Secure Provisioning of Cloud Services
based on SLA Management

SPECS Project - Deliverable 4.5.2

Testing and validation - 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.:	D4.5.2
Deliverable title:	Testing and validation – Intermediary
Deliverable nature:	Report
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), Stefano Marrone (CeRICT), Roberto Nardone (CeRICT), Heng Zhang (TUDA), Alain Pannetrat (CSA), Silviu Panica (IeAT), Giancarlo Capone (CeRICT)
Reviewers:	Dana Petcu (IeAT), Andrew Byrne (EMC)
Task contributing to the deliverable:	T4.5
Total number of pages:	92

Executive summary

Validation encompasses a variety of activities along the software development life cycle with the common objective of ensuring that the developed product is secure, of good quality, and that it complies with all requirements. This deliverable presents the final validation methodology to be used for both, the Enforcement module and the other elements of the entire framework.

Verification and testing methodologies defined in this task will be adopted on the project level. Results for the Enforcement module are covered by deliverables produced in this task, whereas outcomes of validation activities for other work packages will be presented in dedicated prototype deliverables.

Based on what was explored in year 1 of the SPECS project in D4.5.1, the current document presents the finalized verification and testing methodologies and tools to be used in the project. Common development rules are briefly summarized and code quality analysis approaches are specified. Some initial functional tests are reported, code quality analysis is demonstrated with an example, and approach to the security review is presented.

Table of contents

Index of figures

Index of tables

1. Introduction

Task T4.5 focuses on validation and testing of the Enforcement module to provide a loop between design and implementation activities in WP4, and to provide a basis for similar activities in other work packages.

In SPECS, verification activities were planned in an agile way as much as possible. The plan was to define user stories, build validation scenarios, elicit requirements, design the framework, and then verify the design in early stages to allow the implementation and integration to be more efficient. After the design of the architecture and the definition of all processes, the set of elicited requirements was re-evaluated (some requirements were discarded, added, merged, or remapped to new components/modules), initial validation scenarios were amended and enriched, and the architecture itself was revamped. The adopted process is depicted in Figure 1.

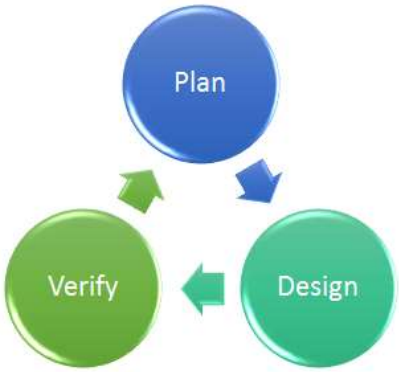

Figure 1. Verification activities

A similar approach was adopted in the next stage, namely the testing phase (see Figure 2). Implemented pieces have been tested and test results were evaluated to improve the code.

Figure 2. Testing activities

There have also been some overlaps between the verification and testing phases. Some testing outcomes resulted in adjusting the architecture and some implementation and integration issues caused re-evaluation of requirements and validation scenarios.

In this deliverable, the focus is twofold. On one hand, the goal is to finalize the validation and testing approach, which will be adopted in other work packages. The second focus is on

verifying the Enforcement module and presenting testing activities conducted under the WP4 umbrella during the last year.

The first release of this document (D4.5.1) outlined possible validation approaches, presented possible methodologies to adopt and tools to use in the testing phase, and discussed the initial verification of the Enforcement module. The current document presents the final validation methodology and reports about verification of the refined Enforcement module. The proposed validation methodology will be adopted on the project level and results will be reported in dedicated prototype deliverables (for WP1, WP2, and WP3). This deliverable also presents the final testing approach. Collaborative development rules are summarized, the final set of testing methodologies and technologies are discussed, and the security review approach is defined. Initial results of code quality analysis and functional testing of Enforcement module are presented.

The final iteration of this deliverable (D4.5.3, due at M30) will report about final validation results and present non-functional testing activities and outcomes of the security review conducted for the Enforcement module. All tests performed at the integration and system level will be reported in deliverables of task T1.5, and all tests conducted at the component level for other modules will be presented and discussed in dedicated prototype deliverables.

The document is structured as follows. In Section 2, relationships between this document and other deliverables of the project are discussed. Section 3 briefly summarizes the requirements and the design of the Enforcement module. Validation methodology, testing techniques and tools, and security review technique, adopted in the project, are described in Sections 4 and 6, respectively. Intermediary results of validation and testing activities (including code quality assessment and functional tests) are reported in Sections 5 and 7, respectively. The document concludes with a brief summary of current results and future plans.

2. Relationship with other deliverables

Testing and validation is an important part of the development cycle. With clear validation and testing methodologies we can determine whether the developed framework complies with all elicited requirements and whether it supports functionalities according to the design.

For the purpose of testing and validating the Enforcement module, the following inputs are required:

- Refined user stories and validation scenarios discussed in D5.1.2.
- Requirements elicited in D4.1.2.
- Architecture of the module presented in D4.2.2 and associated APIs defined in D1.3.
- The module's prototypes demonstrated in D4.3.2.
- The initial testing and validation methodology defined in D4.5.1.
- Feedback from the Platform's prototype demonstrated in D1.6.1.

The methodologies defined in task T4.5 are not aimed only at the Enforcement module. In order to assure a smooth integration process, techniques are also adopted by developers of components and modules in other tasks and other WPs. Negotiation and Monitoring modules will report testing and validation results in dedicated prototype deliverables (D2.3.2 and D3.4.2, respectively). Test and validation results for the Platform will be reported in D1.6.2. Credential Service and Security Tokens will be tested and validated under the dedicated task T4.4 (presented in D4.4.2), and Auditing component will be tested and validated under task T1.4 (see D1.4.1 and D1.4.2). The final results for the (entire) Enforcement module will be presented in D4.5.3.

Results presented in this deliverable will be the main input for the final prototypes of the Enforcement components, which will be demonstrated in D4.3.3. All integration and system related testing activities will be reported in D1.5.1 and D1.5.2.

All above mentioned relationships among SPECS' deliverables are depicted in Figure 3.

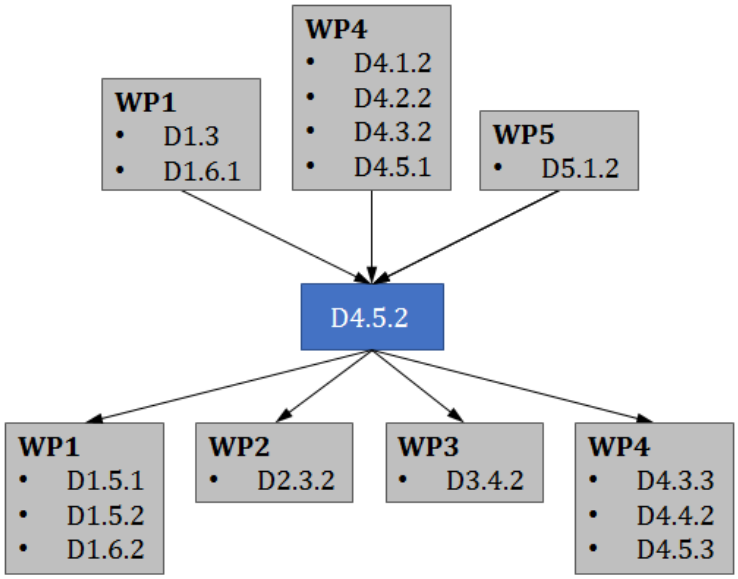

Figure 3. Relationship with other deliverables

3. The Enforcement module

Core components of the Enforcement module orchestrate the SLA implementation and SLA remediation phases. The initial set of requirements for Enforcement's functionalities is reported in D4.1.2, and the initial design is defined in D4.2.2. Due to the feedback received from implementation activities, the processes driven by the Enforcement module have been slightly refined.

Similarly, a few changes occurred in the design of security mechanisms. Developers' feedback and outcomes of exploitation activities (additional requests from stakeholders) resulted in a couple of new requirements associated with security mechanisms, and further refinements of the architecture reported in D4.3.2.

In the remainder of this section a list of new requirements are discussed (Section 3.1), the refined, final design is briefly presented for both, main Enforcement components (Section 3.2) and the set of security mechanisms (Section 3.3). For more details about main Enforcement components, security mechanisms, and the enforcement process itself see D4.3.2.

3.1. Enforcement requirements

The original set of Enforcement requirements is reported and discussed in D4.1.2. Here only a set of newly elicited requirements are declared. The final list of all Enforcement requirements is provided at the end of this document in Appendix 1.

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 1. New requirements for SPECS security mechanisms

New requirements outline the need for a new security mechanism which has been designed and developed in T4.3. Details are provided in D4.3.2.

Old and new requirements have been thoroughly analysed in terms of coverage by design and coverage of validation scenarios. Results are reported in Section 5.

3.2. Main Enforcement components and the enforcement process

The architecture of the Enforcement module remains the same as it was initially designed in D4.2.2. What has evolved are the details of the enforcement process. Refinements are reported at the end of this subsection.

There are four main Enforcement components, namely Planning, Implementation, Diagnosis, and Remediation Decision System (RDS), which oversee SLA implementation and SLA remediation phase. The Auditing component is being developed in T4.3, but its functionalities are offered to other modules as well, thus it has been placed into the Vertical Layer[1].

[1] For the final SPECS architecture see D1.1.3.

After the End-user (EU) negotiates[2] security requirements, the Planning component has to build one or more valid supply chains that implement the set of EU's requirements. When the Negotiation module builds SLAs according to the build supply chains, and the EU signs one of them, the Planning prepares an implementation plan. The Implementation component (with support of the Broker mechanism) acquires resources and deploys and configures components as specified in the implementation plan. After a successful execution of the implementation plan the Planning updates the Monitoring Policy[3], and the SLA enters the monitoring phase.

If/when the Diagnosis component receives a notification of a possible alert or a violation, it performs a classification (i.e., determines whether the event represents an alert, a violation, or a false positive), analysis (i.e., determines the effect on an SLA and determines the risk/severity level of the event with respect to the affected SLA), and prioritization (i.e., puts the SLA in the priority queue according to the assigned risk/severity level) of the notified monitoring event. Each alerted/violated SLA is then pushed to the RDS to determine the root cause, find the best proactive/reactive actions, and builds a remediation plan later executed by the Implementation component.

EUs have an option to renegotiate signed SLAs and to terminate them before the expiration dates. In some cases renegotiation or termination is required after an unsuccessful remediation of an SLA violation. In this case the Planning component prepares a reaction plan according to the old and new SLA/supply chain to reconfigure target services. The reaction plan is later executed by the Implementation component.

All details about the Auditing component are available in deliverables D1.4.1 and D1.4.2.

As reported in D4.3.2 and outlined again here in the tables below, enforcement process has been amended to support all diagnosis and remediation activities, but mostly to support planning and implementation activities after remediation, renegotiation, and termination.

Main enforcement component	Planning
Year 1	**Year 2**
• Validates supply chains. • Builds implementation plans.	• Builds valid supply chains. • Builds implementation plans and associated supply chains. • Builds reaction plans to reconfigure target services after SLA renegotiation and SLA termination. • Updates Monitoring Policy.
Comments	• In year 2, generation and validation of supply chains is merged into one step. • Building reaction plans has been added after refinement of renegotiation. • Updating Monitoring Policy functionality has been moved from the Implementation component to the Planning component.

Table 2. Overview of the Planning component

[2] For details about the negotiation process see D2.2.2.
[3] For details about the monitoring process and the Monitoring Policy see D3.3.

Main enforcement component	Implementation
Year 1	**Year 2**
• Executes implementation plans. • Updates Monitoring Policy.	• Executes implementation plans. • Executes remediation plans to reconfigure target services during SLA remediation. • Executes reaction plans to reconfigure target services after SLA renegotiation or SLA termination.
Comments	• Updating Monitoring policy functionality has been assigned to the Planning component. • Executions of remediation and reaction plans have been added due to refinements of remediation and renegotiation processes.

Table 3. Overview of the Implementation component

Main enforcement component	Diagnosis
Year 1	**Year 2**
• Classifies, analyses, and prioritizes monitoring events. • Determines root causes of monitoring events.	• Classifies, analyses, and prioritizes monitoring events.
Comments	• Identifying root causes of monitoring events functionality has been moved to the Remediation Decision System component during the refinement of the remediation process.

Table 4. Overview of the Diagnosis component

Main enforcement component	Remediation Decision System (RDS)
Year 1	**Year 2**
• Searches for redressing techniques.	• Determines root causes of monitoring events. • Searches for redressing techniques. • Builds remediation plans.
Comment	• Identifying root causes of monitoring events functionality has been moved from the Diagnosis to the Remediation Decision System component during the refinement of the remediation process. • Building remediation plans functionality has been added during refinements of the SLA remediation phase.

Table 5. Overview of the RDS component

3.3. Security mechanisms

Prototypes demonstrated in D4.3.2 and D4.4.2 comprise the following security mechanisms:

- Broker (Secure Provisioning),
- WebPool (Secure Web Server),
- DBB,
- E2EE,
- SVA Security,
- TLS Security,
- Credential Service, and
- Security Tokens.

The remaining two, namely AAA and DoS, will be demonstrated in D4.3.3 at the end of the project.

Note that all details related to Credential Service and Security Tokens are provided in the dedicated deliverable D4.4.2.

In the remainder of this section we report design details for each security mechanism (except the ones yet to be demonstrated at M30 or the ones discussed in T4.4). If the architecture of a mechanism has been refined due to the feedback received from developers, the refinements are reported as well.

Security mechanism	Broker	
Component	**Description**	**Comment**
Broker Configuration Manager	Manages the Broker configuration. Communicates with the SLA Platform, in order to make the component available and synchronized with the Platform.	Integrated into Resource Broker component.
Resource Broker	Acquires and configures IAAS resources from a Cloud Service Provider (i.e. configures the firewall and the public/private keys in order to enable the access to the machine via ssh, etc.).	Integrates Chef Server.

Table 6. Overview of the Broker mechanism

Security mechanism	WebPool	
Component	**Description**	**Comment**
Web Container Pool Manager	Forwards all incoming requests to one of the Pool Agents according to the scheduling policy defined in its property file. By default it uses round robin algorithm.	
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.	

Table 7. Overview of the WebPool mechanism

Security mechanism	DBB	
Component	**Description**	**Comment**
DBB Main Server	Handles put and get requests and orchestrates all associated operations (writes/reads data to/from DBB Main DB, performs backups).	Initially considered as E2EE Encryption Configurator.
DBB Main DB	Stores original data.	
DBB Backup Server	Responsible for backups and restorations.	
DBB Backup DB	Manages backup data.	
DBB Client	Provides a web interface for uploading and downloading files.	Initially considered as part of E2EE Client component.
DBB Auditor	Performs auditing (monitors write-serializability and read-freshness).	Introduced in year 2.
DBB Monitoring adapter	Monitors availability of DBB servers and DBs.	

Table 8. Overview of the DBB mechanism

Security mechanism	E2EE	
Component	**Description**	**Comment**
E2EE Client	Provides a web interface for uploading and downloading files. Sent and received data is encrypted/decrypted on the client's side.	Previously named Client-side Encryptor.
E2EE Monitoring Adapter	Monitors certification status of the E2EE Client.	Introduced in year 2.

Table 9. Overview of the E2EE mechanism

Security mechanism	SVA	
Component	**Description**	**Comment**
SVA Enforcement	Manages vulnerability list, orchestrates vulnerability scans, checks for updates and upgrades of vulnerable libraries.	Initially integrated scanners. In year 2, scanners are considered as separate components. Automatic updating and patching have been discarded.
SVA Monitoring	Monitors SVA security metrics.	
SVA Dashboard	Presents vulnerability list and SVA reports.	
OpenSCAP Scanner	Performs vulnerability scans.	Initially integrated in SVA Enforcement.
OpenVAS Scanner	Performs vulnerability scans. Supports penetration testing.	Initially considered as separate monitoring mechanism. To be integrated in year 3.
Nikto Scanner	Performs vulnerability scans. Supports penetration testing.	Added in year 2. To be integrated in year 3.

Table 10. Overview of the SVA mechanism

Security mechanism	TLS	
Component	**Description**	**Comment**
TLS Reasoner	Decides what security configuration to be used for TLS Terminator. The security configuration consists of a group of cryptographic ciphers that meet the cryptographic strength level negotiated.	Initially integrated in TLS security mechanism.
TLS Terminator Configurator	Creates configuration templates used by TLS Terminator to offer the security metrics negotiated.	Initially integrated in TLS security mechanism.
TLS Terminator Controller	Manages the TLS Terminator behaviour allowing administrative tasks (start, stop, and status query) over the TLS Terminator service.	Initially integrated in TLS security mechanism.
TLS Prober	Monitors TLS security metrics.	Initially integrated in TLS security mechanism.

Table 11. Overview of the TLS mechanism

4. Validation methodology

This section describes the overall process that is related to the definition of the chosen testing levels and techniques. The objective is to relate the module-level testing activity concerned in this deliverable to the user-oriented system-level testing approach adopted in T5.1 and described in D5.1.1 and D5.1.2. This section also specifies which kind of testing (e.g., functional, non-functional, security, etc.) will be performed on Enforcement components as well as the components of other SPECS modules. Figure 4 depicts the followed process.

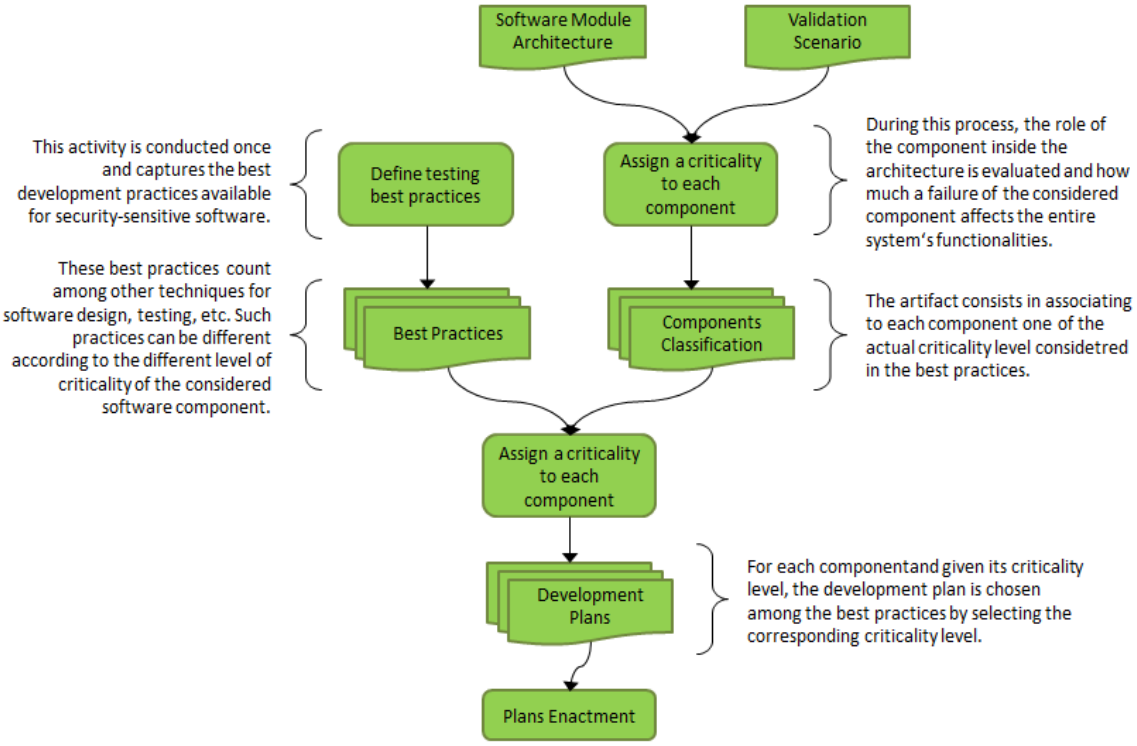

Figure 4. Component validation process

The first step is the definition of development and testing best practices that define, according to a criticality level, some techniques and methodological tools that are recommended to ensure a proper quality level for software components. Since there are different components in our software system, each of one with a different role and a different set of responsibilities, it is not realistic to use the same techniques for all of them. The components are classified according to three different levels of criticality (low, medium, and high).

The best practices table (Table 12) for which the structure has been introduced in D4.5.1, has on the rows the different component life-cycle phases:

- **Specification**: definition of a set of functional and/or security requirements.
- **Design**: definition of a software architecture, software components, and their interactions and interfaces.
- **Coding**: implementation of defined components.
- **Verification**: definition, execution and analysis of test cases aimed at verifying the correct implementation of specified requirements.
- **Operation & Maintenance**: continuous process related to the set up runtime environment, collection and analysis of execution traces.

The definition of best practices is based on this above described classification, while the activity of the followed process is in charge of assigning a criticality level to each component. The inputs for this assignment phase are the Validation Scenarios (VSs) introduced in D5.1.1 and refined in D5.1.2, and the SPECS software architecture summarized in D1.1.3. By using these inputs, the relationships and the dependency among the components are analysed in particular in the interaction scenarios that are most sensitive from a user-perspective (the VSs). This assignment is not made on the basis of a formalised approach, such as the HAZOP, or other risk assessment approaches. Instead, the criticality is mainly assigned on the base of the experience of the SPECS designers and architects. The reasoning behind this choice is because the objective of this task is not the certification of the developed products but rather a demonstration of the overall quality of the produced software.

Once each component has a criticality level assigned to it, by using the best practices defined, it is possible to extract the specific set of techniques to apply for each component and apply it during the component life-cycle.

It is important to underline that some development and verification activities will be performed for all the components independently from their criticality level: requirements engineering and traceability, UML modelling, version control, etc. In particular, functional tests will be performed for all the components while, for non-functional properties and testing activities, we proceed in the following way:

- For each Enforcement core component we determine the criticality of the test types.
- For each test type (security, interoperability, dependability, robustness) we define specific testing techniques and goals to apply to the component under test.

Table 12 lists the best practices detected for the components as stated in Figure 4. Rows represent the phases of the development life-cycle, while columns represent possible criticality levels. Each cell contains the techniques applicable for all the components characterized by a certain level of criticality (column) at a certain development phase (row).

	High	**Medium**	**Low**
Specification	• Traceability • Peer-review inspection	• Traceability	• Traceability
Design	• Interfaces and behavioural UML modelling	• Interfaces and behavioural UML modelling	• Interfaces and behavioural UML modelling
Coding	• Secure programming • Coding standard	• Coding standard	• Structured Programming
Verification	• Code quality analysis • Black box functional testing • Branch coverage white box testing • Security testing • Security review • Interoperability testing • Dependability and robustness testing	• Black box functional testing • Unit testing • Statement coverage white box testing	• Black box functional testing • Unit testing
Operation & Maintenance	• Use of versioning software systems • Use of an open problems log	• Use of versioning software systems	• Use of versioning software systems

Table 12. Methods/Criticality assignment example

As an example, if we consider a component X and consider it as a medium critical component, during its validation, the techniques listed in the related cell should be applied (i.e., black box functional testing, unit testing, and statement coverage white box testing).

The following list briefly covers the single individual technique used:

- *Traceability*: The requirements are traced on design components and tests. This enables, during the verification phase, the definition of which component/requirement represents some errors and which is the part of the system that is affected by a requirements' change.
- *Peer-review inspection*: The requirements are checked in order to detect inconsistencies and/or incompleteness.
- *Interfaces and behavioural UML modelling*: By using UML, interfaces between the components as well as the dynamic behaviour of components and their interactions are clear and documented.
- *Secure Programming*: Using a secure programming coding standard it is possible to limit the occurrence of security bugs such as buffer overrun, etc.
- *Coding Standard*: By using a coding standard, the developer knows if some construct can or cannot be used. This has the effect of producing more stable and clear software (an example is constituted by inhibiting the usage of function pointers in C/C++).
- *Structured Programming*: Avoiding non-structured programming approaches is in keeping with clear programming practices and quality improvement.
- *Black box testing*: The software is tested by simply defining input/output couples. The software is not inspected but the verification that input and output match is conducted. This testing practice is in general associated with the verification of functional requirements.
- *Unit Testing*: Has the scope of testing single components by means of the definition of stubs and driver software in general under the guidance of widespread testing frameworks such as *xUnit* [32].
- *Statement coverage white-box testing*: White-box testing looks inside the structure of a component; covering the statements means to cover the greatest part of executable statements at least once.
- *Branch coverage white-box testing*: White-box testing looks inside the structure of a component; covering the branches means to cover the greatest part of branches at least once. It is important to underline that branch coverage is a stronger condition than statement coverage.
- *Code Quality Analysis*: By means of quantitative metrics, some quality indicators can be computed indicating the overall quality of the software (lines of code per function, software complexity, specific object-oriented metrics, etc.).
- *Security Testing*: Security testing aims at finding software vulnerabilities, which can be exploited by an attacker. By finding them as soon as possible, they can be fixed and the overall vulnerability level of the software can be reduced.
- *Security Review*: While the aim is the same as the Security Testing approach, the Security Review is mainly conducted by defining and following security checklists by static human based inspection of the software.
- *Interoperability Testing*: The objective of this phase is to check the adherence of produced software to international standards and to ensure that the produced software package can properly work in different operative contexts (in the SPECS case,

we have to check that the SPECS modules, platform, and applications work with different CSPs).

- *Dependability and Robustness Testing*: The objective here is to test the reliability of the software also in the event of faulty conditions (network error, erroneous/malicious inputs, etc.).
- *Use of versioning software systems*: Software configuration management is a critical process in taming the complexity of large software projects. Through a comprehensive configuration management process, product releases can be defined in an accurate way by choosing the proper version of all the software artefacts.
- *Use of an open problems log*: By keeping, feeding and periodically reviewing an open problem log, the software development team is able to keep software issues under control and is able to plan improved releases according to a shared priority.

It is beyond the scope of this section to present the results of these approaches, instead only presenting the approaches, providing motivation for the approaches, and defining the proper development and testing framework. Concrete techniques and supporting tools used for the single purposes (coverage, software metrics, etc.) will be presented in Section 6. For functional testing activities refer to Section 7.2. For results on integration refer to D1.5.1/D1.5.2, and for activities related to non-functional testing refer to D4.5.3 (M30). Moreover, all the life-cycle steps that have been highlighted, are touched in different tasks and demonstrations of the achieved results is provided in different deliverables:

- Specification was performed in Y1 (see D4.1.2).
- Design was conducted in Y1 (see D4.2.2) and updated in Y2 (see D4.3.2).
- The coding phase started in Y1 (see D4.3.1) and continued in Y2 (refer to D4.3.2).
- Verification partially started in Y1 (D4.5.1), continued with component testing in Y2 and will result with complete results in Y3 (refer to D4.5.3).

5. Validation of the Enforcement module

As anticipated in the introduction and discussed in D4.3.2, in the year 2 of the SPECS project the Enforcement module has been improved to support not only all the steps of the refined remediation phase but also to support implementation activities after renegotiation and termination. Similarly, rethinking enforcement and monitoring of security metrics resulted in a few amendments of the architecture of security mechanisms (for details see D4.3.2 and D4.4.2). Moreover, exploitation activities produced a few new requirements implying the need for a new security mechanism (DBB mechanism; see D4.3.2). This resulted in some changes in coverage of requirements by the main Enforcement components and security mechanisms.

A year of research, development, and integration also resulted in a new version of validation scenarios reported in D5.1.2. Steps for each scenario have been revised, refined, and further details have been added.

In order to verify the Enforcement module (to validate the intermediary design/prototype), in the next subsections the new coverage matrices are presented and discussed, and summaries of main changes are reported.

Note that all scenarios and requirements directly associated to Credential Service and Security Tokens are verified in deliverable D4.4.2, and all scenarios and requirements strictly related to the Auditing component are discussed in deliverable D1.4.1. Verification of all scenarios and requirements associated to AAA and DoS mechanisms will be presented in D4.5.3.

5.1. Coverage of validation scenarios

This section provides verification results for the Enforcement module with respect to validation scenarios introduced in D5.1.1 and refined in D5.1.2. In order to evaluate the intermediary Enforcement prototypes demonstrated in D4.3.2 and D4.4.2, each validation scenario is presented in terms of coverage by main Enforcement components and security mechanisms and by Enforcement requirements (see the tables below).

Coverage analysis of validation scenarios is an important step in the validation process since validation scenarios serve as basis for defining integration scenarios (that will be presented in deliverables of task T1.5).

Note that in the initial discussion related to coverage of scenarios by components the focus was on the entire SPECS framework, whereas in this document the focus is on Enforcement module only. In addition, the coverage analysis presented in this section is focused only on components and mechanisms described in D4.3.2 and validated in this deliverable (i.e., core Enforcement components, Broker, WebPool, TLS, SVA, DBB, E2EE).

Validation of other modules is performed in dedicated tasks and results are reported in dedicated deliverables. The final validation of the entire framework will be performed at the end of the project and results will be reported in the final iteration of this deliverable.

Scenario ID	*SST-01 Secure_Storage_Selection*
Scenario description	The End-user aims at acquiring a secure storage service from a cloud provider, which fulfils specific security-related requirements. To achieve this service, the End-user negotiates the desired features with SPECS. In this validation scenario, the desired features are entirely implemented by an external CSP, while SPECS only provides the End-user with the functionalities to search, rank and select a service which is compliant to her/his requirements. Moreover, in this scenario, SPECS supports the End-user in signing an SLA with the selected provider.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains)	• *ENF_PLAN_R1-R4* • *ENF_PLAN_R10-R12*

Table 13. Coverage of *Secure_Storage_Selection* scenario

Scenario ID	*SST-02 Secure_Storage_Brokering_with_Client_Crypto*
Scenario description	The End-user aims at acquiring a secure storage service from a remote cloud provider, which fulfils specific security-related requirements. Specifically, the End-user needs the two capabilities of Database-as-a-Service and End-2-End Encryption in order to detect and prove security-related violations and to locally encrypt her/his data. To achieve this service, the End-user negotiates the desired features with SPECS and signs an SLA including all service terms and guarantees. SPECS acquires the Database-as-a-Service on behalf of the End-user (registered on SPECS) and provides her/him with end-2-end encryption security mechanism. In this scenario, SPECS also provides monitoring functionalities.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components) • Broker mechanism (acquires resources) • DBB mechanism (offers secure storage with backup) • E2EE mechanism (provides client-side encryption)	• *ENF_PLAN_R1-R7* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_BROKER_R1-R5* • *ENF_CRYPTO_R1-R4* • *ENF_DBB_R1-R2*

Table 14. Coverage of *Secure_Storage_Brokering_with_Client_Crypto* scenario

Scenario ID	*SST-03 Secure_Storage_with_Defined_CSP*
Scenario description	The End-user aims at storing encrypted data on a known remote cloud provider which offers a Database-as-a-service. The End-user asks SPECS for End-2-End Encryption capability, needed to locally encrypt her/his data. To achieve this service, the End-user also gives SPECS her/his credentials on the chosen provider; SPECS manages these credentials and uses them to log into the chosen provider and store User's data. In this scenario, SPECS also provides monitoring functionalities.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components) • Broker mechanism (acquires resources) • DBB mechanism (offers secure storage with backup) • E2EE mechanism (provides client-side encryption)	• *ENF_PLAN_R1-R7* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_BROKER_R1-R5* • *ENF_CRYPTO_R1-R4* • *ENF_DBB_R1-R2*

Table 15. Coverage of *Secure_Storage_with_Defined_CSP* scenario

Scenario ID	*SST-04 Secure_Storage_Brokering_with_Client_Crypto_Alert*
Scenario description	The End-user aims at acquiring a secure storage service from a remote cloud provider, which fulfils specific security-related requirements. Specifically, the End-user needs the two capabilities of Database-as-a-Service and End-2-End Encryption in order to detect and prove security-related violations and to locally encrypt her/his data. To achieve this service, the End-user negotiates the desired features with SPECS and signs an SLA including all service terms and guarantees. SPECS acquires the Database-as-a-Service on behalf of the End-user (registered on SPECS) and provides her/him with end-2-end encryption security mechanism. In this scenario, SPECS also provides monitoring functionalities. In this scenario, an alert is raised since the Encryption Server component is detected to be down and, since no data are sent from the End-user during the down time, no violation occurs.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components, executes remediation plan) • Diagnosis component (analyses and classifies the monitoring event) • RDS component (builds remediation plan) • Broker mechanism (acquires resources) • DBB mechanism (offers secure storage with backup) • E2EE mechanism (provides client-side encryption)	• *ENF_PLAN_R1-R8* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_DIAG_R1- R18* • *ENF_REM_R2-R9* • *ENF_BROKER_R1-R5* • *ENF_CRYPTO_R1-R4* • *ENF_DBB_R1-R2* • *SLANEG_R31*

Table 16. Coverage of *Secure_Storage_Brokering_with_Client_Crypto_Alert* scenario

Scenario ID	*SST-05 Secure_Storage_Brokering_with_Client_Crypto_Violation*
Scenario description	The End-user aims at acquiring a secure storage service from a remote cloud provider, which fulfils specific security-related requirements. Specifically, the End-user needs the two capabilities of Database-as-a-Service and End-2-End Encryption in order to detect and prove security-related violations and to locally encrypt her/his data. To achieve this service, the End-user negotiates the desired features with SPECS and signs an SLA including all service terms and guarantees. SPECS acquires the Database-as-a-Service on behalf of the End-user (registered on SPECS) and provides her/him with end-2-end encryption security mechanism. In this scenario, SPECS also provides monitoring functionalities. In this scenario, a violation is detected since the Encryption Server component is detected to be down.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components, executes remediation plan) • Diagnosis component (analyses and classifies the monitoring event) • RDS component (builds remediation plan) • Broker mechanism (acquires resources) • DBB mechanism (offers secure storage with backup) • E2EE mechanism (provides client-side encryption)	• *ENF_PLAN_R1-R8* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_DIAG_R1- R18* • *ENF_REM_R2-R9* • *ENF_BROKER_R1-R5* • *ENF_CRYPTO_R1-R4* • *ENF_DBB_R1-R2* • *SLANEG_R31*

Table 17. Coverage of *Secure_Storage_Brokering_with_Client_Crypto_Violation* scenario

Scenario ID	*SWC-01 Secure_Web_Container_Selection*
Scenario description	The End-user aims at acquiring a web container from an Infrastructure-as-a-Service CSP, represented by a VM hosting the Web Server, which fulfils specific security requirements. To achieve this service, the End-User negotiates the desired features with SPECS. In this validation scenario, the desired features are entirely implemented by an Infrastructure-as-a-Service CSP. SPECS only returns to the End-user the reference to such provider.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains)	• *ENF_PLAN_R1-R4* • *ENF_PLAN_R10-R12*

Table 18. Coverage of *Secure_Web_Container_Selection* scenario

Scenario ID	*SWC-02 Secure_Web_Container_Brokering*
Scenario description	The End-user aims at acquiring a web container from an Infrastructure-as-a-Service CSP, represented by a VM hosting the Web Server, which fulfils specific security-related requirements. To achieve this service, the End-User negotiates the desired security features with SPECS. In this validation scenario, the desired features are entirely implemented by an Infrastructure-as-a-Service CSP. SPECS acquires the resources on behalf of the End-user (registered on SPECS) and sets up some monitoring functionalities in order to monitor the SLA achievement.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components) • Broker mechanism (acquires resources) • WebPool mechanism (offers a secure web container)	• *ENF_PLAN_R1-R7* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_BROKER_R1-R5* • *ENF_POOL_R1-R5*

Table 19. Coverage of *Secure_Web_Container_Brokering* scenario

Scenario ID	*SWC-03 Secure_Web_Container_TLS_Enhanced*
Scenario description	The End-user aims at acquiring a web container from an Infrastructure-as-a-Service CSP, represented by a VM hosting the Web Server, which fulfils specific security-related requirements. In particular, the End-user requires the adoption of Transport Layer Security (TLS) protocol to protect the Web Server communications, DoS detection and mitigation mechanisms. To achieve this service, the End-user negotiates the desired features with SPECS. In this validation scenario, the VM (without TLS) is provided by an Infrastructure-as-a-Service CSP while the TLS protocol and the DoS detection and mitigation mechanisms are provided by SPECS. SPECS acquires the resources on behalf of the End-user (registered on SPECS), adds the TLS protocol, and sets up some monitoring functionalities in order to monitor the TLS communication. In this scenario, an alert regarding a DoS attack is detected, and SPECS reacts by activating proper mitigation strategies. The scenario ends without any other alert.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components, executes remediation plan) • Diagnosis component (analyses and classifies the monitoring	• *ENF_PLAN_R1-R8* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_DIAG_R1- R18*

event) • RDS component (builds remediation plan) • Broker mechanism (acquires resources) • WebPool mechanism (offers a secure web container) • TLS mechanism (provides TLS protocol)	• *ENF_REM_R2-R9* • *ENF_BROKER_R1-R5* • *ENF_POOL_R1-R5* • *ENF_TLS_R1-R5* • *SLANEG_R31*

Table 20. Coverage of *Secure_Web_Container_Enhanced* scenario

Scenario ID	*SWC-04 Secure_Web_Container_SVA_Enhanced_Alert*
Scenario description	The End-user aims at acquiring a web container from an Infrastructure-as-a-Service CSP, represented by a VM hosting the Web Server, which fulfils specific security-related requirements. In particular, the End-user requires the adoption of Software Vulnerability Assessment (SVA) tools to protect the Web Server environment. To achieve this service, the End-user negotiates the desired features with SPECS. In this validation scenario, the VM (without SVA) is provided by an Infrastructure-as-a-Service CSP while the SVA agent is installed by SPECS. SPECS acquires the resources on behalf of the End-user (registered on SPECS), adds the SVA agents, and sets up some monitoring functionalities. This scenario includes the raising of an alert due to a deviation of some metrics; SPECS reacts by updating the software (redressing). The scenario ends without any other alerts.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components, executes remediation plan) • Diagnosis component (analyses and classifies the monitoring event) • RDS component (builds remediation plan) • Broker mechanism (acquires resources) • WebPool mechanism (offers a secure web container) • SVA mechanism (provides SVA security services)	• *ENF_PLAN_R1-R8* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_DIAG_R1- R18* • *ENF_REM_R2-R9* • *ENF_BROKER_R1-R5* • *ENF_POOL_R1-R5* • *ENF_SVA_R1-R4* • *SLANEG_R31*

Table 21. Coverage of *Secure_Web_Container_SVA_Enhanced_Alert* scenario

Scenario ID	*SWC-05 Secure_Web_Container_TLS_SVA_Enhanced_Violation*
Scenario description	The End-user aims at acquiring a web container from an Infrastructure-as-a-Service CSP, represented by a VM hosting the Web Server, which fulfils specific security-related requirements. In particular, the End-user requires the adoption of Software Vulnerability Assessment (SVA) tools to protect the Web Server environment. To achieve this service, the End-user negotiates the desired features with the SPECS. In this validation scenario, the VM (without SVA) is provided by an Infrastructure-as-a-Service CSP while the SVA agents are installed by SPECS. SPECS acquires the resources on behalf of the End-user (registered on SPECS), adds the SVA agents, and sets up some monitoring functionalities in order to detect the presence of exposed vulnerabilities. This scenario includes the raising of an alert regarding a vulnerability threat which corresponds to a violation of the agreed SLA. SPECS reacts by renegotiating the SLA; the End-user asks for the adoption of Transport Layer Security (TLS) protocol to protect the Web Server communications. The renegotiated SLA is hence signed and properly monitored by SPECS.
Involved Enforcement components/mechanisms	**Related requirements**

• Planning component (builds valid supply chains, builds implementation plan, builds reaction plan, updates MoniPoli) • Implementation component (configures resources and SPECS components, executes remediation plan, executes reaction plan) • Diagnosis component (analyses and classifies the monitoring event) • RDS component (builds remediation plan) • Broker mechanism (acquires resources) • WebPool mechanism (offers a secure web container) • SVA mechanism (provides SVA security services) • TLS mechanism (offers TLS protocol)	• *ENF_PLAN_R1-R12* • *ENF_IMPL_R1-R10* • *ENF_DIAG_R1- R18* • *ENF_REM_R1-R9* • *ENF_BROKER_R1-R5* • *ENF_POOL_R1-R5* • *ENF_TLS_R1-R5* • *ENF_SVA_R1-R4* • *SLANEG_R30-R31*

Table 22. Coverage of *Secure_Web_Container_TLS_SVA_Enhanced_Violation* scenario

Scenario ID	*SWC-06 Secure_Web_Container_TLS_Multitenancy*
Scenario description	Two End-users aim at acquiring different web containers Infrastructure-as-a-Service CSPs, represented by VMs hosting the Web Servers, which fulfil different security requirements. In addition, both End-users require the adoption of Transport Layer Security (TLS) protocol to protect the communications of Web Servers. To achieve this service, the first End-user negotiates the desired features with SPECS. The VM (without TLS) is provided by an Infrastructure-as-a-Service CSP while the TLS protocol is added by SPECS setting up proper resources (e.g., reverse proxy). The second End-user negotiates the desired features with the SPECS framework. A different VM (without TLS) is provided by an Infrastructure-as-a-Service CSP (either the same or a different one) while the TLS protocol is added by SPECS reusing, for scalability purposes, the same resources adopted for the first End-user. This validation scenario considers the multi-tenancy in the usage of shared resources between End-users.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components) • Broker mechanism (acquires resources) • WebPool mechanism (offers a secure web container) • TLS mechanism (offers TLS protocol)	• *ENF_PLAN_R1-R7* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_BROKER_R1-R5* • *ENF_POOL_R1-R5* • *ENF_TLS_R1-R5*

Table 23. Coverage of *Secure_Web_Container_TLS_Multitenancy* scenario

Scenario ID	*SWC-07 Secure_Web_Container_Web_Pool_Replication_Enhanced_Alert*
Scenario description	The End-user aims at acquiring a set of web containers from an Infrastructure-as-a-Service CSP, each of them represented by a VM hosting the Web Server, which fulfil specific security-related requirements. In particular, the End-user requires a specific level of redundancy and session persistence among web container replicas. To achieve this service, the End-user negotiates the desired features with SPECS. In this validation scenario, the VMs are provided by an Infrastructure-as-a-Service CSP while session persistence among replicas is implemented by the SPECS web pool mechanism. SPECS acquires the resources on behalf of the End-user (registered on SPECS), adds the web pool components, and sets up proper resources to handle HTTP request through proxying functionality in order to

	forward the requests to one of the available the web container. In this scenario, the proxy functionality is added, by SPECS, on a dedicated VM. This scenario includes the rising of an alert regarding a vulnerability threat on a specific web container; SPECS reacts by updating the implemented forwarding policy (redressing) and removes the affected web container from the pool of available web containers. The scenario ends without any other alerts.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components, executes remediation plan) • Diagnosis component (analyses and classifies the monitoring event) • RDS component (builds remediation plan) • Broker mechanism (acquires resources) • WebPool mechanism (offers a secure web container)	• *ENF_PLAN_R1-R8* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_DIAG_R1- R18* • *ENF_REM_R2-R9* • *ENF_BROKER_R1-R5* • *ENF_POOL_R1-R5* • *SLANEG_R31*

Table 24. Coverage of *Secure_Web_Container_Web_Pool_Replication_Enhanced_Alert* scenario

Scenario ID	*SWC-08 Secure_Web_Container_Web_pool_Replication_Enhanced_Violation*
Scenario description	The End-user aims at acquiring a precise number of web containers from an Infrastructure-as-a-Service CSP, each of them represented by a VM hosting the Web Server, which fulfil specific security requirements. In particular, the End-user requires a specific level of redundancy and session persistence among web container replicas. To achieve this service, the End-user negotiates the desired features with SPECS. In this validation scenario, the VMs are provided by an Infrastructure-as-a-Service CSP while the session persistence among replicas is implemented through the SPECS web pool mechanism by SPECS. SPECS acquires the resources on behalf of the End-user (registered on SPECS), adds the web pool components, and sets up proper resources to handle HTTP request through proxying functionality in order to forward the requests to one of the available the web container. In this scenario, the proxy functionality is added, by SPECS, on a dedicated VM. This scenario includes the rising of an alert regarding a vulnerability threat on a specific web container; SPECS reacts by removing the affected web container from the pool of available web containers. The signed SLA is hence violated since the number of available VMs is not sufficient to fulfil the SLA.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains, builds implementation plan, updates MoniPoli) • Implementation component (configures resources and SPECS components, executes remediation plan) • Diagnosis component (analyses and classifies the monitoring event) • RDS component (builds remediation plan) • Broker mechanism (acquires resources) • WebPool mechanism (offers a secure web container)	• *ENF_PLAN_R1-R8* • *ENF_PLAN_R10-R12* • *ENF_IMPL_R1-R8* • *ENF_IMPL_R10* • *ENF_DIAG_R1- R18* • *ENF_REM_R2-R9* • *ENF_BROKER_R1-R5* • *ENF_POOL_R1-R5* • *SLANEG_R31*

Table 25. Coverage of *Secure_Web_Container_Web_Pool_Replication_Enhanced_Violation* scenario

Scenario ID	*NGDC-01 Data_Center_Bursting_for_Storage_Resources*
Scenario description	A CSP hosting its own ngDC acting within a CSC role aims at using the SPECS framework to perform Cloud bursting in order to extend its Secure Storage as a Service (SStaaS) capabilities during a period of increased storage demand beyond

	its own ngDC storage capabilities by its CSCs and/or End-users. The CPS considers its storage as first class storage due the capability to tune all the security parameters. The CSP will allocate the first class storage to the End-User that don't need high-security capability. Otherwise it will allocate storage to an external provider throw SPECS. All that process is transparent to the End-user. Note while the CSP acquiring external CSP storage resources is typically considered an End-user, it is not in the context of a SPECS defined End-user. That is, the CSP intends to resell its acquired external storage resources and so is considered a CSC (in the context of SPECS). For ease of exposition 'customer' is used as a syntactic sugar to refer to either a CSC or End-user of the CSP hosting the ngDC.
Involved Enforcement components/mechanisms	**Related requirements**
• Planning component (builds valid supply chains)	• *ENF_PLAN_R1-R4* • *ENF_PLAN_R10-R12*

Table 26. Coverage of *Data_Center_Bursting_for_Storage_Resources* scenario

In summary, all reported validation scenarios involve at least one Enforcement component and at least one security mechanism. The traceability matrix outlining correlation between validation scenarios and Enforcement design is provided in Table 27 below.

Enforcement component / mechanism	SST-01	SST-02	SST-03	SST-04	SST-05	SWC-01	SWC-02	SWC-03	SWC-04	SWV-05	SWC-06	SWC-07	SWC-08	NGDC-01
Planning	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Implementation		X	X	X	X		X	X	X	X	X	X	X	
Diagnosis				X	X			X	X	X		X	X	
RDS				X	X			X	X	X		X	X	
Broker		X	X	X	X		X	X	X	X	X	X	X	
WebPool							X	X	X	X	X	X	X	
DBB		X	X	X	X									
E2EE		X	X	X	X									
SVA									X	X				
TLS								X		X	X			

Table 27. Coverage of validation scenarios by Enforcement components/mechanisms

All requirements for the Enforcement module identified on the basis of user stories and validation scenarios introduced in D5.1.1 cover all old/refined validation scenarios. Details are discussed in the following subsection.

5.2. Coverage of requirements

In the first iteration of this deliverable, the "main" Enforcement requirements (the ones associated with core Enforcement components) have been analysed in terms of coverage with respect to validation scenarios. After refinement of validation scenarios, improvement of the enforcement process, receipt of initial feedback from implementation activities, and elicitation of some new requirements, a more detailed coverage matrix for the "main" Enforcement requirements and even for ones associated to security mechanisms is provided in the Table 28 below.

Since mapping between requirements and validation scenarios is already provided in Section 5.1, Table 28 focuses on explaining HOW each Enforcement requirement is covered by prototypes developed so far.

REQ_ID	Requirement	Comment
ENF_PLAN_R1	***Get SLA to enforce***	The Planning component parses the SLA to build supply chains and to prepare implementation plans.
ENF_PLAN_R2	***Define security mechanisms related to SLOs***	The Planning component considers the set of SLOs in the SLA (while building supply chains) and determines which kind of security mechanisms are to be applied.
ENF_PLAN_R3	***Get security components***	The Planning component (while building supply chains) retrieves all security mechanisms able to implement negotiated SLA.
ENF_PLAN_R4	***Select best security component***	The Planning component (while building supply chains) selects the best available security mechanisms able to implement negotiated SLA.
ENF_PLAN_R5	***Activate implementation***	The Planning component triggers execution of an implementation plan by invoking the Implementation component.
ENF_PLAN_R6	***Log component activation and deactivation***	The Planning component logs its activation and deactivation.
ENF_PLAN_R7	***Build an implementation plan***	The Planning component prepares an implementation plan based on the signed SLA and associated supply chain. Building implementation plan includes deducing alert thresholds.
ENF_PLAN_R8	***Build a reaction plan***	The RDS component is able to build a reaction plan after an alert or a violation. *The Planning component has to be able to build a reaction plan after renegotiation. This should be covered by the final prototype demonstrated at M30.*
ENF_PLAN_R9	***Build a migration plan***	The RDS component is able to build a migration plan after an alert or a violation. *The Planning component has to be able to build a migration plan after renegotiation. This should be covered by the final prototype demonstrated at M30.*
ENF_PLAN_R10	***Get monitoring systems***	The Planning component (while building supply chains) retrieves all monitoring systems able to monitor negotiated SLA.
ENF_PLAN_R11	***Select best monitoring systems***	The Planning component (while building supply chains) selects the best available monitoring systems able to monitor negotiated SLA.
ENF_PLAN_R12	***Validate an SLA***	The Planning component builds only valid supply chains. Consequently (in the SLA negotiation process) only valid SLAs are built.
ENF_IMPL_R1	***Implement Plan***	The Implementation component executes implementation plan by orchestrating the acquisition of the needed resources, their configuration, and the activation of involved services.
ENF_IMPL_R2	***Acquire resources***	The Implementation component (the Broker)

		acquires all resources needed to realize an implementation plan.
ENF_IMPL_R3	***Deploy and configure***	The Implementation component deploys and configures all acquired resources according to the implementation plan.
ENF_IMPL_R4	***Start services***	The Implementation component activates all services deployed and configured on top of acquired resources.
ENF_IMPL_R5	***Trigger monitoring agent activation or deactivation***	The Implementation component activates/deactivates all monitoring agents deployed and configured on top of acquired resources.
ENF_IMPL_R6	***Log service activation***	The Implementation component logs a successful activation of each security service related to the implemented SLA.
ENF_IMPL_R7	***Update SLA state***	The Implementation component updates the state of an SLA after its successful implementation.
ENF_IMPL_R8	***Log component activation or deactivation***	The Implementation component logs its activation or deactivation.
ENF_IMPL_R9	***Implement reaction plan***	*The Implementation component has to be able to implement a reaction plan built after renegotiation. This should be covered by the final prototype demonstrated at M30.*
ENF_IMPL_R10	***Update monitoring policy***	The Planning component updates the monitoring policy (with violation and alert thresholds) according to a signed SLA.
ENF_DIAG_R1	***Get monitoring event notification***	The Diagnosis component receives notifications of monitoring events from the Monitoring module.
ENF_DIAG_R2	***Get monitoring event information***	The Diagnosis component is able to retrieve all information related to a monitoring event by accessing the Auditing component.
ENF_DIAG_R3	***Identify SLOs affected by a monitoring event***	The Diagnosis component analyses notified monitoring events and identify the SLOs at risk or violated.
ENF_DIAG_R4	***Update SLA state***	The Diagnosis component updates the state of an SLA (to *Alerted* or *Violated*) depending on the classification of the notified monitoring event.
ENF_DIAG_R5	***Get SLAs affected by a monitoring event***	The Diagnosis component identifies and retrieves all SLAs affected by a notified monitoring event.
ENF_DIAG_R6	***Activate reaction***	The Diagnosis component activates the RDS component to react to an alert or a violation.
ENF_DIAG_R7	***Express SLA violation in terms of KPI***	*This requirement could possibly be covered by the final prototype demonstrated at M30.*
ENF_DIAG_R8	***Query metric***	The Diagnosis component is able to query the metric data stored inside the Event Archiver (Monitoring module) when evaluating the status of a notified monitoring event.
ENF_DIAG_R9	***Log component activation or deactivation***	The Diagnosis component logs its activation and deactivation.
ENF_DIAG_R10	***Determine effect on***	For each SLA affected by a monitoring event, the

	an SLA	Diagnosis component determines the effect the monitoring event has on the SLA (i.e., is it alerted or violated).
ENF_DIAG_R11	***Log SLA impact***	The Diagnosis component logs all event related information.
ENF_DIAG_R12	***Classify event***	The Diagnosis component classifies all notified monitoring events.
ENF_DIAG_R13	***Identify root cause***	The Diagnosis component performs a root cause analysis of each monitoring event.
ENF_DIAG_R14	***Log root cause***	The Diagnosis component logs all event related information.
ENF_DIAG_R15	***Analyse monitoring event***	The Diagnosis component analyses each notified monitoring event.
ENF_DIAG_R16	***Prioritize events***	The Diagnosis component prioritizes monitoring events (actually, SLAs affected by notified events) according to their risk/severity levels.
ENF_DIAG_R17	***Log priority queue***	The Diagnosis component logs all event related information.
ENF_DIAG_R18	***Log queue change***	The Diagnosis component must be able to compare the current metric/SLO data with the alert/violation thresholds specified for an alerted/violated SLA to verify if the severity of the alert/violation has changed.
ENF_REM_R1	***Trigger renegotiation***	The RDS component triggers renegotiation when available remediation activities are unable to resolve SLA violations.
ENF_REM_R2	***Log component activation or deactivation***	The RDS logs its activation and deactivation.
ENF_REM_R3	***Get SLA state***	The RDS checks the state of an SLA in order to identify proper remediation actions.
ENF_REM_R4	***Update SLA state***	The RDS component updates SLA's state (to *Proactive Redressing* or *Remediating*) depending to the type of the notified event (alert or violation).
ENF_REM_R5	***Get SLA***	The RDS component retrieves an alerted/violated SLA in order to identify required remediation actions.
ENF_REM_R6	***Get SLA impact***	The RDS component is able to retrieve all information related to an alert/violation.
ENF_REM_R7	***Get security components***	The RDS component retrieves all event related security components.
ENF_REM_R8	***Search for redressing techniques***	The RDS identifies remediation actions based on the event information and affected SLAs.
ENF_REM_R9	***Notify End-user***	When End-user's decision is needed in the process of managing an alert or a violation, the RDS component communicates the issue with the End-user through the SPECS Application.
ENF_BROKER_R1	***Enable CSP***	The SPECS Administrator is able to configure and enable the Broker to access and use an external CSP.
ENF_BROKER_R2	***Acquire cluster***	The Broker component is able to acquire a cluster of VMs on one of the enabled CSPs.
ENF_BROKER_R3	***Delete cluster***	The Broker component is able to delete a cluster of

		VMs previously acquired.
ENF_BROKER_R4	***Add user***	The Broker component is able to add a new user to the available cluster of VMs.
ENF_BROKER_R5	***Execute script on node***	The Broker component is able to execute on or more scripts on a cluster of VMs.
ENF_POOL_R1	***Diversity***	This requirement is satisfied by acquiring many VMs and configuring a different web server engines on them.
ENF_POOL_R2	***Load balancing***	This requirement is satisfied by configuring a proxy that is able to forward all incoming requests to one of the VMs hosting the web server engines. The scheduling policy can be configured.
ENF_POOL_R3	***Survivability***	This requirement is satisfied by acquiring and configuring the same web server engine on more than one VM.
ENF_POOL_R4	***Session sharing***	This requirement is satisfied by configuring each web server engine with a cache accessible to all web server engines,
ENF_POOL_R5	***Incident management***	*This requirement could possibly be covered by the final prototype demonstrated at M30.*
ENF_TLS_R1	***Translate TLS constraints***	TLS Reasoner translates security high level constraints and requirements in configuration templates that are used by both TLS Terminator, to enforce, and TLS Prober, to monitor and generate events.
ENF_TLS_R2	***Verify TLS constraints***	TLS Configurator verifies if the configuration templates do not overlap or generate misconfigurations by adding contradictory features or configurations.
ENF_TLS_R3	***Instantiate TLS configuration***	TLS Terminator Configurator will add to the TLS Terminator the right configuration templates that meet the negotiated requirements.
ENF_TLS_R4	***Deploy TLS configuration***	TLS Terminator Controller will deploy the configuration instantiated by the TLS Terminator Configurator.
ENF_TLS_R5	***Probe TLS endpoint configuration***	TLS Prober will periodically check if the instantiated and deployed configuration template is not altered during the lifecycle of the component.
ENF_SVA_R1	***Detect vulnerabilities and misconfigurations***	The SVA mechanism is able to detect software vulnerabilities.
ENF_SVA_R2	***Report vulnerabilities and misconfigurations***	The SVA mechanism reports about the detected software vulnerabilities.
ENF_SVA_R3	***Upgrade libraries and fix misconfigurations***	*Due to complexity of the automatically upgrading libraries and fixing misconfigurations, this requirement will most likely remain uncovered.*
ENF_SVA_R4	***Visualize detected vulnerabilities and misconfigurations***	The SVA Dashboard presents all software vulnerability reports, list of published vulnerabilities, and status of scans and measurements taken under the umbrella of the SVA mechanism.

ENF_CRYPTO_R1	***Provide client-side encryption tool as a plugin/extension***	The E2EE mechanism provides client-side encryption with the E2EE Client component.
ENF_CRYPTO_R2	***Configure and deploy encryption tools***	Encryption tools (components of the E2EE mechanism) are configurable.
ENF_CRYPTO_R3	***Encrypt data***	The E2EE mechanism enables local encryption of files.
ENF_CRYPTO_R4	***Decrypt data***	The E2EE mechanism enables local decryption of encrypted files.
ENF_DBB_R1	***Offer secure storage***	The DBB mechanism automatically offers secure storage in the cloud.
ENF_DBB_R2	***Assure business continuity with backup***	The DBB mechanism comprises components orchestrating backup services.
SLANEG_R30	***Remediation through SLA renegotiation***	The RDS component considers renegotiation of an existing signed SLA as a potential remedy to apply in case of alerts and violations.
SLANEG_R31	***Alerts/violations affecting multiple elements of the secure SLA hierarchy***	The RDS component considers interrelationships among SLOs when choose the optimal redressing technique in case of SLA alerts and violations.

Table 28. Coverage of Enforcement requirements with respect to validation scenarios

For the sake of completeness, Table 29 summarizes coverage of requirements by core Enforcement components and security mechanisms.

Requirement ID	***Planning***	***Implementation***	***Diagnosis***	***RDS***	***Broker***	***WebPool***	***DBB***	***E2EE***	***SVA***	***TLS***
ENF_PLAN_R1-R12	X									
ENF_PLAN_R8-R9	X			X						
ENF_IMPL_R1-R9		X								
ENF_IMPL_R10	X									
ENF_DIAG_R1-R18			X							
ENF_REM_R1-R11				X						
SLA_NEG_R30-R31				X						
ENF_BROKER_R1-R5					X					
ENF_POOL_R1-R6						X				
ENF_TLS_R1-R5										X
ENF_SVA_R1-R4									X	
ENF_CRYPTO_R1-R4								X		
ENF_DBB_R1-R2							X			

Table 29. Enforcement components/mechanisms and related requirements

With respect to the mapping considered in the first year (reported in D1.1.2), the following changes have been made (for details see D4.3.2):

- Requirements *ENF_PLAN_R8* and *ENF_PLAN_R9*, related to building a reaction and a migration plan, have been initially covered by the Planning component. Current prototypes move these two functionalities to the RDS component.
- Requirement *ENF_IMPL_R10*, related to updating the monitoring policy, has been moved from the Implementation component to the Planning.
- Newly defined requirements *ENF_DBB_R1* and *ENF_DBB_R2* are covered by the DBB mechanism.

If any changes occur during the final steps of the implementation and integration phases, they will be reported in the final iteration of this document.

6. Testing techniques and technologies

Testing is an essential part of any software development life cycle. Having the right methods and tools to detect faults at the right time is a primary factor for success in software development. The first iteration of this deliverable presented a brief survey of all testing techniques and technologies that could possibly be adopted in SPECS. In this section we report the actual methods and tools adopted by project's developers and integrators.

Although testing has a crucial role in software development, it is impossible to test all preconditions, all possible inputs, all interactions, and all software's characteristics. Therefore testing must be conducted efficiently and systematically to optimize its effectiveness within the given time frame. In SPECS, testing activities have been carefully scheduled in accordance with implementation and integration plans in order to support agile development as much as possible.

During the development stage of the project, developers working on the framework's components are predominantly performing functional, unit and component tests which are driven by elicited requirements. Unit and component testing plays a very important role in finding faults before the integration process. Testing outcomes provide a base for further development and improvements. In order to reduce integration issues, to detect integration errors as quickly as possible, and to support a development of a cohesive framework more rapidly, the continuous integration approach has been adopted together with continuous integration testing. Further details related to (continuous) integration will be provided in deliverables D1.5.1 and D1.5.2. Nevertheless, some aspects (collaborative development guidelines and code quality analysis) are briefly discussed here as they are also related to unit and component testing.

To increase development productivity, some testing has been manually conducted with code walkthroughs and (informal) technical reviews. During this phase requirements were analysed with the purpose of identifying any overlapping or missing requirements, designs were examined with the purpose of identifying any defects and possible performance issues, and interfaces were reviewed in order to avoid inconsistent specifications.

A code quality analysis tool is used by developers in order to assure the quality of all developed components and the quality of the entire system. Monitoring and fixing detected code quality issues on one hand guarantees the quality of the product and on the other hand assures easier and faster integration. Details are discussed in Section 6.2.

In order to choose the best testing techniques and technologies for automatic testing, a number of factors have been taken into account (e.g., used programming languages, testing objectives, and time constraints). To ensure smooth integration, functional tests have been and will be conducted on component, integration, and system level. All details on methodologies for functional testing are provided in Section 6.3. On the integration and system level, some non-functional aspects will also be tested. Considering the complexity of the entire framework, interoperability will be put to test. The framework's dependability and robustness will be evaluated. And most importantly, considering that the framework offers security services, the framework itself will be tested in terms of security. Methods and tools used to execute non-functional tests are discussed in Section 6.4.

Thanks to the Application Security Verification Standard (ASVS 2.0) proposed by OWASP [31], all modules and security mechanisms developed in the project will be assessed in terms of security as well. For details about the security review see Section 6.5.

Note that this section only provides information about the chosen code quality assessment tools, testing methodologies and technologies, and approach to a security review. Initial results of code quality analysis and components testing are reported in Section 7. Results of integration and system testing will be reported in deliverable D1.5.2. Non-functional evaluation of the Enforcement module and its security review will be reported in the final iteration of this deliverable, namely in D4.5.3.

Note that testing activities will be performed in other work packages as well (adopting the methodology presented in this task), and all designed and executed tests will be reported in dedicated prototype deliverables.

Before discussing testing approaches, the collaboration guidelines are presented (for the entire SPECS project, not only the Enforcement module).

6.1. Collaborative development guidelines

Since SPECS is a collaborative project, following the same rules and principles is very important. Some guidelines for developers have been already presented in D4.5.1, but for the sake of completeness, the summary is again reported here.

Initially, two repositories were anticipated for the project's code, namely GitHub and Bitbucket, with two different accounts each. During the development and integration, the decision was made to simplify the work and only use one repository with one account. All code (official and experimental) produced in SPECS is available at Bitbucket [1].

In order to make development consistent and transparent, the SPECS repository can only be used for software components like prototypes and tests, patched third-party software components, and any other projects needed and developed in SPECS (e.g., scripts, recipes).

Project's repositories are very different in terms of the content. There are repositories for components, repositories for Chef cookbooks, and repositories for all other utilities (e.g., data models). To simplify the process of integration and to make the project's Bitbucket web site more readable, a naming convention has been adopted. More details will be provided in deliverable D1.5.1 focused on integration.

Bitbucket not only provides revision control and source code management, but also supports bug tracking system which simplifies development, testing, and integration process. Each creator or owner of a project simply activates the issue tracking functionality [2], which outlines feature requests and bug reports.

In summary, some very basic rules for developing and committing code are followed for better overview of the project and for easier integration:

- Committed code passes all basic tests (e.g., unit tests) which are also committed. Unit tests cover at least 50% of the code.

- Committed code is analysed in terms of quality and at least issues outlining security or performance flaws are fixed.
- Each repository/project contains at least a basic documentation providing installation and usage guides for both, the code and the associated tests.
- The common data models defined in T1.3 are used.
- Naming convention for repositories is adopted.
- Each repository contains a README file with a clear description, installation and usage guides, and tests.
- Each repository contains a LICENCE file summarizing the copyrights.

6.2. Code quality analysis

Assessment of the quality of the code for each component is conducted through metrics like code complexity, number of detected issues and their severity, coverage and success of accompanied unit tests, etc.

As anticipated in D4.5.1, SonarQube [3] is used for the purpose of code quality assessment, since it is an efficient open source tool that supports many different programming languages, and is able to evaluate all proposed code quality metrics for each component and all accompanied unit tests. Also, SonarQube can be easily coupled with Atlassian Bamboo, i.e., the system used for continuous integration.

SonarQube evaluates the following aspects and metrics[4]. Note that the following is only the description of the tool; the evaluation of the code with this tool is presented in Section 7.1.

Complexity. Highly complex code may be hard to understand and is more prone to bugs. Code bases with high complexity value are reviewed and (if possible) broken down to several pieces of code. SonarQube metrics:

- Complexity: The cyclomatic complexity, known as McCabe metric (for details see [5]). Whenever the control flow of a function splits, the complexity counter gets incremented by one. In Java, for example, keywords incrementing the complexity are: `if`, `for`, `while`, `case`, `catch`, `throw`, etc.
- Class complexity: Average complexity by class.
- File complexity: Average complexity by file.
- Function complexity: Average complexity by function.

Each function has a minimum complexity of 1. According to McCabe [5] the value of 10 is considered as the threshold between acceptable (i.e., low risk) code and too complex (i.e., high risk) code. In SPECS, this threshold of complexity not exceeding value 10 is set to classes, files, and functions.

Design. Unnecessary dependencies among files not only decrease code readability but also affect incremental build time. SonarQube metrics:

- Directory tangle index: Level of directory interdependency. Best value of 0% means that there is no cycle and worst value of 100% means that directories are really tangled.

[4] For more thorough descriptions of SonarQube's metrics see [4].

- File cycles: Minimal number of file cycles detected inside a directory.
- Dependencies to cut between directories: Directory dependencies to cut in order to remove cycles among directories.
- Dependencies to cut between files: File dependencies to cut in order to remove cycles between files inside a directory.

In SPECS, code with a high amount of dependencies is reviewed and revised.

Documentation. Comments in code, especially in headers of public APIs, represent a main source for documentation. Therefore they are key for the source code to be understandable and useable. SonarQube metrics:

- Comment lines: Number of lines containing either comment or commented-out code.
- Comments: Density of comment lines, where 50% means that the number of lines of code equals the number of comment lines and 100% means that the file only contains comment lines.
- Public documented API: Number of public APIs with comments header.
- Public undocumented API: Number of public APIs without comments header.
- Documentation: Density of documented public APIs.

Duplications. In some cases (where they are unnecessary), code duplications may cause issues in code maintenance and decrease code readability. They are usually solved by factorizing the duplicated code. SonarQube metrics:

- Duplicated blocks: Number of duplicated blocks of lines.
- Duplicated files: Number of files involved in a duplication.
- Duplicated lines: Number of lines involved in a duplication.
- Duplication density: Density of duplicated lines.

In SPECS, the goal is to bring the level of duplications to almost non-existent.

Issues. In order to improve the quality of the code and assure its correctness, detection of any minor or major issues affecting code's usability and productivity or even security is of high importance. SonarQube metrics:

- Number of issues: Number of detected issues.
- Severity: The severity level of detected issues, where:
    - Blocker: Operational/security risk: This issue might make the whole code unstable in production.
    - Critical: Operational/security risk: This issue might lead to an unexpected behaviour in production without impacting the integrity of the whole application.
    - Major: This issue might have a substantial impact on productivity.
    - Minor: This issue might have a potential and minor impact on productivity.
    - Info: Not known or yet well-defined security risk or impact on productivity.
- Debt: Effort to fix all issues.

SonarQube provides a detailed list and description of all detected issues. SPECS developers are encouraged to address all detected issues and fix at least the most critical ones (blockers, critical, and major issues).

Size. Since the size of the code is often correlated with number of duplications, issues, and the overall complexity, the size of the code can have an impact on code quality. SonarQube metrics:

- Lines of code: Number of physical lines that contain at least one character which is neither a whitespace or a tabulation or part of a comment.
- Number of files: Number of files in analysed package.
- Number of directories: Number of directories in analysed package.
- Number of lines: Number of physical lines (number of carriage returns).
- Number of functions: Number of functions, methods, or paragraphs (depends on a language).
- Number of classes: Number of classes (including nested classes, interfaces, enums, and annotations).
- Number of statements: Number of statements in analysed code. In Java, statements counter gets incremented by one each time a following key word is encountered: `if`, `else`, `while`, `do`, `for`, `etc`.
- Number of accessors: Number of getter and setter functions used to get (reading) or set (writing) a class property.

In SPECS, developers are encouraged to keep the code as simple (in terms of size) as possible.

Tests. As already mentioned, testing is a very important part of software development life cycle. The basic testing includes a set of unit tests which have to cover a large portion of logic in order to assure the correctness of behaviour. SonarQube metrics:

- Unit tests coverage: The density of unit test coverage. SonarQube checks how much of the source code has been covered by the unit tests.
- Line coverage: Number of lines of code that have been executed during the execution of unit tests.
- Condition coverage: The density of possible conditions in flow control structures that have been followed during unit tests execution. On each line of code containing some Boolean expression, SonarQube checks whether each Boolean expression has been evaluated both to true and false.
- Unit test success: Density of succeeded unit tests.
- Failures: Number of unit test that have failed with an unexpected exception.
- Errors: Number of unit tests that have failed.
- Tests: Number of unit tests.
- Execution time: Time required to execute all the unit tests.

Each developer in SPECS builds a set of unit tests for each component. The agreement is that tests have to cover at least 50% of the code and all tests have to succeed.

For code coverage, JaCoCo [8], has been chosen since it is the most widely adopted and recommended open source solution to be used with SonarQube.

SonarQube can also evaluate code's technical debt in an objective, accurate, reproducible, and automated way. Technical debt is measured according to SQUALE[5] method in terms of the

[5] SQUALE stands for Software Quality Assessmnet based on Lifecycle Expectations. For details see http://www.sqale.org/.

time needed to fix all detected issues. For this purpose SonarQube uses two metrics, namely Technical Debt Ratio (TDR) and SQUALE rating. Technical debt ratio is the ratio between estimations of the effort needed to fix detected issues and the effort needed to develop the code from scratch. The SQUALE rating depends on the technical debt ratio and is defined as shown in Figure 5. For details on the evaluation of these technical debt metrics see [6].

Figure 5. SQUALE rating

An example of a code quality evaluation is presented and discussed in Section 7.

6.3. Functional testing

Functional testing is performed to ensure that the developed software conforms to all elicited requirements and design specifications. As mentioned above, functional testing is performed on component level (with unit and component testing) and also on integration and system level. All details are provided in the following subsections.

Some tests for the Enforcement module are reported in Section 7. Testing of integration and system will be conducted in task T1.5 and reported in deliverable D1.5.2 at the end of the project.

6.3.1. Unit and component testing

Each module in SPECS consists of a set of components. For example, as seen in Section 3.2, the Enforcement module comprises Planning, Implementation, Diagnosis, RDS, and Auditing. Before integrating individual components and the entire module into the framework, components need to be thoroughly tested. Developers are required to test individual components' functionalities in terms of unit tests, and when enough code coverage is achieved with unit testing, they are required to test entire components in terms of component tests. Prior to unit and component testing, some manual tests are conducted for each component of each module with code walkthroughs and (informal) technical reviews to identify defects and possible performance and integration issues.

The architecture of the Enforcement module itself may not be so complex. But because of the high complexity of the enforcement process, there are many dependencies on other modules/components. To isolate the behaviour of each tested component (when executing unit and component tests), external dependent components are replaced by mocks that simulate their behaviour.

Development efforts in SPECS are scattered over several teams and individuals who use different technologies and different programming languages (e.g., Java, Python, Go). Some differences are also necessary due to the fact that some pieces of the framework are just adapted and integrated existing open source tools (e.g., OpenVAS, Nmap). Therefore unit and component testing is executed with different technologies as outlined in Table 30.

Testing aspect	Tool
Mocking	WireMock [13], Mockito [14], Fongo [15]
Unit and component tests	JUnit [16], MockMvc [17], PyUnit [19]

Table 30. Tools used for unit and component testing

Testing activities covering functional aspects of the Enforcement module are discussed in Section 7.2. Each discussed executed test is presented in the form of the following table.

Test ID	
Test objective	
Verified requirements	
Inputs	
Expected results	
Outputs	
Comments	

Table 31. Test case template

6.3.2. Integration and system testing

SPECS' core modules (Negotiation, Monitoring, Enforcement, SLA Platform) and security mechanisms comprise a large set of components which are interrelated and interdependent. To make sure that all components are working together as expected and that the data flow among them is as specified, integration testing has to be conducted during all steps of integration (i.e., after each component or a small set of components is added to already integrated parts).

Considering that task T1.5 is specifically focused on the entire integration process (and thus also on integration and system testing), this section will only provide a brief description of the task.

Integration scenarios that will be detailed and reported in D1.5.1 are based on validation scenarios defined in T5.1 and briefly described in Section 5.1. Components are implemented and (continuously) integrated in such a way that EU's and developer's most valuable requirements can be verified and tested as soon as possible, and that integration scenarios can be implemented and tested as soon as possible. This allows the developers to receive the feedback and use it to fix and eliminate any functional or non-functional flaws, bugs, and errors.

In D4.5.1, continuous integration software Jenkins [7] was anticipated for the use in integration testing. Jenkins provides an easy continuous integration tool which not only enables easy integration but also conducts integration tests continuously. Considering that Atlassian Bamboo [18] offers the same functionalities as Jenkins, but provides better integration with other Atlassian tools (Bitbucket), the later has been adopted in the project.

Detailed implementation, integration, and testing plans will be discussed in D1.5.1 where, as mentioned, integration scenarios will be introduced and discussed. Implementation details and integration tests will be provided in D1.5.2.

6.4. Non-functional testing

In order to evaluate the quality and readiness of the entire system, some non-functional tests will be performed on the entire system.

Considering the complexity of the entire framework and the fact that many different programming languages are used, the system's interoperability will be put to test. On one hand this includes testing external interoperability in terms of evaluating readiness of the entire SPECS system to be hosted/used by different CSPs, and on the other hand this also includes evaluating internal interoperability in terms of testing compatibility among components/modules of the SPECS framework. Details on the methodology and tools are provided in Section 6.4.3.

Stability and robustness of the system will be evaluated by means of stress testing and perturbation analysis. Details on the technique and technology are reported in Section 6.4.4.

Last but not least, security level of the entire framework will be evaluated to identify possible flaws and weaknesses in terms of security, and determine how the developed system behaves in the presence of malicious attacks. Details are discussed in Section 6.4.5.

Outcomes of all non-functional tests related to the Enforcement module will be reported at the end of the project in D4.5.3, and for the entire framework in D1.5.2. As anticipated in the description of the validation methodology in Section 4, each Enforcement core component will be evaluated in terms of which test types are more or less critical.

6.4.3. Interoperability testing

Interoperability testing will be conducted on the SPECS Platform with the main objective of evaluating the interoperability level reached by SPECS itself. Specifically, interoperability testing verifies possible executions of the same scenarios in presence of different external CSPs offering a target service, and/or hosting CSPs which hosts the SPECS Platform. Moreover, interoperability among SPECS' core modules is assured by adherence with the SPECS REST APIs reported in D1.3. Additionally, SPECS' core modules allow for interoperability with new security mechanisms, which can be added by following the development and deployment guidelines reported in D1.1.3.

Interoperability testing scenarios are identified from the integration scenarios, defined for system functional testing purposes (see Section 6.3.2) and reported in D1.5.1, as well as from validation scenarios, defined in T5.1 and reported in D5.1.1 and D5.1.2. Specifically, starting from an existing scenario, some variants are identified if possible. These variants involve mainly a different external CSP and/or hosting CSP and evaluate the capability of SPECS of offering the same target service, by involving different CSPs. The definition and execution of all possible interoperability scenarios with available CSPs is infeasible given the scope of this deliverable.

The adopted interoperability testing methodology, hence, includes the following steps:

1. *Identification of scenarios*: Among the available scenarios (both validation scenario and integration), those which can be adopted to evaluate the interoperability level are selected. The selection criterion is mainly the possibility to involve a different CSP (either external or hosting) with respect to those used in the scenario.

2. *Definition of variants*. Starting from the identified scenarios, possible variants are defined. A variant is similar to the source scenario in terms of sequence of steps; it differs in some details as, for example, the usage of different brokering features requires the involvement of a different CSP.
3. *Executions of scenario*. The defined variants are executed.
4. *Collection of results and analysis*. The obtained results, for all possible variants, are collected. A comparison between the source scenario and its related variants is performed.
5. *Corrective actions*. If some variants discover bugs related to interoperability issues, corrective actions shall be implemented and variants shall be executed again. According to the specific implemented corrective action, an impact analysis could be necessary in order to reduce the number of scenarios to execute.

Given the strict relationships between interoperability and integration and system testing, both Jenkins [7] and Atlassian Bamboo [18] will be adopted to perform the interoperability testing activities.

6.4.4. Dependability and robustness testing

This section investigates experimental processes to assess the dependability and robustness of the SPECS implementation. We will refer to these tests as Perturbation Analysis (PA) where PA entails the deliberate introduction of perturbations to examine how well the SPECS system can tolerate data deviations on the input or interfaces, i.e., deviation tolerance. As perturbations are typically encountered at the operational level, classical analytical testing approaches such as static analysis (that focus on specification based testing) can only partially address operational perturbations. Hence, the perturbation analysis is conducted on the SPECS implementation and utilizes an experimental testing process. Such a perturbation process (especially if it results in observable deviations of behaviour) enables the SPECS developers to identify the weak elements in the design of the implemented system to make corrective changes as needed.

The content of this section presents the conceptual methodology of conducting perturbation analysis on the SPECS system. The methodology, as a supplement to functional and interoperability testing, provides guidelines on test case design for verifying the correctness of the implementation and its sustainability against the deliberate injection of perturbations at different components, interface API's and/or architectural data flows as warranted by the specifications.

It is important to highlight upfront that any experimental injection process, by its inherent nature of statistical coverage[6] of the operational state space, does not provide for completeness of the perturbation analysis of the SPECS Enforcement module. Hence, the experimental injection process described in this section presents two elements as:

a) The general perturbation analysis methodology.

[6] While static analysis techniques can potentially claim completeness of the state space to the degree of detail of the available specifications (including source code though only for limited code sizes), the operational state space is infinite for software. Consequently, operational testing techniques such as random testing, or directed techniques such as bit flips, data types testing, among others, are invariably based on statistical approaches to target focused partial areas of the operational state space.

b) Templates that constitute guidelines to develop implementation-specific perturbation testing cases for robustness and dependability analysis.

The basic process of Perturbation Analysis was initially developed by TUDA in the ABC4Trust project for testing crypto architectures [20]. While the ABC4Trust's hardware-oriented crypto architecture fundamentally differs from the SPECS middleware-level Enforcement Module, in SPECS the foundations and experiences from ABC4Trust are re-utilized, refined and re-oriented for usage in the API-focused perturbation analysis methodology applicable to the SPECS enforcement module. Furthermore, unlike the crypto reference implementation analysis in ABC4Trust, SPECS develops the Enforcement module API testing guidelines demo (see Scenarios #1, #2, #3, and #4 in Section 6.4.4.3) with Java-style pseudo code.

The objective of conducting perturbation analysis is to experimentally evaluate the dependability and robustness of the SPECS implementation against potential stress or failure scenarios. In this document, the terminologies of "dependability and robustness" refer to "the correctness of SPECS implementation on encountering perturbations namely failures, incorrect inputs and/or outlier conditions". The following sections present the general framework for implementing such a perturbation analysis in order to serve as guideline to develop specific and precise perturbation test cases. The perturbation analysis framework puts emphasis on assessing the correctness of SPECS implementation via varied testing campaigns such as injecting outliers and executing stress tests, etc.

The following is the basic terminology of dependability testing as background for understanding the perturbation and analysis processes (for more details, the interested readers can refer to [21]).

- **Robustness** refers to the correctness of implementation (in particular referring to availability and integrity) in the presence of failures.
- **Perturbation** refers to a deliberately-introduced misuse or abuse event that has the potential to interrupt the target system's correct operations, and consequently affect system robustness.
- **Perturbation analysis (PA).** The objective of perturbation analysis is to investigate how a target system, or parts of the system, behave under anomalous (i.e., perturbed) operational conditions. Perturbation analysis is able to highlight the types of outputs a target system produces under those anomalous circumstances. In practice, perturbation analysis simulates various scenarios to represent deviations from standard system specification (also called "misuse cases"). The underlying assumption is that those anomalous cases have not been taken into account during the system designing stage and the corresponding reactions might not have been specified. Contrary to traditional functional testing (correctness) and penetration testing (usually a stable architecture and source code level implementation details are needed), the primary target of perturbation analysis is to assess system robustness. (c.f., Figure 6). It is important to mention that perturbation analysis is not an approach to determine system correctness, but primarily to assess the performance of robustness/fault-tolerant mechanisms of the target system when encountering perturbations.

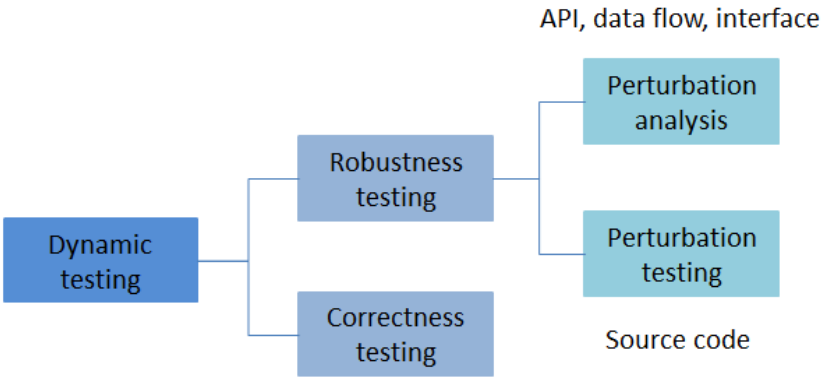

Figure 6. Dynamic testing category

- **Perturbation Campaign** Perturbation analysis does not focus on testing an individual misuse case. It is implemented in the form of a perturbation campaign, which is a collective set (or class) of misuse cases. A specific campaign might target different elements of a system (cf. ET below).
- **Evaluation Target (ET)** The evaluation target refers to specific system elements (e.g., component, block, API, etc.) that have been selected for conducting perturbations. In this document, perturbation analysis focuses on architectural data flows and various interfaces. Therefore, it can offer some degree of isolation from minor maintenance-purpose modification at source code level of target components. In general, an evaluation target subjected to a perturbation is expected to observe a successful "fail-safe" reaction behaviour (i.e., in a perturbation analysis scenario, the evaluation target has to respond correctly to ensure that the functional behaviour of other system modules remains intact on occurrence of a perturbation or failure). A typical "fail-safe" approach entails raising an exception when encountering a failure and halting the subsequent operations immediately at a pre-defined safe state to prevent that failure propagating to other modules. The halting of operations involves either completely stopping all operations or switching to a pre-defined degraded form of operations. Alternate types of "fail-safe" actions involve raising flags and requiring user inputs to handle the identified anomaly in order to proceed with execution.

6.4.4.1. Perturbation analysis framework

In this document, the approach to perturbation analysis is based on the framework shown in Figure 7 where an ET (from the SPECS implementation) is selectively exposed to a perturbation based on the functional test cases for evaluating system robustness. The selection of an ET is based on the following criteria:

- Focusing on a particular stage of SPECS's life cycle.
- Selecting the:
  (i) Data flows at the SPECS architecture level.
  (ii) Components and interfaces of SPECS.

The perturbations in this document are based on perturbing the functional behaviour of the target Enforcement module, with the goal of assessing the degree of robustness (to the perturbations) of the SPECS implementation. Based on this framework, it is possible to develop a comprehensive approach (to the limits of any experimental approach) that can cover a spectrum of real or speculative perturbations conducted against the SPECS system. It conceptually incorporates perturbations derived from system specifications at different levels of abstraction during system construction, as well as feedback from operational conditions.

The results of perturbation analysis can be used by designers and developers to improve the robustness of the SPECS implementation in its final edition. This document adopts some existing perturbation frameworks (such as those that target ET assessment of availability and integrity in the presence of failures [21]). The next section presents the methodology that implements the proposed PA framework.

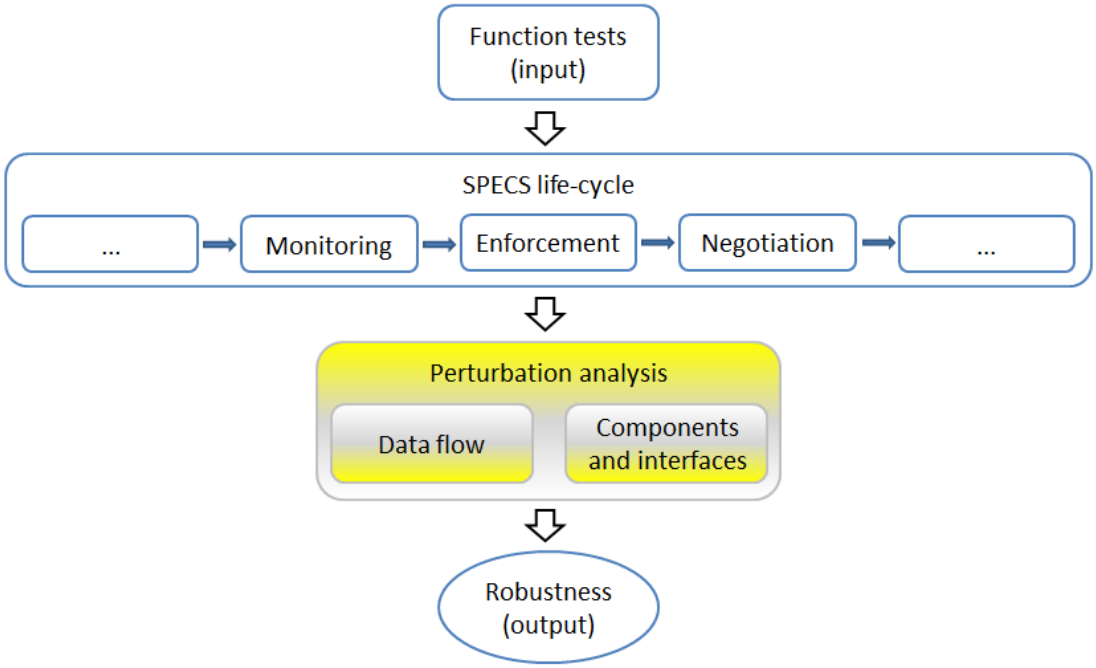

Figure 7. Perturbation analysis framework

6.4.4.2. Methodology

Perturbation analysis methodology consists of multiple steps illustrated in Figure 8 below. It starts with identifying an ET based on the framework (see Figure 7 above), i.e., the SPECS life-cycle and its associated flows/components and interfaces from relevant deliverables. In Steps 2-4, ET is first classified into different categories (namely, data flow, component or interface) such that corresponding perturbation campaigns can be composed and implemented. In Step 5, generated perturbation results are analysed for developing possible corrective actions. However, it is significant to be aware that corrective actions may change the ET type. Thus the analysis of possible corrective actions can be used as feedback for further design and development in the final edition of the SPECS implementation.

It is considered best-practice to document each perturbation as "misuse case scenarios", which specifies details about applied perturbations, observed results and any mitigation/corrective actions that have been taken. At the state of the art, there is no commonly-recognized format for documenting misuse cases. Therefore, in Table 32, we propose a template for developing specific perturbation analysis test cases by developers. Such a template can also be helpful to support system testing (e.g., trying to reproduce the failure once a corrective action has been deployed).

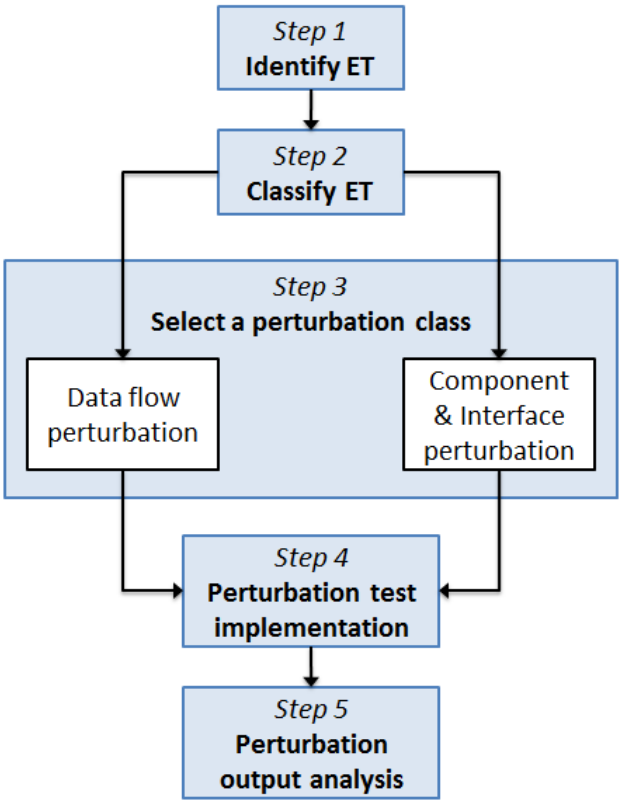

Figure 8. Perturbation analysis methodology

Scenario #. Name of PA misuse case scenario	
Summary	Brief description of misuse case scenario.
Evaluation Target (ET)	Description of ET (for example, SPECS component, SPECS API).
ET Type	ET classification as 1. Architecture Data Flow (***Arch***) or 2. Component/Interface (***Comp***).
Normal flow	Description of correct ET flow/usage.
Perturbation	Description of perturbation tests in the specific misuse case.
Perturbation Class	Perturbation classification for applying different testing class as followed, 1. Data Flow Level or 2. Component/Interface Level.
Testing Result	Documentation of output of perturbation tests in the specific misuse case. The output of perturbation analysis can be categorised into one of the following: 1. ***Compliant:*** The execution follows the documented specifications during the perturbation testing phase, i.e., failures detected successfully once exceptions triggered. Alternatively, no unexpected behaviour (e.g., high resource consumption) is observed during the perturbation phase. 2. ***Non-complaint***: The execution does not follow the documented specification during the perturbation testing phase, i.e., failures cannot be successfully detected even if the exceptions are triggered. Alternatively, anomalous behaviour (e.g., high resource consumption) is observed successfully during the perturbation testing phase.

	3. ***Inconclusive***: A conclusion cannot be made on the compliance of documented specification during the perturbation testing phase for some reason, possibly the test execution requires a significant amount of time.
Mitigation Action	Document the candidate mitigation actions based on the observed faults. When the perturbation was handled by an ET correctly, the correctness of implemented mechanism should be documented here.

Table 32. Misuse case scenario template for perturbation analysis

Following with the methodology illustrated in Figure 8, perturbation analysis cases can be designed or modified according to specific situations. The methodology comprises of five steps:

Step1. Identify the ET
The PA starts with analysing the entire target system to identify the components that can compromise the overall robustness. For example, applying a perturbation to an API call's parameter might result in a specific type of exception (which can be managed at run-time by a component), however applying a perturbation to a security level parameter might result in altering original system parameters that can be used by the other components and cause exceptions.

Step2. Classify the ET
After identifying the ET at Step 1, the perturbation analysis classifies ETs into different categories, which will be used in the next step to select the appropriate perturbation. The ET categories used in SPECS are:

1. Architecture flow.
2. Component/Interface.

We note that the PA at the component level will be focused on the SPECS API's. This allows focus on the functional interfaces representing data flow across components. It is also a convenient point to inject faults before any specific implementation primitive get used (e.g., at the transport level). This approach (originally proposed by Nik [22], [23]) allows data level control at the functional level needed to apply perturbations.

Step3. Select a Perturbation Class
The classes of perturbation are selected from the list in Table 33. Each class defines a set of tests. Each test is designed as starting from valid functional test cases, and then derived according to perturbation class. For example, in Table 33, the Data Flow-Stress case (i.e., DF-S) is used to test a particular class of Data Flow-Outlier cases (i.e., DF-O) that were derived by introducing sustained concurrent requests. DF-S tests will consider two parameters: the number ***k*** of concurrent requests and the time interval ***t*** in seconds. The test keeps ***k*** concurrent requests during a period of ***t*** seconds. The Component Data Type (C-DT) and Component Outlier (C-O) tests are performed by selecting inputs over a set of invalid inputs. Invalid inputs are identified by combining the syntax and semantics of the API call function parameters. The selection can be done manually or automated by using a distribution function e.g., uniform distribution.

Type of ET	Type of Perturbation	Expectations on Perturbation Injection
Architecture Data Flow	Data Flow Outlier Case(DF-O)	Perturbations test values that appear to deviate significantly from other members of the sample in which it occurs. DF-O includes stress cases Perturbations (DF-S) aimed at taking a system to an extreme working mode (close to its DoS threshold).
Component/Interface	Component Data Type (C-DT)	Perturbations test values that are valid for the same type of parameter, but that are invalid for the specification. Typical DT perturbations must consider the potential factors as comprehensively as possible [24], [25], [26], [27], e.g., for an integer parameter include: param--, param++, 1, 0, -1, INT_MAX and INT_MIN. In Service Oriented Architectures, the use of DT perturbations is useful and more efficient than other techniques (e.g., bit flipping) for testing fault tolerance mechanisms.
	Component Outlier (C-O)	Perturbations test values that appear to deviate significantly from other members of the sample in which it occurs.

Table 33. Perturbation Test Classes

Step4. Test Perturbation
After selecting the class of perturbation, the corresponding perturbation campaign is executed for the ET. This may require access to the running system (physical/remote) and to the code (e.g., to increase log verbosity). In any case, the perturbation analysis must guarantee that the perturbation is repeatable under the same conditions documented in the misuse case. Stress tests are executed within a finite amount of time. If the test does not produce an outcome within the time window, then the test is considered inconclusive. The size of the time window is set by the specification of a different test. The results of the test execution are documented for each designed scenario using the template shown in Table 32.

Step5. Analyze Output
During the test execution, the outputs are monitored and documented as part of the misuse case. This is a critical step as corrective actions will be designed and deployed in the final edition of the SPECS implementation based on these observations.

6.4.4.3. Perturbation analysis demo

Based on the PA methodology outlined in the prior sections, this section develops two example scenarios as guidance for PA on the Enforcement module API. As PA constitutes a natural extension to functional testing, the intent of the developed scenarios is to outline representative API calls illustrating how to set up perturbations on the call parameters.

As specified in deliverable D4.2.2, the Enforcement module's purpose is to create an enforcement system to manage SLA implementation, enforce security services, and carry out necessary prevention and recovery actions. For facilitating the design and implementation of specific perturbation tests, a set of representative perturbation analysis testing cases for the Enforcement module are presented as follows. The (representative) stress-oriented and

string-oriented test cases are demonstrated for PA purpose, which form the most common PA testing scenarios. With the selected PA test cases we can verify the system's robustness for compliance with the specified exception handling behavior (e.g., "fail and stop" mechanism). These two PA demos also cover the two main different PA categories, namely, Data-flow level PA and component level PA. These representative PA categories form the basis for PA testers to organize, implement, and evaluate different part of the system (ETs) via the full API list applicable for the Enforcement module.

Two considerations are important to highlight in the context of PA, namely:

a) Defining the actual perturbation(s) to conduct (e.g., parameter types, data rates, ET type, API functionality) requires detailed implementation information to be meaningful. As PA naturally supplements functional testing, the advocated basis is to formulate PA tests as a) focussing on parameters already being used in functional/interoperability testing, b) utilizing outliers on parameter specifications, and c) defining extreme cases based on deliberate misspecification or stress conditions (e.g., data rates, simultaneous invocation of multiple parameters) that breach functional specifications. Equally important, based on the system threat model, is to determine if single PA instances are desired or combinations of PA's are needed. Consequently, given the immense variety of parameter attribute combinations, the composition of PA test cases is typically done as an extension of functional/interoperability testing with the functional tests as a baseline.
b) The essence of PA, apart from setting up the perturbations, is to interpret the 'Testing Result' in the PA Scenarios where a designer has to interpret if the result was Compliant/Non-Compliant/Inconclusive. This determination requires (a) detailed understanding of the operational behaviour to assess valid/invalid responses, and more importantly (b) comprehensive understanding of the data/information flows across components as the perturbation in one component often manifests as a deviation only in another component. Hence, meaningful PA requires having a comprehensive system view of inter-component behaviour and data flows.

Based on these considerations, and for the dependency on implementation details for component and inter-component functionality, the PA process is guided by the illustrative representative scenarios #1 and #2 (respectively covering data-flow and component level PA) to help compose and conduct the actual tests as extensions to functional/interoperability testing. Two additional representative scenarios #3 and #4 (at the component and interface levels) are developed to respectively outline the object-oriented and mutation based PA cases. These scenarios outline the PA process, on the representative scenario classes, for guiding actual testing on the implementation.

Scenario #1: Data Flow-Level Perturbation

The following content demonstrates an example of data flow-level misuse cases considered for the Enforcement module, which covers two sets of SLA phases in SPECS, namely implementation and remediation phase.

The selected scenarios particularly focus on assessing the robustness of the Enforcement module under stress state, which is a typical situation for perturbation analysis in practice. The involved API calls (described in detail in D1.3) are:
/sla-enforcement/diag-activities,

/sla-enforcement/diag-activities/{da-id},
/sla-enforcement/diag-activities/{da-id}/status,
/sla-enforcement/diag-activities/{da-id}/sla-id,
/sla-enforcement/diag-activities/{da-id}/classification.

Scenario 1. Stress PA on SPECS Enforcement module	
Summary	A stress perturbation implemented on SPECS Enforcement module to assess the resilience against denial of service (DoS).
Evaluation Target (ET)	All components of the Enforcement module.
ET Type	***Arch***
Normal flow	Enforcement module supports continuously receiving monitoring events from the Monitoring module, which is specified in deliverable D1.3 as "*The Diagnosis component must be able to receive notifications from the Platform about monitoring events captured by the Monitoring module.*"

Perturbation	Stress the Enforcement module by sending ***k*** monitoring events during a short period of ***t*** seconds. The purpose is to check the resource consumption (i.e., memory consumption, CPU usage rate, IO idle status, etc.) and availability and performance of the service (i.e., processing time of incoming monitoring events, response time of incoming monitoring events, etc.), When applicable, this perturbation test should be repeatable for all involved components.
	The parameter *k* and *t* are the following:
	• ***k*** ∈ {50, 2500, 125000}
	• ***t*** ∈ {1, 2, 3}
	The pseudo-algorithm is described in the following(*taking memory consumption, for example*):
	
	`public class Perturbation_Pseudo_Stress{`
	`  memBase = getFreePhysicalMemorySize();`
	
	`  while(Timer(t)){`
	`  sendMonitoringEvent(k);`
	`  }`
	
	`  memCost = getFreePhysicalMemorySize()-memBase;`
	`  Logger.getLogger(t, k, memCost);`
	`}`
	
	Document the outputs and assess the consumption of memory resource.

Perturbation Class	DF-S
Testing Result	***The following content is ONLY for demonstration purpose*** ***Compliant:*** Enforcement module has successfully passed the stress-oriented perturbation. Unexpected resource consumption has not been observed during the perturbation testing phase. Enforcement module works normally under all these three stress scenarios.
Mitigation Action	None

Table 34. Data Flow-Level Perturbation

Scenario #2: Component and Interface-Level Perturbation

The following content demonstrates an example of component level misuse cases considered for the Enforcement module. Remediation Decision System is the main component of the Enforcement module for managing SLA alerts and SLA violations as well as finding the most suitable mitigation actions. In this context, the API call of */sla-enforcement/rem-activities/{ra-id}* is selected as the perturbation target. As aforementioned, perturbation analysis is specially focusing on parameters that can potentially compromise system robustness. For instance, we decide to test SPECS system resilience against perturbations affecting the value of parameter *ra-id* with regard to API call */sla-enforcement/rem-activities/{ra-id}*, which might propagate the deviated value to other component and trigger unforeseeable system crash events.

Scenario 2. Outlier PA against parameter in */sla-enforcement/rem-activities/{ra-id}*	
Summary	This perturbation targets to test the robustness of API call */sla-enforcement/rem-activities/{ra-id}*, by using values deviated from the specification of parameter *ra-id*.
Evaluation Target (ET)	RDS component
ET Type	***Comp***
Normal flow	The RDS component should return the corresponding Remediation Activity object based on the string identifier *ra-id*.

Perturbation	As the data model of the Remediation Activity described in D1.3, the parameter "*rem-act-id*" should be the type of "*string*". Considering the data model is JSON-compatible, the perturbation tests consider the following cases:
	• Perturbation of string overflow.
	The test introduces over-long strings as the value of "*rem-act-id*" against this API call. Due to JSON having no size limitation on itself, but servers having a security property called "*MaxJsonLength*" for JSON file parsing. For example, IBM WebSphere specified "*Maximum Value String Length*" with the size of 2,097,152 (2 MB) characters [28].
	
	• Perturbation of subtly crafted strings.
	The test introduces a set of intentionally-crafted Escape sequence as the value of "*rem-act-id*". e.g., `""\"\\ag63645n", "\u0062\u0022\u0035\u0001"`
	
	Assuming in total ***n*** string samples available for the data type perturbation, the first ***m*** string samples target string overflow testing and the remaining samples are for intentionally-crafted string testing. The pseudo-algorithm is described in the following:
	
	`public class Perturbation_Pesudo_DateType{`
	
	`String TestStringArray[]=new String[n];`
	
	`//Define test sample for perturbation of string overflow`
	`TestStringArray[0] = "bwgyp";`
	`TestStringArray[1] = "sdanvfkqiz";`
	`…`
	`TestStringArray[m-1] = "he36...$^&%"; //str.length > 2097152`
	
	`//Define test sample for perturbation of subtly crafted`

	strings
	TestStringArray[m] = “”\”\\ag63645n”;
	…
	TestStringArray[n-1] = “\u0062\u0022\u0035\u0001”;
	
	int Counter = 0;
	while (Counter < TestStringArray.size()){
	rem-act-id = TestStringArray.get(Counter);
	//
	invoking API /sla-enforcement/rem-activities/{ra-id}
	//
	Logger.getLogger();
	}
	}
	
	Document the outputs and observe the correctness of results or implementation of exception handling mechanism.

Perturbation Class	C-DT
Testing Result	***The following content is ONLY for demonstration purpose*** ***Non-complaint***: EM has failed to switch to the “fail-stop” mode. During the perturbation testing phase, failures cannot be detected successfully.
Mitigation Action	Document the potential mitigation actions against the observed faults.

Table 35. Component and Interface-Level Perturbation

Scenario #3: Component and Interface-Level Perturbation (Object-oriented based)

The following content demonstrates an example of object oriented perturbation analysis at component level considered for the Enforcement module. As Java language is used to develop the Diagnosis component of the Enforcement module, it is necessary to execute object-oriented PA test cases for the purpose of robustness assessment. One common practice of object-oriented PA is to perturb corresponding behaviours against public Java methods by introducing special instances of different types. By applying this object-oriented perturbation analysis, the performance of the targeted public methods can be evaluated to help prevent incorrect inputs propagating further in SPECS implementation. The DiagnosisActivityController.java is selected as the perturbation target for demonstrated of the general object-oriented cases.

Scenario 3. Object Oriented PA against Java object type	
Summary	This perturbation test targets to test the robustness of Diagnosis component in Enforcement module, by introducing object type manipulation perturbation.
Evaluation Target(ET)	Diagnosis component
ET Type	***Comp***
Normal flow	The Diagnosis component should return correct value of “ResponseEntity” and correct Diagnosis Activity object in the end.
Perturbation	The DiagnosisActivityController.java (as included in the Diagnosis component of

	the Enforcement module) defines a class to act as the diagnosis activity controller. An object type perturbation test case is introduced to assess its robustness and detect dependability issues. In the example the method `httpheaders()` is used to inject incorrect headers.
	
	The pseudo-algorithm is described in the following:
	
	`public Perturbation_Pseudo_OO_Type( ) {`
	`  try {`
	`  headers = (OtherType) super.clone();`
	`  return headers;`
	`  } catch (CloneNotSupporedException cnse) {`
	`  throw new RuntimeException(cnse);  }`
	`}`
	
	`//////////////////////////`
	`Target Java method to perturb`
	`//////////////////////////`
	
	……
	`  public ResponseEntity<DiagnosisActivity> create(@RequestBody Notification notification, HttpServletRequest request) {`
	`  DiagnosisActivity diagnosisActivity = daService.create(notification);`
	
	`  URI location = URI.create(request.getRequestURL().append("/").append(diagnosisActivity.getId()).toString());`
	`  HttpHeaders headers = new HttpHeaders();`
	
	`//////////////////////////////////////////////////`
	`Invoking object type perturbation against headers`
	`//////////////////////////////////////////////////`
	
	`  headers.setLocation(location);`
	`  return new ResponseEntity<DiagnosisActivity>(diagnosisActivity, headers, HttpStatus.CREATED);`
	`  }`
	
	`  ……`
	`//////////////////////////`
	
	Document the outputs and observe the correctness of results or implementation of exception handling mechanism.

Perturbation Class	C-DT
Testing Result (example)	***<u>The following content is ONLY for demonstration purpose</u>*** ***Non-complaint***: The Enforcement module has failed to switch to the "fail-stop" mode. During the perturbation testing phase, failures cannot be detected successfully.
Mitigation Action	Document the potential mitigation actions against the observed faults.

Table 36. Component and Interface-level Perturbation (Object-oriented based)

Scenario #4: Component and Interface-Level Perturbation (Mutation based)

The following content demonstrates an example of component level misuse cases considered for Enforcement module. The Diagnosis component is the component for conducting diagnosis process. In order to assess its robustness, the process to outline mutation based perturbation analysis is introduced. As a common PA practice, mutation perturbation testing consists of several various forms such as call sequence shifting, intentional call ignoring, variable value alternation among others. By applying the mutation perturbation analysis, one can assess whether an ET can successfully prevent the fault propagation in SPECS implementation. For demonstration purpose, the DiagnosisActivityController.java is selected as the perturbation target. As aforementioned, the emphasis of perturbation analysis is focusing on erroneous input information that can potentially compromise system robustness.

Scenario 4. Mutation PA against parameter daID in DiagnosisActivityController.java	
Summary	This perturbation targets to test the robustness of Diagnosis component in Enforcement module, by introducing value mutation against parameter daID
Evaluation Target(ET)	Diagnosis component
ET Type	***Comp***
Normal flow	The Diagnosis component should return correct diagnosis activity based on the parameter daID.

Perturbation	The DiagnosisActivityController.java (as included in the Diagnosis component of the Enforcement module) defines a class to act as the diagnosis activity controller. A mutation perturbation test case is introduced to assess its robustness and dependability. The bit flip method is used in the following perturbation case:
	
	`public Perturbation_Pseudo_Mutation( ) {`
	`  string test=daID;`
	`  test_bits =test.getBytes();`
	`  new_test_bits=test_bits.flipBit(n); //flip the n-th bit`
	`  string new_daID=new string(new_test_bits, "UTF-8");`
	`  return new_daID;`
	`}`
	
	`//////////////////////////`
	`Target code to perturb`
	`//////////////////////////`
	`……`
	`@RequestMapping(value = "/{daId}", method = RequestMethod.GET,`
	`produces = {MediaType.APPLICATION_JSON_VALUE})`
	`……`
	`//////////////////////////`
	
	Document the outputs and observe the correctness of results or implementation of exception handling mechanism.

Perturbation Class	C-O
Testing Result (example)	***The following content is ONLY for demonstration purpose*** ***Non-complaint***: The Enforcement module has failed to switch to the “fail-stop”

	mode. During the perturbation testing phase, failures cannot be detected successfully.
Mitigation Action	Document the potential mitigation actions against the observed faults.

Table 37. Component and Interface-Level Perturbation (Mutation based)

It is important to note that that actual knowledge of the detailed API and implementation level behaviour is needed to compose the actual PA tests. The pseudo code level scenarios #1, #2, #3, and #4 provide the guidelines for composing the tests. These scenarios also cover the two main representative PA categories of Data-flow level PA and component level PA as applicable to the SPECS Enforcement module, and serve as process guidelines to design and implement the specific perturbation cases as actual tests over D4.5.3.

6.4.4.4. Scope and limitations of the methodology

The tests associated with the perturbation analysis documented were based on the experimental version of the SPECS implementation (available on project's Bitbucket site [1]) and architectural data flows (discussed in deliverables D4.4.2 and D4.3.2). The architecture (and its corresponding implementation) will be referenced as "SPECS architecture" in the rest of this document.

For the PA, both the data-flow level and component-level perturbations (example scenarios #1, #2, #3, and #4) are designed by focusing on SPECS and core components invoked by the APIs while performing the tests.

6.4.4.5. Enforcement-related API List

The prior sections have outlined both the Perturbation Analysis methodology and also representative scenarios #1, #2, #3, and #4 that serve as guidelines for formulating and conducting the PA on the full set of API calls involved in both the implementation and remediation phases carried out by the Enforcement module. These scenarios, described with Java style pseudo code, provide the guidelines for actual tests to be conducted on the Enforcement module.

Composing tests essentially requires (a) details of the implementation, (b) details of the functional behavior, and (c) detailed understanding what constitutes normal versus anomalous behavior. Hence, the example scenarios #1, #2, #3, and #4 establish the basis (in conjunction with possessing knowledge of the functional and implementation details) to compose executable tests. The following table lists the relevant API calls needed for testing. The details of these API's are present in D1.3.

Resource ID of the API
/sla-enforcement/sc-activities
/sla-enforcement/sc-activities/{sca-id}
/sla-enforcement/sc-activities/{sca-id}/status
/sla-enforcement/sc-activities/{sca-id}/sc-list
/sla-enforcement/supply-chains
/sla-enforcement/supply-chains/{sc-id}
/sla-enforcement/notifications

/sla-enforcement/notifications/{n-id}
/sla-enforcement/diag-activities
/sla-enforcement/diag-activities/{da-id}
/sla-enforcement/diag-activities/{da-id}/status
/sla-enforcement/diag-activities/{da-id}/sla-id
/sla-enforcement/diag-activities/{da-id}/classification
/sla-enforcement/plan-activities
/sla-enforcement/plan-activities/{pa-id}
/sla-enforcement/plan-activities/{pa-id}/status
/sla-enforcement/plan-activities/{pa-id}/plansnum
/sla-enforcement/plan-activities/{pa-id}/planlist
/sla-enforcement/plan-activities/{pa-id}/active
/sla-enforcement/plans
/sla-enforcement/plans/{p-id}
/sla-enforcement/impl-activities
/sla-enforcement/impl-activities/{ia-id}
/sla-enforcement/impl-activities/{ia-id}/status
/sla-enforcement/reconfigs
/sla-enforcement/rem-plans
/sla-enforcement/rem-plans/{rp-id}
/sla-enforcement/rem-plans/{rp-id}/result
/sla-enforcement/rem-activities
/sla-enforcement/rem-activities/{ra-id}
/sla-enforcement/rem-activities/{ra-id}/status

Table 38. Enforcement API as defined in D1.3

6.4.5. Security testing

SPECS aims at offering Security-as-a-Service. Thus ensuring that the produced software itself is secure which is of the utmost importance. The five basic security aspects that should be supported by and enforced with the developed system are:

- **Confidentiality**. All data stored, used, and/or produced by the software is secure from theft is disclosure to unauthorised entities.
- **Integrity**. All data stored, used, and/or produced by the software cannot be altered over its life cycle, either through accidental corruption of the data or targeted manipulation by a malicious party.
- **Authentication**. Valid credentials that grant access to the data stored, used, and/or produced by the software are provided to authorised entities.
- **Authorization**. Software's users should only have access to authorized functions and authorized data according to a set of policies or rules.
- **Non repudiation**. The software has to ensure that senders and receivers of data cannot deny having sent or received it.

There are many different ways to compromise software and exploit its weaknesses [33]. By following a few simple guidelines for secure development, the developers of SPECS components can ensure an adequate level of security of the entire framework and individual SPECS components and applications (e.g., always keeping sensitive data encrypted, planning

for session time out after a specific time if the user is not active, validating all inputs and outputs, authenticating each access request to every object). However, since many software vulnerabilities result from defects that are unintentionally introduced in the software during design, some basic secure software development principles were followed in the design phase (for example, architecting a clear, simple design, using a strong authentication mechanism, integrating known and already tested tools as much as possible).

As discussed in D4.5.1, there are many tools available that can easily uncover security vulnerabilities and determine whether software's data and resources are protected from possible intruders. In SPECS, some individual components, but mostly integrated parts are undergoing vulnerability scans and penetration tests with vulnerability scanners developed and integrated in SPECS' SVA security mechanism (i.e., with OpenSCAP and OpenVAS).

Further details about the outcomes of security testing of the Enforcement module will be reported in D4.5.3.

6.5. Security review

In order to review the security of the Enforcement module, several strategies were considered:

- Use a cloud security assessment standard such the CCM developed by CSA [29], a partner in the SPECS project.
- Use a product-centric assessment framework such as common criteria [30].
- Review the application of best practices in security in web application development, such as those provided by OWASP [31].

We chose the last solution for following reasons.

As described in D1.1.3, the SPECS platform is likely to help End-users (customers) to implement some key controls of the CCM [29]. However, the CCM is not the right tool to evaluate the security of the SPECS platform or its individual components such as the Enforcement module described in this deliverable. Control frameworks such as the CCM are designed to evaluate the implementation of an information system by an actual organisation, taking into account aspects as diverse as human resources, national regulations or physical security of data centres. In contrast, this document describes mainly a software system, for which we can find better tools to conduct a security assessment.

The most well known approach for conducting an in-depth security analysis of a product is Common Criteria [30]. It is used in particular in the banking sector to evaluate the software and hardware security of smart cards used in payment systems. Unfortunately, common criteria evaluations are typically long and expensive: bankcards typically take more than a year to get certified, including all necessary preparatory work and lab analysis. As a cloud application is expected to be much more complex than a smart card, a certification of the enforcement module would probably take an order of magnitude more work, and way beyond the resources of this project. To our best knowledge no cloud application has undergone such a certification.

In essence, the Enforcement module, as with other modules in the SPECS project is a web application centred on a RESTful API. As such, we finally considered that the best strategy for a security evaluation is therefore to focus on risks that are specifically related to the

development of web applications, with an emphasis on the API platform. To this end, we propose to base our analysis on the Application Security Verification Standard (ASVS 2.0) proposed by OWASP [31]. The work of OWASP is well recognized in the community, and in fact the very first control of CSA's CCM (AIS-01) explicitly mentions this approach as an example of best practices for secure cloud application development. As further evidence of the value in adopting this strategy, the latest version of ASVS 2.0 was released in 2014 and is currently used as a certifiable standard for the evaluation of web applications.

The ASVS 2.0 is presented as a security checklist which asks a series of questions against which you can assess your security features. Since the Enforcement module is mainly a REST API server, we decided to create our own security assessment checklist by taking the ASVS 2.0 as a foundation and removing any security check that relates more to client side security and/or HTML/Javascript considerations, focusing solely on server side requirements. For example, we removed the following question "*Verify all password fields do not echo the user's password when it is entered*", since it relates to HTML form fields.

The resulting checklist is provided in Appendix 2.

7. Testing of the Enforcement module

This section presents tests designed and executed during the development stage in the second year of the project. As anticipated in the introduction and further discussed in Section 6, this document covers tests performed on the component level only. All analyses and tests are conducted in accordance to methodologies defined in Section 6.

In the first subsection 7.1 an example of a code quality analysis is shown. Subsection 7.2 covers test cases designed and executed for functional testing of the Enforcement module.

7.1. Code quality analysis

Evaluating the quality of the code in SPECS is performed with SonarQube [3]. An example report of the code quality analysis for the Planning component is presented in Figure 9 and Figure 10.

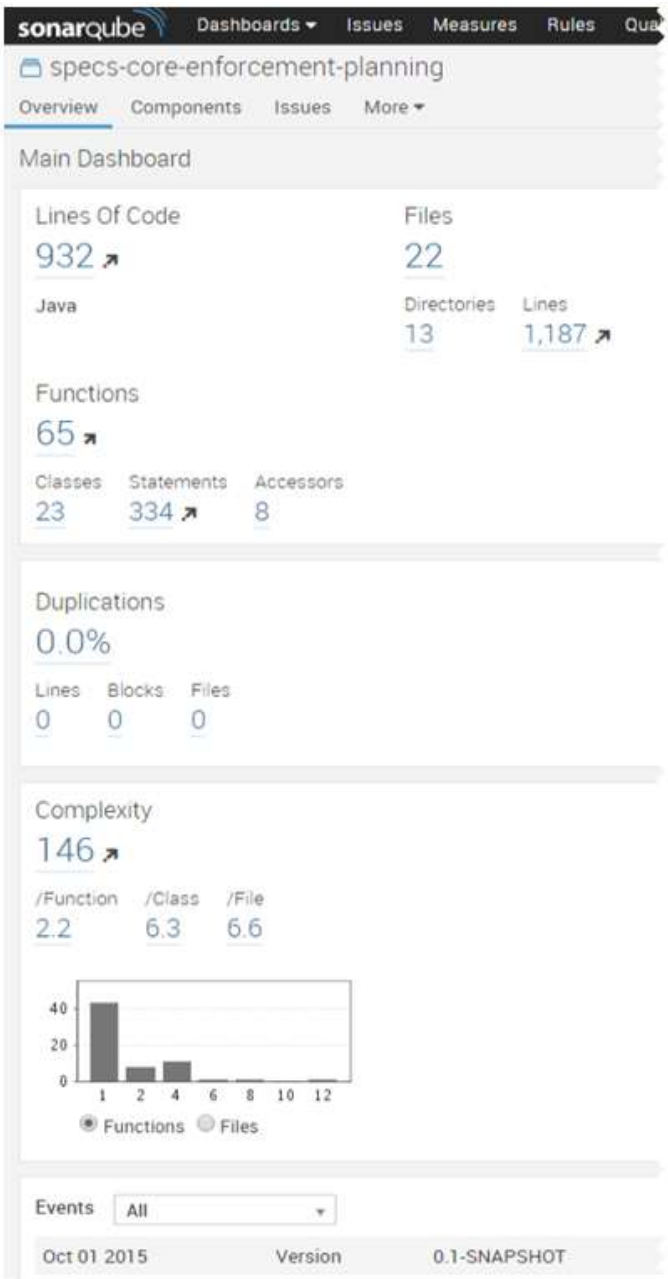

Figure 9. Code quality analysis report for the Planning component – part 1

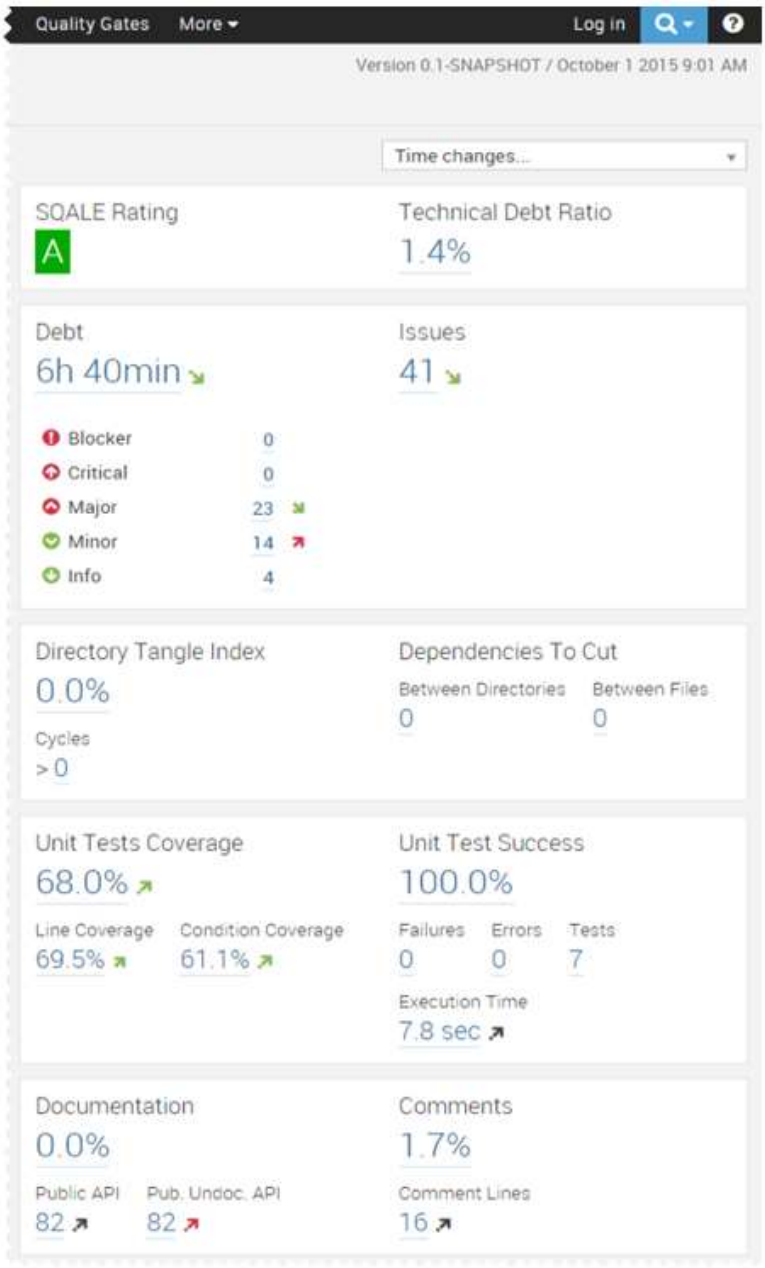

Figure 10. Code quality analysis report for the Planning component – part 2

Code quality analysis for the Planning component shows the following:

- The complexity of classes, functions and files is below the threshold of 10, which implies that the code is of low risk.
- Directory tangle index of 0.0% means that there were no cycles detected inside the directories.
- Documentation density of 0.0% and comment density of 1.7% implies that the code is well documented, but more comments need to be added.
- The code does not contain any duplications.
- SonarQube identified 43 issues, where none of them are blockers or of critical severity, and 25 of them are of major severity. The list of issues is reported in Figure 11.

- The code for the Planning component in spread over 22 files in 13 directories, and contains 1185 lines (930 lines of code). 65 functions containing 332 statements are defined in 23 classes. The code is considered of acceptable size.
- The Planning component is accompanied with 7 unit tests for which the success rate is 100% (no failures and no errors). Tests executed 69.5% of all lines and 61.1% of all conditions in the code. This means that almost 70% of the code is covered by unit tests.
- Technical debt ratio is extremely low (1.5%) which results in SQUALE rating A.

Figure 11. Code quality issues for the Planning component

Quality reports for other pieces of code for the Enforcement module can be found on dedicated repositories on Bitbucket [1].

7.2. Functional testing

Each component developed in SPECS is subject to a set of unit and component tests. For readability's sake, only tests for one core Enforcement component and one security mechanism are presented here. Tests for other Enforcement components and the rest of the security mechanisms are available at dedicated Bitbucket web sites (together with code).

Note that the reported tests do not cover all requirements associated to the Planning component and the SVA security mechanism. This is due to the fact that some tests will be performed under the integration task (reported in D1.5.2), and due to further development plans (see D4.3.2). Also note that some tests only verify correct behaviour of the code and do not cover any requirement directly.

7.2.1. The Planning component

The following are the tests executed for the Planning component [12]. Further details are available in D4.3.2.

Test ID	test_supply_chain_activity_repository
Test objective	Test Supply Chain Activity repository operations.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R2*, *ENF_PLAN_R3*, *ENF_PLAN_R4*, *ENF_PLAN_R10*, *ENF_PLAN_R11*, *ENF_PLAN_R12*
Inputs	A test Supply Chain Activity object.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed successfully.

Test ID	test_supply_chain_repository
Test objective	Test Supply Chain repository operations.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R2*, *ENF_PLAN_R3*, *ENF_PLAN_R4*, *ENF_PLAN_R10*, *ENF_PLAN_R11*, *ENF_PLAN_R12*
Inputs	A test Supply Chain object.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed successfully.

Test ID	test_planning_activity_repository
Test objective	Test Planning Activity repository operations.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R7*
Inputs	A test Planning Activity object.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed successfully.

Test ID	test_supply_chain_activity_service
Test objective	Test service class that provides operations for dealing with Supply Chain Activity objects.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R2*, *ENF_PLAN_R3*, *ENF_PLAN_R4*, *ENF_PLAN_R10*, *ENF_PLAN_R11*, *ENF_PLAN_R12*
Inputs	• A test SLA Template document in the XML format. • A test Supply Chain Activity input data (as sent by the Supply Chain Manager component).
Expected results	• A valid Supply Chain Activity is created. • A valid Supply Chain is created.

	• All operations execute successfully.
Outputs	None.
Comments	• All operations executed successfully. • Uses SLA Platform's Service Manager mock service.

Test ID	test_supply_chain_activity_service_error_behaviour
Test objective	Test the *build supply chains* method's behaviour when an invalid input data is given.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R2*, *ENF_PLAN_R3*, *ENF_PLAN_R4*, *ENF_PLAN_R10*, *ENF_PLAN_R11*, *ENF_PLAN_R12*
Inputs	• A test SLA Template document in the XML format. • An invalid test Supply Chain Activity input data.
Expected results	• The status of the created Supply Chain Activity is ERROR. • The annotation property contains the error description and the stack trace.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_supply_chain_service
Test objective	Test service class that provides operations for dealing with Supply Chain objects.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R2*, *ENF_PLAN_R3*, *ENF_PLAN_R4*, *ENF_PLAN_R10*, *ENF_PLAN_R11*, *ENF_PLAN_R12*
Inputs	A test Supply Chain object.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed successfully.

Test ID	test_planning_activity_service
Test objective	Test service class that provides operations for dealing with Planning Activity objects.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R7*
Inputs	A test Supply Chain object.
Expected results	• A valid Implementation Plan object is created. • All operations execute successfully.
Outputs	None.
Comments	Uses Enforcement Implementation mock service and Monitoring module mock service.

Test ID	test_planning_activity_service_error_behaviour
Test objective	Test the *create planning activity* method's behaviour when an invalid input data is given.
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R7*
Inputs	A test Supply Chain object with some invalid data.
Expected results	• The status of the created Planning Activity is ERROR.

	• The annotation property contains the error description and the stack trace.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_solver
Test objective	Test the Solver functionality (building supply chains).
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R2*, *ENF_PLAN_R3*, *ENF_PLAN_R4*, *ENF_PLAN_R10*, *ENF_PLAN_R11*, *ENF_PLAN_R12*
Inputs	A test Supply Chain Activity object.
Expected results	Valid Supply Chain objects are created corresponding to the Supply Chain Activity data.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_dump
Test objective	Test dump method of the JsonDumper class.
Verified requirements	/
Inputs	A test Java object.
Expected results	Java object is serialized to a valid JSON string.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_app_config
Test objective	Test application configuration loading from a file.
Verified requirements	/
Inputs	A test application configuration file.
Expected results	Application configuration properties are set correctly.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_api_supply_chains
Test objective	Perform full test of the functionalities of the Planning component related to supply chains through the REST API provided by the planning-api (part of the Planning component [12]).
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R2*, *ENF_PLAN_R3*, *ENF_PLAN_R4*, *ENF_PLAN_R10*, *ENF_PLAN_R11*, *ENF_PLAN_R12*
Inputs	• A test SLA Template document in the XML format. • A test supply chain activity input data (as sent by the Supply Chain Manager).
Expected results	• A valid Supply Chain Activity object is created. • A valid Supply Chain objects are created according to the input SLA Template, Supply Chain Activity object input data, and security mechanisms. • All operations execute successfully.

Outputs	None.
Comments	• Uses SLA Platform's Service Manager mock service. • All operations executed as expected.

Test ID	test_api_planning_activity
Test objective	Perform full test of the functionalities of the Planning component related to planning activities through the REST API provided by the planning-api (part of the Planning component [12]).
Verified requirements	*ENF_PLAN_R1*, *ENF_PLAN_R7*
Inputs	A test Supply Chain object.
Expected results	• A valid Planning Activity object is created. • A valid Implementation Plan object is created according to the input Supply Chain, and security mechanisms. • All operations execute successfully.
Outputs	None.
Comments	• Uses Enforcement's Implementation mock service and Monitoring module mock service. • All operations executed as expected.

7.2.2. The SVA security mechanism

In next subsections we present tests for all SVA components. For details of the mechanism's design see D4.3.2.

7.2.2.1. SVA Enforcement component

The following set of tests has been performed to test code for the SVA Enforcement component [9]. Detailed description of the component is provided in D4.3.2.

Test ID	test_download_ovals
Test objective	Test if oval (list of published vulnerabilities) is downloaded successfully.
Verified requirements	*ENF_SVA_R1*
Inputs	URL to oval repository.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_reconfigure_repository
Test objective	Test if repository is successfully changed.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_run_scanner
Test objective	Test if the scanner generates the scanning report.
Verified requirements	*ENF_SVA_R1*
Inputs	A test vulnerability list.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_send_scanning_report
Test objective	Test if the scanning report is successfully sent to the django server.
Verified requirements	*ENF_SVA_R2*
Inputs	Django server IP.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• The Django server is the server used by the SVA Dashboard. • All operations executed as expected.

Test ID	test_generate_up_report
Test objective	Test if the update/upgrade report is successfully generated.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_send_up_report
Test objective	Test if the update/upgrade report is successfully sent to the django server.
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• The Django server is the server used by the SVA Dashboard. • All operations executed as expected.

Test ID	test_vulnerability_list_command
Test objective	Test if the vulnerability list command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_vulnerability_scan_command
Test objective	Test if vulnerability scan command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	A test vulnerability list.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_up_report_command
Test objective	Test if update/upgrade report command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_reconfigure_repository_command
Test objective	Test if reconfigure repository command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

7.2.2.2. SVA Monitoring component

The following tests have been executed to test the code for the SVA Monitoring component [10]. For details about components activities and about measurements taken by the component to evaluate validity of SVA metrics, see D4.3.2.

Test ID	test_get_list_age_if_list_update_frequency_not_selected
Test objective	Tests if the *vulnerability list age* measurement is taken in case when the *list update frequency* metric is not selected.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully (measurement is not taken).
Outputs	None.
Comments	• When the *list update frequency* metric is not selected, the SVA Monitoring component should not be taking *vulnerability list age* measurements. • All operations executed as expected.

Test ID	test_send_list_age
Test objective	Test if the *vulnerability list age* measurement is sent to the Monitoring module (to the Event Hub) and the SVA Dashboard.
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• Both the SVA Dashboard and the Event Hub should be running or else test fails. • All operations executed as expected.

Test ID	test_send_repository_availability
Test objective	Test if the *repository availability* measurement is sent to the Monitoring module (to the Event Hub).
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• The Event Hub must be running or else test fails. • All operations executed as expected.

Test ID	test_get_basic_report_age_if_basic_scan_frequency_not_selected
Test objective	Test if the *basic scan report age* measurement is taken in case when the *scanning frequency – basic scan* metric is not selected.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully (measurement is not taken).
Outputs	None.
Comments	• When the *scanning frequency – basic scan* metric is not selected, the SVA Monitoring component should not be taking *basic scan report age* measurements. • All operations executed as expected.

Test ID	test_send_basic_report_age
Test objective	Test if the *basic scan report age* measurement is sent to the Monitoring module (to the Event Hub) and the SVA Dashboard.
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• Both the SVA Dashboard and the Event Hub should be running or else test fails. • All operations executed as expected.

Test ID	test_send_list_availability
Test objective	Test if the *list availability* measurement is sent to the Monitoring module (to the Event Hub).
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• The Event Hub must be running or else test fails. • All operations executed as expected.

Test ID	test_send_scanner_availability
Test objective	Test if the *scanner availability* measurement is sent to the Monitoring module (to the Event Hub).
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• The Event Hub must be running or else test fails. • All operations executed as expected.

Test ID	test_get_up_report_age_if_up_report_frequency_not_selected
Test objective	Test if the *update/upgrade report age* measurement is taken in case when the *up report frequency* metric is not selected.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully (measurement is not taken).
Outputs	None.
Comments	• When the *up report frequency* metric is not selected, the SVA Monitoring component should not be taking *update/upgrade report age* measurements. • All operations executed as expected.

Test ID	test_send_up_report_age
Test objective	Test if the *update/upgrade report age* measurement is sent to the Monitoring module (to the Event Hub) and the SVA Dashboard.
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• Both the SVA Dashboard and the Event Hub should be running or else test fails. • All operations executed as expected.

Test ID	test_send_scan_report_availability
Test objective	Test if the *scan report availability* measurement is sent to the Monitoring module (to the Event Hub).
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• The Event Hub must be running or else test fails. • All operations executed as expected.

Test ID	test_send_up_report_availability
Test objective	Test if the *up report availability* measurement is sent to the Monitoring module (to the Event Hub).
Verified requirements	*ENF_SVA_R2*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	• The Event Hub must be running or else test fails. • All operations executed as expected.

Test ID	test_invoke_msr_repository_availability
Test objective	Test if the command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_invoke_msr_list_availability
Test objective	Test if the command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_invoke_msr_scanners_availability
Test objective	Test if the command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.

Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_invoke_msr_scan_report_availability
Test objective	Test if the command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_invoke_msr_up_report_availability
Test objective	Test if the command is executed without errors.
Verified requirements	*ENF_SVA_R1*
Inputs	None.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

7.2.2.3. SVA Dashboard

The following tests have been executed to test the code for the SVA Dashboard [11]. For further details about the component see D4.3.2.

Test ID	test_scan_report_post
Test objective	Test if the scan report is uploaded successfully.
Verified requirements	*ENF_SVA_R4*
Inputs	A test scan report.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_oval_report_post
Test objective	Test if the vulnerability list is uploaded successfully.
Verified requirements	*ENF_SVA_R4*
Inputs	A test vulnerability list.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_up_report_post
Test objective	Test if the update/upgrade report is uploaded successfully.
Verified requirements	*ENF_SVA_R4*
Inputs	A test up report.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_scan_report_available
Test objective	Test if the scan report is available in the database after the upload.
Verified requirements	*ENF_SVA_R4*
Inputs	A test scan report.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_vulnerability_list_available
Test objective	Test if the vulnerability list is available in the database after the upload.
Verified requirements	*ENF_SVA_R4*
Inputs	A test vulnerability list.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_up_report_available
Test objective	Test if the update/upgrade report is available in the database after the upload.
Verified requirements	*ENF_SVA_R4*
Inputs	A test up report.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_report_url
Test objective	Test if URL for a VM is available.
Verified requirements	*ENF_SVA_R4*
Inputs	A test URL for a VM.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_wrong_vm
Test objective	Test if the client is redirected to the index page after accessing a virtual machine, which is not in the database.
Verified requirements	*ENF_SVA_R4*
Inputs	A test URL for a VM not in the database.
Expected results	Redirected to index page.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_oval_file_url
Test objective	Test if the client is able to view the vulnerability list.
Verified requirements	*ENF_SVA_R4*
Inputs	A test vulnerability list.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_scanning_report_file_url
Test objective	Test if the client is able to view the scanning report.
Verified requirements	*ENF_SVA_R4*
Inputs	A test scan report.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_up_report_file_url
Test objective	Test if the client is able to view the update/upgrade report.
Verified requirements	*ENF_SVA_R4*
Inputs	A test update/upgrade report.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

8. Conclusions

This document finalizes the validation and testing methodologies adopted in the project. Although this particular deliverable is focused on the Enforcement module, all methodologies and technologies proposed here are to be used on the project level. Details will be provided in dedicated WP1, WP2, and WP3 prototype deliverables at M24 and M30.

With respect to the first iteration of this document, many improvements are reported. Apart from the

- complete and final testing and validation approach in terms of criticality assignment procedure, code quality assurance method, chosen test types, testing methodologies and technologies to be used, and
- security review procedure to be adopted,

some initial results are presented and discussed as well:

- Report of the code quality analysis is demonstrated for one main Enforcement component.
- Functional tests (unit and component tests) are demonstrated for one main Enforcement component and one security mechanism.

All other functional tests for Enforcement module are available at dedicated Bitbucket web sites [1]. Integration and system tests will be presented and discussed in T1.5 deliverable D1.5.2.

The non-functional characteristics of the Enforcement module will be evaluated and all associated tests will be reported in the final release of this deliverable, namely in D4.5.3 at the end of the project. The final iteration of this document will also summarize the validation of the entire Enforcement module and outcomes of the security review.

9. Bibliography

[1] SPECS, "*SPECS Team*", 2015. [Online]. Available: https://bitbucket.org/specs-team/.

[2] Atlassian, "*Use the issue tracker*", 2015. [Online]. Available: https://confluence.atlassian.com/bitbucket/use-the-issue-tracker.

[3] SonarSource, "*SonarQube*", 2015. [Online]. Available: http://www.sonarqube.org/.

[4] SonarSource, "*Metric definitions*", 2015. [Online]. Available: http://docs.sonarqube.org/display/SONAR/Metric+definitions.

[5] T. J. McCabe, "*A complexity Measure*", IEEE Transactions on Software Engineering, SE-2(4):308-320, 1976.

[6] SonarSource, "*Technical Debt Evaluation*", 2015. [Online]. Available: http://www.sonarsource.com/products/plugins/governance/sqale/.

[7] Atlassian, "*Meet Jenkins*", 2015. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins.

[8] Mountainminds GmbH & Co. KG and Contributors, "*JaCoCo Java Code Coverage Library*", 2015. [Online]. Available: http://www.eclemma.org/jacoco/.

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

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

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

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

[13] Tom Akehurst, "*WireMock*", 2015. [Online]. Available: http://wiremock.org/.

[14] Szczepan Faber, "*Mockito*", 2015. [Online]. Available: http://mockito.org/.

[15] Fongo Inc., "*Fongo*", 2015. [Online]. Available: https://www.fongo.com/.

[16] JUnit, "*JUnit*", 2015. [Online]. Available: http://junit.org/.

[17] Pivotal Software, "Spring Framework – *Class MockMvc*", 2015. [Online]. Available: http://docs.spring.io/spring-framework/docs/3.2.0.RC2/api/org/springframework/test/web/servlet/MockMvc.html.

[18] Atlassian, "*Bamboo*", 2015. [Online]. Available: https://www.atlassian.com/software/bamboo.

[19] Steve Purcell, "*PyUnit*", 2015. [Online]. Available: http://pyunit.sourceforge.net/pyunit.html.

[20] J. Luna., N. Suri, G. Pellegrino, H. Zhang, M. Bladt Stausholm, "*D4.3 Final Perturbation Analysis of the Implementation*". ABC4Trust project, July 2014.

[21] J. M. Vaas, K. W. Miller, "*Software Teastability: The New Verification*". Proc. of IEEE Software, vol. 12, no. 3, pp. 17—28, May 1995.

[22] L. Nik, M. Munro, J. Xu, "*Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection*". Proc. of Intl. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), 2003.

[23] L. Nik, M. Munro, J. Xu, "*Simulating errors in web services*". International Journal of Simulation Systems, Science & Technology, pp. 29—37, 2004.

[24] P. Koopman, et al., "*Comparing operating systems using robustness benchmarks*". Proc. of the Symposium on Reliable Distributed Systems (SRDS), pp. 72—79, 1997.

[25] S. Winter, C. Sârbu, N. Suri, et al., "*The impact of fault models on software robustness evaluations*". Software Engineering (ICSE 2011), 33rd International Conference on Software Engineering, pp. 51—60, 2011.

[26] S. Winter, N. Suri, et al., "*simFI: From single to simultaneous software fault injections*". 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2013.

[27] S. Winter, N. Suri, et al., "*No PAIN, No Gain? The Utility of PArallel Fault INjections*". In Proceedings of the 37th International Conference on Software Engineering (ICSE) (to appear), 2015.

[28] IBM Knowledge Center, "*Considerations for JSON parser limits*", 2015. [Online]. Available: https://www-01.ibm.com/support/knowledgecenter/SS9H2Y_7.1.0/com.ibm.dp.doc/json_parserlimits.html.

[29] Cloud Security Alliance "*Cloud Control Matrix version 3.0.1*", [Online]. Available: https://cloudsecurityalliance.org/download/cloud-controls-matrix-v3-0-1/.

[30] The Common Criteria Recognition Agreement Members. "*Common criteria for information technology security evaluation*", 2006. [Online]. Available: http://www.commoncriteriaportal.org/.

[31] The Open Web Application Security Project, "*Application Security Verification Standard (2014)*", 2014. [Online]. Available: https://www.owasp.org/images/5/58/OWASP_ASVS_Version_2.pdf.

[32] Outercurve Foundation, "*xUnit.net*", 2015. [Online]. Available: http://xunit.github.io/.

[33] Microsoft, "*Security Threats*", 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/cc723507.aspx.

Appendix 1. Requirements associated to the Enforcement module

The following table presents a list of all requirements associated to the Enforcement module.

REQ_ID	Requirement	Description
ENF_PLAN_R1	***Get SLA to enforce***	The Planning component must be able to retrieve and parse the SLA to implement, by invoking proper functionalities offered by the Platform.
ENF_PLAN_R2	***Define security mechanisms related to SLOs***	The Planning component must be able to determine which kind of security mechanisms are to be applied, given a set of high-level SLOs contained in the SLA to implement.
ENF_PLAN_R3	***Get security components***	The Planning component must be able to retrieve the available Enforcement security components that implement the security mechanisms related to the fulfilment of the SLOs defined in the SLA to implement.
ENF_PLAN_R4	***Select best security components***	Based on the selected target service and on the negotiated SLA, the Planning component must be able to select the best available Enforcement components to invoke, among different technology stacks, in order to meet the SLOs defined in the SLA.
ENF_PLAN_R5	***Activate implementation***	The Planning component must be able to activate the selected plan, by properly invoking the Implementation component.
ENF_PLAN_R6	***Log component activation and deactivation***	The Planning component must be able to report about its activation or deactivation for accountability purposes.
ENF_PLAN_R7	***Build an implementation plan***	After the set of high-level SLOs specified in an SLA have been correlated to the appropriate security mechanisms and the best associated security components have been retrieved, the Planning component must be able to prepare an implementation plan. Building implementation plan includes deducing alert thresholds.
ENF_PLAN_R8	***Build a reaction plan***	The Planning component must be able to plan the actual activation of the redressing technique selected by the Remediation Decision System component. This may include different strategies (e.g., the definition of a chain of service invocations or the activation of a new configuration of a running service).
ENF_PLAN_R9	***Build a migration plan***	The Planning component must be able to plan the strategy to migrate from the target service currently being delivered to the new version of it, if this is a part of a redressing technique chosen by the Remediation Decision System component.
ENF_PLAN_R10	***Get monitoring systems***	The Planning component must be able to retrieve a list of available monitoring systems/agents, associated to security components that fulfil the requirements of the SLA.
ENF_PLAN_R11	***Select best monitoring systems***	The Planning component must be able to select the appropriate monitoring systems/agents that will monitor metrics/SLOs specified in the SLA.
ENF_PLAN_R12	***Validate an SLA***	The Planning component has to be able to validate an SLA

		by verifying that it can be enforced (*ENF_PLAN_R1*).
ENF_IMPL_R1	***Implement Plan***	The Implementation component must be able to actually realize the plan built by the Planning component, by orchestrating the acquisition of the needed resources, their configuration, and the activation of involved services.
ENF_IMPL_R2	***Acquire resources***	The Implementation component must be able to acquire all the resources needed, based on the plan built by the Planning component.
ENF_IMPL_R3	***Deploy and configure***	The Implementation component must be able to deploy and configure all the resources, based on the plan built by the Planning component.
ENF_IMPL_R4	***Start services***	The Implementation component must be able to properly start the needed services on top of the acquired resources, in order to build the plan.
ENF_IMPL_R5	***Trigger monitoring system agent activation or deactivation***	The Implementation component must be able to trigger activation/deactivation or reconfiguration of the appropriate monitoring agents by accessing the functionalities offered by the Platform.
ENF_IMPL_R6	***Log service activation***	The Implementation component must be able to log a successful activation of each security service related to a certain SLO in an SLA.
ENF_IMPL_R7	***Update SLA state***	The Implementation component must be able to update the state of an SLA after its successful implementation.
ENF_IMPL_R8	***Log component activation or deactivation***	The Implementation component must be able to report about its activation or deactivation for accountability purposes.
ENF_IMPL_R9	***Implement reaction plan***	The Implementation component must be able to apply the reaction and migration plans previously defined in the reaction plan.
ENF_IMPL_R10	***Update monitoring policy***	The Implementation component must be able to update the monitoring policy according to each signed SLA.
ENF_DIAG_R1	***Get monitoring event notification***	The Diagnosis component must be able to receive notifications from the Platform about monitoring events captured by the Monitoring module.
ENF_DIAG_R2	***Get monitoring event information***	The Diagnosis component must be able to retrieve all information, related to a monitoring event notified through the Platform, by accessing the Auditing component.
ENF_DIAG_R3	***Identify SLOs affected by a monitoring event***	The Diagnosis component must be able to identify the SLOs at risk or violated by processing a monitoring event that has been notified by the Platform.
ENF_DIAG_R4	***Update SLA state***	The Diagnosis component must be able to update the state of an SLA by accessing the proper functionalities offered by the Platform.
ENF_DIAG_R5	***Get SLAs affected by a monitoring event***	Given a monitoring event which has been notified by the Platform, the Diagnosis component must be able to retrieve all SLAs affected by such an event.
ENF_DIAG_R6	***Activate reaction***	The Diagnosis component must be able to activate the Remediation System component to react to an alert or a violation and find the best redressing techniques or remediation actions, respectively.

ENF_DIAG_R7	***Express SLA violation in terms of KPI***	The Diagnosis component must express the SLA violation detection in terms of KPI rules.
ENF_DIAG_R8	***Query metric***	The Diagnosis component can query the metric data stored inside the monitoring results repository in the Platform.
ENF_DIAG_R9	***Log component activation or deactivation***	The Diagnosis component must be able to log its activation or deactivation for accountability purposes.
ENF_DIAG_R10	***Determine effect on an SLA***	For each SLA affected by a monitoring event, the Diagnosis component must be able to determine the effect the monitoring event has on the SLA (i.e., is it alerted or violated).
ENF_DIAG_R11	***Log SLA impact***	When all SLOs affected by a monitoring event are identified, and the severity of the impact of the monitoring event has been determined, the Diagnosis component must be able to log this information.
ENF_DIAG_R12	***Classify event***	The Diagnosis component must be able to classify a monitoring event with regard to each affected SLA, based on the information provided by the Monitoring component and the affected SLOs and SLAs.
ENF_DIAG_R13	***Identify root cause***	The Diagnosis component must be able to perform a root cause analysis of each monitoring event that causes alerts or violations of one or more monitored SLAs.
ENF_DIAG_R14	***Log root cause***	The Diagnosis component must be able to log the information about the root cause of a monitoring event.
ENF_DIAG_R15	***Analyse monitoring event***	The Diagnosis component must be able to analyse each monitoring event related to an alert or a violation of one or more monitored SLAs.
ENF_DIAG_R16	***Prioritize events***	After the impact of a monitoring event on each of the affected SLAs is known and the root cause of the monitoring event is identified, the Diagnosis component must be able to create a priority queue.
ENF_DIAG_R17	***Log priority queue***	The Diagnosis component must be able to log the information about the priority queue.
ENF_DIAG_R18	***Verify SLA state***	The Diagnosis component must be able to compare the current metric/SLO data with the alert/violation thresholds specified for an alerted/violated SLA to verify if the severity of the alert/violation has changed.
ENF_REM_R1	***Trigger renegotiation***	The Remediation Decision System component will provide a mechanism to trigger renegotiation activities, by accessing the proper Platform functionalities.
ENF_REM_R2	***Log component activation or deactivation***	The Remediation Decision System component must be able to log its activation or deactivation.
ENF_REM_R3	***Get SLA state***	The Remediation Decision System component must be able to check the state of an SLA in order to react either to an alert or a violation.
ENF_REM_R4	***Update SLA state***	In the process of reacting to an event, the Remediation Decision System component must be able to update SLA's state.
ENF_REM_R5	***Get SLA***	The Remediation System Component must be able to

		retrieve an SLA.
ENF_REM_R6	***Get SLA impact***	The Remediation Decision System component must be able to retrieve information about the impact of a monitoring event to an affected SLA, provided by the Diagnosis component through the Auditing component.
ENF_REM_R7	***Get security components***	In the process of searching for the best actions to apply in order to mitigate the risk of having a violation or to recover from a violation, the Remediation Decision System component must be able to retrieve all relevant security components.
ENF_REM_R8	***Search for redressing techniques***	Based on the event information, associated SLAs and security mechanisms available, the Remediation Decision System component must be able to find redressing techniques to invoke in case of an alert or a violation.
ENF_REM_R9	***Notify End-user***	When End-user's decision is needed in the process of managing an alert or a violation, the Remediation Decision System component must be able to communicate the issue with the End-user through the SPECS Application.
ENF_AUD_R1	***Create log***	The Auditing component must be able to create different types of logs (e.g., activation/deactivation of a component, service activation, priority queue, etc.).
ENF_AUD_R2	***Query log***	The Auditing component must enable all components of the SPECS framework to query for different types of logs (retrieving logs from the database using different search criteria).
ENF_AUD_R3	***Support different communication technologies***	The Auditing component has to support different (software) communication technologies (REST, Thrift, etc.).
ENF_AUD_R4	***Support log correlation***	The Auditing component has to support correlation among different logs, i.e. consolidate logs created due to the same request or event into a workflow.
ENF_AUD_R5	***Support different databases***	The Auditing component has to support the use of different databases.
ENF_BROKER_R1	***Enable CSP***	The SPECS Administrator must be able to configure and enable the Broker to access and use an external CSP.
ENF_BROKER_R2	***Acquire cluster***	The Broker component must be able to acquire a cluster of VMs on one of the enabled CSPs.
ENF_BROKER_R3	***Delete cluster***	The Broker component must be able to delete a cluster of VMs.
ENF_BROKER_R4	***Add user***	The Broker component must be able to add a new user to the available cluster of VMs.
ENF_BROKER_R5	***Execute script on node***	The Broker component must enable the execution of scripts on a cluster of VMs.
ENF_POOL_R1	***Diversity***	A minimum (with respect to End-user's requirements and technological constraints) *Level of Diversity* must be ensured, through the availability of a *pool* of different web server engines for hosting End-user's applications.
ENF_POOL_R2	***Load balancing***	Load balancing features should be provided, to enable the distribution of the workload generated by the End-user's web applications across multiple servers.
ENF_POOL_R3	***Survivability***	A minimum (with respect to End-user's requirements

		and technological constraints) *Level of Redundancy* must be ensured: in case some web containers become unavailable, the End-user's web application shall still run on the other web containers belonging to the *pool*. If all web containers in a *pool* fail, the End-user's web application will become unavailable until at least one of those web containers become healthy again.
ENF_POOL_R4	***Session sharing***	All web containers belonging to a *pool* must be able to access the shared session variables saved into a distributed caching system. This ensures session data sharing among different web servers. Also this system part exploits the advantages of replication.
ENF_POOL_R5	***Incident management***	Incident management features must be provided, enabled by the interaction with the SPECS Monitoring module and the Enforcement components, and consisting in isolating the VMs affected/targeted by some incident while ensuring business continuity to the End-user.
ENF_TLS_R1	***Translate TLS constraints***	Based on high-level constraints and requirements, the TLS component must be able to generate technology independent configuration parameters.
ENF_TLS_R2	***Verify TLS constraints***	The TLS component must be able to verify that the high-level constraints and requirements are valid and not contradictory.
ENF_TLS_R3	***Instantiate TLS configuration***	Based on technology independent configuration parameters, the TLS component must be able to generate technology dependent parameters, ready for deployment.
ENF_TLS_R4	***Deploy TLS configuration***	Taking as input the technology dependent configuration parameters, the TLS component must be able to configure a target server.
ENF_TLS_R5	***Probe TLS endpoint configuration***	The TLS component must be able to periodically check the actual exposed parameters by a TLS endpoint.
ENF_SVA_R1	***Detect vulnerabilities and misconfigurations***	Software modules/libraries that should be upgraded to resolve known issues in older versions of the monitored software, as well as misconfigurations enabling known vector attacks, must be detected.
ENF_SVA_R2	***Report vulnerabilities and misconfigurations***	Software modules/libraries that need an upgrade and any detected misconfigurations must be reported to the Platform.
ENF_SVA_R3	***Upgrade libraries and fix misconfigurations***	Upgrade or reconfiguration of the vulnerable libraries must be supported.
ENF_SVA_R4	***Visualize detected vulnerabilities and misconfigurations***	A dashboard for the visualization of detected vulnerabilities and misconfigurations as well as of the policies and rules defined by the Enforcement module must be provided.
ENF_CRYPTO_R1	***Provide client-side encryption tool as a plugin/extension***	The mechanism must provide client-side encryption in the form of a plugin/extension to download and add to the browser, in order to avoid MITM attacks. It needs to be provided as a plugin or extension (Chrome) to avoid modifications of the tool when it is being transferred to the user's machine.
ENF_CRYPTO_R2	***Configure and***	Encryption tools must be configurable. They should

	deploy encryption tools	support asymmetric/symmetric encryption, different key management techniques, file sharing etc.
ENF_CRYPTO_R3	***Encrypt data***	The mechanism should enable encryption of files – either locally (end-to-end) or on server (depending on the security requirements).
ENF_CRYPTO_R4	***Decrypt data***	The mechanism should enable decryption of files.
ENF_AAA_R1	***Support different authentication sources***	The AAA mechanism should support different authentication sources, i.e., internal/external software components providing authentication services (e.g., LDAP server, DB, social networks).
ENF_AAA_R2	***Manage different accounts for a user***	In case of multiple supported authentication sources, the AAA mechanism must properly manage the different accounts associated to an End-user (for example, via a federation identity).
ENF_AAA_R3	***Link different identities to a single account***	The AAA mechanism must allow an End-user to create a personal account on the target system, and to associate one or more external identities to this account.
ENF_AAA_R4	***Login***	The AAA mechanism must allow End-users owning a valid account to login with such account or with any of the other identities associated with it.
ENF_AAA_R5	***Authenticate***	The AAA mechanism must apply access control policies to an End-user, whenever it invokes a service provided by the target system.
ENF_AAA_R6	***Dynamically manage access control policies***	The AAA mechanism must envision a dynamic management of access control policies carried out by an administrator.
ENF_AAA_R7	***Logout***	The AAA mechanism must provide a user with the capability of logging out of a target system.
ENF_AAA_R8	***Authentication and authorization independency***	The AAA mechanism must include authentication and authorization modules which are independent one from the other and can be configured dynamically.
ENF_AAA_R9	***Confidentiality and integrity***	The AAA mechanism itself must be protected from external compromise.
ENF_CRED_R1	***Target service authentication schemes support***	The Credential Service mechanism needs to implement the relevant part of the chosen target service authentication schemes, namely the one involving the usage of credentials.
ENF_CRED_R2	***Access control policies to the credentials usage***	The Credential Service mechanism needs to provide support for limiting the usage of certain credentials to a well-defined set of clients.
ENF_CRED_R3	***Multiple credentials for the same target service***	The Credential Service mechanism needs to allow the concurrent usage of different credentials for the same target service.
ENF_CRED_R4	***Credentials usage auditing***	The Credential Service mechanism needs to provide credentials usage auditing. It needs to provide enough information to identify the software component requesting access.
ENF_CRED_R5	***Disjoint credentials data management and storage***	The Credential Service mechanism needs to separate the actual credentials usage from their long term storage, in order to, possibly, prevent or reduce losses and risks in case of successful attack on any of the credential service

		components.
ENF_TOK_R1	***Support offline token validation***	The offline validation of tokens without calling the central validation service must be supported to provide scalability: if every API request would require calling central validation service, the validation service might become a serious bottleneck.
ENF_TOK_R2	***Send tokens in HTTP header***	Tokens must be small enough to fit into HTTP header. Indeed, putting the token in request body might be a problem for methods that expect for example MIME type image/jpg. Also, putting the token for example in query string is not fully secure.
ENF_TOK_R3	***Obtain security tokens issued by a centralized service***	A centralized service that issues security tokens for subjects and maintains the token revocation list must be provided. The security tokens must contain the claims about the specified subject and must be stored in a proper database. A client application that wants to invoke a REST API adopting tokens for authentication/authorization must be able to obtain, if provided with valid credentials, the tokens to send with the request.
ENF_TOK_R4	***Request, parse and validate tokens***	A client for requesting, parsing and validating tokens must be provided. It must enable offline validation of token signatures and must be able to determine whether a token has been revoked.
ENF_TOK_R5	***Revoke tokens***	Tokens revocation must be supported if needed. It may be carried out by marking the specified token in the database as revoked and by storing its revocation date. The revoked token remains in the database till the token expiration.
ENF_TOK_R6	***Generate token revocation lists***	The possibility of retrieving all revoked tokens from the database must be supported, to enable the generation of token revocation lists used to validate requests.
ENF_TOK_R7	***Sign certificates***	The functionalities needed to sign a certificate (i.e. to digitally sign a token) must be provided.
ENF_TOK_R8	***Decode tokens***	Proper functionalities to decode tokens and retrieve the required information from them must be provided.
ENF_TOK_R9	***Determine access rights according to SLA***	Security Tokens mechanism has to be able to determine access rights according to SLAs.
ENF_DOS_R1	***Detect DoS attack***	DoS attack detection features must be provided.
ENF_DOS_R2	***Classify detected DoS attacks***	Detected DoS attacks must be correctly classified: there are numerous DoS attack types based on consumption of computational resources, disruption of configuration, obstructing the communication media, etc.
ENF_DOS_R3	***Mitigate DoS attacks***	Mitigation functionalities must be provided. Note that mitigation depends on type of attack (e.g., filters may be used to block illegitimate traffic, using reverse proxies).
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.
SLANEG_R30	***Remediation***	Enforcement should consider the renegotiation of an

	through SLA renegotiation	existing SLA as a potential remedy to apply in case of alerts and violations.
SLANEG_R31	***Alerts/violations affecting multiple elements of the secure SLA hierarchy***	A detected alert/violation might affect more than one element of the SPECS security SLA hierarchy. Enforcement should consider interrelationships along SLA elements to choose the optimal redressing technique (e.g., renegotiation might help to manage multiple alerts/violations).

Appendix 2. Secure web application checklist

Derived from OWASP Application Security Verification Standard 2.0 (2014).

a. Authentication Verification Requirements

ID	Requirement
SC1	Verify all resources require authentication except those specifically intended to be public (Principle of complete mediation).
SC2	Verify all authentication controls are enforced on the server side.
SC3	Verify all authentication controls (including libraries that call external authentication services) have a centralized implementation.
SC4	Verify all authentication controls fail securely to ensure attackers cannot log in.
SC5	Verify all account identity authentication functions (such as registration, update profile, forgot username, forgot password, disabled / lost token, help desk or IVR) that might regain access to the account are at least as resistant to attack as the primary authentication mechanism.
SC6	Verify users can safely change their credentials using a mechanism that is at least as resistant to attack as the primary authentication mechanism.
SC7	Verify that all authentication decisions are logged. This should include requests with missing required information, needed for security investigations.
SC8	Verify that account passwords are salted using a salt that is unique to that account (e.g., internal user ID, account creation) and use bcrypt, scrypt or PBKDF2 before storing the password.
SC9	Verify that credentials, and all other identity information handled by the application(s), do not traverse unencrypted or weakly encrypted links.
SC10	Verify that the forgotten password function and other recovery paths do not reveal the current password and that the new password is not sent in clear text to the user.
SC11	Verify that username enumeration is not possible via login, password reset, or forgot account functionality.
SC12	Verify there are no default passwords in use for the application framework or any components used by the application (such as “admin/password”).
SC13	Verify that a resource governor is in place to protect against vertical (a single account tested against all possible passwords) and horizontal brute forcing (all accounts tested with the same password e.g. “Password1”). A correct credential entry should incur no delay. Both these governor mechanisms should be active simultaneously to protect against diagonal and distributed attacks.
SC14	Verify that all authentication credentials for accessing services external to the application are encrypted and stored in a protected location (not in source code).

SC15	Verify that forgot password and other recovery paths send a link including a time-limited activation token rather than the password itself. Additional authentication based on soft-tokens (e.g. SMS token, native mobile applications, etc.) can be required as well before the link is sent over.
SC16	Verify that forgot password functionality does not lock or otherwise disable the account until after the user has successfully changed their password. This is to prevent valid users from being locked out.
SC17	Verify that there are no shared knowledge questions/answers (so called "secret" questions and answers).

b. Access Control Verification Requirements

ID	Requirement
SC18	Verify that users can only access secured functions or services for which they possess specific authorization.
SC19	Verify that users can only access secured URLs for which they possess specific authorization.
SC20	Verify that users can only access secured data files for which they possess specific authorization.
SC21	Verify that direct object references are protected, such that only authorized objects or data are accessible to each user (for example, protect against direct object reference tampering).
SC22	Verify that access controls fail securely.
SC23	Verify that all user and data attributes and policy information used by access controls cannot be manipulated by end users unless specifically authorized.
SC24	Verify that all access controls are enforced on the server side.
SC25	Verify that there is a centralized mechanism (including libraries that call external authorization services) for protecting access to each type of protected resource.
SC26	Verify that all access control decisions are logged and all failed decisions are logged.
SC27	Aggregate access control protection – verify the system can protect against aggregate or continuous access of secured functions, resources, or data. For example, possibly by the use of a resource governor to limit the number of edits per hour or to prevent the entire database from being scraped by an individual user.

c. Malicious Input Handling Verification Requirements

ID	Requirement
SC28	Verify that the runtime environment is not susceptible to buffer overflows, or that security controls prevent buffer overflows.
SC29	Verify that all input validation failures result in input rejection.
SC30	Verify that a character set, such as UTF-8, is specified for all sources of input.
SC31	Verify that all input validation or encoding routines are performed and enforced on the server side.
SC32	Verify that a single input validation control is used by the application for each type of data that is accepted.
SC33	Verify that all input validation failures are logged.
SC34	Verify that all input data is canonicalized for all downstream decoders or interpreters prior to validation.
SC35	Verify that the runtime environment is not susceptible to SQL Injection, or that security controls prevent SQL Injection.
SC36	Verify that the runtime environment is not susceptible to OS Command Injection, or that security controls prevent OS Command Injection.
SC37	Verify that the runtime environment is not susceptible to XML External Entity attacks or that security controls prevents XML External Entity attacks.
SC38	Verify that the runtime environment is not susceptible to XML Injections or that security controls prevents XML Injections.
SC39	If the application framework allows automatic mass parameter assignment (also called automatic variable binding) from the inbound request to a model, verify that security sensitive fields such as “accountBalance”, “role” or “password” are protected from malicious automatic binding.
SC40	Verify that for each type of output encoding/escaping performed by the application, there is a single security control for that type of output for the intended destination.

d. Cryptography at Rest Verification Requirements

ID	Requirement
SC41	Verify that all cryptographic functions used to protect secrets from the application user are implemented server side.
SC42	Verify that all cryptographic modules fail securely.

SC43	Verify that access to any master secret(s) is protected from unauthorized access (A master secret is an application credential stored as plaintext on disk that is used to protect access to security configuration information).
SC44	Verify that all random numbers, random file names, random GUIDs, and random strings are generated using the cryptographic module's approved random number generator when these random values are intended to be unguessable by an attacker.
SC45	Verify that cryptographic modules used by the application have been validated against FIPS 140-2 or an equivalent standard.
SC46	Verify that cryptographic modules operate in their approved mode according to their published security policies.
SC47	Verify that there is an explicit policy for how cryptographic keys are managed (e.g., generated, distributed, revoked, expired). Verify that this policy is properly enforced.

e. Error Handling and Logging Verification Requirements

ID	Requirement
SC48	Verify that the application does not output error messages or stack traces containing sensitive data that could assist an attacker, including session id and personal information.
SC49	Verify that all error handling is performed on trusted devices
SC50	Verify that all logging controls are implemented on the server.
SC51	Verify that error handling logic in security controls denies access by default.
SC52	Verify security logging controls provide the ability to log both success and failure events that are identified as security-relevant.
SC53	Verify that each log event includes: a timestamp from a reliable source, the severity level of the event, an indication that this is a security relevant event (if mixed with other logs), the identity of the user that caused the event (if there is a user associated with the event), the source IP address of the request associated with the event, whether the event succeeded or failed, and a description of the event.
SC54	Verify that all events that include untrusted data will not execute as code in the intended log viewing software.
SC55	Verify that security logs are protected from unauthorized access and modification.
SC56	Verify that there is a single application-level logging implementation that is used by the software.
SC57	Verify that the application does not log application-specific sensitive data that could assist an attacker, including user's session identifiers and personal or sensitive

	information. The length and existence of sensitive data can be logged.
SC58	Verify that a log analysis tool is available which allows the analyst to search for log events based on combinations of search criteria across all fields in the log record format supported by this system.
SC59	Verify that all non-printable symbols and field separators are properly encoded in log entries, to prevent log injection.
SC60	Verify that log fields from trusted and untrusted sources are distinguishable in log entries.
SC61	Verify that logging is performed before executing the transaction. If logging was unsuccessful (e.g. disk full, insufficient permissions) the application fails safe. This is for when integrity and non-repudiation are a must.

f. Data Protection Verification Requirements

ID	Requirement
SC62	Verify that the list of sensitive data processed by this application is identified, and that there is an explicit policy for how access to this data must be controlled, and when this data must be encrypted (both at rest and in transit). Verify that this policy is properly enforced.
SC63	Verify that all sensitive data is sent to the server in the HTTP message body (i.e., URL parameters are never used to send sensitive data).
SC64	Verify that all cached or temporary copies of sensitive data stored on the server are protected from unauthorized access or purged/invalidated after the authorized user accesses the sensitive data.
SC65	Verify that there is a method to remove each type of sensitive data from the application at the end of its required retention period.
SC66	Verify the application has the ability to detect and alert on abnormal numbers of requests for information or processing high value transactions for that user role, such as screen scraping, automated use of web service extraction, or data loss prevention. For example, the average user should not be able to access more than 5 records per hour or 30 records per day, or add 10 friends to a social network per minute.

g. Communications Security Verification Requirements

ID	Requirement
SC67	Verify that a path can be built from a trusted CA to each Transport Layer Security (TLS) server certificate, and that each server certificate is valid.
SC68	Verify that failed TLS connections do not fall back to an insecure HTTP connection.

SC69	Verify that TLS is used for all connections (including both external and backend connections) that are authenticated or that involve sensitive data or functions.
SC70	Verify that backend TLS connection failures are logged.
SC71	Verify that certificate paths are built and verified for all client certificates using configured trust anchors and revocation information.
SC72	Verify that all connections to external systems that involve sensitive information or functions are authenticated.
SC73	Verify that all connections to external systems that involve sensitive information or functions use an account that has been set up to have the minimum privileges necessary for the application to function properly.
SC74	Verify that there is a single standard TLS implementation that is used by the application that is configured to operate in an approved mode of operation (See http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf).
SC75	Verify that specific character encodings are defined for all connections (e.g., UTF-8).

h. HTTP Security Verification Requirements

ID	Requirement
SC76	Verify that the application accepts only a defined set of HTTP request methods, such as GET and POST and unused methods are explicitly blocked.
SC77	Verify that every HTTP response contains a content type header specifying a safe character set (e.g., UTF-8).
SC78	Verify that HTTP headers in both requests and responses contain only printable ASCII characters.
SC79	Verify that HTTP headers added by a frontend (such as X-Real-IP), and used by the application, cannot be spoofed by the end user.
SC80	Verify that the HTTP headers do not expose detailed version information of system components.

i. Malicious Controls Verification Requirements

ID	Requirement
SC81	Verify that no malicious code is in any code that was either developed or modified in order to create the application.
SC82	Verify that the integrity of interpreted code, libraries, executables, and configuration files is verified using checksums or hashes.

SC83	Verify that all code implementing or using authentication controls is not affected by any malicious code.
SC84	Verify that all code implementing or using access controls is not affected by any malicious code.
SC85	Verify that all input validation controls are not affected by any malicious code.
SC86	Verify that all code implementing or using output validation controls is not affected by any malicious code.
SC87	Verify that all code supporting or using a cryptographic module is not affected by any malicious code.
SC88	Verify that all code implementing or using error handling and logging controls is not affected by any malicious code.
SC89	Verify all malicious activity is adequately sandboxed.
SC90	Verify that sensitive data is rapidly sanitized from memory as soon as it is no longer needed and handled in accordance to functions and techniques supported by the framework/library/operating system.

j. Business Logic Verification Requirements

ID	Requirement
SC91	Verify the application processes or verifies all high value business logic flows in a trusted environment, such as on a protected and monitored server.
SC92	Verify the application does not allow spoofed high value transactions, such as allowing Attacker User A to process a transaction as Victim User B by tampering with or replaying session, transaction state, transaction or user IDs.
SC93	Verify the application does not allow high value business logic parameters to be tampered with, such as (but not limited to): price, interest, discounts, PII, balances, stock IDs, etc.
SC94	Verify the application has defensive measures to protect against repudiation attacks, such as verifiable and protected transaction logs, audit trails or system logs, and in highest value systems real time monitoring of user activities and transactions for anomalies.
SC95	Verify the application protects against information disclosure attacks, such as direct object reference, tampering, session brute force or other attacks.
SC96	Verify the application has sufficient detection and governor controls to protect against brute force (such as continuously using a particular function) or denial of service attacks.
SC97	Verify the application has sufficient access controls to prevent elevation of privilege

	attacks, such as allowing anonymous users from accessing secured data or secured functions, or allowing users to access each other's details or using privileged functions.
SC98	Verify the application will only process business logic flows in sequential step order, with all steps being processed in realistic human time, and not process out of order, skipped steps, process steps from another user, or too quickly submitted transactions.
SC99	Verify the application has additional authorization (such as step up or adaptive authentication) for lower value systems, and / or segregation of duties for high value applications to enforce anti-fraud controls as per the risk of application and past fraud.
SC100	Verify the application has business limits and enforces them in a trusted location (as on a protected server) on a per user, per day or daily basis, with configurable alerting and automated reactions to automated or unusual attack. Examples include (but not limited to): ensuring new SIM users don't exceed $10 per day for a new phone account, a forum allowing more than 100 new users per day or preventing posts or private messages until the account has been verified, a health system should not allow a single doctor to access more patient records than they can reasonably treat in a day, or a small business finance system allowing more than 20 invoice payments or $1000 per day across all users. In all cases, the business limits and totals should be reasonable for the business concerned. The only unreasonable outcome is if there are no business limits, alerting or enforcement.

k. Files and Resources Verification Requirements

ID	Requirement
SC101	Verify that file names and path data obtained from untrusted sources is canonicalized to eliminate path traversal attacks.
SC102	Verify that files obtained from untrusted sources are scanned by antivirus scanners to prevent upload of known malicious content.
SC103	Verify that parameters obtained from untrusted sources are not used in manipulating filenames, pathnames or any file system object without first being canonicalized and input validated to prevent local file inclusion attacks.
SC104	Verify that parameters obtained from untrusted sources are canonicalized, input validated, and output encoded to prevent remote file inclusion attacks, particularly where input could be executed, such as header, source, or template inclusion
SC105	Verify that web or application server is configured by default to deny access to remote resources or systems outside the web or application server.
SC106	Verify the application code does not execute uploaded data obtained from untrusted sources.