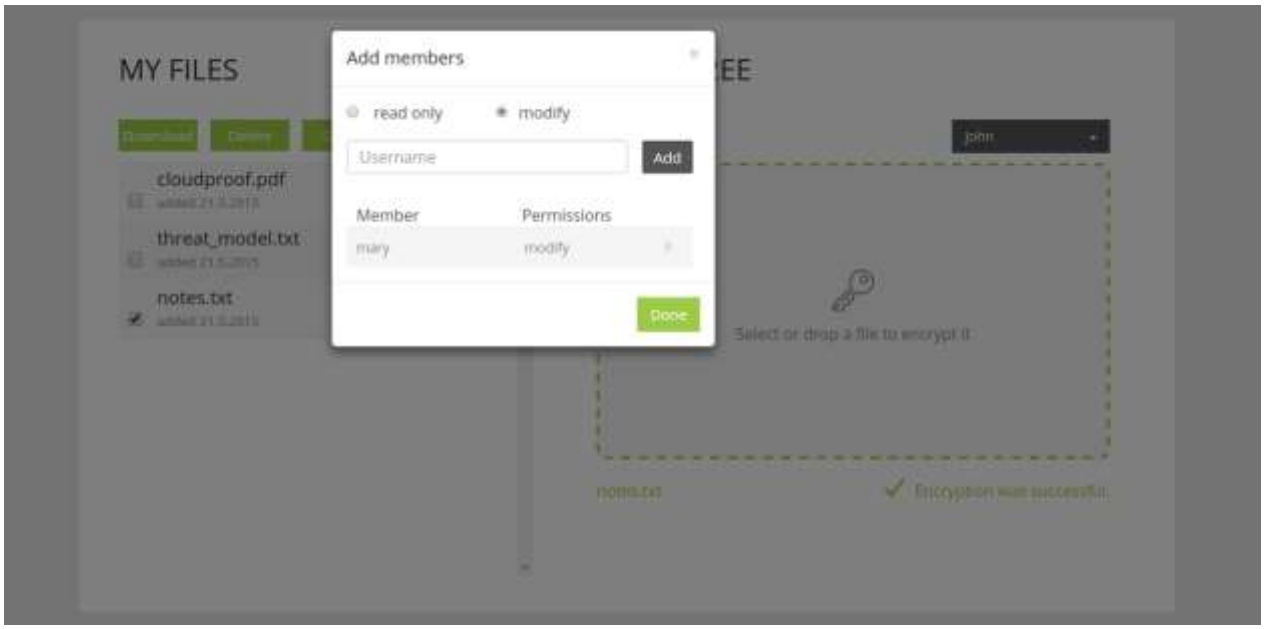

Figure 32. Encrypting/decrypting and sharing files

4.4. SVA mechanism

In this section, we present a description and implementation details for the mechanisms involved in the following validation scenarios defined in T5.1 (see D5.1.2):

- *SWC-04 Secure_Web_Container_SVA_Enhanced_Alert*
- *SWC-05 Secure_Web_Container_TLS_SVA_Enhanced_Violation*

4.4.1. Overview

Software Vulnerability Assessment mechanism (SVA) comprises a set of components that enhance the security of cloud services with the following functionalities:

- Periodically check for and report about published software vulnerabilities (using different repositories[8] to extract information about known vulnerabilities).
- Periodically scan target services (with a possibility of using different scanners), and check for known software vulnerabilities.
- Periodically check and report about available updates and upgrades of vulnerable libraries installed on target services.

4.4.1.1. Architecture

As introduced in D4.2.2 and initially developed in D4.3.1, the mechanism is implemented with three components:

- **SVA Enforcement** enforces security metrics: manages (generates and updates) vulnerability lists, orchestrates scans, checks for updates/upgrades of vulnerable libraries installed on the EU's target services, and build reports.
- **SVA Monitoring** monitors security metrics: monitors all parameters associated to each metric (e.g., age of reports, availability of repository, and responsiveness of scanners).
- **SVA Dashboard** presents vulnerability list and scanning results, and reports about available updates/upgrades of vulnerable libraries.

Figure 33 presents the SVA mechanism's architecture through an example where an EU requested SVA mechanism with Secure Web Container service (three VMs were acquired; two for web servers and one for balancer). The Dashboard component is deployed on the VM where the WebPool's balancer resides. The combination of SVA Enforcement, SVA Monitoring, and scanners is deployed on each of the VMs hosting web servers.

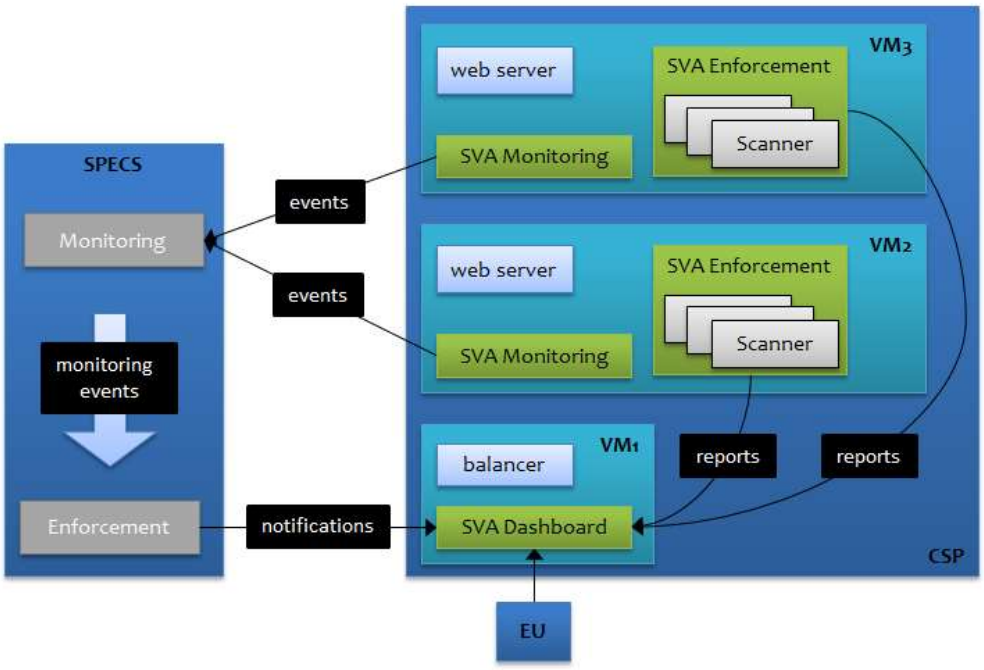

Figure 33. Architecture of the SVA mechanism in case of Secure Web Server service

[8] For example, http://ftp.suse.com/pub/projects/security/oval/ or https://support.novell.com/security/oval/.

In case of Secure Storage service, the SVA Dashboard is deployed on VM_5, and a combination of SVA Enforcement, SVA Monitoring, and scanners is deployed on each of the machines VM_1-VM_4 (see Figure 18 in Section 4.3). More details will be provided in D4.3.3

In the SLA implementation phase (see Sections 3.2 and 3.3), the Enforcement module deploys and configures SVA mechanism on EU's target service. Configurations depend on EU's choice of security controls and security metrics. During the SLA monitoring phase[9], SVA Monitoring components continuously perform measurements to evaluate the state of EU's SLOs. Measurement results (events) are sent to the Monitoring module which determines whether an event indicates possible alert/violation and should therefore be further analysed by the Enforcement module. Whenever an event reaches the Enforcement module and it presents and actual SLO alert or a violation, the root cause of the event has to be determined and an appropriate remediation action has to be applied (see Sections 3.4 and 3.5).

4.4.1.2. Security metrics and controls

Security metrics associated to the SVA mechanism are defined in the following five tables. For each metric we provide a description, possible values with units, default values, and actions that need to be taken in order to enforce the metric. These actions are periodically performed by the SVA Enforcement component deployed on EU's target services. Note that setting a metric to its default value ensures the maximum possible level of security associated to that metric.

Name		Value	Default value	Unit
List update frequency (**LUF**)		int > 0	24	hours
Description	This metric sets the frequency of updates of the list of disclosed vulnerabilities. For example, for *list_update_frequency=12*, SPECS ensures that the list of published vulnerabilities will be updated and presented at least once every 12 hours.			
Actions taken to enforce the metric	First, vulnerability list is generated. Then periodically: 1. Update and present the vulnerability list.			

Table 33. SVA security metric LUF

Name		Value	Default value	Unit
Scanning frequency – basic scan (**BSF**)		int > 0	24	hours
Description	This metric sets the frequency of a basic software vulnerability scan. For example, for *scanning_frequency=24*, SPECS ensures that software vulnerability scans will be performed at least once every day.			
Actions taken to enforce the metric	Before the first scan, vulnerability list is generated. Then periodically: 1. Perform scan. 2. Build and present scanning report.			

Table 34. SVA security metric BSF

[9] For details see D3.3.

Name		Value	Default value	Unit
Scanning frequency – extended scan (**ESF**)		int > 0	24	hours
Description	This metric sets the frequency of an extended software vulnerability scan. For example, for *scanning_frequency=48*, SPECS ensures that software vulnerability scans will be performed at least once every two days. Scans are performed with two scanners and both scanning reports are presented.			
Actions taken to enforce the metric	Before the first scan, vulnerability list is generated. Then periodically: 1. Perform scan with both scanners. 2. Build and present both scanning reports.			

Table 35. SVA security metrics ESF

Name		Value	Default value	Unit
Up report frequency (**URF**)		int > 0	24	hours
Description	This metric sets the frequency of checks for updates and upgrades of vulnerable installed libraries. SPECS first updates vulnerability list, performs the vulnerability scan of the system, and then checks for available updates and upgrades of libraries on which vulnerabilities have been detected). For example, for *up_report_frequency=24*, SPECS ensures that checks for updates and upgrades are performed at least once every day.			
Actions taken to enforce the metric	Before the first check, vulnerability list is generated, vulnerability scan is performed, and scanning report is generated. Then periodically: 1. Check for updates/upgrades. 2. Build and present report.			

Table 36. SVA security metric URF

Name		Value	Default value	Unit
Penetration testing activated (**PTA**)		yes / no	yes	n/a
Description	This metric activates the penetration testing activity. The metric can be chosen together with metrics related to vulnerability scans. If chosen, scanner with penetration testing functionality is deployed.			
Actions taken to enforce the metric	If the metric is chosen, scanners with penetration testing functionality are installed. After installation of chosen scanners is successful, we consider metric respected through the entire SLA life-cycle.			

Table 37. SVA security metric PTA

With metric *Scanning frequency – extended scan* an EU has an opportunity to install two different vulnerability scanners (OpenSCAP [26], OpenVAS [27], and/or Nikto [28]) on the target service. If two scanners are used and scans are performed for the same list of disclosed vulnerabilities, comparison of both scanning reports can quickly outline possible false positives or false negatives. In case the EU chooses the extended scan, SPECS provides two scanning reports.

If an EU requires extended vulnerability scans, SPECS installs OpenSCAP and OpenVAS. If

penetration testing is also required (through security metric *Penetration testing activated*), OpenVAS and Nikto are the installed scanners. When the EU requires basic scan, the primary scanners are OpenSCAP and OpenVAS (the decision depends on the value of the metric *Penetration testing activated*).

Note that by selecting metrics BSF, ESF, and URF, the SVA mechanism periodically generates scanning and update/upgrade reports that are accessible to the EU via the SVA Dashboard.

Each SVA metric is associated with one basic and one or more additional measurements (with which the alert/violation thresholds are set and MoniPoli rules are built). The following tables present all measurements together with MoniPoli rules associated to SVA metrics.

Metric	List update frequency (LUF)
SLO	list_update_frequency = N hours
Measurements	**MoniPoli rules**
list_age	list_age ≤ N
repository_availability	repository_availability = yes

Table 38. Measurements and MoniPoli rules associated to SVA metric LUF

Metric	Scanning frequency – basic scan (BSF)
SLO	scan_basic_frequency = N hours
Measurements	**MoniPoli rules**
report_basic_age	report_basic_age ≤ N
list_availability	list_availability = yes
scanners_availability	scanners_availability = yes

Table 39. Measurements and MoniPoli rules associated to SVA metric BSF

Metric	Scanning frequency – extended scan (ESF)
SLO	scan_extended_frequency = N hours
Measurements	**MoniPoli rules**
report_extended_age	report_extended_age ≤ N
list_availability	list_availability = yes
scanners_availability	scanners_availability = yes

Table 40. Measurements and MoniPoli rules associated to SVA metric ESF

Metric	Up report frequency (URF)
SLO	up_report_frequency = N hours
Measurements	**MoniPoli rules**
up_report_age	up_report_age ≤ N
scan_report_availability	scan_report_availability = yes
up_report_availability	up_report_availability = yes

Table 41. Measurements and MoniPoli rules associated to SVA metric URF

Metric	Penetration testing activated (PTA)
SLO	penetration_testing = yes/no
Measurements	**MoniPoli rules**
pen_testing_activated	pen_testing_activated = PTA_value

Table 42. Measurements and MoniPoli rules associated to SVA metric PTA

In order to generate list of published vulnerabilities, the repository from where the information is extracted has to be available. Hence measurement *repository_availability* for metric LUF (in Table 38). In order to ensure that the list has been generated in right time, the age of the vulnerability list is monitored.

Since for vulnerability scans responsive scanners are needed and since vulnerability scanners require a list of published vulnerabilities, metrics BSF and ESF are mapped to measurements *list_availability* and *scanners_availability*. In order to ensure that scans have been performed in time, the age of scanning reports are monitored (with measurements *basic_report_age* and *extended_report_age*).

Update/upgrade report presents a list of libraries installed on the system that have been labelled as vulnerable by the scanners and for which updates and/or upgrades are available. So in order to check for available updates and upgrades, scanning report has to be available (hence *scan_report_availability* measurement). Similarly as before, in order to evaluate if the update/upgrade report has been generated in time, the measurement *up_report_age* has been introduced and mapped to the URF metric. One additional measurement (*up_report_availability*) is needed in the SLA remediation phase.

Metric PTA has only one associated measurement, namely *pen_testing_activated*. With this measurement responsiveness of the scanner with penetration testing functionality is observed.

Note that the current prototype does not yet support enforcement of metrics ESF and PTA since OpenVas and Nikto have not been integrated. Integration will be made by the end of the project and presented in D4.3.3.

The defined SVA metrics implement NIST and CCM security controls presented in the following table.

Control Family/Group	Control Name	Control ID	Security metric				
			LUF	BSF	ESF	URF	PTA
NIST							
Security Assessment and Authorization	Continuous Monitoring	CA-7		✓	✓	✓	
	Continuous Monitoring | Trend Analyses	CA-7 (3)	✓				
	Penetration testing	CA-8					✓
Risk Assessment	Vulnerability Scanning	RA-5		✓	✓	✓	
	Vulnerability Scanning | Update Tool Capability	RA-5 (1)	✓				
CCM							
Threat and Vulnerability Management	Vulnerability/Patch Management	TVM-02	✓	✓	✓	✓	✓

Table 43. Mapping of SVA metrics to NIST and CCM security controls

4.4.1.3. Remediation

As discussed in Section 3.5, each measurement defines one monitoring event. Table 44 below lists all possible monitoring events related to SVA metrics that can be detected by the Monitoring module.

ID	Condition	Affected metrics		Event type
SVA-E1	list_age > LUF_value	LUF		violation
SVA-E2	report_basic_age > BSF_value	BSF		
SVA-E3	report_extended_age > ESF_value	ESF		
SVA-E4	up_report_age > URF_value	URF		
SVA-E5	pen_testing_activated != PTA_value	PTA		
SVA-E6	repository_availability = no	LUF		alert
SVA-E7	list_availability = no	BSF	ESF	
SVA-E8	scanners_availability = no	BSF	ESF	
SVA-E9	scan_report_availability = no	URF		
SVA-E10	up_report_availability = no	URF		

Table 44. Monitoring events related to SVA metrics

The following table presents actions needed to remediate SVA alerts and violations.

ID	Description
SVA-A1	Check if the configured repository is available.
SVA-A2	Reconfigure repository and check if it is available.
SVA-A3	Check if vulnerability list is available.
SVA-A4	Delete vulnerability list, generate ne vulnerability list and check if it is available.
SVA-A5	Check if installed scanners are available.
SVA-A6	Delete old scanning report, scan again, and check if the new scanning report is available.
SVA-A7	Reinstall scanners and check if they are available.
SVA-A8	Check if the scanning report is available.
SVA-A9	Delete old up report, check for updates/upgrades and check if the new up report is available.

Table 45. SVA remediation actions

The next three figures report remediation plans related to alerts and violations of SVA metrics. For details on the structure of a remediation plan see Section 3.5.

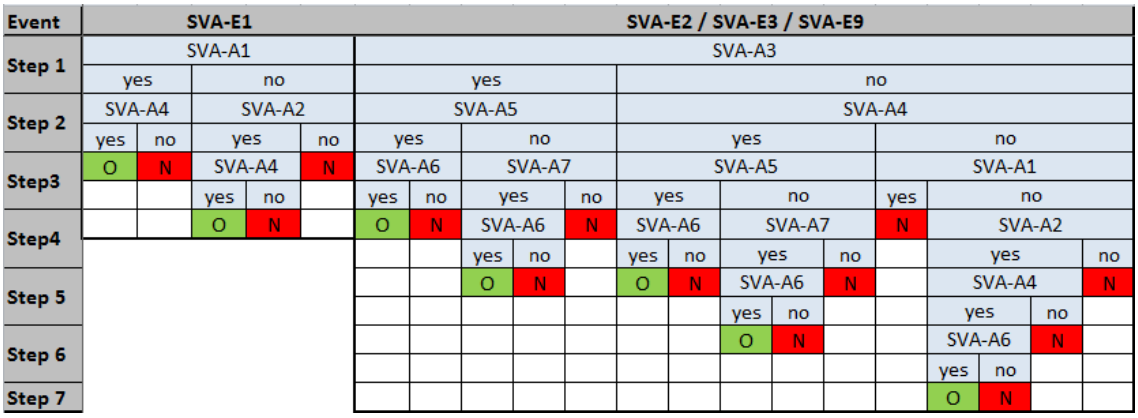

Figure 34. Remediation plans for monitoring events SVA-E1, SVA-E2, SVA-E3, and SVA-E9

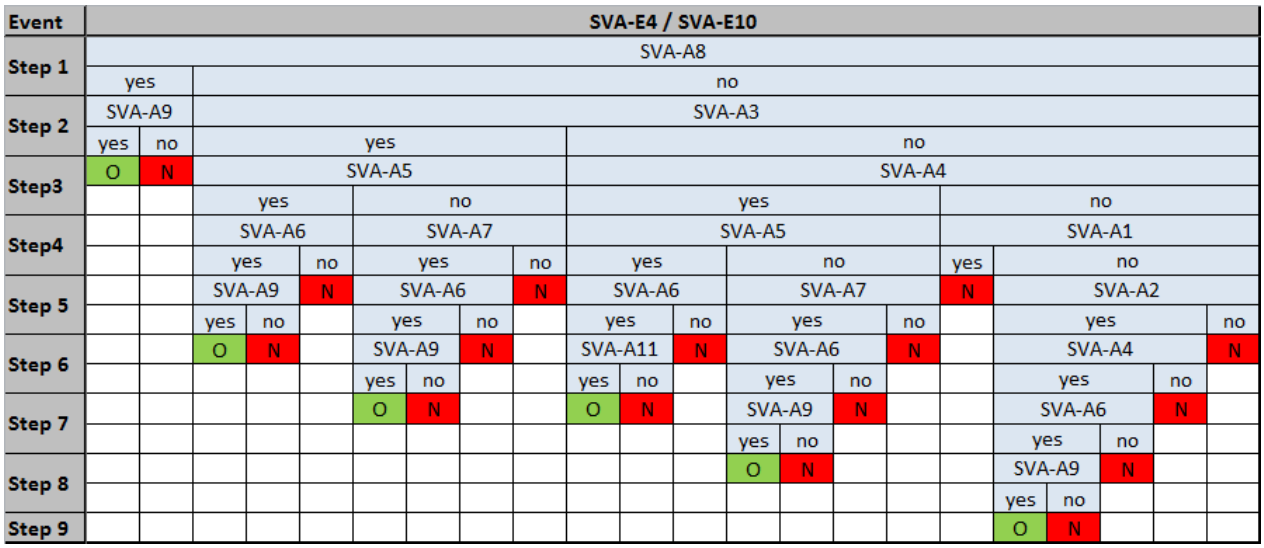

Figure 35. Remediation plans for monitoring events SVA-E4 and SVA-E10

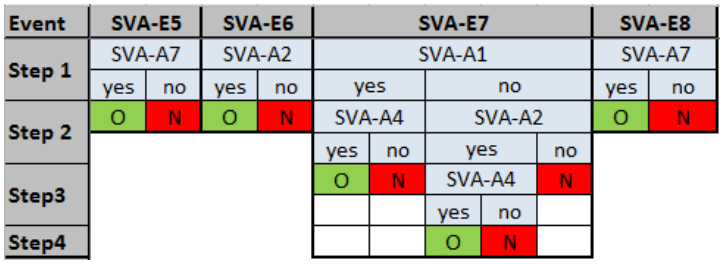

Figure 36. Remediation plans for monitoring events SVA-E5, SVA-E6, SVA-E7, and SVA-E8

All mechanism's implementation, configuration, and remediation details are available on project's Bitbucket [19]. The code for all three components is also available on project's Bitbucket repository:

- SVA Core (containing all common files for SVA Enforcement and SVA Monitoring components) is available on [5].
- SVA Dashboard component is available on [6].
- SVA Enforcement component (including OpenSCAP scanner) is available on [7].
- SVA Monitoring component is available on [8].

4.4.1.4. Development

With respect to the initial design of the mechanism (presented in D4.3.1), the main changes occurred with the introduction of additional vulnerability scanners. Also, new security metrics have been defined (initial prototype only supported metrics BSF and LUF) and remediation actions for the entire set of SVA metrics have been defined and detailed. Introduction of new metrics resulted in the need to adopt the initial prototype to support them (e.g., the need to adjust the SVA Monitoring component to take measurements mapped to new metrics, the need to adjust the SVA Dashboard to present the status of SLOs related to new metrics, the need to adjust SVA Enforcement to be able to change repository to extract known vulnerabilities). Some improvements of the initial functionalities have also been conducted.

Due to the delicate nature of automatically applying patches and fixing software vulnerabilities, this functionality has not been integrated. This aspect may be explored during the last period of the mechanism's development.

The design of the mechanism integrates existing open-source tools for vulnerability assessment (OpenVAS, Nikto), but the majority of mechanism's components (SVA Enforcement, SVA Monitoring, SVA Dashboard) were developed in the context of the project according to elicited requirements and the design of the core Enforcement components, SVA Enforcement had to be developed to manage vulnerability lists, scans, and reports, SVA Monitoring component had to be developed to support automatic remediation activities, and SVA Dashboard had to be developed to provide to the EU all information related to SVA activities.

The following subsections briefly describe repositories and provide installation and usage guides.

4.4.2. Repository

Each SVA component has its own repository on Bitbucket; SVA Enforcement on [7], SVA Monitoring on [8], and SVA Dashboard on [6].

4.4.3. Description and design

The repositories for SVA Enforcement and SVA Monitoring consist of two modules. One for the source code and one for unit tests.

The repository for the SVA Dashboard comprises four modules. Dashboard and dashboard-web are Django root folder and Django settings folder, respectively. Static module includes all static files (javascript, less, css, etc.). The last module is for unit tests.

4.4.4. Installation

Prerequisites:

- Redis
- PostgreSQL
- Django

Core repository [5] contains core functionaility of both SVA Enforcement and SVA monitoring component.

```
hg clone https://bitbucket.org/specs-team/specs-mechanism-enforcement-sva_core
specs_sva_core
```

Install postgresql

```
zypper install postgresql-devel
zypper install postgresql
zypper install postgresql-contrib
zypper install python-devel
service postgresql start
sudo -u postgres psql -c "ALTER USER postgres PASSWORD 'sva';"
sudo -u postgres createdb sva
sudo -u postgres createuser -P sva
```

You will be prompted for password, the password should be *sva*.

```
sudo -u postgres psql -c "GRANT ALL PRIVILEGES ON DATABASE sva TO sva;"
```

Enforcement component installation guide:

Switch to root user:

```
su root
```

Create a virtual enviroment with:

```
pip install virtualenv
virtualenv /path/to/env
source /path/to/env/bin/activate
```

Clone repository:

```
hg clone https://bitbucket.org/specs-team/specs-mechanism-enforcement-sva_vulnerability_manager specs_enforcement_sva
```

Install requirements:

```
pip install -r /path/to/specs_enforcement_sva/requirements.txt
```

Monitoring component installation guide:

You can use same virtual enviroment as above since enforcement and monitroing share the same package requirements.

Clone repository:

```
hg clone https://bitbucket.org/specs-team/specs-mechanism-monitoring-sva
specs_monitorign_sva
```

Dashboard component installation guide:

Switch to root user:

```
su root
```

Install required packages and configure postgresql:

```
zypper install postgresql-devel
zypper install postgresql
zypper install postgresql-contrib
zypper install python-devel
zypper install redis
zypper install nodejs
zypper install mercurial
zypper install python-pip
service postgresql start
sudo -u postgres psql -c "ALTER USER postgres PASSWORD 'dashboard';"
sudo -u postgres createdb dashboard
sudo -u postgres createuser -P dashboard
```

You will be prompted for password, the password should be *dashboard*.

```
sudo -u postgres psql -c "GRANT ALL PRIVILEGES ON DATABASE dashboard TO dashboard;"
```

Clone django repository:

```
hg clone https://bitbucket.org/specs-team/specs-mechanism-enforcement-sva_dsahboard
specs-enforcement-sva-dashboard
```

Install project requirements using pip:

```
python -m pip install -r /path/to/specs-enforcement-sva-dashboard/dashboard/requirements.txt
```

Install less:

```
npm install -g less
```

Run redis server, which is used for asynchroneous execution of celery tasks:
```
redis-server –daemonize yes
```

Migrate django database:
```
python /path/to/specs-enforcement-sva-dashboard/dashboard/manage.py
migrate
```

Open port on firewall so server is accessible from others virtual machines:
```
SuSEfirewall2 open EXT TCP 8000
SuSEfirewall2 stop
SuSEfirewall2 start
service SuSEfirewall2 restart
```

Run celery worker:
```
cd /path/to/specs-enforcement-sva-dashboard/dashboard/
screen -S celeryWorkers -m -d celery -A dashboard_web worker -B --
loglevel=INFO –concurrency=10
```

Run django server:
```
screen -S djangoServer -m -d python /path/to/specs-enforcement-sva-
dashboard/dashboard/manage.py runserver 0.0.0.0:8000
```

Django server is now accessible on http://localhost:8000 and ready to receive reports and display them.

4.4.5. Usage

Once all SVA components are installed and running the following should serve as a guideline.

Enforcement component usage
Switch to root user:
```
su root
```

Activate virtual enviroment:
```
source /path/to/env/bin/activate
```

Run the SVA Enforcement component with three arguments (scanning_frequency, list_update_frequency, up_report_frequency) in seconds. Script will automatically run basic scan, download oval files, and generate upgrade/update report:
```
python /path/to/specs_enforcement_sva/src/enforcement.py run_enforcement
3600 3600 3600
```

Or you can run each metric separately as follows.

Will download oval vulnerability list:
```
python /path/to/specs_enforcement_sva/src/enforcement.py
vulnerability_list
```
Will perform basic scan using OpenSCAP and send results to the SVA Dashbaord:
```
python /path/to/specs_enforcement_sva/src/enforcement.py
vulnerability_scan
```
Will generate upgrade/update report and send results to SVA Dashbaord:
```
python /path/to/specs_enforcement_sva/src/enforcement.py upgrade_report
```

Will reconfigure repository for fetching oval files:
```
python /path/to/specs_enforcement_sva/src/enforcement.py
reconfigure_repository
```

Monitoring component usage
Switch to root user:
```
su root
```

Activate virtual enviroment:
```
source /path/to/env/bin/activate
```

Run the SVA Monitoring component with three arguments (scanning_frequency, list_update_frequency, up_report_frequency) in seconds. Script will automatically run availability checks and send event reports to the Event Hub (monitoring module) and the SVA Dashbaord:
```
python /path/to/specs_monitoring_sva/src/monitoring.py run_monitoring
3600 3600 3600
```

Or you can also run each measurements separately as follows.

Checks for availability of the current repository and sends report to the Event Hub:
```
python /path/to/specs_monitoring_sva/src/monitoring.py invoke_msr6
```

Checks if vulnerability list is available and sends report to the Event Hub:
```
python /path/to/specs_monitoring_sva/src/monitoring.py invoke_msr7
```

Checks availability of installed scanner (OpenSCAP) and sends report to the Event Hub:
```
python /path/to/specs_monitoring_sva/src/monitoring.py invoke_msr8
```

Checks if the scanning report is available and sends report to the Event Hub:
```
python /path/to/specs_monitoring_sva/src/monitoring.py invoke_msr9
```

Checks if the upgrade/update report is available and sends report to the Event Hub:
```
python /path/to/specs_monitoring_sva/src/monitoring.py invoke_msr10
```

Dashboard component usage
Open http://localhsot:8000 in browser. See Figure 37 for the snapshot of the SVA Dashboard.

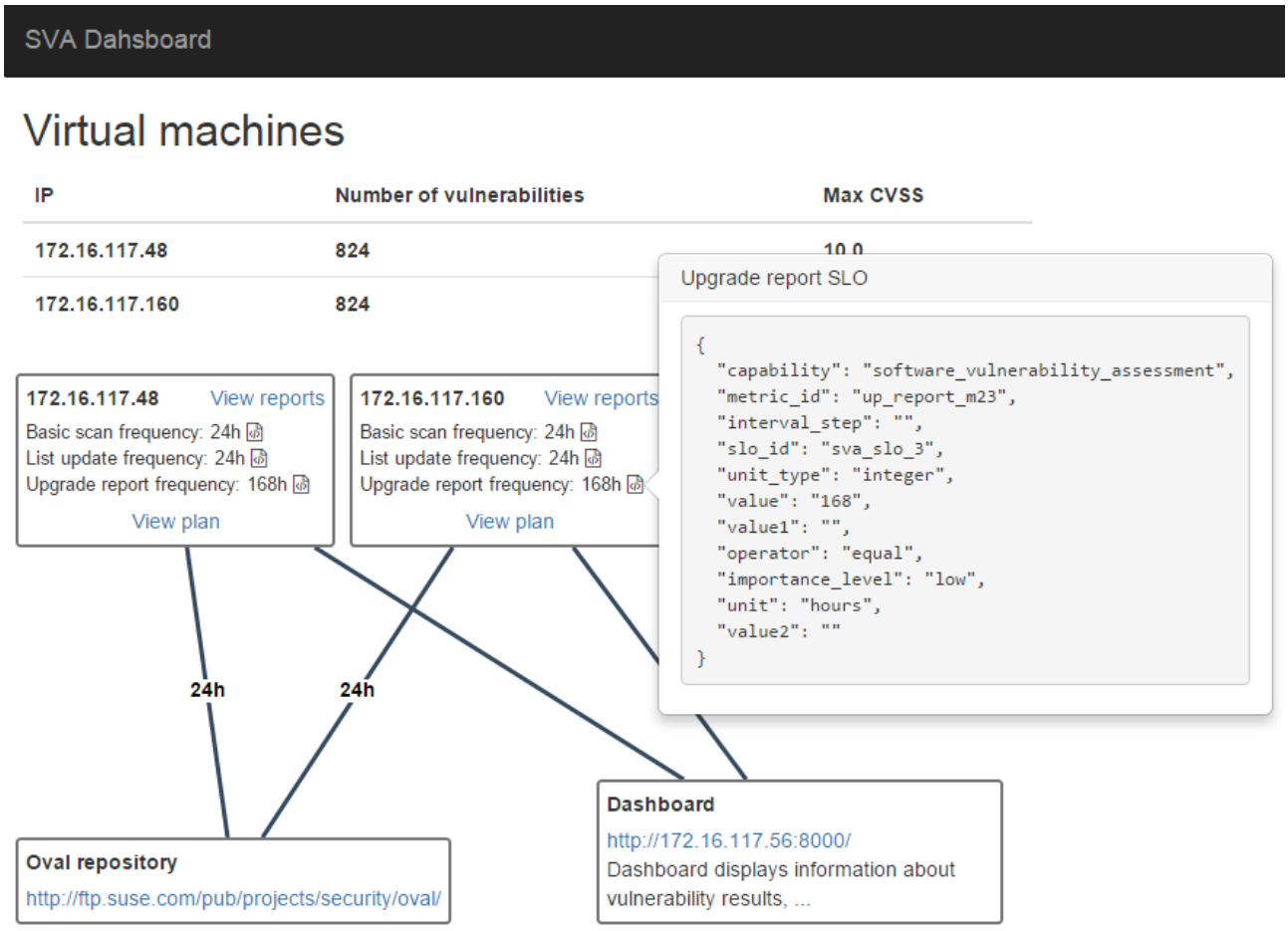

Figure 37. Snapshot of the SVA Dashboard

Clicking on one of the virtual machines will redirect you to a new page, where you can see all detected vulnerabilities and track age of reports (see Figure 38 for an excerpt).

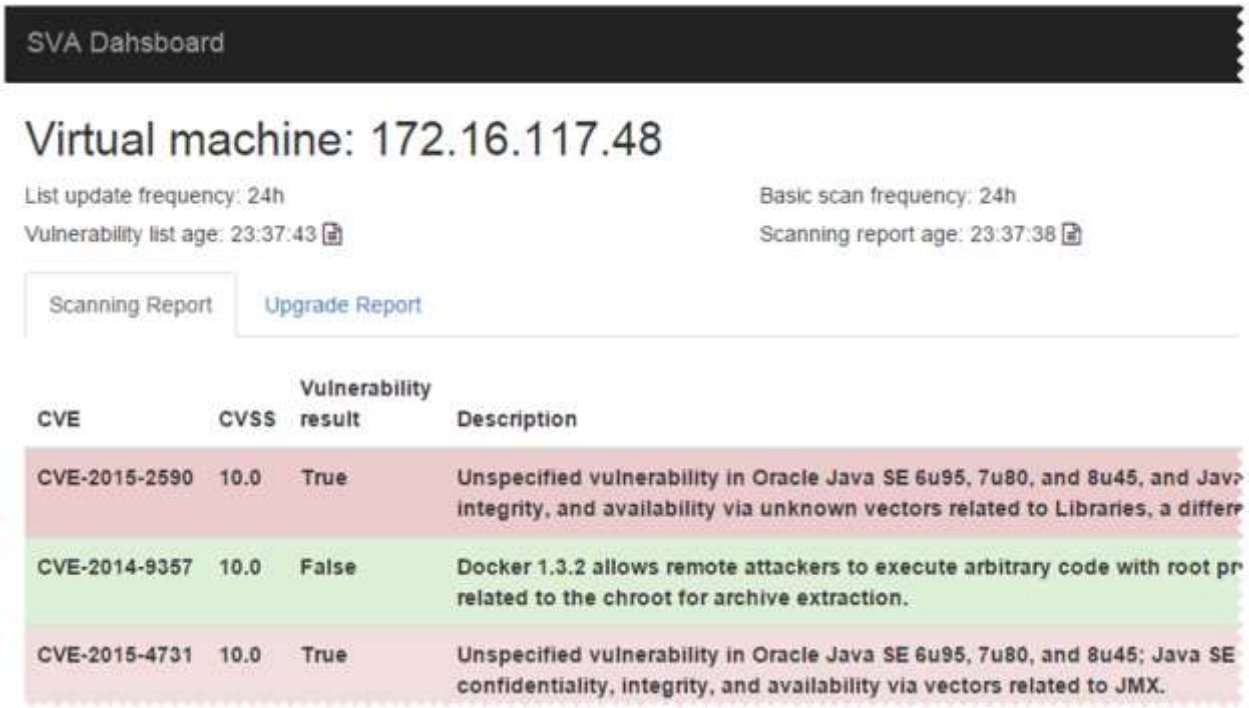

Figure 38. SVA reports for a VM

4.5. TLS mechanism

In this section, we present a description and implementation details for the mechanisms involved in the following validation scenarios defined in T5.1 (see D5.1.2):

- *SWC-03 Secure_Web_Container_TLS_Enhanced*
- *SWC-05 Secure_Web_Container_TLS_SVA_Enhanced_Violation*
- *SWC-06 Secure_Web_Container_TLS_Multitenancy*

4.5.1. Overview

SPECS TLS represents a security mechanism that ensures data integrity and mutual authentication between two or more communicating actors. SPECS TLS is a secure proxy that allows services to securely communicate with clients although they are not designed to deliver data over the secure communication channel. In this way any service can be easily secured at communication level by using the SPECS TLS mechanism.

4.5.1.1. Architecture

SPECS TLS architecture consists of several components that ensure the entire functionality as illustrated in Figure 39:

- **TLS terminator** is the main component that acts as a proxy for secure communication between clients and targeted services secured by SPECS. TLS Terminator is able to support multiple targeted services using a single (common) or multiple security credentials (TLS certificates) used for secure communication. Moreover it also supports different HTTPs security features (explained below) to be enforced based on the configuration generated from the SLA agreed between the clients and SPECS.
- **TLS terminator configurator** acts as a configuration module able to create different configuration templates for the TLS Terminator and TLS Prober. The configuration templates enable different TLS features based on the metrics that are described in the SLA to be enforced and monitored.
- **TLS terminator controller** module is in charge with the management of the TLS Terminator. It ensures the availability of the TLS Terminator and offers controls for starting, stopping and reconfiguring the service.
- **TLS prober** is a monitoring component that need to: monitor that the initial configuration templates are not change during the lifecycle and in case of anomalies (example: configuration changes) to generate monitoring events with the detected change that affects a specific metric.
- **TLS reasoner** is a module that decides what configuration rules must be added in a configuration template based on a list of metrics needed to be enforced and monitored. The reasoner will translate metrics into TLS security definitions in order to ensure the final TLS functionality.
- **TLS endpoint** (web server) represents a list of services where the HTTP request must be sent. This endpoint is not maintained by TLS security mechanism. TLS will only monitor the availability of the TLS endpoint.

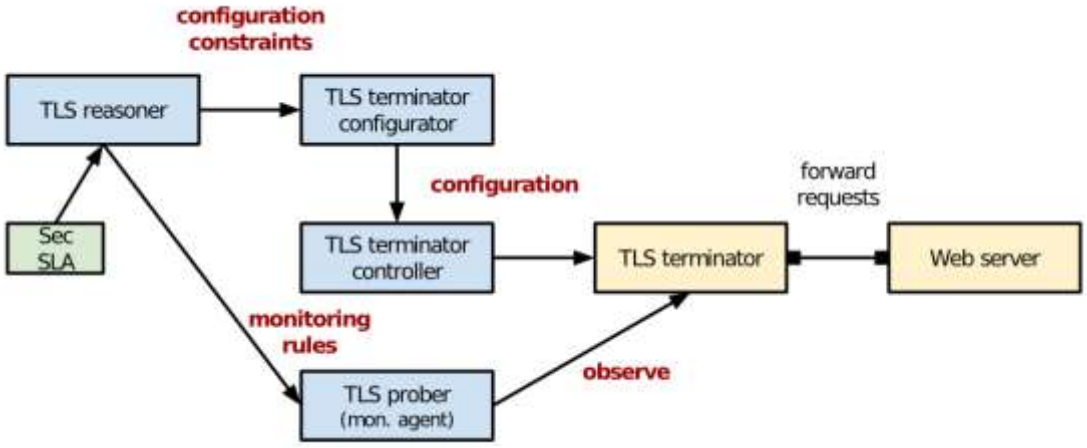

Figure 39. Architecture of the TLS mechanism

4.5.1.2. Security metrics and controls

SPECS TLS security mechanism is able to enforce and monitor specific security metrics related to HTTP transport security. The supported security metrics are described in the following tables.

	Value	Default value	Unit
TLS cryptographic strength (**TCS**)	0<int<8	7	-
Description	This metric sets the cryptographic strength to be used by the TLS Terminator. TLS Terminator Configurator will choose the appropriate cryptographic ciphers that meet the negotiated level, and configure TLS Terminator accordingly.		
Actions taken to enforce the metric	Generate TLS Terminator configuration to support the chosen cryptographic strength level and periodically monitor the value.		

Table 46. TLS security metric TCS

	Value	Default value	Unit
Forward secrecy (**FS**)	yes/no	no	-
Description	This metric ensures that the encrypted data sent through a session of the TLS secure channel cannot be decrypted even if the cryptographic data, used to generate the cryptographic credentials for that session, are compromised.		
Actions taken to enforce the metric	Generate TLS Terminator configuration to support forward secrecy periodically check the configuration.		

Table 47. TLS security metric FS

	Value	Default value	Unit
HTTP strict transport security (**HSTS**)	yes/no	no	-
Description	This metric is a feature of HTTP transport layer that declares the web content available only over a secure HTTP connection.		
Actions taken to enforce the metric	Generate TLS Terminator configuration to support HTTP strict transport security periodically check the configuration.		

Table 48. TLS security metric HSTS

	Value	Default value	Unit
HTTP to HTTPS redirects (**HHSR**)	yes/no	no	-
Description	This metric is a feature of HTTP delivery service that forces clients to use only secure HTTP protocol.		
Acions taken to enforce the metric	Generate TLS Terminator configuration to support HTTP to HTTPS redirects periodically check the configuration.		

Table 49. TLS security metrics HHSR

	Value	Default value	Unit
Secure cookies (**SC**)	yes/no	no	-
Description	This metric is a feature of HTTP protocol to force the clients to download session cookies, delivered by the HTTP services, only through a secured HTTP communication		
Actions taken to enforce the metric	Generate TLS Terminator configuration to support secure cookies periodically check the configuration.		

Table 50. TLS security metric SC

	Value	Default value	Unit
Certificate pinning (**CP**)	yes/no	no	-
Description	This metric is a feature of HTTP protocol allowing the verification of the SSL certificates between the client and the HTTP service where the hash of the public certificate is pinned into the HTTP response.		
Actions taken to enforce the metric	Generate TLS Terminator configuration to support certificate pinning, generate SSL certificate hash and periodically check the configuration.		

Table 51. TLS security metrics CP

Based on the above metrics a set of measurements are defined to ensure that the metrics are enforced during the SLA life cycle. The list of defined measurements for each TLS metric is represented in the following tables.

Metric	TLS cryptographic strength (TCS)
SLO	tls_crypto_strength = N
Measurements	MoniPoli rules
tls_crypto_strength	tls_crypto_strength = N
tls_terminator_availability	tls_terminator_availability = yes
tls_endpoint_availability	tls_endpoint_availability = yes

Table 52. Measurements and MoniPoli rules associated to TLS metric TCS

Metric	Forward secrecy (FS)
SLO	forward_secrecy = yes/no
Measurements	MoniPoli rules
forward_secrecy	forward_secrecy = FS_value
tls_terminator_availability	tls_terminator_availability = yes
tls_endpoint_availability	tls_endpoint_availability = yes

Table 53. Measurements and MoniPoli rules associated to TLS metric FS

Metric	HTTP strict transport security (HSTS)
SLO	hsts = yes/no
Measurements	MoniPoli rules
hsts	hsts = HSTS_value
tls_terminator_availability	tls_terminator_availability = yes
tls_endpoint_availability	tls_endpoint_availability = yes

Table 54.Measurements and MoniPoli rules associated to TLS metrics HSTS

Metric	HTTP to HTTPS redirect (HHSR)
SLO	http_redirect = yes/no
Measurements	MoniPoli rules
http_redirect	http_redirect = HHSR_value
tls_terminator_availability	tls_terminator_availability = yes
tls_endpoint_availability	tls_endpoint_availability = yes

Table 55. Measurement and MoniPoli rules associated to TLS metrics HHSR

Metric	Secure cookies (SC)
SLO	secure_cookies = yes/no
Measurements	MoniPoli rules
secure_cookies	secure_cookies = SC_value
tls_terminator_availability	tls_terminator_availability = yes
tls_endpoint_availability	tls_endpoint_availability = yes

Table 56. Measurements and MoniPoli rules associated to TLS metric SC

Metric	Certificate pinning (CP)
SLO	certificate_pinning = yes/no
Measurements	MoniPoli rules
certificate_pinning	certificate_pinning = CP_value
tls_terminator_availability	tls_terminator_availability = yes
tls_endpoint_availability	tls_endpoint_availability = yes

Table 57. Measurements and MoniPoli rules associated to TLS metric CP

The above defined TLS security metrics implement NIST and CCM security controls presented in the following table.

Control Family/Group	Control Name	Control ID		Security metric				
			TCS	TFS	THS	THR	TSC	TCP
NIST								
System and Communications Protection	Cryptographic protection	SC-13	✓					
	Public Key Infrastructure Certificates	SC-17						✓
	Heterogeneity	SC-29					✓	
	Cryptographic Key Establishment And Management	SC-12		✓				
	Transmission Confidentiality And Integrity	SC-8				✓		
	Usage Restrictions	SC-43			✓			
CCM								

Encryption & Key Management	Entitlement	EKM-01	✓					
	Sensitive Data Protection	EKM-03		✓		✓	✓	
Identity & Access Management	Credential Lifecycle / Provision Management	IAM-02			✓			
	User Access Authorization	IAM-09						✓

Table 58. Mapping of TLS metrics to NIST and CCM security controls

4.5.1.3. Remediation

Each measurement is defined by a value that needs to be checked periodically by the TLS prober in order to ensure that the configuration values are compliant with the initial values. The measurements are translated into events (Table 59) that can generate actions to be followed by the remediation decision system described in Section 3.5.

ID	Condition	Affected metrics	Event type
TLS-E1	tls_crypto_strength_level < TCS_value	TCS	violation
TLS-E2	tls_forward_secrecy != TFS_value	TFS	
TLS-E3	tls_hsts != THS_value	THS	
TLS-E4	tls_http_to_https_redirect != THR_value	THR	
TLS-E5	tls_force_secure_cookies != TSC_value	TSC	
TLS-E6	tls_certificate_pinning != TCP_value	TCP	
TLS-E7	tls_terminator_availability = no	All	alert
TLS-E8	tls_endpoint_availablity = no	All	

Table 59. Monitoring events related to TLS metrics

The following table presents actions needed to remediate TLS alerts and violations.

ID	Description
TLS-A1	Reconfigure TLS cryptographic strength to TCS_value and check if TLS_crypto_strength has the initial TCS_value.
TLS-A2	Restart TLS Terminator and check if it is available.
TLS-A3	Check if TLS_crypto_strength is >= TCS_value.
TLS-A4	Reconfigure TLS forward secrecy to TFS_value and check if TLS_forward_secrecy has the initial TFS_value.
TLS-A5	Check if TLS_forward_secrecy has the initial TFS_value.
TLS-A6	Reconfigure TLS HSTS to THS_value and check if TLS_hsts has the initial THS_value.
TLS-A7	Check if TLS_hsts has the initial THS_value.
TLS-A8	Reconfigure TLS HTTP2HTTPS to THR_value and check if TLS_https_to_https_redirect has the initial THR_value.
TLS-A9	Check if TLS_http_t_https_redirect has the initial THR_value.
TLS-A10	Reconfigure TLS FSC to TSC_value and check if TLS_force_secure_cookies has the initial TSC_value.
TLS-A11	Check if TLS_force_secure_cookies has the initial TSC_value.
TLS-A12	Reconfigure TLS CP to TCP_value and check if TLS_certificate_pinning has the initial TCP_value.
TLS-A13	Check if TLS_certificate_pinning has the initial TCP_value.
TLS-A14	Request, to an external service, TLS Endpoint restart and check if the TLS Endpoint is available.

Table 60. TLS remediation actions

The next three figures report remediation plans related to alerts and violations of TLS metrics. For details on the structure of a remediation plan see Section 3.5.

Event	TLS-E1				TLS-E2				TLS-E3				TLS-E4			
Step 1	TLS-A1				TLS-A4				TLS-A6				TLS-A8			
	yes	no			yes	no			yes	no			yes	no		
Step 2	O	TLS-A2			O	TLS-A2			O	TLS-A2			O	TLS-A2		
		yes	no			yes	no			yes	no			yes	no	
Step3		TLS-A3	N			TLS-A5	N			TLS-A7	N			TLS-A9	N	
		yes	no			yes	no			yes	no			yes	no	
Step4		O	N			O	N			O	N			O	N	

Figure 40. Remediation plans for monitoring events TLS-E1, TLS-E2, TLS-E3, and TLS-E4

Event	TLS-E5				TLS-E6				TLS-E7		TLS-E8	
Step 1	TLS-A10				TLS-A12				TLS-A2		TLS-A14	
	yes	no			yes	no			yes	no	yes	no
Step 2	O	TLS-A2			O	TLS-A2			O	N	O	N
		yes	no			yes	no					
Step3		TLS-A11	N			TLS-A13	N					
		yes	no			yes	no					
Step4		O	N			O	N					

Figure 41. Remediation plans for monitoring events TLS-E5, TLS-E6, TLS-E7, and TLS-E8

All mechanism's implementation, configuration, and remediation details are available on project's Bitbucket [29].

4.5.1.4. Development

TLS security mechanism integrates open-source tools for HTTP proxy and SSL features enforcement but most of the TLS components (Terminator, Terminator Configurator, and Terminator Controller) were developed in the context of the project with respect to the requirements and the design of the core Enforcement components.

TLS security mechanism design is finalised and there will be no modifications in the overall architecture by the end of the project. However, the implementation of the mechanism is partially finalised as some changes are expected in order to improve the overall performance and for a better integration with the Service Manager (SLA Platform) used by the underlying operating system.

The following subsections briefly describe repositories and provide installation and usage guides.

4.5.2. Repository

TLS Mechanism source code is hosted as a Bitbucket repository available at [29].

4.5.3. Description and design

TLS Mechanism has two parts that are hosted under the same repository:

- HTTP Proxy component;
- TLS security component;

HTTP Proxy component is in charge with the management actions of the proxy technology used for HTTP protocol communication (both secure and unsecure) intermediation. Under the

repository the files that start with "*proxy-*" are related to this purpose.

TLS security component tackles the aspects of transport layer security that are strictly related to SSL aspects of HTTP protocol communication (features used to enforce and monitor a particular metric associated with TLS Mechanism).

4.5.4. Installation

Prerequisites:

- HAProxy, min. version 1.5.14
- OpenSSL, min. version 1.0.x

```
hg clone https://bitbucket.org/specs-team/specs-mechanism-enforcement-tls
/opt/specs-mechanism-enforcement-tls
zypper install haproxy
zypper install openssl
export PATH=$PATH:/opt/specs-mechanism-enforcement-tls
```

If TLS security mechanism is not installed and configured through SPECS Chef service, then the monitoring related information must be manually added in `/opt/specs-mechanism-enforcement-tls/etc/proxy-config.sh` file:

- `_proxy_prober_monitoring_url=`
  - monitoring event-hub endpoint in URI format;
- `_proxy_prober_monitoring_username=`
  - (optional) monitoring event-hub username;
- `_proxy_prober_monitoring_password=`
  - (optional) monitoring event-hub password
- `_proxy_prober_monitoring_component=`
  - monitoring event-hub component related information;
- `_proxy_prober_monitoring_object=`
  - monitoring event-hub object related information;
- `_proxy_prober_monitoring_labels=`
  - monitoring event-hub labels related information;
- `_proxy_prober_monitoring_type=`
  - monitoring event-hub type related information;
- `_proxy_prober_monitoring_data=`
  - (optional) monitoring event-hub data related information;

4.5.5. Usage

Once TLS security mechanism is installed the following commands should run as starting point:

TLS Enforcement

```
# enforce cryptographic strength level < 7
tls-configurator --m3
# define TLS endpoint details
tls-configurator --tls-backend BACKEND_IP:BACKEND:PORT
# start TLS Terminator
tls-controller start
```

TLS Monitoring

```
# check cryptographic strength level < 7
tls-prober --m3
# check if the TLS Terminator is online
tls-prober --tls-msr7
```

For more detailed information please check Bitbucket repository Wiki page or use the `--help` switch:

```
tls-configurator --help
tls-controller --help
tls-prober --help
```

5. Conclusions

This document presents prototypes of the core part of the Enforcement module and demonstrates all security mechanisms considered in year 2 of the SPECS project. The choice of security mechanisms presented in this deliverable was based on implementation plans reported at M12 which were based on EU's requirements.

Note that two security mechanisms intended to secure interactions among SPECS components, namely Credential Service and Security Tokens, although part of the Enforcement module, are discussed in deliverables of the dedicated task T4.4 (see D4.4.2). Similarly, one of the core Enforcement components, namely the Auditing, offering its functionalities to all elements of the SPECS framework and thus considered as part of the Vertical Layer, is presented in deliverables of task T1.4 (see D1.4.1 and D1.4.2). Moreover, all testing activities are discussed in D4.5.2.

This document is an extension of two M12 deliverables, namely D4.2.2 presenting the Enforcement architecture and D4.3.1 related to the initial prototypes. All design changes which were needed due to updates in other tasks and due to the feedback received from developers and integrators, and all refinements in the enforcement process which were anticipated in D4.2.2, are presented in current deliverable:

- Refinements of the SLA implementation phase to support generation of valid supply chains and to support improved generation of implementation plans and their executions.
- Refinements of the SLA remediation phase to support diagnosis and remediation activities improved in year two.
- Prototypes of all Enforcement core components.
- Integration of Broker with the Implementation component.
- Design and demonstration of newly developed mechanism providing storage and backup as-a-Service (DBB).
- Demonstration of fully developed mechanism providing client side encryption (E2EE).
- Demonstration of upgraded software vulnerability assessment mechanism (SVA; integration of improved SVA mechanism demonstrated at M12 with a monitoring system OpenVAS).
- Demonstration of TLS mechanism.

The final iteration of this deliverable (namely D4.3.3) will focus on the following:

- Adding planning and implementation steps after renegotiation or a termination of an SLA. These activities are related to the Planning and Implementation components.
- Possibly developing a more meaningful methodology to determine risk and severity levels of SLA alerts and violations. This activity is related to the Diagnosis component.
- Applying any changes and improvements that would be needed due to integration issues. These also include changes related to improving performance and refer to all components and mechanisms.
- Possibly considering automatically applying patches and fixing software vulnerabilities with SVA mechanism.
- Elaborating on detectable attacks and system failures for each security mechanism.
- Development of the remaining two security mechanisms, namely AAA and DoS.

For detailed implementation plan see Figure 42.

Component		Y2		Y3
		M13-M18	M19-M24	M25-M30
main Enforcement components	Planning			
	Implementation			
	Diagnosis			
	RDS			
security mechanisms and controls	Secure Provisioning (Broker)			
	Secure Web Server (WebPool)			
	TLS			
	SVA			
	E2EE			
	DBB			
	AAA			
	DoS			

Figure 42. Implementation plan for the Enforcement module

6. Bibliography

[1] SPECS, "*SPECS Core Enforcement Planning*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-planning.

[2] SPECS, "*SPECS Core Enforcement Implementation*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-implementation.

[3] SPECS, "*SPECS Core Enforcement Diagnosis*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-diagnosis.

[4] SPECS, "*SPECS Core Enforcement RDS*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-rds.

[5] SPECS, "*SPECS Mechanism Enforcement SVA Core*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-sva_core.

[6] SPECS, "*SPECS Mechanism Enforcement SVA Dashboard*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-sva_dashboard.

[7] SPECS, "*SPECS Mechanism Enforcement SVA Vulnerability Manager*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-sva_vulnerability_manager.

[8] SPECS, "*SPECS Mechanism Monitoring SVA*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-monitoring-sva.

[9] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, L. Zhuang, "*Enabling Security in Cloud Storage SLAs with CloudProof*", In Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11).

[10] A. Albeshri, C. Boyd, J. G. Nieto, "*Enhanced GeoProof: improved geographic assurance for data in the cloud*", International Journal of Information Security, 13(2):191-198, 2014.

[11] D. Cash, A. Küpçü, D. Wichs, "*Dynamic Proofs of Retrievability via Oblivious RAM*", Advances in Cryptology (EUROCYRPT 2013), Lecture Notes in Computer Science, 7881:279-295, 2013.

[12] A. Juels, B. Kalinski, "*PORs: Proofs of Retrievability for Large Files*", In CCS Proceedings of the 14th ACM conference on Computer and communication security, 584-597, 2007.

[13] H. Shacham, B. Waters, "*Compact Proofs of Retrievability*", Advances in Cryptology (ASIACRYPT 2008), Lecture Notes in computer Science, 5350:90-107, 2008.

[14] E. Shi, E. Stefanov, C. Papamanthou, "*Practical Dynamic Proofs of Retrievability*", In Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security, 325-336, 2013.

[15] SPECS, "*SPECS Enforcement E2EE Server*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-e2ee-server.

[16] SPECS, "*SPECS Enforcement E2EE Client*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-e2ee-client.

[17] SPECS, "*SPECS Enforcement E2EE Auditor*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-monitoring-e2ee-auditor.

[18] JOptimizer, "*JOptimizer*", 2015. [Online]. Available: http://www.joptimizer.com/.

[19] SPECS, "*SPECS Core Enforcement Repository"*, 2015. [Online]. Available:

https://bitbucket.org/specs-team/specs-core-enforcement-repository/overview.

[20] SpiderOak, "*Crypton*", 2015. [Online]. Available: https://crypton.io/.

[21] C. Pedersen, D. Dahl, "*Crypton: Zero-knowledge Application Framework*". Whitepaper, 2013. [Online]. Available: https://crypton.io/crypton.pdf.

[22] Redis Labs, "*Redis*", 2015. [Online]. Available: http://redis.io/.

[23] The PostgreSQL Global Development Group, "*PostgreSQL*", 2015. [Online]. Available: http://www.postgresql.org/.

[24] NIST National Institute of Standards and Technology, "*Security and privacy controls for federal information systems and organizations*", NIST 800-53v4, 2013. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf.

[25] Cloud Security Alliance, "Cloud Controls Matrix Working Group", 2015. [Online]. Available: https://cloudsecurityalliance.org/group/cloud-controls-matrix/.

[26] Red Hat, "*OpenSCAP*", 2015. [Online]. Available: http://www.open-scap.org/page/Main_Page.

[27] OpenVAS, "*OpenVAS*", 2015. [Online]. Available: http://www.openvas.org/.

[28] C. Sullo, D. Lodge, "*Nikto2*", 2015. [Online]. Available: https://cirt.net/Nikto2.

[29] SPECS, "*SPECS Mechanism Enforcement TLS*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-tls/overview.

[30] SPECS, "*SPECS Core Enforcement Broker*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-broker/overview.

[31] Chef Software, "Chef", 2015. [Online]. Available: https://www.chef.io/chef/.

[32] SPECS, "*SPECS Mechanism Enforcement WebPool*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-enforcement-webpool/overview.

[33] SPECS, "SPECS Core Enforcement Repository", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-repository/src.

[34] HAProxy, "*HAProxy*", 2015. [Online]. Available: http://www.haproxy.org/.

[35] Nginx, "*Nginx*", 2015. [Online]. Available: http://nginx.org/en/.

[36] The Apache Software Foundation, "*Apache*", 2015. [Online]. Available: http://www.apache.org/.

[37] SPECS, "*SPECS Utility Data Model*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-utility-data-model.

[38] SPECS, "*SPECS Core Enforcement Repository E2EE*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-repository/src/master/E2EE/.

[39] SPECS, "*SPECS Core Enforcement Repository DBB*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-core-enforcement-repository/src/master/DBB/.

[40] Pivotal Software, "*Spring*", 2015. [Online]. Available: https://spring.io/.

[41] SPECS, "*SPECS Enforcement E2EE Monitoring Adapter*", 2015. [Online]. Available: https://bitbucket.org/specs-team/specs-mechanism-monitoring-e2ee-adapter.

Appendix 1. Solving the planning problem

The first phase of the process of automatic provisioning of cloud services is negotiation. SLA negotiation phase consist of eliciting End-user's (EU's) desired security requirements, mapping them to available security mechanisms, and building supply chains that implement EU's choice of security features. In what follows we focus on *formalizing and solving the planning problem*, i.e., on *modelling and determining the optimal deployment of security mechanisms' components* for the implementation of an SLA.

In SPECS, the generation of valid supply chains is orchestrated by the Enforcement's Planning component according to the input provided by the Negotiation module. During the negotiation phase[10], the EU specifies the desired security requirements. Negotiation module formalizes the requirements in terms of an SLA and identifies security mechanisms able to enforce and monitor all security parameters specified in the SLA. Once the Planning component receives the list of supported CSPs, the list of SLOs included in the SLA, and a list of security mechanisms able to implement it, the planning process begins.

The planning problem's input consist of

- CSP related information (e.g., maximum acquirable number of VMs),
- security mechanism related information (i.e., mechanisms' metadata), and
- EU's security requirements (i.e., SLOs).

First, the set of CSPs is parsed. For each CSP a list of zones, VM types, and maximum acquirable number of VMs per CSP per zone is extracted. Then all implementation details for each security mechanism are retrieved. Metadata for each mechanism includes a list of enforcement and monitoring components belonging to the mechanism, all configuration related parameters, and also some other information, for example, resource consumption, and dependencies and incompatibilities among components.

For each security mechanism, the Planning component has to identify the actual components needed to implement the set of SLOs. This solves the *first part of the planning problem*. According to the final set of components to be deployed, the Planning prepares a set of associated constraints. According to the constraints, the Planning component has to solve the *second part of the planning problem* (named as *allocation problem*), i.e., to determine

- the number of instances of each component to deploy,
- the number of resources needed to deploy all required components, and
- the distribution of components over acquired resources.

Note that the allocation problem has to be solved for each supported CSP separately, i.e., separately for each combination {CSP, zone, VM type}.

The described allocation problem (depicted in Figure 43) in practice becomes very challenging, not only because we need to consider infrastructure limitation of resources (e.g., highest load on each VM, maximum acquirable number of VMs), but also because we need to take into account constraints related to security mechanisms (e.g., dependencies and incompatibilities among components). And since each SLA can imply a different set of security mechanisms and a different set of available CSPs, both parts of the planning problem are

[10] For details see D2.2.2.

modelled dynamically at runtime.

All considered constraints can be expressed in a linear way. And since the goal is to find the minimum number of resources to acquire to implement an SLA, the allocation problem is modelled in the form of an Integer Linear Programming problem (ILP). In SPECS, in order to solve the allocation problem, the Planning component uses a Solver, a subcomponent able to process a set of constraints and identify the optimal acquisition and allocation solution.

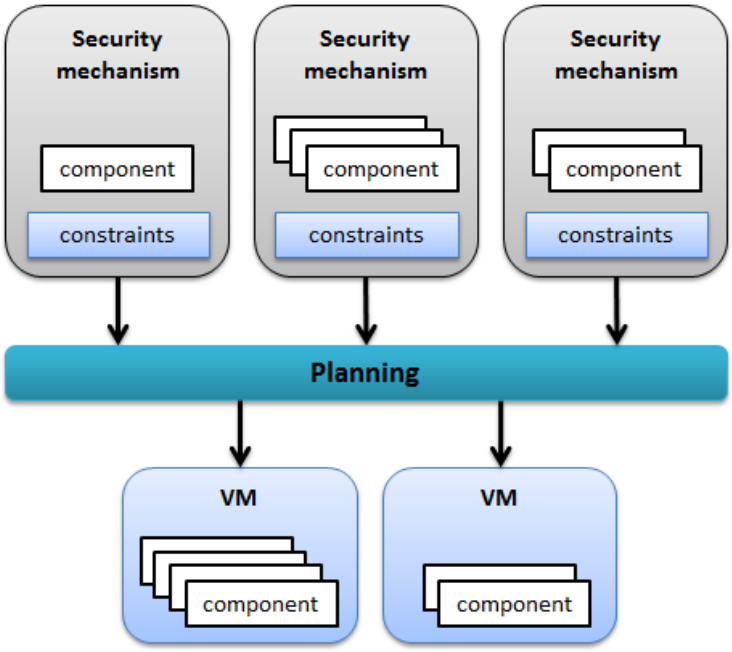

Figure 43. Input and output of the allocation problem

Note that a set of security requirements is not always implementable with supported CSPs. This results in Solver not always being able to find a solution of the allocation problem. In this case, the EU might be asked to adjust the set of desired security features.

As mentioned above, the allocation model has three main inputs, namely

- a set of components $C = c_1, c_2, \dots, c_N$ to deploy,
- a set of available VMs $V = v_1, v_2, \dots, v_M$, where M is the maximum acquirable number of VMs for the considered CSP, and
- a list of constraints related either to the set of components C or to some general allocation principles.

Each component $c_i \in C$ is assigned a computational load l_i, $i = 1,2, \dots, N$, based on resource consumption features (e.g., required RAM or CPU) included in the metadata as mentioned above. Similarly, all VMs are characterized by a maximum allowed load $VMLmax$. Note that the allocation problem is modelled and solved for one CSP at a time and as a provider we consider a combination of one CSP, one zone, and one VM type. Thus all VMs considered in one allocation problem are of the same type.

The Planning component (i.e., the Solver) has to determine the minimum number of VMs to acquire so that all components C are deployed on the set of acquired VMs according to the constraints.

As mentioned above, the allocation problem is subject to various constraints. We have to consider *general constraints* related to basic allocation rules, and *mechanism-specific constraints* arising from the definition and design of mechanisms. In the following we present both types of constraints. A mathematical formulation for each of them will be presented in D4.3.3.

General constraints. We identified the following constraints arising from basic allocation principles:

- **GC1**. Each component must be allocated to at least one VM.
- **GC2**. The maximum allowed load $VMLmax$ on each acquired VM cannot be exceeded.
- **GC3**. A component can be allocated to a VM only if it is acquired.

Mechanism-specific constraints. As anticipated, some constraints may apply to components due to the specific design of mechanisms. These constraints are prepared by mechanisms' developers who may not be aware of the other mechanisms, their components, and their characteristics. As a consequence, we consider either *intra-mechanism constraints* that express rules of deployment among components of the same mechanism, or *inter-mechanism constraints*, expressing general rules of deployment that only indirectly involve components belonging to other mechanisms. These mechanism-specific constraints are divided into classes by grouping together constraints of the same type:

- **SC1 (incompatibility)**. Such constraints express incompatibilities among components implying that involved components cannot be deployed on the same VM:
  - **SC1a (simple incompatibility)**. Component c_α cannot be allocated to a VM together with a set $\hat{C}$ of other components, where c_α and $\hat{C}$ belong to the same mechanism.
  - **SC1b (full incompatibility/exclusive use)**. Component c_α needs exclusive use of a machine. Note that, if full incompatibility is only related to components of the same mechanism, this is a particular case of SC1a and can be written in the same way as SC1a considering $\hat{C} = C - c_\alpha$.
- **SC2 (number of instances)**. Such constraints refer to the number of instances of the same component to deploy:
  - **SC2a (number of instances of a component)**. The number of instances of a component c_α must comply with an expression:
    - **SC2a-1**: The number of instances of a component c_α must be equal to n.
    - **SC2a-2**: The number of instances of a component c_α must be greater or equal to n.
    - **SC2a-3**: The number of instances of a component c_α must be less or equal to n.
    - **SC2a-4**: The number of instances of a component c_α must be in the range between n_1 and n_2.
  - **SC2b (number of instances of a set of components)**. The total amount of instances of components $c_i \in \hat{C}$ must comply with an expression:
    - **SC2b-1**: The total amount of instances of components $c_i \in \hat{C}$ must be equal to n.

        - **SC2b-2**: The total amount of instances of components $c_i \in \hat{C}$ must be greater or equal to n.
        - **SC2b-3**: The total amount of instances of components $c_i \in \hat{C}$ must be less or equal to n.
        - **SC2b-4**: The total amount of instances of components $c_i \in \hat{C}$ must be in a range between n_1 and n_2.
    - **SC2c (full deployment of a component)**. Component c_α must be deployed on each acquired VM. Clearly, in order to make the problem solvable, such a constraint must take into account possible incompatibilities with other components. Therefore the developer of each mechanism must also indicate the set of components $c_i \in \hat{C}$ that are incompatible with c_α.
- **SC3 (minimum number of VMs)**. Such constraint expresses the need for a minimum number $n \leq M$ of VMs to acquire, and may be explicitly introduced by an SLO.
- **SC4 (dependency)**. The number of instances of a component c_α depends on the number of instances of a component c_β according to an expression:
    - **SC4-1**: The number of instances of a component c_α must be equal to the number of instances of a component c_β.
    - **SC4-2**: The number of instances of a component c_α must be greater than or equal to the number of instances of a component c_β.
    - **SC4-3**: The number of instances of a component c_α must be less than or equal to the number of instances of a component c_β.
    - **SC4-4**: Fore very n instances of a component c_β there must be one instance of a component c_α.

Note that avoiding conflicts among constraints for one mechanism is the responsibility of the mechanism's developer. However, conflicts among constraints belonging to different mechanisms may still appear. For example, combination of constraints SC1b (full incompatibility) and SC2c (full deployment) assigned to two components c_α and c_β of two different mechanisms may result in no feasible solution. Thus the Planning component has to modify them to avoid this situation. If the Planning component recognizes that both constraints have been specified for the set of mechanisms to deploy, it automatically rewrites constraint SC2c by including component c_α in the set $\hat{C}$ of components that are incompatible with c_β.

Developers of security mechanisms have to be provided with a clear and simple syntax to define mechanism's constraints. Prepared constraints are included in mechanism's metadata along with other configuration details. Both constraints and metadata in general can be specified in several formats. In SPECS, the JSON format has been adopted due to the fact that it is language-independent and easy to read and write for humans, and easy to parse and generate for machines. The JSON schema for mechanism-specific constraints is presented in the listing below.

The JSON schema includes the ID of the constraint (*ctype*), two possible arguments (*arg1* and *arg2*) related to components (e.g., $arg1 = c_\alpha$ and $arg2 = \hat{C}$), an operator (*op*), and two possible integer values (*val1* and *val2*) related to constraints SC2a, SC2b, SC3, and SC4-4.

```
{
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "ctype": {"type": "string"},
      "arg1": {"type": "array", "items": {"type": "string"}},
      "arg2": {"type": "array", "items": {"type": "string"}},
      "op": {"type": "string"},
      "val1": {"type": "string"},
      "val2": {"type": "string"}
    },
    "required": ["ctype"]
  }
}
```

The following table summarizes all considered constraints and illustrates to developers how to prepare mechanism's metadata (i.e., how to fill the JSON schema) in an easy way.

ctype	arg1	arg2	op	val1	val2
SC1a	c_α	$c_i \in \hat{C}$	-	-	-
SC1b	c_α	-	-	-	-
SC2a-1	c_α	-	$=$	n	-
SC2a-2	c_α	-	$\geq$	n	-
SC2a-3	c_α	-	$\leq$	n	-
SC2a-4	c_α	-	in	n_1	n_2
SC2b-1	$c_i \in \hat{C}$	-	$=$	n	-
SC2b-2	$c_i \in \hat{C}$	-	$\geq$	n	-
SC2b-3	$c_i \in \hat{C}$	-	$\leq$	n	-
SC2b-4	$c_i \in \hat{C}$	-	in	n_1	n_2
SC2c	c_α	$c_i \in \hat{C}$	-	-	-
SC3	-	-	-	n	-
SC4-1	c_α	c_β	$=$	-	-
SC4-2	c_α	c_β	$\geq$	-	-
SC4-3	c_α	c_β	$>$	-	-
SC4-4	c_α	c_β	$*$	n	-

Table 61. Mechanism-specific constraints

When the Planning component solves the first part of the planning problem (i.e., identifies all components to be deployed), it creates a list of constraints and builds the allocation model. The list of constraints in the ILP format is passed to an automatic solver running inside the Planning component. The solution, if exists, is then translated from the ILP format into a supply chain format as will be presented later with an example.

Several open-source libraries exist that allow building and solving ILP problems. For the purpose of the following example we translated the model in the Mosel[11] language and

[11] http://www.fico.com/en/wp-content/secure_upload/Xpress-Mosel-Libraries.pdf

processed it with the Xpress Optimization Suite[12], a commercial product that provides an educational release, adopted to conduct the experiments.
In order to illustrate the introduced approach to solving the planning/allocation problem, we consider the case of provisioning a web container service provided by a CSP and enriched with two SPECS mechanisms, namely *WebPool* and *WebIDS*.

The WebPool (as presented in Section 4.2) offers resilience to attacks and failures by means of redundancy and diversity. The mechanism is offered through two security metrics, namely Level of redundancy (LoR; number of replicas of web servers) and Level of diversity (LoD; number of different web server types), and is implemented with a set of different web servers (*Nginx* and *Apache*) and a load balancer (*HAProxy*). The following are the constraints given by the developer of the mechanism:

- The balancer cannot be deployed on the same VM where any of web servers reside.
- Different web servers cannot be deployed together on the same VM.
- Exactly one balancer component has to be instantiated.
- The total amount of instances of web servers must be at least equal to $LoR + 1$.
- The number of different web server types must be equal to LoD.
- At least LoR+1 VMs must be acquired.

The WebIDS protects from unauthorized and potentially dangerous accesses by means of intrusion detection tools. The mechanism is offered through one security metric Report generation Frequency (RGF; the frequency of generation of the intrusion detection report in hours), and is implemented with an agent (*IDSagent*) that protects a resource and a server (*IDSserver*) which collects the data gathered by agents. The following are the constraints given by the mechanism's developer:

- Server needs an exclusive use of a VM.
- One agent must be allocated on every acquired VM except on the one with the server.
- There must be one server for every 10 instances of an agent.

Let us consider one CSP (offering one VM type with $VMLmax = 10$ in one zone where the maximum number of acquirable VMs is $M = 20$) and the following set of SLOs:

- $LoR = 3$
- $LoD = 2$
- $RGF = 2$

The Planning component first identifies the components needed to be deployed and the minimum number of VMs to acquire. According to the value of the LoR metric, we need at least 4 VMs. To assure the requested level of diversity, we have to deploy two different web servers. And to deploy WebIDS, we need all its components. Thus the final set of components to be deployed is {*HAProxy*, *Apache*, *Nginx*, *IDSserver*, *IDSagent*}.

Tables Table 62 and Table 63 present mechanism-specific constraints for the chosen set of components (separately for each mechanism).

[12] http://www.fico.com/en/products/fico-xpress-optimization-suite

ctype	arg1	arg2	op	val1	val2
SC1a	HAProxy	{Apache, Nginx}	-	-	-
SC1a	Apache	Nginx	-	-	-
SC2a-1	HAProxy	-	=	1	-
SC2b-2	{Apache, Nginx}	-	≥	3	-
SC3	-	-	-	4	-

Table 62. WebPool-specific constraints

ctype	arg1	arg2	op	val1	val2
SC1b	IDSserver	-	-	-	-
SC2c	IDSagent	IDSserver	-	-	-
SC4-4	IDSserver	IDSagent	*	10	-

Table 63. WebIDS-specific constraints

If we denote $HAProxy = c_1$, $Apache = c_2$, $Nginx = c_3$, $IDSserver = c_4$, and $IDSagent = c_5$, and we merge all constraints, we get a full set of mechanism-specific constraints for the set of EU's requirements. The full set is presented in Table 64 below.

ctype	arg1	arg2	op	val1	val2
SC1a	c_1	$\{c_2, c_3\}$	-	-	-
SC1a	c_2	c_3	-	-	-
SC1b	c_4	-	-	-	-
SC2a-1	1	-	=	1	-
SC2b-2	$\{c_2, c_3\}$	-	≥	3	-
SC2c	c_5	c_4	-	-	-
SC3	-	-	-	4	-
SC4-4	c_4	c_5	*	10	-

Table 64. Mechanism-specific constraints for the full set of components to be deployed

In the next step, the Planning component prepares the allocation model (in the mathematical form of an ILP problem) considering all general and all mechanism-specific constraints. The model is fed to the Solver which finds the following solution, translated into the supply chain format:

- Number of needed VMs: 5
- Allocation:
  - VM_1: {Apache, IDSagent}
  - VM_2: {Nginx, IDSagent}
  - VM_3: {IDSserver}
  - VM_4: {*HAProxy*}
  - VM_5: {Apache, IDSagent}

Appendix 2. Example of the implementation plan and the associated SLA with alerts

The following example presents the implementation plan generated for an SLA with the following attributes:

- Provider: aws-ec2, us-east-1, t1.micro.
- Service: Secure Web Container
- Capabilities:
  - Web Resilience, SLOs:
    - Level of redundancy = 2 with high importance
    - Level of diversity = 2 with medium importance
  - Software Vulnerability Assessment, SLOs:
    - Basic Scan Frequency = 24h with high importance
    - Vulnerability List Update Frequency = 24h with medium importance

For this set of requirements two security mechanisms need to be deployed, namely WebPool (see Section 4.2) and SVA (see Section 4.4) to cover Web Resilience and Software Vulnerability Assessment capabilities, respectively.
The chosen mechanisms are implemented with the following supply chain:

- Provider: {aws-ec2, us-east-1, t1.micro}.
- Number of needed VMs: 3
- Allocation:
  - VM_1: {HAProxy, SVA Dashboard}
  - VM_2: {Apache, SVA Enforcement, SVA Monitoring, OpenSCAP}
  - VM_3: {Nginx, SVA Enforcement, SVA Monitoring, OpenSCAP}

The following is the associated implementation plan built by the Planning component. Some IDs and configuration details (e.g., IP addresses and firewall rules) have been left out for readability sake. The complete JSON schema is described in D1.3.

```
{
  …,
  "iaas": {
    "provider": "aws_ec2",
    "zone": "us_east_1",
    "appliance": "ami_ff0e0696",
    "hardware": "c1_medium"
  },
  "pools": [
    {
      "pool_name": "webpool",
      "pool_seq_num": 1,
      …,
      "vms": [
        {
          "vm_seq_num": 1,
          "components": [
            {
              "component_name": "wp_haproxy",
              "cookbook": "webpool",
              "recipe": "wp_r5",
              …
```

```
            },
            {
              "component_name": "sva_dashboard",
              "cookbook": "sva",
              "recipe": "sva_r6",
              …
            }
          ]
        },
        {
          "vm_seq_num": 2,
          "components": [
            {
              "component_name": "wp_apache",
              "cookbook": "webpool",
              "recipe": "wp_r6",
              …
            },
            {
              "component_name": "sva_enforcement",
              "cookbook": "sva",
              "recipe": "sva_r4",
              ...
            },
            {
              "component_name": "sva_monitoring",
              "cookbook": "sva",
              "recipe": "sva_r5",
              …
            },
            {
              "component_name": "sva_openscap",
              "cookbook": "sva",
              "recipe": "sva_r1",
              …
            }
          ]
        },
        {
          "vm_seq_num": 3,
          "components": [
            {
              "component_name": "wp_nginx",
              "cookbook": "webpool",
              "recipe": "wp_r7",
              …
            },
            {
              "component_name": "sva_enforcement",
              "cookbook": "sva",
              "recipe": "sva_r4",
              …
            },
            {
              "component_name": "sva_monitoring",
              "cookbook": "sva",
              "recipe": "sva_r5",
              …
            },
            {
              "component_name": "sva_openscap",
              "cookbook": "sva",
```

```
              "recipe": "sva_r1",
              …
            }
          ]
        }
      ]
    }
  ],
  "slos": [
    {
      "slo_id": "wp_slo_1",
      "capability": "web_resilience",
      "metric_id": "level_of_redundancy_m1",
      "unit_type": "integer",
      "unit": "machines",
      "value": "2",
      "operator": "greater_equal",
      "importance_level": "high"
    },
    {
      "slo_id": "wp_slo_2",
      "capability": "web_resilience",
      "metric_id": "level_of_diversity_m2",
      "unit_type": "integer",
      "unit": "machines",
      "value": "2",
      "operator": "equal",
      "importance_level": "medium"
    },
    {
      "slo_id": "sva_slo_1",
      "capability": "software_vulnerability_assessment",
      "metric_id": "basic_scan_m13",
      "unit_type": "integer",
      "unit": "hours",
      "value": "24",
      "operator": "equal",
      "importance_level": "high"
    },
    {
      "slo_id": "sva_slo_2",
      "capability": "software_vulnerability_assessment",
      "metric_id": "list_update_m14",
      "unit_type": "integer",
      "unit": "hours",
      "value": "24",
      "operator": "equal",
      "importance_level": "medium"
    }
  ],
  "measurements": [
    {
      "msr_id": "number_of_servers_wp_msr1",
      "msr_description": "Number of responsive web servers.",
      "frequency": "1h",
      "metrics": [
        "level_of_redundancy_m1"
      ],
      "monitoring_event": {
        "event_id": "redundancy_too_low_wp_e1",
        "event_description": "The number of responsive web servers is too low.",
        "event_type": "violation",
```

```
        "condition": {
          "operator": "less",
          "threshold": "2"
        }
      }
    },
    {
      "msr_id": "diversity_level_wp_msr2",
      "msr_description": "Number of web server types active.",
      "frequency": "1h",
      "metrics": [
        "level_of_diversity_m2"
      ],
      "monitoring_event": {
        "event_id": "diversity_too_low_wp_e2",
        "event_description": "The number of web server types is too low.",
        "event_type": "violation",
        "condition": {
          "operator": "less",
          "threshold": "2"
        }
      }
    },
    {
      "msr_id": "report_basic_age_sva_msr1",
      "msr_description": "Age of the scanning report.",
      "frequency": "24h",
      "metrics": [
        "basic_scan_m13"
      ],
      "monitoring_event": {
        "event_id": "basic_report_too_old_sva_e1",
        "event_description": "Scanning report (basic scan) is too old.",
        "event_type": "violation",
        "condition": {
          "operator": "greater",
          "threshold": "24h"
        }
      }
    },
    {
      "msr_id": "list_age_sva_msr2",
      "msr_description": "Age of the vulnerability list.",
      "frequency": "24h",
      "metrics": [
        "list_update_m14"
      ],
      "monitoring_event": {
        "event_id": "list_too_old_sva_e2",
        "event_description": "Vulnerability list is too old.",
        "event_type": "violation",
        "condition": {
          "operator": "greater",
          "threshold": "24h"
        }
      }
    },
    {
      "msr_id": "repository_availability_sva_msr6",
      "msr_description": "Availability of the OVAL/NVD repositories.",
      "frequency": "6h",
      "metrics": [
```

```
        "list_update_m14"
      ],
      "monitoring_event": {
        "event_id": "repository_unavailable_sva_e6",
        "event_description":     "Repository     for     extracting     published
vulnerabilities is unavailable.",
        "event_type": "alert",
        "condition": {
          "operator": "equal",
          "threshold": "no"
        }
      }
    },
    {
      "msr_id": "list_availability_sva_msr7",
      "msr_description": "Availability of vulnerability list.",
      "frequency": "6h",
      "metrics": [
        "basic_scan_m13"
      ],
      "monitoring_event": {
        "event_id": "list_unavailable_sva_e7",
        "event_description": "Vulnerability list is unavailable.",
        "event_type": "alert",
        "condition": {
          "operator": "equal",
          "threshold": "no"
        }
      }
    },
    {
      "msr_id": "scanner_availability_sva_msr8",
      "msr_description": "Availability of the installed scanner.",
      "frequency": "6h",
      "metrics": [
        "basic_scan_m13"
      ],
      "monitoring_event": {
        "event_id": "scanner_unavailable_sva_e8",
        "event_description": "Installed scanner is unavailable.",
        "event_type": "alert",
        "condition": {
          "operator": "equal",
          "threshold": "no"
        }
      }
    }
  ]
}
```


As described in Section 3.2, each metric is associated to a set of measurements with which we can detect alerts and violations. These measurements are reported and mapped to metrics in the "measurement" section of the implementation plan. With these measurements a list of alert and violation thresholds is created in order to update the MoniPoli.

```
…
<specs:SLO name="specs_webpool_M1" metric-name="Level of Redundancy">
<specs:description>
The number of replicas of the Web Container that are set-up and kept active
throughout the service operation to ensure redundancy.
```

```
</specs:description>
<specs:expression unit="machines" op="eq">2</specs:expression>
<specs:importance_weight>HIGH</specs:importance_weight>
</specs:SLO>

<specs:SLO name="specs_webpool_M2" metric-name="Level of Diversity">
<specs:description>
The number of different software and/or hardware versions of the Web Container
service that are set-up and kept active throughout the service operation to
increase the protection from attacks and vulnerabilities exploits.
</specs:description>
<specs:expression unit="machines" op="eq">2</specs:expression>
<specs:importance_weight>MEDIUM</specs:importance_weight>
</specs:SLO>
…
-<specs:objectiveList>
      -
<specs:SLO name="specs_openvas_M13" metric-name="Scanning Frequency - Basic
Scan">
<specs:description>
The frequency of report generation (e.g., a value of "7*24h" requires that
reports are generated at least once per week).
</specs:description>
<specs:expression unit="hours" op="eq">24</specs:expression>
<specs:importance_weight>HIGH</specs:importance_weight>
</specs:SLO>
      -
<specs:SLO name="specs_openvas_M14" metric-name="List Update Frequency">
<specs:description>
The frequency of vulnerability list updates (e.g., a value of "24h" requires
that the list of known vulnerabilities is updated at least once per day).
</specs:description>
<specs:expression unit="hours" op="eq">24</specs:expression>
<specs:importance_weight>MEDIUM</specs:importance_weight>
</specs:SLO>

<specs:SLO name="report_basic_age_MSR1" metric-name="report_basic_age_SVA"
type="alert">
<specs:description>Age of the scanning report (basic scan)</specs:description>
<specs:expression unit="hours" op="eq">24</specs:expression>
<specs:importance_weight>HIGH</specs:importance_weight>
</specs:SLO>

<specs:SLO name="list_age_MSR2" metric-name="list_age_MSR2_SVA" type="alert">
<specs:description>Age of the vulnerability list.</specs:description>
<specs:expression unit="hours" op="eq">24</specs:expression>
<specs:importance_weight>MEDIUM</specs:importance_weight>
</specs:SLO>

<specs:SLO name="repository_availability_MSR6" metric-
name="repository_availability_MSR6_SVA" type="alert">
<specs:description>Availability of the OVAL/NVD repository.</specs:description>
<specs:expression unit="boolean" op="eq">yes</specs:expression>
<specs:importance_weight>HIGH</specs:importance_weight>
</specs:SLO>

<specs:SLO name="list_availability_MSR7" metric-
name="list_availability_MSR7_SVA" type="alert">
<specs:description>Availability of the vulnerability list</specs:description>
<specs:expression unit="boolean" op="eq">yes</specs:expression>
<specs:importance_weight>HIGH</specs:importance_weight>
</specs:SLO>
```

```
<specs:SLO name="scanners_availability_MSR8" metric-
name="scanners_availability_MSR8_SVA" type="alert">
<specs:description>Availability of the installed scanner.</specs:description>
<specs:expression unit="boolean" op="eq">yes</specs:expression>
HIGH
</specs:SLO>
…
```

Appendix 3. Example of the diagnosis process for an SVA alert

Let us take the SLA and implementation plan defined in Appendix 2, and let us consider the following notification of a monitoring event, sent from the Monitoring module to the Diagnosis component:

```
{
  "component": "specs_enforcement_sva",
  "object": "sva_monitoring_adapter",
  "labels": [
    "sla_id_9b0f908e",
    "security_mechanism_sva",
    "measurement_list_availability_sva_msr7"
  ],
  "type": "boolean",
  "data": "no",
  "timestamp": 1438945443
}
```

Note that the notification is in the SPECS event format introduced in D3.3.

The Diagnosis logs the start of the process, and retrieves the SLA with ID `9b0f908e` from the SLA Manager and queries Chef Server for the associated implementation plan. Based on the reported measurement (`list_availability_sva_msr7`), the affected SLOs are identified. From the "measurements" part of the implementation part it is clear that the considered measurement is related to metric `basic_scan_m13`. According to the "slos" part of the implementation plan this metric is mapped to one SLO with ID `sva_slo_1`.

During the classification of the notified event, the Diagnosis checks in the implementation plan the "monitoring_event" attribute of the reported measurement, presented below:

```
{
  "msr_id": "list_availability_sva_msr7",
  "msr_description": "Availability of vulnerability list.",
  "frequency": "6h",
  "metrics": [
    "basic_scan_m13"
  ],
  "monitoring_event": {
    "event_id": "list_unavailable_sva_e7",
    "event_description": "Vulnerability list is unavailable.",
    "event_type": "alert",
    "condition": {
      "operator": "equal",
      "threshold": "no"
    }
  }
}
```

By comparing the condition of the monitoring event and the data value in the event notification, the Diagnosis concludes that the notified event actually represents an alert or a violation (the false positive is discarded). Event type associated to the measurement reveals that the event classifies as an alert for the affected SLO.

Considering there is only one SLO affected by the notified event, the entire SLA is labelled as Alerted.

In the next steps the Diagnosis has to analyse the alert and determine its impact on the SLA. Since the event represents an alert and the affected SLO is of high importance, the risk level assigned to the affected SLO and thus to the entire SLA is 3 (see Table 7).

The state of the SLA is updated to *Alerted* and the SLA is put in the priority queue according to the assigned risk level. All gathered information about the alert is logged.

When all SLAs with higher risk/severity level have been pushed to the RDS, the Diagnosis takes the SLA considered in this example and verifies its state. This means that it queries Event Archiver for all data related to the SLA (with ID `9b0f908e`) and related to the measurement reported in the event notification (`list_availability_sva_msr7`) that refers to the time after the initial event occurred (with timestamp higher than `1438945443`).

If the Event Archiver has no such data or if the data that is retrieved reports the same condition as it was initially reported (`list_availability_sva_msr7 = no`), the SLA is pushed to the RDS.

If the Event Archiver returns data that implies that the alert has escalated to a violation or that the initial alert/violation still persists but the initial reported value for the same measurement changed, the Diagnosis discards the event and handles the next SLA in the priority queue, because this would actually mean that the Diagnosis has been in the meantime notified about another monitoring event related to this same measurement of this same SLA.

Appendix 4. Configuration details for the WebPool mechanism

The following are configuration details for the WebPool mechanism prepared by its developer. The complete JSON schema is described in D1.3.

```
{
  "security_mechanism_id": "86dacd86-3ce5-11e5-a151-feff819cdc9f",
  "security_mechanism_name": "webpool",
  "sm_description": "This security mechanisms aims at offering web servers and
the capabilities of surviving to security incidents involving a web server, by
implementing proper strategies aimed at preserving business continuity, achieved
through redundancy and/or diversity.",
  "security_capabilities": [
    "web_resilience"
  ],
  "enforceable_metrics": [
    "level_of_redundancy_m1",
    "level_of_diversity_m2"
  ],
  "monitorable_metrics": [
    "level_of_redundancy_m1",
    "level_of_diversity_m2"
  ],
  "measurements": [
    {
      "msr_id": "number_of_servers_wp_msr1",
      "msr_description": "Number of responsive web servers.",
      "frequency": "1h",
      "metrics": [
        "level_of_redundancy_m1"
      ],
      "monitoring_event": {
        "event_id": "number_of_servers_too_low_wp_e1",
        "event_description": "The number of responsive web servers is too low.",
        "event_type": "violation",
        "condition": {
          "operator": "less",
          "threshold": "level_of_redundancy_m1"
        }
      }
    },
    {
      "msr_id": "diversity_level_wp_msr2",
      "msr_description": "Number of web server types active.",
      "frequency": "1h",
      "metrics": [
        "level_of_diversity_m2"
      ],
      "monitoring_event": {
        "event_id": "diversity_level_too_low_wp_e2",
        "event_description": "The number of web server types is too low.",
        "event_type": "violation",
        "condition": {
          "operator": "less",
          "threshold": "level_of_diversity_m2"
        }
      }
    }
  ],
  "metadata": {
    "components": [
```

```
    {
      "component_name": "wp_haproxy",
      "component_type": "balancer",
      "recipe": "webpool",
      "cookbook": "wp_r5",
      "implementation_step": 1,
      "pool_seq_num": 1,
      "pool_id": "webpool",
      "vm_requirement": {
        "hardware": "t1_micro",
        "usage": "100",
        "acquire_public_ip": "true",
        "private_ips_count": 1,
        "firewall": {
          "incoming": {
            "source_ips": [
              "0.0.0.0/0"
            ],
            "source_nodes": [
              "string"
            ],
            "interface": "public",
            "proto": [
              "TCP"
            ],
            "port_list": [
              "22",
              "80",
              "443"
            ]
          },
          "outcoming": {
            "destination_ips": [
              "0.0.0.0/0"
            ],
            "destination_nodes": [],
            "interface": "public,private:1",
            "proto": [
              "TCP"
            ],
            "port_list": [
              "*"
            ]
          }
        }
      }
    },
    {
      "component_name": "wp_apache",
      "component_type": "web_server",
      "recipe": "webpool",
      "cookbook": "wp_r6",
      "implementation_step": 1,
      "pool_seq_num": 1,
      "pool_id": "webpool",
      "vm_requirement": {
        "hardware": "t1_micro",
        "usage": "50",
        "acquire_public_ip": "false",
        "private_ips_count": 1,
        "firewall": {
          "incoming": {
```

```
            "source_ips": [],
            "source_nodes": [
              "wp_haproxy"
            ],
            "interface": "private:1",
            "proto": [
              "TCP"
            ],
            "port_list": [
              "22",
              "80",
              "443"
            ]
          },
          "outcoming": {
            "destination_ips": [],
            "destination_nodes": [
              "wp_haproxy"
            ],
            "interface": "private:1",
            "proto": [
              "TCP"
            ],
            "port_list": [
              "*"
            ]
          }
        }
      }
    },
    {
      "component_name": "wp_nginx",
      "component_type": "web_server",
      "recipe": "webpool",
      "cookbook": "wp_r7",
      "implementation_step": 1,
      "pool_seq_num": 1,
      "pool_id": "webpool",
      "vm_requirement": {
        "hardware": "t1_micro",
        "usage": "50",
        "acquire_public_ip": "false",
        "private_ips_count": 1,
        "firewall": {
          "incoming": {
            "source_ips": [],
            "source_nodes": [
              "wp_haproxy"
            ],
            "interface": "private:1",
            "proto": [
              "TCP"
            ],
            "port_list": [
              "22",
              "80",
              "443"
            ]
          },
          "outcoming": {
            "destination_ips": [],
            "destination_nodes": [
```

```
                "wp_haproxy"
              ],
              "interface": "private:1",
              "proto": [
                "TCP"
              ],
              "port_list": [
                "*"
              ]
            }
          }
        }
      }
    ],
    "constraints": [
      {
        "ctype": "SC1a",
        "arg1": [
          "wp_haproxy"
        ],
        "arg2": [
          "wp_apache",
          "wp_nginx"
        ]
      },
      {
        "ctype": "SC1a",
        "arg1": [
          "wp_apache"
        ],
        "arg2": [
          "wp_nginx"
        ]
      },
      {
        "ctype": "SC2a_1",
        "arg1": [
          "wp_haproxy"
        ],
        "op": "=",
        "n1": "1"
      },
      {
        "ctype": "SC2b_2",
        "arg1": [
          "wp_apache",
          "wp_nginx"
        ],
        "op": ">=",
        "n1": "level_of_redundancy_m1"
      },
      {
        "ctype": "SC3",
        "n1": "level_of_redundancy_m1+1"
      }
    ]
  },
  "remediation": {
    "remediation_actions": [
      {
        "name": "wp_a1",
        "action_description": "Check if the number of responsive web servers is
```

```
greater than or equal to level_of_redundancy_m1 value.",
        "recipes": [
          "wp_r1"
        ]
      },
      {
        "name": "wp_a2",
        "action_description": "Restart unresponsive web server and check if
number of responsive web servers is greater than or equal to
level_of_redundancy_m1 value.",
        "recipes": [
          "wp_r2",
          "wp_r1"
        ]
      },
      {
        "name": "wp_a3",
        "action_description": "Replace unresponsive web server and check if
number of responsive web servers is greater than or equal to
level_of_redundancy_m1 value.",
        "recipes": [
          "wp_r3",
          "wp_r1"
        ]
      },
      {
        "name": "wp_a4",
        "action_description": "Check if the number of responsive web server
types is greater than or equal to level_of_diversity_m2 value.",
        "recipes": [
          "wp_r4"
        ]
      },
      {
        "name": "wp_a5",
        "action_description": "Restart unresponsive web server and check if the
number of responsive web server types is greater than or equal to
level_of_diversity_m2 value.",
        "recipes": [
          "wp_r2",
          "wp_r4"
        ]
      },
      {
        "name": "wp_a6",
        "action_description": "Replace unresponsive web server and check if the
number of responsive web server types is greater than or equal to
level_of_diversity_m2 value.",
        "recipes": [
          "wp_r3",
          "wp_r4"
        ]
      }
    ],
    "remediation_flow": [
      {
        "name": "redundancy_too_low_wp_e1",
        "action_id": "wp_a1",
        "yes_action": "observe",
        "no_action": {
          "action_id": "wp_a2",
          "yes_action": "observe",
```

```
          "no_action": {
            "action_id": "wp_a3",
            "yes_action": "observe",
            "no_action": "notify"
          }
        }
      },
      {
        "name": "diversity_too_low_wp_e2",
        "action_id": "wp_a4",
        "yes_action": "observe",
        "no_action": {
          "action_id": "wp_a5",
          "yes_action": "observe",
          "no_action": {
            "action_id": "wp_a6",
            "yes_action": "observe",
            "no_action": "notify"
          }
        }
      }
    ]
  },
  "chef_recipes": [
    {
      "name": "wp_r1",
      "recipe_description": "Take measurement wp-msr1 and label the event as
remediation-event.",
      "associated_metrics": [],
      "associated_measurements": [],
      "dependent_components": [
        "wp_haproxy"
      ]
    },
    {
      "name": "wp_r2",
      "recipe_description": "Restart unresponsive servers.",
      "associated_metrics": [],
      "associated_measurements": [],
      "dependent_components": [
        "wp_haproxy",
        "wp_apache",
        "wp_nginx"
      ]
    },
    {
      "name": "wp_r3",
      "recipe_description": "Replace unresponsive servers.",
      "associated_metrics": [
        "specs:level_of_redundancy:M1",
        "specs:level_of_diversity:M2"
      ],
      "associated_measurements": [],
      "dependent_components": [
        "wp_haproxy",
        "wp_apache",
        "wp_nginx"
      ]
    },
    {
      "name": "wp_r4",
      "recipe_description": "Take measurement wp-msr2 and label the event as
```

```
remediation-event.",
      "associated_metrics": [],
      "associated_measurements": [],
      "dependent_components": [
        "wp_haproxy"
      ]
    },
    {
      "name": "wp_r5",
      "recipe_description": "Install HAProxy.",
      "associated_metrics": [],
      "associated_measurements": [
        "specs:level_of_redundancy:M1",
        "specs:level_of_diversity:M2"
      ],
      "dependent_components": [
        "wp_apache",
        "wp_nginx"
      ]
    },
    {
      "name": "wp_r6",
      "recipe_description": "Install Apache.",
      "associated_metrics": [],
      "associated_measurements": [],
      "dependent_components": [
        "wp_haproxy"
      ]
    },
    {
      "name": "wp_r7",
      "recipe_description": "Install Nginx.",
      "associated_metrics": [],
      "associated_measurements": [],
      "dependent_components": [
        "wp_haproxy"
      ]
    }
  ]
}
```

Appendix 5. List of security metrics

In the following table a list of all metrics offered and implemented by security mechanisms demonstrated in this document is provided.

The entire list of security metrics (the ones presented in this document, the ones that will be presented in the final iteration of this document, and the ones focused on ngDC and developed in WP5) together with all the details is available online in the SPECS metrics catalogue[13].

Name	Description
Level of redundancy (LOR)	This metric sets the minimum number (with respect to EU's requirements and technological constraints) of web server engines which are set-up and kept active throughout the service operation to increase the protection from attacks and vulnerabilities exploits. For example, *level_of_redundancy = 3*, ensures that there are at least three web servers running.
Level of diversity (LOD)	This metric sets the number of different web server types available on target VMs. For example, for *level of diversity = 2*, SPECS ensures that there are at least two different types of web servers deployed and available.
Write-serializability (WS)	This metric ensures the EU that any WS violations to the stored data will be detected in a defined period of time (detection periods are less than 2**epoch*). In case of WS violations, the EU will be notified, and the system will be restored to the state of the last finished *epoch*.
Read-Freshness (RF)	This metric ensures the EU that any RF violations to the stored data will be detected in a defined period of time (detection periods are less than 2**epoch*). In case of RF violations, the EU will be notified, and the system will be restored to the state of the last finished *epoch*.
Client-side encryption certification (EC)	This metric ensures that the E2EE Client component available at the provided address is certified and thus grants the security of the encryption.
List update frequency (LUF)	This metric sets the frequency of updates of the list of disclosed vulnerabilities. For example, for *list_update_frequency=12*, SPECS ensures that the list of published vulnerabilities will be updated and presented at least once every 12 hours.
Scanning frequency – basic scan (BSF)	This metric sets the frequency of a basic software vulnerability scan. For example, for *scanning_frequency=24*, SPECS ensures that software vulnerability scans will be performed at least once every day.
Scanning frequency – extended scan (ESF)	This metric sets the frequency of an extended software vulnerability scan. For example, for *scanning_frequency=48*, SPECS ensures that software vulnerability scans will be performed at least once every two days. Scans are performed with two scanners and both scanning reports are presented.
Up report frequency	This metric sets the frequency of checks for updates and upgrades

[13] http://apps.specs-project.eu/specs-app-security_metric_catalogue/

(URF)	of vulnerable installed libraries. SPECS first updates vulnerability list, performs the vulnerability scan of the system, and then checks for available updates and upgrades of libraries on which vulnerabilities have been detected). For example, for *up_report_frequency=24*, SPECS ensures that checks for updates and upgrades are performed at least once every day.
Penetration testing activated (PTA)	This metric activates the penetration testing activity. The metric can be chosen together with metrics related to vulnerability scans. If chosen, scanner with penetration testing functionality is deployed.
TLS cryptographic strength (TCS)	This metric sets the cryptographic strength to be used by the TLS Terminator. TLS Terminator Configurator will choose the appropriate cryptographic ciphers that meet the negotiated level, and configure TLS Terminator accordingly.
Forward Secrecy (FS)	This metric ensures that the encrypted data sent through a session of the TLS secure channel cannot be decrypted even if the cryptographic data, used to generate the cryptographic credentials for that session, are compromised.
HTTP strict transport security (HSTS)	This metric is a feature of HTTP transport layer that declares the web content available only over a secure HTTP connection.
HTTP to HTTPS redirects (HHSR)	This metric is a feature of HTTP delivery service that forces clients to use only secure HTTP protocol.
Secure cookies (SC)	This metric is a feature of HTTP protocol to force the clients to download session cookies, delivered by the HTTP services, only through a secured HTTP communication
Certificate pinning (CP)	This metric is a feature of HTTP protocol allowing the verification of the SSL certificates between the client and the HTTP service where the hash of the public certificate is pinned into the HTTP response.

Table 65. WP4 security metrics