D6.1 – TREDISEC FRAMEWORK IMPLEMENTATION

<table>
<thead>
<tr>
<th>Work Package</th>
<th>WP 6, Development, delivery and evaluation of the TREDISEC framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lead Author (Org)</td>
<td>José F. Ruiz (ATOS)</td>
</tr>
<tr>
<td>Contributing Author(s) (Org)</td>
<td>Carlos H. Arce (ATOS), Dimitris Mitropoulos (GRNET), Wenting Li (NEC)</td>
</tr>
<tr>
<td>Reviewers</td>
<td>David Vallejo (ARSYS), Andreas Fischer (SAP)</td>
</tr>
<tr>
<td>Due Date</td>
<td>30.09.2017 (M30)</td>
</tr>
<tr>
<td>Date</td>
<td>20.10.2017</td>
</tr>
<tr>
<td>Version</td>
<td>1.0 FINAL</td>
</tr>
</tbody>
</table>

Dissemination Level

X PU: Public
CO: Confidential, only for members of the consortium (including the Commission)

The work described in this document has been conducted within the project TREDISEC. This project has received funding from the European Union's Horizon 2020 (H2020) research and innovation programme under the Grant Agreement no 644412. This document does not represent the opinion of the European Union, and the European Union is not responsible for any use that might be made of its content.
## Versioning and contribution history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>16.06.2017</td>
<td>Jose Francisco Ruiz (ATOS)</td>
<td>Definition of the table of contents</td>
</tr>
<tr>
<td>0.2</td>
<td>03.07.2017</td>
<td>Jose Francisco Ruiz (ATOS)</td>
<td>Extending sections of the deliverable</td>
</tr>
<tr>
<td>0.3</td>
<td>17.07.2017</td>
<td>Wenting Li (NEC)</td>
<td>Contribution to section 3</td>
</tr>
<tr>
<td>0.4</td>
<td>28.07.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Contribution to section 3</td>
</tr>
<tr>
<td>0.5</td>
<td>25.08.2017</td>
<td>Jose Francisco Ruiz (ATOS)</td>
<td>Contribution to various sections of the document</td>
</tr>
<tr>
<td>0.6</td>
<td>29.08.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Review and comments in Sections 3-4</td>
</tr>
<tr>
<td>0.7</td>
<td>31.08.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Review and integration of contributions</td>
</tr>
<tr>
<td>0.8</td>
<td>01.09.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Review of the document and comments for contributors</td>
</tr>
<tr>
<td>0.9</td>
<td>04.09.2017</td>
<td>Jose Fran. Ruiz (ATOS), Dimitris Mitropoulos (GRNET)</td>
<td>Extension of Section 3, review of comments and refinement of processes</td>
</tr>
<tr>
<td>0.10</td>
<td>04.09.2017</td>
<td>Jose Fran. Ruiz (ATOS), Dimitris Mitropoulos (GRNET)</td>
<td>Contributions to Sections 3-4</td>
</tr>
<tr>
<td>0.11</td>
<td>04.09.2017</td>
<td>Jose Fran. Ruiz (ATOS), Dimitris Mitropoulos (GRNET)</td>
<td>Contribution to Section 2</td>
</tr>
<tr>
<td>0.12</td>
<td>04.09.2017</td>
<td>Jose Fran. Ruiz (ATOS), Dimitris Mitropoulos (GRNET)</td>
<td>Review of Section 3 and extension</td>
</tr>
<tr>
<td>0.13</td>
<td>05.09.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Contribution to Section 3</td>
</tr>
<tr>
<td>0.14</td>
<td>05.09.2017</td>
<td>Jose Francisco Ruiz (ATOS)</td>
<td>Generation of clean version</td>
</tr>
<tr>
<td>0.15</td>
<td>05.09.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Contribution to Section 3</td>
</tr>
<tr>
<td>0.16</td>
<td>06.09.2017</td>
<td>Jose Francisco Ruiz (ATOS)</td>
<td>Review and extended contributions</td>
</tr>
<tr>
<td>0.17</td>
<td>07.09.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Extending information to Section 4</td>
</tr>
<tr>
<td>0.18</td>
<td>11.09.2017</td>
<td>Jose Francisco Ruiz (ATOS)</td>
<td>Compiling updates of sub-sections</td>
</tr>
<tr>
<td>Release</td>
<td>Date</td>
<td>Author/Contributor</td>
<td>Comments</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>0.19</td>
<td>11.09.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Comments about installation/integration of TREDISEC Framework and TTE</td>
</tr>
<tr>
<td>0.20</td>
<td>12.09.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Refinement and extension of Section 4</td>
</tr>
<tr>
<td>0.21</td>
<td>13.09.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Integration of contributions and review of Section 3</td>
</tr>
<tr>
<td>0.22</td>
<td>14.09.2017</td>
<td>Wenting Li (NEC)</td>
<td>Extension of Section 4 and refinement</td>
</tr>
<tr>
<td>0.23</td>
<td>15.09.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Refinement and extension of contribution to Section 4</td>
</tr>
<tr>
<td>0.24</td>
<td>20.09.2017</td>
<td>Wenting Li (NEC)</td>
<td>Integration of contributions, review of document, refinement using comments of the reviewers</td>
</tr>
<tr>
<td>0.25</td>
<td>26.09.2017</td>
<td>Dimitris Mitropoulos (GRNET)</td>
<td>Refinement of Section 3 and 4</td>
</tr>
<tr>
<td>0.26</td>
<td>27.09.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Integration of final refinements and review</td>
</tr>
<tr>
<td>0.27</td>
<td>29.09.2017</td>
<td>Beatriz Gallego, Jose Fran. Ruiz (ATOS)</td>
<td>Review language and fixed comments of approval</td>
</tr>
<tr>
<td>0.28</td>
<td>29.09.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Integrated reviewed English and format of Sections 1-2</td>
</tr>
<tr>
<td>0.30</td>
<td>02.10.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Review Section 3</td>
</tr>
<tr>
<td>0.31</td>
<td>10.10.2017</td>
<td>Jose Fran. Ruiz, Beatriz Gallego (ATOS)</td>
<td>Integration of the review of language of Sections 3-4, Formatting and style reviewed, Integration of comments of David Vallejo (reviewer)</td>
</tr>
<tr>
<td>0.32</td>
<td>16.10.2017</td>
<td>Jose Fran. Ruiz (ATOS)</td>
<td>Integration of review/comments of David Vallejo</td>
</tr>
</tbody>
</table>
Disclaimer

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.
# Table of Contents

Executive Summary ........................................................................................................... 8  

Glossary ................................................................................................................................. 9  

1 Introduction ......................................................................................................................... 10  
   1.1 Purpose and scope ............................................................................................................. 10  
   1.2 Structure of the document ............................................................................................... 11  

2 Roles of the TREDISEC Framework .................................................................................... 12  
   2.1 Description ....................................................................................................................... 12  
   2.2 TREDISEC Admin ............................................................................................................. 12  
   2.3 Security Expert Engineer ................................................................................................. 12  
   2.4 Security Technology Provider ......................................................................................... 12  
   2.5 TREDISEC End-user ......................................................................................................... 13  

3 TREDISEC Framework ....................................................................................................... 14  
   3.1 Description ....................................................................................................................... 14  
   3.2 Access and Authorization ................................................................................................. 15  
   3.3 User List............................................................................................................................ 17  
   3.4 Tenant Management ........................................................................................................ 18  
   3.5 Testing Environment ....................................................................................................... 19  
   3.6 Security Primitives .......................................................................................................... 21  
      3.6.1 Description .................................................................................................................. 21  
      3.6.2 Security Primitive Pattern .......................................................................................... 22  
      3.6.3 Security Primitive Implementation ............................................................................. 28  
      3.6.4 TREDISEC Recipe ...................................................................................................... 33  
      3.6.5 Management of Deletion/Updating Artefacts .............................................................. 37  
   3.7 Search and Advanced Search ......................................................................................... 40  
   3.8 Security Primitives Packaging ....................................................................................... 41  
      3.8.1 Guideline of Artefact Packaging .............................................................................. 41  
      3.8.2 General Description of the Packages ....................................................................... 47  
   3.9 Testing Process ................................................................................................................ 50  
      3.9.1 Description .................................................................................................................. 50  
      3.9.2 Packaging ................................................................................................................... 50  
      3.9.3 Lifecycle ...................................................................................................................... 51  
      3.9.4 Technical Tests .......................................................................................................... 53  
      3.9.5 Performance Tests .................................................................................................... 53  
   3.10 Testing Environment ...................................................................................................... 54  
      3.10.1 Description ................................................................................................................ 54  
      3.10.2 Technical description ............................................................................................... 55  
      3.10.3 Management ............................................................................................................. 55  
   3.11 Deployment Process ....................................................................................................... 55
3.11.1 Description.................................................................................................55
3.11.2 Packaging.................................................................................................56
3.11.3 Lifecycle....................................................................................................56
3.11.4 Exchange of information between the TREDISEC Framework and a TTE ................56
4 TREDISEC Framework Technical Specifications ..................................................58
  4.1 Description.....................................................................................................58
  4.2 TREDISEC Framework deployment components ........................................58
  4.3 TREDISEC Framework Interfaces ................................................................59
  4.4 Data Storage ................................................................................................60
  4.5 TREDISEC Framework Development Environment .....................................62
    4.5.1 Django ....................................................................................................62
    4.5.2 Sphinx ....................................................................................................62
    4.5.3 APIMAS ..................................................................................................62
    4.5.4 Celery .....................................................................................................62
  4.6 TREDISEC Framework Testing: process and supporting tools .......................63
    4.6.1 Source code version management ..........................................................63
    4.6.2 Internal Testing .......................................................................................64
    4.6.3 Labelled issues .......................................................................................64
  4.7 How to install and configure your own TREDISEC Framework .....................65
    4.7.1 Technical requirements of the environment ..........................................65
    4.7.2 Installation instructions for the framework application ...........................66
    4.7.3 Redis/Celery ...........................................................................................67
5 Conclusions and Future Work .............................................................................69
6 Annexes ...............................................................................................................70
  Annex I: Examples of functional tests of EPICA TREDISEC Recipe .....................70
  Annex II: Categories JSON file of EPICA TREDISEC Recipe ..............................73
  Annex III: Creation process of a security primitive pattern ................................74
  Annex IV: Creation process of a security primitive implementation ..................81
  Annex V: Creation process of a TREDISEC Recipe ...........................................89

List of Tables

Table 1 Interfaces of the TREDISEC Framework ..................................................60
Table 2 Structure of storing of packaging in the database ....................................62

List of Figures

Figure 1 - Roles of TREDISEC ..........................................................................12
Figure 2: TREDISEC Framework High-level Description .......................................14
Figure 3: Log-in in the TREDISEC Framework ...................................................15
Figure 4: TREDISEC Framework main interface (home) as a user .........................16
Figure 5: TREDISEC Framework main interface (home) as an admin user ............16
Figure 6: User list ...............................................................................................17
Figure 7: Create new user interface ................................................................. 18
Figure 8: Information of tenants ................................................................. 19
Figure 9: List of cloud platforms of the TREDISEC Framework ......................... 20
Figure 10: Wizard for creating a new cloud platform .................................. 21
Figure 11: Security Primitives Concept ....................................................... 22
Figure 12 Accessing interface of Security Primitive Pattern ................................ 23
Figure 13 Listing of all security primitive patterns in the TREDISEC Framework ........ 23
Figure 14 Wizard for creating a security primitive pattern (step 1) ......................... 24
Figure 15 Wizard for creation of security primitive pattern (step 2) ...................... 25
Figure 16 Detailed information of a security primitive pattern .......................... 26
Figure 17 HTML-based documentation associated to a primitive pattern ......... 27
Figure 18 Interface for modifying a security primitive pattern .......................... 27
Figure 19 Access to the list of Security Primitive Implementations .................. 28
Figure 20 Listing of security primitive implementations .................................. 28
Figure 21 Selecting creation of a security primitive implementation ................. 29
Figure 22 Creation of a security primitive implementation (step 1) ................... 29
Figure 23: Creation of a security primitive implementation (step 2) ................... 30
Figure 24: Detailed information of a security primitive implementation (owned) .... 31
Figure 25 Interface for modifying a security primitive implementation .......... 32
Figure 26: Accessing list of TREDISEC Recipes ........................................ 33
Figure 27: List of TREDISEC Recipes ......................................................... 33
Figure 28: Wizard for creating a TREDISEC Recipe (1) .................................. 34
Figure 29: Wizard for creating a TREDISEC Recipe (step 2) ............................ 35
Figure 30: Detailed information of a TREDISEC Recipe and actions .................. 36
Figure 31: Interface for modifying a TREDISEC Recipe ............................... 37
Figure 32: Deletion of Artefacts ............................................................... 38
Figure 33: Updating of Artefacts ............................................................... 39
Figure 34 Advanced Search input form ......................................................... 40
Figure 35 Results of the Advanced Search .................................................. 41
Figure 36: Overview of a Security Primitive Pattern .................................... 49
Figure 37: Known issues of the Remote Attestation Primitive Implementation .... 50
Figure 38 Upload TREDISEC primitive package with "Testing" code and information .................................................. 51
Figure 39 Use of testing package, upload of results and visualization ............. 52
Figure 40 Testing tab functionalities of a TREDISEC Recipe in the TREDISEC Framework .................................................. 53
Figure 41 Checking package format ............................................................. 54
Figure 42 Test/Deploy TREDISEC Recipe .................................................... 55
Figure 43 Interface for deploying or testing a TREDISEC Recipe ..................... 57
Figure 44 Testing information in the TREDISEC Framework ................. 57
Figure 45 TREDISEC Framework modular architecture .................................. 58
Figure 46 TREDISEC Framework system ..................................................... 59
Figure 47 Database objects ........................................................................ 61
Figure 48 A view of the main page of the TREDISEC framework GitLab project .... 63
Figure 49 View of the branches of framework Git repository .......................... 64
Figure 50 Labels specified in Git for testing of TREDISEC Framework .......... 65
Figure 51 Definition of secret token in TTE ............................................... 68
Executive Summary

The TREDISEC project has two main objectives:

- Design and develop solutions that fulfill both security and functional requirements of cloud-based systems
- Develop a framework that supports users in designing, managing and using such solutions.

The solutions are offered through the framework in the form of security primitive patterns, security primitive implementations and TREDISEC Recipes. Deliverables D2.3 and D2.4 described the architecture design of the TREDISEC framework, and detailed the lifecycle of security primitives, in their three flavours, and how the framework supports that, providing different functionalities and specific features to the four user roles identified, namely: TREDISEC Security Admin, TREDISEC end-user, Security Expert engineer, Security Technology Provider..

Deliverable D6.1 is a software implementation of the TREDISEC framework architecture design, as it is described in its final version in D2.4. The present document is a description of the actual software, which is available, as a stable version at M30, from the following sources:

- **GitLab repository of source code**: hosted by Atos. The framework software source code can be obtained from URL:
  

  Access must be requested in advance, by contacting TREDISEC project coordinator: beatriz.gallego-nicasio@atos.net

- **TREDISEC Framework instance running at the Test environment**: hosted by GRNET. The framework software is deployed, up and running, publicly available at the following URL:
  
  [https://tredisec.dev.grnet.gr/](https://tredisec.dev.grnet.gr/)

  Credentials to access and use the framework can be requested by contacting TREDISEC project coordinator: beatriz.gallego-nicasio@atos.net

Following, we describe in detail the TREDISEC Framework software implementation, starting with the roles we support and how they should use the framework, the functionalities offered, making especial emphasis in three processes: packaging (critical for the creation of actionable security primitive patterns, security primitive implementations and TREDISEC Recipes), primitives testing and deployment.

The testing of security primitive implementations and TREDISEC Recipes, which can either focus in functional requirements (e.g. correct functionality of the solution, etc.) or performance (e.g. increase or reduction of time of processing after deploying the solution) is supported by the framework by making available the so-called TREDISEC Testing Environments (TTEs). These TTEs are basically virtual environments (VMs) that users of the framework can use to test the capabilities of the primitives before actually downloading/using them in their own Cloud environments. These TTEs can be also used for deploying TREDISEC recipes and play around with them, e.g. by connecting via ssh.

Additionally we provide technical information of the software implementation building blocks, technologies used, communication channels and interfaces exposed, and procedures to build, install and configure your own instance of the TREDISEC Framework.

Finally, we include the conclusions and future work we will perform in the latest stage of the project together with the initial status and expected functionality.
Glossary

**TREDISEC Artefacts:** this concept is used to describe all the different types of artefacts that are managed and used in TREDISEC (security primitive pattern, security primitive implementation and TREDISEC Recipe)

**Security primitive pattern:** it is a security and functional solution expressed at abstract level. It only describes architecture and high-level functionality

**Security primitive implementation:** it is a security and functional solution that can be used in software systems. It implements a specific security primitive pattern following its architecture, guidelines, interfaces, etc. It could exist more than one security primitive implementation for one security primitive pattern as they could be done with different technical requirements, needs, etc.

**TREDISEC Recipe:** it is a composition of one or more security primitive implementation in order to provide a complex security and functional solution. The implementation of the TREDISEC Recipe contains the natural integration of the security primitive implementations that compose it so the use of several of them is transparent to the end-user.

**TREDISEC Framework:** the framework that allows working with the different artefacts of TREDISEC (e.g. creation, management, use, etc.)

**End-to-end security:** security solution that is implemented exclusively on the endpoints of a connection, therefore providing a high-level security for communications

**TREDISEC Testing Environment:** cloud environment used by the TREDISEC Framework for the deployment and testing of security primitive implementations and TREDISEC Recipes. More information about it can be found at Section 3.10

**TREDISEC Security Admin:** administrator of the TREDISEC Framework

**Security Expert Engineer:** security expert with knowledge of high-level solutions of security and functionality for the cloud

**Security Technology Provider:** expert with expertise in create security and functional solutions for the cloud in the form of security primitive implementations or TREDISEC Recipes

**TREDISEC User:** end-user of the TREDISEC Framework. She uses the TREDISEC Recipes in her cloud environment for having security and functional properties

**TREDISEC Tenants:** the tenants are the groups that create and manage the TREDISEC artefacts in the TREDISEC Framework. More information about them can be found at Section 3.4
1 Introduction

1.1 Purpose and scope

The TREDISEC Framework described in this document implements the final architecture and functionalities defined previously in deliverable D2.4 “Final Architecture and Design of the TREDISEC framework”. This architecture was proposed to give an answer to, on the one hand, the needs stemming from business, expressed by the project use case partners and collected in deliverable D2.2 “Requirements Analysis and Consolidation”; and on the other hand, the technical characteristics of the designs and developments of the WP3, WP4 and WP5 primitives, addressing security and cloud functional concerns. To sum up, the process we followed first identified the requirements from various points of view: business, technical and functional points of view (WP2), following we designed an architecture that could cover all previous needs (WP2) and finally we developed the framework that implements this architecture (WP6).

For practical purposes, the TREDISEC Framework aims to provide two main functionalities:

- allow security engineers and developers of security technology providers to create and make available to other framework users their solutions in three flavours, security primitive patterns, security primitive implementations and TREDISEC Recipes, according to their level of expertise and need;
- allow cloud service providers to search, test and integrate these solutions, in order to fulfil both their particular cloud functional and security and privacy needs.

We separate the functionalities offered by the framework to the different user profiles following a modular approach. It was critical that the TREDISEC Framework provided all these functionalities in a very accessible and friendly way, adapting to the user’s needs.

The creation and management of artefacts (security primitive pattern, security primitive implementation and TREDISEC Recipe) was integrated into a single module so they all could be linked easily (as they follow a continuous flow from security primitive pattern → security primitive implementation → TREDISEC Recipe).

The creation and management of users and tenants is another important module that allows a Framework administrator to create different groups (or tenants) and put in place rules to control access to artefacts, either for creation, modification or use. That way, if the TREDISEC Framework is deployed publicly or is intended to be used by different organizations, it would be possible to implement and advanced system for controlling access to the solutions, offered for free, under a closed license or any other “as a Service” business model.

It is worth highlighting that the TREDISEC Framework allows users to test and deploy a security primitive implementation or TREDISEC Recipe before making it available for its use, either to end-users or other TREDISEC security experts. This is possible thanks to the testing and deployment module, where owners of the TREDISEC Framework prepare and make available different virtual environments (i.e. TTEs) for users of the framework to test and deploy their solutions. TTEs may support the deployment of multiple solutions with different characteristics, giving the opportunity to technology providers to test compatibility of their solutions with others, for example.

TTEs can be also offered to integrators and cloud service providers for testing TREDISEC recipes before downloading (or potentially purchasing them), in order to make sure these indeed meet their requirements.

Finally, we would like to highlight that although the TREDISEC Framework has already integrated all the planned functionalities, we still think it could be updated and refined (maybe even extended) with additional features or small changes if, after we receive feedback from the validation of the use cases, we find comments or ideas to facilitate its use or improve certain functionalities. Therefore, all described functionalities here could be updated in order to improve the usability and user-friendly aspect of the TREDISEC Framework.
1.2 Structure of the document

The rest of the document is composed of the next sections, each of them focusing in a specific element of the TREDISEC Framework:

- Section 2 provides a reminder of the different roles we defined in D2.3 and D2.4 together with a more specific description of how these roles use the TREDISEC Framework.

- Section 3 describes the TREDISEC Framework starting with a quick recap of the architecture and then describing more in-depth, by means of screenshots and examples, the different functionalities supported. Among others we describe how access and authorization is done in the framework, the management of the framework, information about the security primitives (all types covering their goal, creation and management, process, link between them, etc.), the search and expert search, the packaging of the artefacts (with a clear description of how it is done and its composition for each artefact), and the testing and deployment environments. All these descriptions are done from a technical point of view so it describes how the TREDISEC Framework manages and works.

- Section 4 focuses in the technical development of the TREDISEC Framework. Here we describe the different building blocks, the interfaces and communication channels, how to install and configure it, technical pre-requisites, etc.

- Section 5 presents the conclusions and future work of the TREDISEC Framework together with feedback obtained from the testers (currently internal ones) for its enhancement.

- The annex shows different contributions for helping understand different parts of the TREDISEC Framework such as examples of testing (automated) or the structure of a file for documentation of a TREDISEC Recipe. Additionally it describes the current status of the TREDISEC artefacts uploaded in the TREDISEC Framework.
2 Roles of the TREDISEC Framework

2.1 Description
The roles of the users of the TREDISEC Framework are shown in Figure 1 interact with its different functionalities for creating, managing, testing and using the security primitives, or to configure and maintain the framework itself. These roles were first described in D2.3 and refined in D2.4. Following, we present briefly the description of these roles and how they use the TREDISEC Framework.

2.2 TREDISEC Admin
This role is in charge of configuring, maintaining and managing the TREDISEC Framework from an administrative point of view. Therefore she is the one in charge of deploying new versions of the software when available, protecting it from known vulnerabilities and threats, reporting bugs (or fixing if she has the knowledge), etc. Also, a user with this role is in charge of creating new users and tenants, and giving permissions to a certain level: within a tenant, the corresponding tenant admin will have this responsibility. The TREDISEC admin has an important assignment since she’s in charge of ensuring that the security primitives and TREDISEC Recipes available in the framework are ready (in terms of quality) for being offered to other users of the framework (i.e. end-users). She needs to check that these primitives are working as expected and thus, can be trusted. This can be done by checking the documentation conforms to the proper structure and quality, and by running some tests, usually provided by the primitive owner itself, but could be also tested using the admin’s own test suite.

2.3 Security Expert Engineer
This role represents the users of the framework that design security primitives and creates and/or maintains the corresponding pattern artefact. Users with this role are experts in security and cloud architects and they focus in designing solutions that security technology providers can later use for developing new solutions. These security technology providers may be part of the same organization as the Security Expert Engineer or not. The primitive pattern describes the primitive designed by the Security Expert engineer at a high level of abstraction, with no specific references to a particular technology, but detailing the requirements addressed (security and cloud functional), security scheme or protocol, the primitive architecture design and the interfaces offered or used, and the intended mode of use of the primitive in each use case.

2.4 Security Technology Provider
The Security Technology Provider can use the TREDISEC Framework with two purposes:
• find existing security primitive patterns and develop an implementation of it using a specific set of technologies. In this case, the Security Technology provider can create a security primitive implementations based on the selected pattern and offer that too through the framework,

• find existing security primitive implementations in order to create TREDISEC Recipes that are ready to be integrated with a cloud environment with particular characteristics. In order to do this, the security technology provider can use the testing module provided by the TREDISEC Framework, which allows her adapting the primitive implementation to the characteristics of a particular cloud environment. This feature facilitates also the possibility to check the correct functionality of the primitive running together with other primitives, all deployed in the same environment.

2.5 TREDISEC End-user

The end-user of TREDISEC searches the framework looking for a TREDISEC Recipe that can be integrated in or directly used by her own cloud system. Therefore, the end-user is looking for a ready-made solution that addresses not only the specific security and cloud functional needs, but also the platform-specific requirements she has. TREDISEC end-users can benefit from the deployment component offered by the framework and test the TREDISEC Recipes prior to actually downloading and integrating them in their cloud system. It is worth clarifying that this role could be either an individual user or an IT department in an organisation.
3 TREDISEC Framework

3.1 Description

Following, we present the TREDISEC Framework in its current implemented state, corresponding to M30 of the TREDISEC project timeline. The source code of the framework version corresponding to this deliverable is available in a tag created at the project GitLab repository:

https://gitlab.atosresearch.eu/ari/tredisec_wp6/tags/tredisec_framework_v1.0_d6.1_final

Access must be requested in advance, by contacting TREDISEC project coordinator: beatriz.gallegonicasio@atos.net

This version of the framework source code is deployed and running in an instance available at the following URL: https://tredisec.dev.grnet.gr/

There are planned new versions of the framework software to incorporate developments to fix bugs, enhance existing functionalities and features, and to incorporate the feedback received by the users in the validation process. These new versions of the software will be deployed regularly in a separate framework testing environment, used internally by the consortium partners. The management of new developments and versioning supported by GitLab is explained in section 4.6.

Here, we provide a brief reminder of the framework architecture, describing its different components and main functionalities offered.

The TREDISEC Framework is composed by five building blocks, as depicted in Figure 2, which support the four different user roles (described in the previous section) in creating, managing, searching and using the different security primitive artefacts developed. These building blocks, or components, implement the necessary functionalities in an easy-to-use manner, and are described in the following subsections in detail. In this description, we highlight which functionalities are available only for a specific role or if it is a general one (and therefore, accessible by any type of user).

![TREDISEC Framework Diagram](image)

Figure 2: TREDISEC Framework High-level Description

Next we present all these functionalities and components as they are implemented in the current
framework software, with details and screenshots explaining how to interact with them by means of
the web interface. In addition to the basic functionalities, we cover also critical features such as the
access control mechanisms, security primitives packaging, search capabilities, etc. necessary to
improve the user experience.

3.2 Access and Authorization

The first thing to do when accessing the TREDISEC Framework is to log into the system. Figure 3
shows the interface for log-in.

It is important to note that the current version of the TREDISEC Framework is intended to be used
only by members of the TREDISEC consortium and the EU Commission, for testing and validation
purposes. Therefore, the process of creation of accounts is done by the TREDISEC Admin manually,
triggered by sending an email to the project coordinator (<beatriz.gallego-nicasio@atos.net>)
requesting access.

If in the future, the agreement of the consortium is to exploit the TREDISEC Framework publicly, it
would be straightforward to extend and improve the registering process with a more advanced
automated access control mechanisms.

![Login Interface](image)

Figure 3: Log-in in the TREDISEC Framework

Once the user is correctly logged in, the main interface of the TREDISEC Framework (home)
appears. Depending on the user role, there will be more or less options available.

The framework main web interface is structured as follows:

- A **heading** section, which is composed of two levels of headings:
  - First-level: contains the logo of the project on the left side, and a two-row menu bar,
    on the right side. As explained next, the options visible for the user depends on its
    role.
  - Second-level: with the breadcrumb bar, on the left, showing the path to the content
    displayed in the main body section; and quick search box, on the right.

- A **main body** section, which dynamically displays the content corresponding to the menu
  option accessed by the user.

- A **footer** section, which at the moment displays the mandatory text acknowledging the
  European Commission grant, and the current version of the framework deployed. This can be
  easily extended with further information if needed in the future.
For framework testing and validation purposes, we are focusing in two roles in the TREDISEC Framework: TREDISEC end-user and TREDISEC Admin. This simplification eases the testing process, since a normal end-user has very limited functionalities available, while the Admin has all functionalities available. The other two roles are, somehow sub-cases of these two, and the results of the tests can be extended to them. Figure 4 shows the main interface for a normal user while Figure 5 shows the interface as an admin. In both cases, the main body section lists the primitive artefacts owned by the user logged in.

The main differences between them, as can be seen in Figure 4 and Figure 5, are that the Admin has access to the following extra elements in the menu bar of the TREDISEC Framework:

- Users List: it allows management and creation of users in the TREDISEC Framework. It is described more in-depth in Section 3.3
- Testing Environments: it allows the creation and management of the testing environments provided by the TREDISEC Framework, for testing and deploying security primitive implementations and TREDISEC Recipes. It is described more in-depth in Section 3.5

In the following sections we detail the functionalities offered by the framework and accessible by means of the different options of the top menu bar.
3.3 User List

This option allows creation and management of users. Figure 6 shows the graphical interface corresponding to this functionality.

The different elements of the table of users are:
- Username: the username of the user
- Email: the email used for registration of the user
- Tenant: the tenant to which the user belongs
- Admin: if checked, that indicates the user has administrator rights over the framework. This corresponds to the TREDISEC Admin role described in the previous section.
- Tenant Admin: if checked, that indicates the user is the owner/manager of a tenant (i.e. group of users that belong to the same organization, e.g. Atos, Arsys, etc.). Being a tenant admin implies the user has a sub-set of the administrative rights of the TREDISEC Admin role, applicable only to the users and primitive artefacts belonging to the specific tenant.
- Active: if checked, that indicates the account of the user is active in the framework

Bear in mind that by default all users have the role of users. The other options (Admin, Tenant Admin) that grant extra permissions must be explicitly activated.

The creation of users is supported by using the “Create New User” button. When used it redirects to the interface shown Figure 7.
For the creation of a new user it is necessary the following data:

- **Username**: the username of the user (mandatory)
- **Password**: the password of the user. It is self-generated but the user must change it the first time she accesses the system (mandatory)
- **First name**: the first name of the user
- **Last name**: the last name of the user
- **Email**: the email of the user (mandatory)
- **Bio Note**: description/notes of the user
- **Admin**: to be checked if the user has the TREDISEC Admin role
- **Tenant**: the tenant the user belongs to. A tenant is a group composed of different users that belong to the same organization or entity.
- **Tenant Admin**: to be checked if the user is the tenant admin

Once created, the user can use the username and password to access the framework.

### 3.4 Tenant Management

A tenant of the TREDISEC framework is a group of users that belong to the same organization or entity. This concept is used within the framework to handle permissions to access certain functionalities and to manage visibility of the primitive artefacts stored. This way, users that belong to a tenant, would automatically be able to access *protected* primitive artefacts developed by their tenant peers (see more details on visibility of primitives in section 3.4). Also, a tenant may have associated one or various TREDISEC Testing Environments (TTEs), for the purpose of testing the primitives developed by all user members (see more details on TTEs in section 4.7.1.2).

The option “Tenants” in the top menu bar allows access to the list of available tenants (and the corresponding members) in the TREDISEC Framework. In the current version of the framework, the creation and management of the tenants is done by the TREDISEC Admin role manually, since it is not supported by the framework interface. A more user-friendly support can be implemented easily, as part of the developments required to launch the framework to the wide public in a future stage. Figure 8 shows the interface for tenant management.
3.5 Testing Environment

By clicking in the “Testing Environment” menu option, the user accesses the functionalities offered by the testing component of the framework (see Figure 2). By means of this component, the framework provides users the possibility of deploying and testing the security primitive implementations and TREDISEC recipes in different cloud platforms, the so-called testing environments. There are two types of testing environments in the framework, as it is shown in Figure 9:

- **TREDISEC Testing Environments**: the ones supported by the TREDISEC Framework. In other words, virtual environments a TREDISEC Admin has configured and made available for uses directly through the framework. These virtual environments are running in the background and are connected to the framework, but this fact is transparent to the rest of the users (including tenant admins). More information can be found in section 4.7.1.2.

- **Unsupported Testing Environments**: here we can define testing environments that the framework do not manage, and thus, cannot be directly used by users for deployment and testing of primitives. These unsupported environments can be used by developers of security primitive implementations and TREDISEC Recipes for specifying the characteristics of the cloud environments where these artefacts are meant to be used. This way, we enable the search of primitive implementations and TREDISEC recipes by target cloud platform, in addition to the search by name, author or keywords. For example, even if Amazon cloud is not linked to the TREDISEC Framework for testing, a TREDISEC Recipe could be defined for being used in that environment. Later, when an end-user accesses the TREDISEC Framework looking for a security and functional solution that could be deployed in the Amazon cloud, she could use this search feature.

Figure 9 shows the main interface of this section, with some environments we are currently using for testing and deploying TREDISEC Recipes.
The different elements that describe a testing environment are:

- Platform name/host: the name of the platform and its URL.
- Keywords: a list of keywords that describe the cloud platform. This list is later used for the search functionality.
- Used by: shows the number of artefacts (security primitive patterns, security primitive implementations and TREDISEC Recipes) that use this platform. By clicking in the environment name, all the artefacts that use it (i.e. that have been associated to it) are listed.
- Modify button: allows to modify the descriptive elements of the cloud platform
- Delete button: allows to delete the cloud platform

In order to create a new platform the user has to click in the “Create Platform” button. It displays the interface shown in Figure 10.
In this wizard the user has to introduce the name of the platform (descriptive), the URI or host of the cloud platform (if it is included it will count as “supported” while if this field is empty then will be “unsupported”), a list of keywords for describing it (for searching purposes) and a description. The tenant dropdown is used to associate the testing environment to a specific tenant.

Once finished the platform is listed in the list of available testing environments and can be used by users that belong to the tenant, to relate a security primitive implementation or TREDISEC Recipe to a cloud-based environment.

3.6 Security Primitives

The concept of security primitives (security primitive pattern, security primitive implementation and TREDISEC Recipe) is described more in-depth in D2.3\(^1\) and D2.4\(^2\). D2.3 shows the initial description and D2.4 updates them, with the feedback received from partners after going through several iterations in the project technical meetings and conference calls.

This section first, recalls the life cycle of the security primitives (creation, management and use) as conceived in the TREDISEC architecture and following, we define how it is supported by the TREDISEC Framework implementation.

3.6.1 Description

The security primitive concept has been de-composed into three different artefacts: the security primitive pattern, the security primitive implementation and the TREDISEC Recipe. One artefact build upon the other, ranging from the more abstract (i.e. security primitive pattern) to the more technology-specific and environment-oriented (i.e. TREDISEC Recipe). A short description of each of the artefacts, shown in Figure 11, follows:

\(^1\) D2.3 TREDISEC Architecture and Initial Framework Design
\(^2\) D2.4 Final Architecture and Design of the TREDISEC Framework
• **Security primitive pattern**: it is the more abstract definition for a security primitive. It is mainly used for describing the security and functional requirements addressed, providing a high-level view of the approach the solution follows to solve the problem (domain-specific, such as resource isolation).

• **Security primitive implementation**: it implements a specific security primitive pattern, using a particular technology. Therefore, it is envisioned that a security primitive pattern could have one or more security primitive implementations. The implementations would be different from the point of view of the technical characteristics (e.g. programming language, technical requirements, underlying cloud environment, etc.). This artefact could be complemented with deployment and testing capabilities and resources, as it is explained in section 3.8.1.2.

• **TREDISEC Recipes**: this artefact is tailored to the specific characteristics of the target cloud environment where TREDISEC end-users will eventually deploy and integrate it. The TREDISEC Recipe is an integration of one or more security primitive implementations in order to fulfil a set of security and functional capabilities. A relevant element of this artefact is the deployment, and optionally, testing capabilities, which may enable end-users to automatically, or semi-automatically, deploy and test the primitive in a pre-configured testing environment (as already introduced in section 3.5 and further detailed in sections 3.9, 3.10 and 3.11).

![Figure 11: Security Primitives Concept](image)

The following sections provide a detailed description of the support offered by the framework in the process of creation, management and use of the three security primitive artefacts.

### 3.6.2 Security Primitive Pattern

All functionalities to create, view and manage the security primitive pattern artefacts in the framework are accessible by clicking option “Security Primitive Patterns”, from the second-row of the top menu bar of the framework graphical interface (see Figure 12). This action will populate the main body section with the list of primitive patterns that are accessible to the user logged in, as it is depicted in Figure 13.
3.6.2.1 Creation

Once the security primitive pattern main interface is loaded, the user must click the button “Create Security Primitive Pattern”, located at the top-right of the list of primitive patterns, in order to open the creation wizard, as shown in Figure 14.

The information that must be provided in the first page of the wizard is:

- **Security primitive pattern name**: unique name of the artefact.
- **Keywords**: keywords describing the security primitive pattern. This will be used later for the search functionality.
- **Documentation**: here users must provide a .zip containing the documentation of the security primitive pattern. The documentation must follow a specific structure according to the type of object (see section 3.8.1.1). When the .zip file is uploaded, the TREDISEC Framework checks that the structure is correct and generates automatically an HTML-based documentation. It also extracts some pieces of information that the framework will display in tabs, as explained later in section 3.6.2.2.
Figure 14: Wizard for creating a security primitive pattern (step 1)

Once the information listed above is provided (and checked for correctness), the second page of the wizard appears (Figure 15). This second page lets the user define the artefact access levels for other users of the framework:

- **Manage access level:** there are three types of access:
  - Public means everyone can access it,
  - Protected means only users of the same tenant can access it, and
  - Private means only the user who created it can access it (e.g. it is still work in progress and the owner does not want it to be shown yet to the rest of users).

- **Manage Tenancy Protected access:** in the case the access level selected is Protected, the permissions can be defined at a finer granularity level:
  - Tenancy view access: allows specifying which tenants can view the information associated to the security primitive pattern (i.e. list it and see the details)
  - Tenancy download access: allows the specification of the tenants that are allowed to, in addition to view it, download the security primitive pattern (i.e. the zip file associated to the pattern artefact).
3.6.2.2 Information of a security primitive pattern

From the list of available primitive patterns (Figure 13), clicking in a security primitive pattern opens a new interface, the primitive pattern details page. This page shows all the information associated to the artefact (extracted from the files included in the corresponding zip file uploaded when created). This page also gives access to different functionalities to manage the artefact through the buttons row at the bottom of the page.

Figure 16 is a screenshot of the primitive pattern details page corresponding of an example artefact.

The information displayed in the primitive pattern details page is organised as follows, top to down, left to right:

- **Security primitive pattern logo**: a common logo to easily identify the type of artefact displayed.
- **Documentation status**: once the zip file is uploaded, a check is conducted by the framework in the background for correctness (see section 0). The result of this check is displayed here: Passed, Pending (when the process is still going on in the background), Failed.
- **Author/Tenant**: displays the creator user and the tenant the user belongs to.
- **Keywords**: the list of keywords that identify the pattern, used for searching purposes.
- **Tabs section**: these tabs are populated using some specific files included in the zip (see section 0).
  - Tab Overview: gives a brief introduction to the primitive pattern, to understand what it does at a glance.
  - Tab Requirements: describes the requirements the security primitive pattern meets.
- **Buttons row**:
  - Remove: deletes the security primitive pattern.
  - Modify: allows the modification of the security primitive pattern (explained below).
- **Documentation**: opens a new tab/window to display the html-based detailed documentation of the primitive (see Figure 17), generate automatically from the files included in the zip file (see section 0).

- **Manage tenancy access**: allows the modification of the permissions associated to the artefact, as explained in section 3.6.2.1.

- **Create Security Primitive Implementation**: allows the creation of a security primitive implementation. It opens a wizard for a fastest access to the creation process.

---

**Multitenancy ABAC primitive pattern**

![Multitenancy ABAC primitive pattern](image)

**Overview**

The aim of the primitive is to provide an enforcement component for distributed attribute-based access control (ABAC) policies that ensures that authorized users always get access to the selected cloud resource (either data or service) whilst the access is refused to malicious parties, in the context of a multi-tenant cloud infrastructure.

This primitive addresses requirements from the TREDISEC access control model supporting multitenancy which is based on an ABAC model. This advanced access control mechanisms will allow tenants to define fine-grained policies on a per-user basis, and which can be easily integrated in current cloud environments. Using this access control model the cloud service providers will be able to share resources (any kind of resource) among their users.

The main components of the access control model addressed by this primitive are:

- **Policy Enforcement Point (PEP)**: for access request coming from outside the tenant, the access request is submitted to this component in order to generate the XACML request to be sent to the POP.
- **Policy Decision Point (PDP)**: The PDP is the component which evaluates the access request against the policies in order to protect the resource trying to be accessed. Once the decision is made, a response will be sent back to the requesting PEP to take effect with respect to the requested access.
- **Policy Repository**: it stores the access control policies. Trust relations among tenants could be defined and supported extending the policies.
- **Resource Manager (RM)**: the manager of the resource on which the Cloud User requests permission to perform an action. The Resource Manager may not have a direct relation with the cloud provider. The cloud provider assigns resources to a Resource Manager and the Resource Manager can assign permissions for the Cloud Users on the resource. For simplicity we will consider the Resource Manager as a component of the cloud system.

![Multitenancy ABAC primitive pattern](image)

**Figure 16 Detailed information of a security primitive pattern**
3.6.2.3 Management

Users may modify a security primitive pattern they create by first, accessing the primitive pattern details page (see section 3.6.2.2), and once there, clicking the button “Modify”, located at the bottom row. After that, the interface for modifying the pattern information (Figure 18) will be displayed.
Users may also delete any security primitive patterns they create. In this case, the user must click in the button labelled “Remove”, also located at the bottom row of the specific primitive pattern details page. This will remove the artefact from the TREDISEC Framework but could have some implications, which are explained in detail in section 3.6.5.

### 3.6.3 Security Primitive Implementation

All functionalities to create, view and manage the security primitive implementation artefacts in the framework are accessible by clicking option “Security Primitive Implementations”, from the second-row of the top menu bar of the framework graphical interface (see Figure 19). This action will populate the main body section with the list of primitive patterns that are accessible to the user logged in, as it is depicted in Figure 20.

#### 3.6.3.1 Creation

The creation of a security primitive implementation can be done in two different ways:
i) from an existing security primitive pattern (as already explained in section 3.6.2.2),

ii) by clicking the button “Create security primitive implementation” located at the top-right of the list of security primitive implementations (Figure 21).

Once any of these options is selected, the user will access a wizard for creating the security primitive implementation, shown in Figure 22.

The information that must be completed here is:

- **Name**: the name of the security primitive implementation must be unique
- **Version**: the version of the security primitive implementation
- **Keywords**: the list of keywords that define the security primitive implementation
- **Security Primitive Pattern**: the user must select from the list the security primitive pattern that is implemented. In case the user went for option (i) to create the artefact, this will be pre-selected.
- **Package**: this opens the dialog for selection of the security primitive implementation package (i.e. the zip file) from user’s filesystem. This package contains the application (e.g. code, binaries, etc.) of the security primitive implementation, the documentation, the testing and the deployment information, as it is further explained in section 3.8.1.2.

Clicking “submit” gets the user to the next wizard page, which allows defining the permissions to manage the security primitive implementation (shown in Figure 23), in the same way as it is done for security primitive patterns:

- **Manage access level**: there are three types of access:
  - *Public* means everyone can access it,
  - *Protected* means only users of the same tenant can access it, and
  - *Private* means only the user who created it can access it (e.g. it is still work in progress and the owner does not want it to be shown yet to the rest of users).

- **Manage Tenancy Protected access**: in the case the access level selected is *Protected*, the permissions can be defined at a finer granularity level:
  - *Tenancy view access*: allows specifying which tenants can view the information associated to the security primitive implementation (i.e. list it and see the details)
  - *Tenancy download access*: allows the specification of the tenants that are allowed to, in addition to view it, download the security primitive implementation package (i.e. the zip file associated to the artefact).

## Manage Tenancy Access

![Manage Tenancy Access]

Figure 23: Creation of a security primitive implementation (step 2)
3.6.3.2 Information of the security primitive implementation

From the list of available primitive implementations (Figure 20), clicking in a security primitive implementation opens a new interface, the primitive implementation details page. This page shows all the information associated to the artefact (extracted from the files included in the corresponding zip file uploaded when created). This page also gives access to different functionalities to manage the artefact through the buttons row at the bottom of the page.

Figure 24 is a screenshot of the primitive implementation details page corresponding of an example artefact.

![EPICA](image)

**Figure 24: Detailed information of a security primitive implementation (owned)**

The information displayed in the primitive implementation details page is organised as follows, top to down, left to right:

- **Security primitive implementation logo**: a common logo to easily identify the type of artefact displayed.
- **Implements Security Primitive pattern**: indicates which pattern the artefact implements. This is a link that redirects to the information page of the pattern, for user’s convenience.
- **Documentation status**: once the zip file is uploaded, a check is conducted by the framework in the background for correctness (see section 0). The result of this check is displayed here: Passed, Pending (when the process is still going on in the background), Failed.
- **Version**: the version of the artefact.
- **Author/Tenant**: displays the creator user and the tenant the user belongs to.
- **Keywords**: the list of keywords that identify the implementation, used for searching purposes.
- **Tabs section**: these tabs are populated using some specific files included in the zip (see section 0).
  - **Tab Overview**: gives a brief introduction to the primitive implementation, to understand what it does at a glance.
Tab Requirements: describes the requirements the artefact meets, and also technical requirements for the implementation to work properly (e.g. specific hardware needs, processing capability, Hard Disk storage, required external libraries, etc.).

Known issues: describes any known issues related to the artefact a user must take into account (e.g. limitations of the implementation, known existing bugs, etc.).

Buttons row:
- Remove: deletes the security primitive pattern.
- Modify: allows the modification of the security primitive pattern (explained below)
- Documentation: opens a new tab/window to display the html-based detailed documentation of the primitive, generate automatically from the files included in the zip file (see section 0).
- Manage tenancy access: allows the modification of the permissions associated to the artefact, as explained in section 3.6.2.1.
- Testing upload and information: allows access to reports of deployment and testing. This opens a dedicated interface, explained in-depth in section 3.9.
- Download: allows users to download the security primitive implementation package.
- Create TREDISEC Recipe: it opens the wizard for creating a TREDISEC Recipe as described in section 3.6.4.1.

3.6.3.3 Management

![Interface for modifying a security primitive implementation](image)

**Security Primitive Implementation name**

| implementation secure_file_deduplication 4 |

**Version**

| 0.0.4 |

**Keywords**

Comma separated list of keywords

| secure | deduplication |

**Security Primitive Pattern**

Select the security pattern this package implements.

| pattern secure_deletion 3 |

**Security Primitive Implementation package**

The Security Primitive Implementation package should be a ZIP archive.

| Browse... No file selected |
Users can modify the security primitive implementation they create. In order to do so, the user has to access the information page of the specific security primitive implementation (shown in Figure 24) and click on button “Modify” at the bottom row. This will display the interface for modifying all the information (Figure 21).

Similarly, users have also the option to delete their security primitive implementations by clicking button “Remove”. This will remove the artefact from the TREDISEC Framework but there are some implications that need to be taken into account. These are explained in detail in section 3.6.5.

3.6.4 TREDISEC Recipe

All functionalities to create, view and manage TREDISEC Recipe artefacts in the framework are accessible by clicking option “TREDISEC Recipes”, from the second-row of the top menu bar of the framework graphical interface (see Figure 26). This action will populate the main body section with the list of recipes that are accessible to the user logged in, as it is depicted in Figure 27.

3.6.4.1 Creation

In order to create a TREDISEC Recipe, the user may do it by following one of the two options listed next:

i) clicking button “Create TREDISEC Recipe” from a specific security primitive implementation details page (as shown in Figure 24), or

ii) from the list of TREDISEC Recipes in the TREDISEC Framework (Figure 31), by clicking in “Create TREDISEC Recipe” button at the top-right.

Once done, the user accesses the wizard for the creation of a TREDISEC Recipe. This wizard is shown in Figure 28.
The information the user must fulfil in this case is:

- **Name**: the name of the TREDISEC Recipe must be unique
- **Version**: the version of the artefact
- **Testing environment**: the type of cloud environment in which the TREDISEC Recipe runs. This must be linked to an existing testing environment (either supported by the framework or not), as previously explained in section 3.5.
- **Keywords**: the list of keywords that define the TREDISEC Recipe, which will facilitate searching.
- **Security Primitive Implementation**: the user must select from the list the security primitive implementation this TREDISEC Recipe is related with. If the user opted from creation method (i), this will be pre-selected.
- **Package**: from here, the user is able to upload, from user's file-system, the package (i.e. zip file) containing all the information related to the TREDISEC Recipe: the documentation, the testing and the deployment information, the code/binaries/etc. for deployment and use in target cloud environments, etc. More details about this package can be found in section 3.8.1.3.

Similarly to the other primitive artefacts, once this initial information is submitted, a second step of the wizard appears. This wizard is used for defining access management for the TREDISEC Recipe (shown in Figure 29). A more in-depth description of the data needed is:

- **Manage access level**: there are three types of access:
  - **Public** means everyone can access it,
  - **Protected** means only users of the same tenant can access it, and
o **Private** means only the user who created it can access it (e.g. it is still work in progress and the owner does not want it to be shown yet to the rest of users).

- **Manage Tenancy Protected access**: in the case the access level selected is **Protected**, the permissions can be defined at a finer granularity level:
  
  o **Tenancy view access**: allows specifying which tenants can view the information associated to the TREDISEC Recipe (i.e. list it and see the details)
  
  o **Tenancy download access**: allows the specification of the tenants that are allowed to, in addition to view it, download the TREDISEC Recipe (i.e. the zip file associated to the artefact).

**Manage Tenancy Access**

![Recipe Details Page](image)

**Figure 29: Wizard for creating a TREDISEC Recipe (step 2)**

### 3.6.4.2 Information of TREDISEC Recipes

From the list of available TREDISEC Recipes (Figure 27: List of TREDISEC Recipes), clicking in a recipe opens a new interface, the **TREDISEC recipe details page**. This page shows all the information associated to the artefact (extracted from the files included in the corresponding zip file uploaded when created). This page also gives access to different functionalities to manage the artefact through the buttons row at the bottom of the page.

Figure 30 shows a screenshot of an example of TREDISEC Recipe details page. The information displayed in the primitive implementation details page is organised as follows, top to down, left to right:

- **TREDISEC Recipe logo**: a common logo to easily identify the type of artefact displayed.

- **Documentation status**: once the zip file is uploaded, a check is conducted by the framework in the background for correctness (see section 0). The result of this check is displayed here: Passed, Pending (when the process is still going on in the background), Failed.

- **Version**: the version of the artefact.

- **Author/Tenant**: displays the creator user and the tenant the user belongs to.

- **Keywords**: the list of keywords that identify the implementation, used for searching purposes.

- **Tabs section**: these tabs are populated using some specific files included in the zip (see section 0).
o **Tab Overview:** gives a brief introduction to the recipe, to understand what it does at a glance.

o **Tab Requirements:** describes the requirements the artefact meets, and also technical requirements for the recipe to work properly (e.g. need for an existing library or software installed, etc.).

o **Known issues:** describes any known issues related to the artefact a user must take into account (e.g. incompatibility with an existing solution or program running in the system, need to install additionally a library, etc.)

- **Buttons row:**
  
  o **Remove:** deletes the security primitive pattern.
  
  o **Modify:** allows the modification of the security primitive pattern (explained below)
  
  o **Documentation:** opens a new tab/window to display the html-based detailed documentation of the primitive, generate automatically from the files included in the zip file (see section 0).
  
  o **Manage tenancy access:** allows the modification of the permissions associated to the artefact, as explained in section 3.6.4.1.
  
  o **Testing upload and information:** allows access to reports of deployment and testing. This opens a dedicated interface, explained in-deep in section 3.9.
  
  o **Deploy:** it starts the automatic process of deployment of the TREDISEC Recipe in the associated testing environment (only for supported ones, as explained in section 3.5).
  
  o **Download:** allows users to download the security primitive implementation package.

---

**Figure 30: Detailed information of a TREDISEC Recipe and actions**

### 3.6.4.3 Management

Users can modify their own TREDISEC Recipes. In order to do so, the user should select a specific TREDISEC Recipe from the list of available ones (shown in Figure 27) and click there in the button...
“Modify”. This will display the interface for modifying the recipe information (Figure 31).

Similarly, users have also the option for deleting their own TREDISEC Recipes by clicking button “Remove”. The system will ask for a confirmation and then remove the artefact from the TREDISEC Framework.

![Figure 31: Interface for modifying a TREDISEC Recipe](image)

### 3.6.5 Management of Deletion/Updating Artefacts

Working with artefacts that depend on each other (a security primitive implementation implements a security primitive pattern and a TREDISEC Recipe contains one or more security primitive implementations), requires a specific strategy for deleting and updating, to propagate the changes safely and keep them consistent. Next, we describe the strategy we use for both actions in the TREDISEC Framework.

#### 3.6.5.1 Deletion of artefacts

When an artefact is deleted it cannot be longer accessed or used by any user in the TREDISEC Framework, as it is removed from the internal repository. Figure 32 shows a high-level diagram of the implications of removing artefacts and after, we present two possible situations and how the framework will behave in each of them.
Security primitive pattern, which is used in a security primitive implementation, is deleted

The security primitive implementation is still available in the TREDISEC Framework but it is not possible to access the specification of the security primitive pattern. This way the security primitive implementation, which can be created by an expert different than the one that created the pattern, is still available but users will not be able to access the pattern that guided its design.

Security primitive implementation, which is used in a TREDISEC Recipe, is deleted

As in the previous case, the deletion of the artefact doesn’t impact the already created ones. In this case, the TREDISEC Recipe would still be available for using in the TREDISEC Framework but a “deprecated” notice will be displayed. This way, users that want to use the TREDISEC Recipe will be aware that it is no longer possible to access the related security primitive implementation. Additionally, the creator of the TREDISEC Recipe will not be able to access the code of the security primitive implementation, and thus, it could not be used for further modifications (except for the current implementation included in the TREDISEC Recipe).

3.6.5.2 Updating of artefacts

The strategy for updating of artefacts is similar to the previously described for deletion. The TREDISEC Framework marks internally when an artefact (security primitive pattern or security primitive implementation), which is used in another one (security primitive implementation or TREDISEC Recipe), has being updated. Owners of these artefacts could then update theirs with the new versions. As we did in the previous section, Figure 33 reflects this strategy with a high-level diagram and right after, we present a couple of examples that illustrate how the TREDISEC Framework works to implement this strategy.

Figure 32: Deletion of Artefacts
Security primitive pattern, which is being used in a security primitive implementation, is updated

When the security primitive pattern is updated the TREDISEC Framework marks it as updated. The security primitive implementation is then marked as using an old version of the security primitive pattern.

Security primitive implementation, which is being used by a TREDISEC Recipe, is updated

Similar to the previous case, when the security primitive implementation is updated it is marked as updated. Therefore, the TREDISEC Framework adds information to all TREDISEC Recipes that uses the updated implementation as “outdated”, since these are using an old version of the security primitive implementation. It is now the responsibility of the owner of the TREDISEC Recipe, to update it with the new implementation version accordingly.
3.7 Search and Advanced Search

The advanced search allows searching any type of artefact supported by the TREDISEC Framework (security primitive pattern, security primitive implementation and TREDISEC Recipe) using different properties such as the name, author, platform where it runs (see section 3.5 for more details), the tenant that owns the artefact or by keywords associated to the artefact. Figure 34 shows the form to input the search terms. The result of the search is displayed right below the search form, grouping artefacts by type, as it is depicted in Figure 35.

![Advanced Search input form](image-url)

Figure 34 Advanced Search input form
Security Primitive Patterns (0) found

Security Primitive Implementations (0) found

TREDISEC Recipes (3) found

Figure 35 Results of the Advanced Search

3.8 Security Primitives Packaging

Each security primitive artefact (security primitive pattern, security primitive implementation and TREDISEC recipe) that is uploaded to the TREDISEC Framework should be packaged following the guideline we present below. This is critical to ensure that

a. The documentation associated to the artefact is correctly generated, to populate the corresponding tabs of the artefact detailed description page, as well as the html pages (see section 3.6)

b. The automatic testing and deployment capabilities of the artefact (implementation or recipe), are performed as expected by the framework at the supported testing environments.

In brief, a package is a zip file that includes documentation that defines carefully the functionality, security and functional requirements, system model and interfaces of the described artefact. A security primitive implementation and TREDISEC Recipe should include a more concrete description of the primitive, considering the implementation details, technologies used and dependencies, the binaries of the actual primitive software, testing data, scripts, etc.

Following we provide a guideline about the structure and packaging of the artefacts, describing more in-depth each different field/folder.

3.8.1 Guideline of Artefact Packaging
All artefacts created in the TREDISEC Framework must upload a package (i.e. a zip file) containing all the different elements necessary for its use within the framework. The structure of the package is composed of the following four directories:

- **Documentation**: contains the description of the artefact, including the functionalities that the artefact defines or achieves, the interfaces that the artefact defines or provides, and the system requirements for application execution or deployment, etc. This information is provided as text documents, including diagrams, figures, screenshots, etc.

- **Application**: contains the technical implementation of the artefact, including the software and configuration files. It is only mandatory for the security primitive implementation and TREDISEC Recipe

- **Deployment**: contains the full delivery of the artefact software, including libraries and dependencies, as well as any deployment scripts. It is only mandatory for the security primitive implementation and TREDISEC Recipe

- **Testing**: contains the scripts and data for testing the artefact software. It is only mandatory for the security primitive implementation and TREDISEC Recipe

All the documents under the “Documentation” directory should follow the structure guidelines presented in the next subsection. That way the TREDISEC Framework will parse the package according to a pre-defined format and display the documentation in the artefact description page and in the html pages displayed when clicking in button “Documentation” of the artefact page.

The software and scripts under the “Application”, “Deployment”, and “Testing” directories can be processed and executed by the TREDISEC Framework if provided in a specific way (automatic run of scripts).

In case the security primitive implementation or TREDISEC Recipe is delivered to be deployed manually, the users of the TREDISEC Framework have to download the package and configure the artefacts following the instructions provided along with the other mandatory elements of the package.

There is the option to deploy the artefact in an automatic way, for example, to test the TREDISEC artefacts. In this case, when uploading the package into the framework, the user must choose the right target platform, so that the testing components can be executed automatically by the framework as designed by the owner of the artefact.

As indicated previously, for security primitive patterns only the “Documentation” folder content is mandatory in the package. For the other artefacts, all four directories should be included in the package to ensure full support by the framework is provided. In case that some artefact does not include certain elements, and the corresponding directory is left out blank, the framework will not provide support for the corresponding functionalities (e.g. automatic deployment or testing).

Next we provide some guidelines for packaging security primitives, in what concerns the structure and composition of each package. Furthermore, Annex 3 provides more details about the package creation process.

### 3.8.1.1 Security Primitive Pattern

<table>
<thead>
<tr>
<th>Application</th>
<th>Not necessary to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment</td>
<td>Not necessary to include</td>
</tr>
<tr>
<td>Documentation</td>
<td>It includes information describing the solution</td>
</tr>
<tr>
<td>Testing</td>
<td>Not necessary to include</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Documentation source</th>
</tr>
</thead>
<tbody>
<tr>
<td>author.txt (mandatory)</td>
</tr>
<tr>
<td>doc_version.txt (mandatory)</td>
</tr>
<tr>
<td>title.txt (mandatory)</td>
</tr>
<tr>
<td>version.txt (mandatory)</td>
</tr>
</tbody>
</table>

The author of the security primitive pattern

The version of the documentation of the security primitive pattern

The title of the security primitive pattern

The version of the security primitive pattern
<table>
<thead>
<tr>
<th>files</th>
<th>Includes information describing the solution provided by the security primitive pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Describes the information of the interfaces the security primitive pattern (the solution) should provide. It can be refined as necessary including names, parameters, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Documentation\source\files</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>overview.rst² (mandatory)</td>
<td>It explains (as an executive summary) what the security primitive pattern does. Bear in mind the text/information provided here is parsed to appear as such in the generated documentation of the artefact, so you have to include titles, bullet points, etc. as necessary. Images can be referenced and should be included in folder &quot;images&quot;.</td>
</tr>
<tr>
<td>architecture.rst (optional)</td>
<td>Describes the architecture of the security primitive pattern from a technology-agnostic point of view.</td>
</tr>
<tr>
<td>requirements.rst (optional)</td>
<td>Here we describe the requirements (e.g. technical, functional, etc.) and any other additional information regarding the solution of the security primitive pattern</td>
</tr>
<tr>
<td>notes_and_references.rst (optional)</td>
<td>It can be used to describe any additional information useful for describing the security primitive pattern, for example, external reference documents.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Documentation\source\interface</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>codename.txt (optional)</td>
<td>Describes the name of the solution</td>
</tr>
<tr>
<td>interface_description.txt (mandatory)</td>
<td>Here we describe the interface offered by the security primitive pattern and the main functionalities provided. This interface must be implemented by the corresponding security primitive implementation artefacts.</td>
</tr>
<tr>
<td>categories.json (mandatory)</td>
<td>It is used for describing the structure of the calls that compose the interface: name, parameters, return type, etc.</td>
</tr>
</tbody>
</table>

### 3.8.1.2 Security Primitive Implementation

<table>
<thead>
<tr>
<th>Application (folder)</th>
<th>Includes the software of the solution. It can contain source code, binaries, scripts, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment (folder)</td>
<td>It includes the data for the deployment of the solution. The deployment could be: automatic (e.g. scripts, executables, etc.), semi-automatic (same as automatic but requiring some interaction with the user, for example asking for user/password, IP, etc.) or manual (e.g. documentation and diagrams about how to deploy/integrate the solution without any further support)</td>
</tr>
<tr>
<td>Documentation (folder)</td>
<td>Includes the documentation describing the security primitive implementation (solution), interfaces, diagrams, etc. from the perspective of a specific technology used.</td>
</tr>
<tr>
<td>Testing (folder)</td>
<td>It includes all the required information for testing the solution. The testing, similar to the deployment, can be run in three different manners: automatic (e.g. scripts or executables that run the tests and compile the results), semi-automatic (same as automatic but requiring some specific configuration/interaction with the user), manual (includes specification of the tests and may include some software, but requires of major interaction with the user to execute them).</td>
</tr>
</tbody>
</table>

³ For creating or modifying files in this format you can use the online tool: https://livesphinx.herokuapp.com/
Application

No specific structure to include the source code, binaries, etc.

Deployment

deploy (mandatory)
This file specifies the instructions for deploying the solution. This follows the magic number of the executable (https://en.wikipedia.org/wiki/Magic_number_(programming)) for specifying the language or type of file for running. Although the file is mandatory to exist, it should contain the information for running only if the deployment is automatic or semi-automatic. If the deployment is manual it does not need to be completed (but the file should exist).

any other file or folder (optional)
You can include here any other file or folder you need for the deployment

Documentation\source

author.txt (mandatory) The author of the documentation of the security primitive implementation
doc_version.txt (mandatory) The version of the documentation of the security primitive implementation	title.txt (mandatory) The title of the security primitive implementation	version.txt (mandatory) The version of the security primitive implementation

files
Includes information describing the solution provided by the security primitive implementation

Interface

Describes the interfaces offered by the security primitive implementation (the solution). It can be refined as necessary including names, parameters, etc. It should comply with the interface of the implemented security primitive pattern.

Documentation\source\files

overview.rst\(^4\) (mandatory) It explains (as an executive summary) what the security primitive implementation does. It should comply with the implemented security primitive pattern.

architecture.rst (optional) Describes the technical architecture of the security primitive implementation. This architecture is tailored to a specific technology selected for implementing the corresponding security primitive pattern.

requirements.rst (optional) Here we describe the requirements (e.g. technical, functional, etc.) of the implementation and any other additional information, such as dependencies or security-related concerns.

notes_and_references.rst (optional) It can be used to describe any additional information useful for describing the security primitive implementation, such as external reference documentation.

Documentation\source\interface

codename.txt The name of the solution

interface_description.txt (mandatory) Here we describe the interface offered by the security primitive implementation

---

\(^4\) For creating or modifying files in this format you can use the online tool: https://livesphinx.herokuapp.com/
primitive implementation and the main functionalities provided. It should comply with the implemented primitive pattern.

categories.json (mandatory) | Describes the structure of the calls that compose the interface: name, parameters, return type, etc. It should comply with the categories defined for the implemented pattern.

Testing

| run_tests (mandatory) | This file specifies the instructions for testing the security primitive implementation. For specifying the language or type of file for running, it follows the magic number of the executable (https://en.wikipedia.org/wiki/Magic_number_(programming)). Although the file is mandatory to exist, it should contain the information for running the tests only if the testing is automatic or semi-automatic. If the testing is manual it doesn’t need to be completed (but the file should exist).
| any other file or folder (optional) | You can include here any other file or folder needed for testing the software solution.

3.8.1.3 TREDISEC Recipe

| Application (folder) | This includes the software of the TREDISEC Recipe (solution): source code, binaries, scripts, etc.
| Deployment (folder) | This includes the necessary files and data for the correct deployment of the solution. The deployment could be done in different manners: automatic (by directly invoking scripts, executables, etc. without intervention from a human), semi-automatic (same as automatic but requiring some interaction with the user, for example asking for user/password, configure IPs, etc.) or manual (requires a complete or major degree of human intervention, e.g. by following a step by step documentation with diagrams about how to deploy/integrate the solution).
| Documentation (folder) | This includes the documentation describing the TREDISEC Recipe, interfaces, diagrams, etc., making especial emphasis on deployment and integration information. Bear in mind a TREDISEC Recipe bundles one or more security primitive implementations, so the documentation should describe the overall solution as a whole.
| Testing (folder) | This includes all the required information for testing of the solution. Similarly to the deployment folder, the testing can be done in three different manners: automatic (e.g. scripts or executables that run the tests and compile the results without human intervention), semi-automatic (same as automatic but requiring some level of interaction with the user, e.g. to configure some variables in the scripts), manual (includes specification of the tests and may include some software, but requires of major intervention by the user to execute them).

Application

| any file or folder (optional) | No specific structure, just the necessary files (e.g. source code, binaries, etc.)

Deployment

| deploy (mandatory) | This file specifies the instructions for deploying the solution. This follows the magic number of the executable (https://en.wikipedia.org/wiki/Magic_number_(programming)) for specifying the language or type of file for running. Although the file is mandatory to exist, it should contain the information for running only if the deployment is automatic or semi-automatic. If the
deployment is manual it does not need to be completed (but the file should exist).

any other file or folder (optional) You can include here any other file or folder needed for the correct deployment of the solution

### Documentation\source

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>author.txt (mandatory)</td>
<td>The author of the documentation of the TREDISEC Recipe</td>
</tr>
<tr>
<td>doc_version.txt (mandatory)</td>
<td>The version of the documentation of the TREDISEC Recipe</td>
</tr>
<tr>
<td>title.txt (mandatory)</td>
<td>The title of the TREDISEC Recipe</td>
</tr>
<tr>
<td>version.txt (mandatory)</td>
<td>The version of the TREDISEC Recipe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>files</td>
<td>This folder includes information describing the solution provided by the TREDISEC Recipe</td>
</tr>
</tbody>
</table>

### Interface

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>codename.txt</td>
<td>Name of the solution</td>
</tr>
<tr>
<td>interface_description.txt (mandatory)</td>
<td>Here we describe the interface offered by the TREDISEC Recipe and the main functionalities provided. It should comply with the interfaces of the security primitive implementations bundled in it.</td>
</tr>
<tr>
<td>categories.json (mandatory)</td>
<td>This file is used for describing the structure of the calls that</td>
</tr>
</tbody>
</table>

5 For creating or modifying files in this format you can use the online tool: https://livesphinx.herokuapp.com/
compose the interface: name, parameters, return type, etc. It should comply with the interface calls of the security primitive implementations bundled in it.

<table>
<thead>
<tr>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_tests (mandatory)</td>
</tr>
<tr>
<td>This file specifies the instructions for testing the TREDISEC Recipe. For specifying the language or type of file that runs the tests, it should follow the magic number of the executable (<a href="https://en.wikipedia.org/wiki/Magic_number_(programming)">https://en.wikipedia.org/wiki/Magic_number_(programming)</a>). Although the file is mandatory to exist, it should contain the information for running only if the testing process is done in an automatic or semi-automatic manner. If the testing is manual, it does not need to be completed (but the file should exist).</td>
</tr>
<tr>
<td>any other file or folder (optional)</td>
</tr>
<tr>
<td>You can include here any other file or folder needed for testing the TREDISEC Recipe.</td>
</tr>
</tbody>
</table>

3.8.2 General Description of the Packages

There is a component of the TREDISEC Framework that processes and validates the artefact package content and structure, according to the above-described guidelines. More specifically, the files in the documentation folder need to follow a pre-defined format and structure, which is detailed by the guidelines below, so the framework is able to render all the information correctly in the corresponding web pages. When a new package is uploaded to the TREDISEC Framework, an HTML version of the documentation will automatically be generated, in order to allow users (with access to that package) to browse and learn about the primitive from the different levels of information and details provided.

Documentation source files use the ReStructured Text (rst) format. Note that reStructuredText is a file format for textual data used primarily in the Python programming language community for technical documentation. Additionally, thanks to its structured form, one could develop and maintain templates to automatically render Security Primitive interface specifications in different forms (e.g. PDF for deliverables and HTML for the website).

The structure of the documentation used for creating a security primitive pattern, security primitive implementation and TREDISEC Recipe are the same, as presented above. What changes is basically the level of detail of the information provided, with more abstract and high level descriptions expected for patterns, and low-level/technical and technology-tailored descriptions expected for implementations and recipes.

3.8.2.1 Security Primitive Pattern

The structure of the documentation files for a security primitive pattern is the following:

1. The documentation must contain an “Overview” section that explains in the executive summary terms what the security primitive pattern does for its users. Implementation details are not relevant except as examples of how a functionality or property of the system (e.g. security, transparency, etc.) may be achieved. This overview must refer to the relevant industry standards and practices and clearly define the added value to a potential user of the primitive. This overview should be contained in a file named: "overview rst".

2. Optionally, an “Architecture” section may describe aspects of this solution. This is useful to have but not critical as the important part is to define the security primitive interface, and then check how many different implementations exist for different platforms. The description of the architecture must come in a file named "architecture.rst". As an example, consider Figure 36.

---

6 http://docutils.sourceforge.net/rst.html
In this figure, we observe that the tab “Overview” displays the text found in the `overview.rst` and the `architecture.rst` of the “Remote Attestation” primitive.

3. The interface documentation must contain an “Interface Reference”. The reference lists all calls available to the immediate user of the primitive along with sufficient documentation enabling an engineer to design and implement a system that uses the primitive. This documentation can be seen if the user presses the “Documentation” button found at the bottom of the page seen in Figure 36. Specifically we require that:

1. The Interface has a global description addressed to the potential developer. The file that contains the interface description should be named “interface_description”.

2. The Interface has its calls organized into “Categories” by topic, method of access, or any other property deemed appropriate. Each category must have its own description defining what is common along its calls and what differs between this and other categories.

3. Each category has a list of interface calls. Each interface call has its own description addressed at the developer that explains usage and semantics.

4. Each call has a list of arguments and a return value. Each argument has a name, a type, and a description addressed to the developer that explains usage and semantics. The return value also has a type and description but no name.

5. There must be a section “Types” that describes all types referenced in the Interface Specification.

6. There must be a section “Notes and References” where notes and related reading or referenced material is linked. This information should be included in a file named “notes_and_references.rst”.

Figure 36: Overview of a Security Primitive Pattern

The need for specifying the interfaces at this level is because such interfaces describe the basic functionalities of a primitive pattern, and these will become the API eventually, when an implementation is created based on this pattern. Users that want to implement a software solution based on an already existing security primitive pattern should be able to see this interface specification, even if it is at an abstract level, to guide their developments and comply with the expected functionalities.

3.8.2.2 Security Primitive Implementation and TREDISEC Recipe

When preparing the documentation for a security primitive implementation or a TREDISEC Recipe the structure must be the same as the one described for the security primitive pattern. Additionally, a user should also include a "Testing" and "Deployment" folder. These folders should include one executable file each, respectively: run_tests and deploy. These files are responsible for installing all the dependencies needed to either perform the testing of the artefact or deploy it (for more details see section 3.10).

A user may also decide to include descriptions for the processes of testing, deploying, and configuring the artefact, as further support for future artefact consumers. Also, the exact method of accessing the interface offered (e.g. linked library, REST API), a specification of the supported target environment and any other software required to interact with the primitive, so it operates in the right way.
Within the files folder (see the complete listing in section 3.8.1) another .rst file named `known_issues.rst` must be included. This file must contain all known issues regarding the corresponding primitive implementation or recipe.

---

**TRAVIS**

![TRAVIS Image](image)

**Overview**

Remote Attestation is the activity of making a claim about properties of a target by supplying evidence to an appraiser over a network.

The Remote Attestation Component is composed of the appraiser part (Remote Attestation Client) that verifies the evidence, and the target part (Remote Attestation Server) which generates the evidence of whether or not the untrusted platform is running in the known state, and therefore, the result of the Third Party Solution is trustworthy.

**Architecture**

The Remote attestation component being developed is composed of a client part and a server part shown in Figure 37.

Remote Attestation Client

This application runs in the Cloud Service Platform and has the capability to verify the evidence provided by the Remote Attestation Server, by matching the PCR values computed with the ones stored in the learning phase, as well as the business logic to connect to the Remote Attestation Server and launch the methods in order to generate the evidence.

**Figure 37: Known issues of the Remote Attestation Primitive Implementation**

---

### 3.9 Testing Process

#### 3.9.1 Description

For any security primitive implementation or TREDISEC Recipe, the framework allows developers (i.e., security technology providers) to include their test procedure. This permits users of the artefact to see if the primitive is deployed correctly, if their functionalities are complete and work according to the specifications, following the correct testing procedure and with some meaningful test data. Moreover, the TREDISEC Framework provides developers the support for testing the primitives in multiple cloud-based platforms.

#### 3.9.2 Packaging

A package should be prepared to include sufficient testing information for anyone to run the tests on the subject artefact. The testing information can be provided in the form of manual instructions, semi-automated scripts or fully-automated scripts. Manual instructions can be text files with diagrams specifying step by step the test to be performed, together with the expected results. Automated tests and semi/fully-automatic scripts allow the framework, in the first case, and the users, in the second case, to run the prepared tests and generate the report (and match the expected results) in a much straightforward way.
Either option requires that each primitive package includes a mandatory script file test and manual instruction file readme in the “Testing” directory. If no automated scripts are available, test file will be left blank.

3.9.3 Lifecycle

The testing lifecycle supported by the TREDISEC Framework is composed of several different processes. Following we describe all the different phases, the users that are related to them and the possible actions to be taken. When possible, we reinforce the description with diagrams or figures to better understand the approach we propose for each phase. Therefore, the different phases are:

- the creation of tests for a security primitive implementation or TREDISEC recipe
- the usage of the testing information by the end-user
- the upload of the results to the TREDISEC Framework
- the visualization/access to the testing information of the security primitive Implementation/TREDISEC Recipe

The TREDISEC Framework is the orchestrator that allows the different users of TREDISEC to work collaboratively with this functionality.

3.9.3.1 Sequence diagram of creation of a security primitive implementation

Figure 38 shows the process of creating an artefact in the framework, which is packaged according to the description in sections 0 and 0. The security technology provider starts the task by creating a new artefact using the corresponding framework wizard. The framework verifies the package follows the right structure of directories: documentation + application + testing package + deployment, and saves it in the framework’s internal database.

![Sequence diagram of creation of a security primitive implementation](image)

Figure 38 Upload TREDISEC primitive package with “Testing” code and information

3.9.3.2 Test, upload and visualize testing results

The TREDISEC Framework allows end-users to share the results obtained after testing primitives in a particular test environment, and this way provide useful information to other potential users of the primitive, as well as for the providers of it. Figure 39 shows the sequence diagram corresponding to the process, which is also described below.
A tenant user first downloads the primitive package and deploys it on the cloud platform of his choice. Then, using the test scripts or manual instructions included in the package, the user performs the tests. The user then gathers the yielded results and uploads these into the framework, by accessing the artefact details page, clicking on button “Testing Upload and information”. The framework displays the interface shown in Figure 40. The “Upload testing information for package” section of the page allows users to upload the results of the primitive testing. Section “View testing information for package” of the page lists all testing results shared by other users.
3.9.4 Technical Tests

Technical tests check the correct functionality of the solution with regards to the specifications. These can cover either security-related aspects or functional-related ones. The main idea behind these tests is that the tester can check the solution according to some acceptance criteria (verification) and ensure there are no errors (validation).

Therefore, technical tests focus in the correct functioning of the application, covering all different mechanisms implemented, in/out interfaces, etc. It depends on the creator of the tests to provide information about the expected results, how to test the solution, appropriate test data to use in order to obtain meaningful results, etc.

Tests could be manual, semi-automatic or automatic, as already introduced in D2.4 and described in detail in section 3.8.1.

We provide in Annex I an example of a semi-automatic technical test that is included in a TREDISEC Recipe. There we show first, the executable file that prompt the user for the address of the service deployed, and then two examples of tests that are run automatically. This way, the results that are reported back to the user can be uploaded to the TREDISEC Framework, so other users can see them too.

3.9.5 Performance Tests

Performance tests aim to check the impact that the integration of the security primitive in a target system have in terms of processing time, volume of data transferred, data transfer speed, etc. The main idea is that the testers can compare the performance of the solution in two or more cloud-based scenarios, according to different parameters and configurations. This type of test can also be used by developers as a diagnostic tool, for locating communication problems or bottlenecks that hint the part of the system that is not responding as expected.
Among others, some common types of performance tests are *load testing* (understand the behaviour of the system under a specific expected load), *stress testing* (check limits of the system), *soak testing* (if the system can work under continuous expected load), *spike testing* (check how the system reacts when increasing/decreasing the work load under a huge number of users) and *configuration system* (check the effect of configuration changes in the system).

By running these performance tests, it would be easier to calculate how the TREDISEC solution affects the system under development (from the viewpoint of the cloud service provider). By making available this information through the TREDISEC Framework, other users could have a better understanding of the degree of improvement the solution provides, as well as anticipating any trade-off derived from integrating it.

### 3.10 Testing Environment

A TREDISEC Testing Environment (TTE) is a platform that can be used to test or deploy a TREDISEC artefact through the TREDISEC Framework. In brief, users who wish to test or deploy a recipe can select a TTE from a list that displays all the available TTEs. Then the framework connects to the corresponding TTE and attempts to perform the action dictated by the user. Finally, the TTE will send back the results to the framework, which in turn displays them to the users (see Section 3.10).

#### 3.10.1 Description

A TTE is fully aware of the packaging structure described in 3.8.2 and it expects specific files to specific places in order to test or deploy a recipe. Hence, it is important to check if this structure is correct. This is done by the framework for example when a user attempts to upload a TREDISEC Recipe. Figure 41, depicts the corresponding steps taken by the framework when the upload happens.

![Figure 41 Checking package format](image)

Figure 42 shows the basic steps taken when a user attempt to either test or deploy a recipe. Initially, a user selects a recipe (Rx). Then, the framework displays a list of all available TTEs to the user. The user selects a TTE (TTE-0x) and the framework sends the recipe to the specified TTE. The TTE then unpacks the recipe and performs the action that the user selected. Finally, it gathers the results and uploads them back to the framework, which in turn displays them to the user.
Note that in step 3.2 the TTE runs a specific file depending on the user action. This means that the TTE knows where to find what after unpacking the recipe. Specifically, the “Testing” directory contains all the files needed to run the tests. The entry point is the “run_tests” file. This is an executable file, responsible to install all the dependencies needed to run the tests, and start the testing procedure. “run_tests” will be allowed to call whatever method they need to complete their job, e.g. call Ansible or Docker, but they will also be responsible for installing them as dependencies. The tests produce results under the “Testing/artefacts/” directory, which is then archived and uploaded to the framework.

In case of a deployment, the “Deployment” directory contains all the files needed for the deployment process. The entry point is another specified file named “deploy”. “deploy” has a similar role to “run_tests”. In essence, it is an executable that installs all the dependencies needed for the deployment process, and starts the procedure.

3.10.2 Technical description

TTEs work with the help of a daemon. The TREDISEC Daemon comes preinstalled in every TTE. The daemon is actually a web server that listens for commands coming from the TREDISEC Framework. Note that the two communicate through a secure channel. To do so, the admin who creates the TTE needs first to acquire the token that is provided when he or she creates the corresponding TTE instance in the framework. Then the admin must configure the daemon to use this token every time the framework communicates with the TTE.

3.10.3 Management

Only administrators can add TTEs to the framework, through the corresponding Django administration panel. Once a TTE instance is created in the framework, the framework generates a token. This token must be then added to the TTE for authentication purposes. Note that, a TREDISEC TTE must come with some preinstalled packages to provide the functionalities mentioned above. Specifically, a TTE must come with the following packages installed: Python, Django, Node.js Apache and some technologies described in Section 4.5.

3.11 Deployment Process

The deployment process is very similar to the testing one. In this subsection we provide some further details regarding the deployment process and point out its differences with the testing process.

3.11.1 Description
All TREDISEC artefacts (i.e., security primitives and recipes) must include deployment information instructing on how to deploy and integrate the artefact into a target cloud-based platform.

### 3.11.2 Packaging

Similar to the testing process, the deployment package can either include manual instructions about how to deploy and integrate the artefacts into the target platform, or one or several scripts that can deploy the artefacts automatically, given the correct configurations.

Either option requires that each primitive package includes a mandatory script file `deploy` or manual instructions file named `readme`, in folder `Deployment`. In the case where no automated scripts are provided by the developer, the `deploy` file will be left blank.

### 3.11.3 Lifecycle

The deployment lifecycle is supported by the framework in various stages. Consider a user who wants to deploy a specific recipe for which she has access to (i.e. she has the appropriate permission to view and use it).

In the details page of the recipe, the user must press the “Deploy” button. Then a dialog, shown in Figure 44, is presented to the user, containing all the available TREDISEC Testing Environments (TTEs). The user selects the environment where the artefact will be deployed. After that, the framework communicates with the selected TTE in the background. After a successful authentication, some instructions on how to deploy the recipe (as provided by the recipe owner) are followed. The TTE then fetches the artefact from the framework repository, unpacks it and validates the contents. Finally, the TTE runs the `deploy` executable file. Once the process is done, the TTE collects the logs and sends the results back to the TREDISEC Framework. In turn, the framework displays them to the user, as explained in section 3.9.3.2 and shown in Figure 43.

Similarly, to obtain a recipe and deploy it in a cloud-based environment of the end-user's choice (not supported by the framework), the user simply downloads the package and follows the instructions from the Deployment package. If the user's environment is not included in the target platforms of the deployment package, the user can prepare and create another deployment package, thus a new TREDISEC Recipe, for the new platform and upload to the framework.

### 3.11.4 Exchange of information between the TREDISEC Framework and a TTE

As we discussed earlier, users can either test or deploy a recipe into a specified TTE (see Figure 43). An action like this though, is not always successful and can produce different results. Hence, users must be aware of what happened at the TTE. Specifically, depending on the action, they must see: (1) if the deployment was successful or not, and (2) the results of the tests. The former involves a simple notification that users will see (“Successful Deployment” or “Deployment Failed”). In case of a deployment failure, the errors will be also shown. The latter may involve more than one result because the developers of a recipe could provide different tests (unit tests, performance tests and more).

Note that, each action must be related to a specific TTE. Hence, the test results are combined with a corresponding TTE. In addition, the results are saved and every user that has access to this recipe can view them. Figure 44 presents the testing results of a TREDISEC recipe, in a specified TTE (TTE-01).
Select action *

Install

Select TREDISEC Testing Environment *

TTE 01

Figure 43 Interface for deploying or testing a TREDISEC Recipe

Figure 44 Testing information in the TREDISEC Framework
4 TREDISEC Framework Technical Specifications

4.1 Description

As it was described in Section 2 and Section 3, the TREDISEC Framework provides several functionalities. In order to implement all of them we followed a modular approach in which we implemented components (architectural building blocks) targeting specific goals. These components exchange information via communication channels or storage services, allowing for a fast and easy collaboration between all modules. Figure 45 shows a high-level description of the architecture of the TREDISEC Framework following this modular approach.

![Figure 45 TREDISEC Framework modular architecture](image)

As it is shown in Figure 45, each building block focuses in certain specific aspects/functionalities offered to users, more specifically: for managing the artefacts of TREDISEC (security primitive pattern, security primitive implementation and TREDISEC Recipe), creation and management of users and tenants, management of the testing environment, and the web interface and elements (e.g. API) that allow for external exchange of information or integration with an existing system.

4.2 TREDISEC Framework deployment components

The TREDISEC Framework architecture building blocks are implemented to be deployed into two separate environment components: a Framework server and a Testing Platform Environment. The second component is not mandatory but necessary for the framework to provide support for primitive testing and deployment functionalities. A high level description of this deployment environment is shown in Figure 46.

The Framework server hosts the TREDISEC framework web application, which is implemented using Django, a python-based framework with SQLite database for persistence of the data managed. This web application offers a GUI for users to interact with all the functionalities the framework provide, described throughout section 3, corresponding to the four blocks inside the rectangle depicted previously in Figure 45.

The storage of the information related to the primitive artefacts is split into two elements: the zip files associated to each artefact (i.e. packages) are stored in the filesystem, using the structure and format explained in Section 3.8.2; while the SQLite database only stores the corresponding artefact metadata, to enable efficient searches and rendering of the information associated to artefacts in the framework web pages. This is explained in technical terms in section 4.4.
In addition to the web application, and to improve the automatic processing and generation of the primitive artefact documentation, the Framework employs a combination of a Redis server with Celery task processing system. This combination allows the framework to efficiently schedule the generation of HTML documentation of the primitive/recipe on uploading time in an asynchronous manner. The process is queued by the framework and processed in the background, returning back the control to the user to keep working in other features of the framework. The user is informed of the finalization of each queued process by means of a tag displayed beside the artefact name, signalling whether the documentation processing is Pending (orange tag), Passed (green tag) or Failed (red tag).

TREDISEC recipes can be deployed in a separated Testing Platform Environment, as explained in Section 3.10. This environment is a cloud-based system that can be hosted in the same physical infrastructure as the Framework server, or in an external cloud-enabled system. Either case, it must be associated to the Framework server to make it accessible for communication purposes, as explained later in section 0. This environment supports the framework in managing and making available for users different TREDISEC Testing Environments (TTEs), that is virtual environments for the deployment and testing of primitives.

![Framework Server Diagram](image)

**Figure 46. TREDISEC Framework system**

### 4.3 TREDISEC Framework Interfaces

The TREDISEC Framework uses standard REST endpoints to perform actions, such as uploading or editing a primitive package, user management, searching primitives, etc. This way, it's possible to use the TREDISEC Framework without the web interface, by invoking the functionalities in console or with any other tool, as long as it follows the REST standard, by connecting to the endpoints described in Table 1.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Data Needed</th>
<th>Use</th>
<th>POST</th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-create/</td>
<td>User information fields</td>
<td>Creates a new user account for the Framework</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>user-edit/</td>
<td>User information fields</td>
<td>Edits the user account with the provided information</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>user-list/</td>
<td>User information fields</td>
<td>Shows the user list with their respective information</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>platform-list/</td>
<td>Show the platform list with their respective information</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 1 Interfaces of the TREDISEC Framework

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform/</td>
<td>Shows the selected platform information with the list of recipes that are using it</td>
</tr>
<tr>
<td>search/</td>
<td>Lists any primitive or recipe that match the search parameter on any field</td>
</tr>
<tr>
<td>advanced_search/</td>
<td>Lists only the primitives or recipes that match all the corresponding parameters</td>
</tr>
<tr>
<td>remove/(item_type)/(id)</td>
<td>Deletes the item_type that can be either pattern, implementation, recipe or platform, and the given item id</td>
</tr>
<tr>
<td>pattern/</td>
<td>Lists all the security primitive patterns</td>
</tr>
<tr>
<td>pattern/(id)</td>
<td>Returns the whole information regarding the Security primitive pattern with the given ID</td>
</tr>
<tr>
<td>pattern-wizard/(id)</td>
<td>If pattern ID is given in the endpoint, then the wizard is treated as edition of the corresponding pattern, otherwise, it creates a new security primitive pattern with the given information fields</td>
</tr>
<tr>
<td>implementation/</td>
<td>Lists all the security primitive implementations</td>
</tr>
<tr>
<td>implementation/(id)</td>
<td>Returns the whole information regarding the security primitive implementation with the given id</td>
</tr>
<tr>
<td>implementation-wizard/(id)</td>
<td>If implementation ID is given in the endpoint, then the wizard is treated as edition of the corresponding implementation, otherwise, it creates a new security primitive implementation with the given information fields</td>
</tr>
<tr>
<td>recipe/</td>
<td>Lists all the TREDISEC recipes</td>
</tr>
<tr>
<td>recipe/(id)</td>
<td>Returns the whole information regarding the TREDISEC recipe with the given id</td>
</tr>
<tr>
<td>recipe-wizard/(id)</td>
<td>If recipe ID is given in the endpoint, then the wizard is treated as edition of the corresponding recipe, otherwise, it creates a new TREDISEC recipe with the given information fields</td>
</tr>
<tr>
<td>tenancy/</td>
<td>Lists all the Tenants in the Framework with the corresponding members</td>
</tr>
</tbody>
</table>

### 4.4 Data Storage

The TREDISEC Framework uses the Django Web Framework which maps Class models into database tables, and Class instances into the corresponding table rows. The models User and Group are the default classes that are used by Django to handle the Web Framework users and permissions, and that can be expanded to be used in the Web Application itself. In this case, the User model was expanded to have extra information like the bio description, and the Group model was used to handle the Tenancy permissions. This is shown high-level in Figure 47.
The Security Primitive Patterns, Implementations and the TREDISEC Recipes, all inherit from the Class Package which contains basic information regarding the owner, TestingResults, Tenancy, etc.

As specified in the architecture defined in Section 3.1, the Security Primitive Implementation implements a security primitive pattern (Pattern), and the TREDISEC Recipes contain one or more Security Primitive Implementations.

The Django database only stores the metadata of the Security Primitive Patterns, Implementations and TREDISEC Recipes, not the binary file itself, to avoid making the database too big and slow unnecessarily. The file itself is unzipped and stored in the filesystem in a folder within the Django Project in the folder ‘django_root/tredisec_project/tredisec_project/uploads’, using the following structure:

Each Package is stored in a folder with the respective name of the class (Pattern, Implementation, Recipe), and inside, there are folders with the unique ID assigned to the primitive at uploading time (after completing the wizard). Each numbered folder contains files and folders with the structure explained in section 0. The folder of the package is created automatically by unzipping the package of the primitive uploaded by the user to the TREDISEC Framework. So for example, for a Security Primitive Implementation with ID 1, Security Patterns with ID 3 and 5, and a TREDISEC recipe with ID 24, the uploads folder will be as presented in Table 2.
4.5 TREDISEC Framework Development Environment

In order to develop and integrate the different building blocks we have described so far, we created a development environment with several tools aiming to support collaborative work and integration of the different modules and artefacts developed in the project.

4.5.1 Django

The base Web Application is coded in Python 2.7, using the Django Web Framework version 1.8 and Bootstrap 3 library for the UI elements and the look-and-feel in general.

The Framework uses the internal Django database (SQLite3) to store the Django user data, and the Framework objects as explained in section 4.4.

The packages of the primitives are stored in the filesystem, in a data folder inside the Django project, and the Framework objects in the database point to the corresponding file path, to avoid storing large binary files in the database as explained in section 4.4.

4.5.2 Sphinx

Sphinx7 is a documentation generator written and used by the Python community. It is written in Python. Sphinx converts reStructuredText files into HTML websites and other formats including PDF, EPub and more. reStructuredText is extensible, and Sphinx exploits its extensible nature through a number of extensions – for auto generating documentation from source code, writing mathematical notation or highlighting source code, etc. We have employed Sphinx and modify it to generate the documentation of the security primitive patterns, security primitive implementations and TREDISEC Recipes. Specifically, the framework searches for the “Documentation” directory (see Subsection 3.8.2) to generate corresponding HTML pages for each artefact.

4.5.3 APIMAS

APIMAS8 is a framework developed by GRNET. It provides an intuitive way to build scalable REST APIs and serving applications. It is written in Python. The goal of APIMAS is to provide a flexible way for building, modifying and extending your application without the cumbersome management due to the complexity and the size of it. We have modeled the REST API of the TREDISEC Framework using this technology.

4.5.4 Celery

Celery9 is an open source asynchronous task queue or job queue, which is based on distributed message passing. While it supports scheduling, its focus is on operations in real time. The execution units, called tasks, are executed concurrently on one or more worker nodes using multiprocessing, eventlet or gevent. Tasks can execute asynchronously (in the background) or synchronously (wait until ready). Celery is used in production systems, for instance Instagram, to process millions of tasks.

---

8 [https://github.com/grnet/apimas](https://github.com/grnet/apimas)
every day. We employ Celery to generate the documentation of the security primitive patterns, security primitive implementations and TREDISEC Recipes.

4.6 TREDISEC Framework Testing: process and supporting tools

4.6.1 Source code version management

In order to ease collaborative development and increase the stability and the consistency of the code, we have used a version control system with tracking capabilities, codenamed branches and tags, which is Git.\(^{10}\)

In particular, we used an instance of GitLab hosted by Atos, to create a GitLab project dedicated to the development and testing of the TREDISEC framework. Figure 48 shows a screenshot of the GitLab project main page.

![Figure 48 A view of the main page of the TREDISEC framework GitLab project](image)

We also defined a process to manage the development of features as well any changes and updates derived from the testing process, by using a combination of milestones, issues, branches and tags.

There are two main branches: develop and master. The *master* branch keeps a stable version of the code, while *develop* keeps the active and most updated version of the code. Figure 49 shows the list of branches active in a certain point in time of the development process, in particular the two main branches *master* and *develop*.

\(^{10}\) [https://en.wikipedia.org/wiki/Git](https://en.wikipedia.org/wiki/Git)
First, before adding a new feature or fix a bug to the main development branch, the developer creates a sub branch with the type of change and the name of the change (either a feature or issue), and then the name of the feature to add, or the number of issue to fix, for example: feature-advanced-search, or issue-71.

When the feature is stable or the issue is solved, then the branch is merged to the main development branch to make sure that the developers are working with a relatively stable version of the code.

Additionally, the master branch is locked with the most stable version of the code, which is the product of the most recent milestone, which has been tested and debugged for an internal demo in the consortium.

4.6.2 Internal Testing

As mentioned in section 4.6, a new milestone tag is created based on the project internal timeline and needs. For this, the Framework has to pass a battery of tests performed internally by non-developers within the WP6 members.

The results of these tests may generate new issues (bugs reported) that require changes in the code. These changes are handled with branches, according to the process described in the previous section.

After releasing a stable version that addresses all issues reported by testers, the code is merged to the master branch and deployed in a stable (non-development) environment, hosted and maintained by GRNET. Then it’s open for the whole consortium to test.

A specific role in the GitLab project has been used for the tester users (i.e. reporter) from within the consortium, so anyone can report a bug or propose a new feature, using the internal Gitlab ticket system.

4.6.3 Labelled issues

The Gitlab internal ticket system allows assigning color coded labels to issues, in order to ease management, filtering and to improve efficiency and readability.
We have created a number of labels that fit the project needs, and followed best practices to assign them to the newly created issues and the right developer.

The issues reported by the consortium testing accounts are labelled first as “testing”, to separate them from the issues found by the internal WP6 testing, and then, after study them, we apply a more appropriate label (e.g. bug, feature) and assign to a responsible person in the development team. Labels are listed in Figure 50.

![Figure 50 Labels specified in Git for testing of TREDISEC Framework](image)

The labels we use in TREDISEC are:

- The label “blocked” which means that the issue cannot be solved immediately because we require more information, or another issue has to be solved first, such as creating a new feature.

- The label “bug” is self-explanatory, but we only apply this label after confirming a problem in the code, to avoid eventualities like connection issues, or human error.

- The label “documentation” is usually applied for inconsistencies between a document and the Framework itself, for example, the user manual missing a newly introduced feature.

- The label “feature” is for the issues that require a substantial change in the code to introduce a new feature in the Framework or a more efficient way to perform an action.

- The label “Infrastructure” is used for issues that are not related to the code of the Framework itself, but the infrastructure behind it, for example problems with the proxy blocking a port, or the creation of a new testing environment.

- The label “release” is used to group issues together that have priority to be solved and included in the next stable release.

- The label “testing” as explained above, is used as a first step to review issues created by the consortium testers before applying a better classification with one of the other labels. For example, we apply the label “bug” to an issue after reproducing the problem and confirming it, or instead we could apply the label “infrastructure” if the issue was caused by the server being down.

### 4.7 How to install and configure your own TREDISEC Framework

#### 4.7.1 Technical requirements of the environment
4.7.1.1 TREDISEC Framework

The TREDISEC Framework server uses lightweight applications like Django, SQLite, Celery and Redis, so the requirements are mostly proportional to the expected number of users and primitives/recipes uploaded.

For a moderate use, the TREDISEC Framework server is okay with a medium-end computer or virtual machine (4 - 8 GB of RAM), and the disk space required, depending of the size of the uncompressed primitives/recipes, it may need up to 20 GB of free disk space.

4.7.1.2 Testing and Deployment environment

The Testing and Deployment environment specifications depend directly on the artefacts' own needs. As the artefacts' needs vary, so will the specifications for the Testing environment. Some artefacts may require a lot of RAM; other may require a lot of disk space, etc. Hence, users who want to set up their environments should take into account what kind of recipes will they run there.

An example of a testing and deployment environment has been setup by GRNET, with two VMs with 1GB RAM, 1 dedicated CPU and 5GB of disk space.

If the testing and deployment environment are going to be used as a TREDISEC Testing environment, then Node.js must be installed and supported too (see also Subsection 4.7.4).

4.7.2 Installation instructions for the framework application

Currently, the source code is hosted in the Git server of ATOS, and it's necessary to have an account to check-out the project. Eventually, it will be openly available when the development is complete and there is final version (after the validation process).

To download the current version, first request a Git account to a contact from ATOS, and clone the repository:

```
git clone git@gitlab.atosresearch.eu:ari/tredisec_wp6.git
``` 

git checkout develop

The branch develop has the latest code, while the master branch has the most stable code.

In the root folder, you can find a linux script to setup the project for you, after installing the required software and/or components as described below:

```
apt-get install python2.7 python2.7-dev python-django virtualenv python-pip build-essential autoconf libtool pkg-config python-opengl python-imaging python-pyrex python-pyside.qtopenogl idle-python2.7 qt4-dev-tools qt4-designer libqtgui4 libqtcore4 libqt4-xmllibqt4-test libqt4-script libqt4-network libqt4-dbus python qt4 python-qgl libgile3 python-dev texlive-latex-base texlive-formats-extra
```

./setup all

This will automatically download additional dependencies with Pip, and compile them so the python virtual environment will have all the necessary things to run the Django server.

The virtual environment is a local instance of python with customized dependencies created in the root folder of the Git repository, and can be accessed with:

```
source venv/bin/activate
```

This will add “(venv)” to the linux’s prompt to let you know that the current console is using that python’s virtual environment, which is necessary to run the Django instance or the Redis server.

To start the Django server, use the usual manage.py script inside the Django project folder:

```
cd tredisec_framework/django_root/tredisec_project/
./manage.py runserver
```

Or from the setup.sh script in the root folder of the Gitlab repository:
/setup.sh runserver

The first time you run the server, you'll need to do the initial migration of the Django database:

./manage.py makemigrations
./manage.py migrate

Additionally, the first time you run the server, you'll be asked to create the Django admin account, which can be used in the TREDISEC Framework itself as an admin account to create normal users, or other admin accounts.

Before you can create primitives or recipes, you need to start the Celery server as explained below.

### 4.7.3 Redis/Celery

First you need to download and install the last Redis server here: https://redis.io/download

Then, activate the virtual environment as explained in the section 4.7.2

From a console with the virtual environment active, install the celery and redis python packages with pip:

```bash
pip install celery redis
```

In a terminal window with the virtual environment, start the redis server with default settings:

```bash
redis-server
```

Make sure redis-server is running by pinging the server:

```bash
redis-cli ping
```

Now in terminals with the virtual environment active, go to the project folder, and start the Celery server and Celery worker to handle tasks:

```bash
cd tredisec_framework/django_root/tredisec_project/
celery beat -A tredisec_project --loglevel=INFO
(Another terminal)
celery worker -A tredisec_project --loglevel=INFO
```

Now you should be able to create and manage primitives and recipes in the Framework

Installation instructions for the deployment/testing environment of the primitives

A TTE must come with the following packages installed: Python, Django, Apache and Node.js. Note that Node.js is particularly important because the daemon described in Subsection 3.10.2, depends on it. Installing Node.js on an Ubuntu or a Debian distribution is relatively easy:

```bash
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Once Node.js is installed the TREDISEC Daemon needs to be installed too (as a Node.js component):

```bash
npm install -g tredisec-tte-daemon
```

This daemon must be configured with the secret token that the admin acquires when he or she creates the corresponding TTE instance in the framework (see section 3.10). Then, the admin must configure the daemon accordingly. Specifically, the admin must create a file named `secret_token` in the following path: `/etc/tredisec-tte/` and place the token in it (all this on the TTE's side). In this way, the TTE will be able to use it every time the framework communicates with it.
TTE-06

Description
Linux Debian 8 server with Intel CPU 3.2 GHz, 4GB Ram, 250GB SSD.

Secret Authentication Token
39693322-1645-4744-a0bd-1a0f396372b4

Connect to the TREDISEC Testing Environment and copy this token into the file at the this location: /home/username/.tredisec/ttt_data.dat

Figure 51 Definition of secret token in TTE
5 Conclusions and Future Work

The TREDISEC Framework has been developed using as basis the architecture designed in WP2 at the beginning of the project. This architecture design, which went through several iterations, is driven by the requirements and needs of both use cases and technical tasks of the project, so we could satisfy all needs, at both cloud security domain and business level.

The TREDISEC Framework follows a modular design so all the functionalities to be provided are decomposed into specific-purpose building blocks. All of them are connected internally so the exchange of information is done via well-defined interfaces and an efficient data storage system, as described in Section 4.

In order to make the collaborative implementation process as much smooth and efficient as possible, we created a shared development environment with several tools and guidelines that facilitated the joint work between the partners. This served not only at development time, but also during the testing, bug finding and fixing stages.

Currently we support two versions of the TREDISEC Framework, one for stable releases that partners can use for managing their security primitive patterns, security primitive implementations and TREDISEC Recipes, as well as for deploying and testing them. Another framework environment instance is deployed for testing, where we integrate new functionalities, request for changes, bug fixes, etc. and while keep doing in-depth testing. That way we can work in parallel in continuously improving the framework software, while let partners of the project to create content in the framework and benefit from their capabilities.

One of the most interesting elements we have implemented in the framework is the support for primitive deployment and testing functionalities. We have created a module for managing several virtual test environments (called TREDISEC Testing Environment, or TTE) from the framework to support these functionalities, as external elements. That way, administrators of the TREDISEC Framework are not tied to specific cloud-based infrastructures and can chose to provide the ones they think best fit. The tests developed by the owners of the security primitive implementations or TREDISEC Recipes can be run in these TTEs, in order to check its correct functionality or performance results. We also have made possible to upload these results to the framework, so other users that want to use the solution may learn in advance if the tool performs as expected, even before deploying it. This allows for benchmarking of the same primitive implementation in different target cloud environments (i.e. different recipes), or comparison of different implementations of the same primitive pattern that uses different technologies (e.g. different programming languages, databases, etc.).

At the current state of the project we have integrated all the functionalities foreseen in the architecture documents (i.e. D2.3 and D2.4) and are currently refining and updating them in order to make them more user-friendly, intuitive and providing extra functionalities in order to improve its performance. We expect new bugs and requests for improvement will arrive in the use case validation phases (Task 6.3) and when involving external end-users in the validation process.

Also, we want to take further steps towards the idea of making the TREDISEC Framework work both as independent framework that provides (and supports lifecycle) of security and functional solutions and to enable the integration in another system (e.g. an existing Cloud service provider infrastructure). This will foster the framework exploitation possibilities and contribute to a promising sustainability strategy.
6 Annexes

Annex I: Examples of functional tests of EPICA TREDISEC Recipe

Semi-automatic testing main file

```
#!/bin/bash
# This script launches the Policy Administration REST API Tester against a given URL

echo "Welcome to the EPICA Policy Administration Manager REST API Tester"
read serveraddress
read serverport

java -jar PolicyAdministrationApiTester.jar $serveraddress $serverport

echo "Finished tests, press enter to exit"
read

Examples of tests executed automatically in the TREDISEC Testing Environment

/**
   * Tests all the <code>POST</code> methods of the database
   */

@Test
public void testPost() {
    int currentTest = 1;
    int maxTests = 6;
    // Testing valid post
    String postBody = postPolicyStoreData;
    // System.out.println("Read data: "+policyStoreData);
    // System.out.println("For the ID "+POLICY_STORE_ID);
    HttpResponse response = postPolicyStore(POST_POLICY_STORE_ID, postBody);
    String description = "Creating a valid policy store with basic data";
    int expectedCode = 201;
    assertTest(description, expectedCode, response.getStatusLine().getStatusCode(), currentTest, maxTests, "POST");
    currentTest++;
    // Testing with an already existing policy store
    response = postPolicyStore(POST_POLICY_STORE_ID, postBody);
    description = "Creating an already existing policy store (using same data as previous test)";
    expectedCode = 400;
    assertTest(description, expectedCode, response.getStatusLine().getStatusCode(), currentTest, maxTests, "POST");
    currentTest++;
    // Testing with an empty policy store id
    response = postPolicyStore("", postBody);
    description = "Testing with an empty policy store id";
    expectedCode = 405;
    assertTest(description, expectedCode, response.getStatusLine().getStatusCode(), currentTest, maxTests, "POST");
    currentTest++;
    // Testing with invalid data
    response = postPolicyStore(POST_POLICY_STORE_ID, "asfsdf");
    description = "Creating a policy store with invalid data";
```
expectedCode = 400;
assertTest(description, expectedCode, response.getStatusLine()
 .getStatusCode(), currentTest, maxTests, "POST");
currentTest++;
// Testing adding a policy for a first-level resource
response = postPoliciesForResource(POST_POLICY_STORE_ID,
 POST_TEST_RESOURCE_1, basicValidPolicy);
description = "Adding a policy for a first-level resource with the id "
 + POST_TEST_RESOURCE_1;
expectedCode = 201;
assertTest(description, expectedCode, response.getStatusLine()
 .getStatusCode(), currentTest, maxTests, "POST");
currentTest++;
// Testing adding a policy for a second-level resource
response = postPoliciesForResource(POST_POLICY_STORE_ID,
 POST_TEST_RESOURCE_2, basicValidPolicy);
description = "Adding a policy for a second-level resource with the id "
 + POST_TEST_RESOURCE_2;
expectedCode = 201;
assertTest(description, expectedCode, response.getStatusLine()
 .getStatusCode(), currentTest, maxTests, "POST");
currentTest++;
// Testing adding a policy for a second-level resource without parent
response = postPoliciesForResource(POST_POLICY_STORE_ID,
 POST_TEST_RESOURCE_34, basicValidPolicy);
description = "Adding a policy for a second-level resource with a parent without
 policies. The id is: "
 + POST_TEST RESOURCE 34;
expectedCode = 201;
assertTest(description, expectedCode, response.getStatusLine()
 .getStatusCode(), currentTest, maxTests, "POST");
currentTest++;
}/**
 * Tests all the <code>DELETE</code> methods of the database
 */
@Test
public void testDelete() {
 postPoliciesForResource(DELETE_POLICY_STORE_ID, DELETERESOURCE_ID,
 deleteValidPolicy);
int currentTest = 1;
int maxTests = 4;
// Test deleting an existing resource
HttpResponse response = deletePoliciesForResource(
 DELETE_POLICY_STORE_ID, DELETERESOURCE_ID);
String description = "Deleting an existing resource";
int expectedCode = 202;
assertTest(description, expectedCode, response.getStatusLine()
 .getStatusCode(), currentTest, maxTests, "DELETE");
currentTest++;
// testDeletePolicyNotFound
response = deletePoliciesForResource(DELETE_POLICY_STORE_ID,
 "sdfsdfsdf");
description = "Deleting a non existing resource";
expectedCode = 404;
assertTest(description, expectedCode, response.getStatusLine()  
    .getStatusCode(), currentTest, maxTests, "DELETE");
currentTest++;
// Test delete a policy store
response = deletePolicyStore(DELETE_POLICY_STORE_ID);
description = "Deleting an existing policy store";
expectedCode = 202;
assertTest(description, expectedCode, response.getStatusLine()  
    .getStatusCode(), currentTest, maxTests, "DELETE");
currentTest++;
// //Test delete a policy store that does not exist  
// response = deletePolicyStore(  
// DELETE_POLICY_STORE_ID);
// description = "Deleting a non-existing policy store";
// expectedCode = 404;
// assertTest(description, expectedCode, response.getStatusLine()  
// .getStatusCode(), currentTest, maxTests, "DELETE");
}
Annex II: Categories JSON file of EPICA TREDISEC Recipe

```json
{
    "policy_administration_api": {
        "category_name": "PolicyStore",
        "calls": {
            "DELETE": {
                "args": {},
                "return": {}
            },
            "GET": {
                "args": {},
                "return": {
                    "return_description": "Gets the Policy Store in a specific format (JSON)"
                }
            },
            "POST": {
                "args": {
                    "uuid": {
                        "arg_description": "the Policy Store in a specific format (JSON)"
                    }
                },
                "return": {}
            },
            "PUT": {
                "args": {
                    "uuid": {
                        "arg_description": "the Policy Store in a specific format (JSON)"
                    }
                },
                "return": {}
            }
        }
    },
    "access_request_evaluation": {
        "category_name": "Access Request Evaluation Operations",
        "calls": {},
        "category_description": "This category groups the interface calls related to the evaluation of the access control requests"
    },
    "policy_enforcement": {
        "category_name": "Policy Enforcement Operations",
        "calls": {},
        "category_description": "This category groups the interface calls related to the enforcement of access control policies"
    }
}
```
Annex III: Creation process of a security primitive pattern

1. Download/obtain template of the security primitive pattern

You can obtain the template from the TREDISEC Framework in the Section TREDISEC Resources.

2. Extract the .zip file. You will then have four different folders

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>05/03/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>Deployment</td>
<td>21/03/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>Documentation</td>
<td>21/03/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>Testing</td>
<td>21/03/2017</td>
<td>File folder</td>
</tr>
</tbody>
</table>

The description of each folder can be found at the “Security Primitive Pattern package structure”.

3. Modify the mandatory files and add information of the overview, architecture, requirements, and nodes and references. For a security primitive pattern, the mandatory ones are in the folder Documentation.

4. Access to Documentation (folder) -> source (folder). The first set of documents to complete are: author.txt, doc_version.txt, title.txt, version.txt

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>files</td>
<td>21/03/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>interface</td>
<td>21/03/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>author.txt</td>
<td>01/09/2017</td>
<td>Text Document</td>
</tr>
<tr>
<td>doc_version.txt</td>
<td>12/01/2017</td>
<td>Test Document</td>
</tr>
<tr>
<td>title.txt</td>
<td>04/09/2017</td>
<td>Test Document</td>
</tr>
<tr>
<td>version.txt</td>
<td>01/09/2017</td>
<td>Test Document</td>
</tr>
</tbody>
</table>

In order to complete files you can check the section of “Security Primitive Pattern package structure”, which describes each file.

5. Now we have to complete the information in the folders files and interface.
6. We access first the folder files

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>images</td>
<td>05/09/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>architecture.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
<tr>
<td>notes_and_references.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
<tr>
<td>overview.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
<tr>
<td>requirements.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
</tbody>
</table>

The description of each file can be found in the section “Security Primitive Pattern package structure”. The output of these file, after being processed by the tool of the TREDISEC Framework is as follows:

<table>
<thead>
<tr>
<th>File</th>
<th>Section in the TREDISEC Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>overview.rst</td>
<td>Overview tab in the security primitive pattern. It is also shown in the documentation of the security primitive pattern.</td>
</tr>
<tr>
<td>requirements.rst</td>
<td>Requirements tab in the security primitive pattern</td>
</tr>
<tr>
<td>architecture.rst</td>
<td>Overview section in the documentation of the security primitive pattern. It is not shown in the tab of Overview.</td>
</tr>
<tr>
<td>notes_and_references.rst</td>
<td>Notes and references section in the documentation of the security primitive pattern.</td>
</tr>
</tbody>
</table>

The file *architecture.rst* must be referenced in the file *overview.rst* in order to be parsed and included in the documentation of the TREDISEC Framework. In order to do so please add at the end of the file *overview.rst* the following line:

```
include:: architecture.rst
```

In order to modify/complete these files you can use the tool Online Sphinx editor. This editor provides tools for creating different structures in the text (e.g. sections, sub-sections, bullet points, etc.), insert figures, etc. Also, being online makes easy to work with the original text and see directly the result after compiling (what you see is what you get).

Then, users could copy the original text and paste directly in the .rst files we described previously. You can find the tool at the following address:

https://livesphinx.herokuapp.com/
7. **We access now the folder interface**

The file `codename.txt` must include the name of the security primitive pattern. It is used for the naming of the solution in the documentation although it is not mandatory for the creation of the package. This means if it is not provided the generation of the documentation would success but it could have some problems for labelling some elements of the documentation (e.g. titles, references, etc.).

The file `categories.json` contains the description of the interfaces of the security primitive pattern. This element helps to defining the interfaces from the point of view of the calls with their arguments and output. Below we show the structure of this element:
8. After all files have been completed we create a .zip file of the folders shown in the figure of Step 2.
9. We now access the TREDISEC Framework and click in the option Security Primitive Pattern

10. We click in “Create Security Primitive Pattern”

11. We complete now the information of the security primitive pattern
12. We search using the browser for the .zip we just created

13. We click in Submit

14. We select the access level for the security primitive pattern
15. After selecting our preferences we click submit and the security primitive pattern will be correctly created in the TREDISEC Framework.
Annex IV: Creation process of a security primitive implementation

1. **Download template of the security primitive implementation**

   You can obtain the template from the TREDISEC Framework in the Section TREDISEC Resources.

2. **Extract the .zip file. You will then have four different folders**

   - Application
   - Deployment
   - Documentation
   - Testing

   The description of each folder can be found at the “Security Primitive Implementation package structure”.

3. **Modify the mandatory files and add information of the overview, architecture, requirements, and nodes and references.** For a security primitive implementation, the mandatory ones are in the folder Documentation, Deployment and Testing.

4. **Access to Documentation (folder) -> source (folder).** The first set of documents to complete are: author.txt, doc_version.txt, title.txt, version.txt

   In order to complete files you can check the section of “Security Primitive Implementation package structure”, which describes each file.
5. Now we have to complete the information in the folders files and interface

6. We access first the folder files

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>images</td>
<td>05/09/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>architecture.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
<tr>
<td>notes_and_references.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
<tr>
<td>overview.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
<tr>
<td>requirements.rst</td>
<td>19/09/2017</td>
<td>RST File</td>
</tr>
</tbody>
</table>

The description of each file can be found in the section “Security Primitive Implementation package structure”.

The output of these file, after being processed by the tool of the TREDISEC Framework is as follows:

<table>
<thead>
<tr>
<th>File</th>
<th>Section in the TREDISEC Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>overview.rst</td>
<td>Overview tab in the security primitive implementation. It is also shown in the documentation of the security primitive implementation.</td>
</tr>
<tr>
<td>requirements.rst</td>
<td>Requirements tab in the security primitive implementation</td>
</tr>
<tr>
<td>architecture.rst</td>
<td>Overview section in the documentation of the security primitive implementation. It is not shown in the tab of Overview.</td>
</tr>
<tr>
<td>notes_and_references.rst</td>
<td>Notes and references section in the documentation of the security primitive implementation.</td>
</tr>
</tbody>
</table>

The file architecture.rst must be referenced in the file overview.rst in order to be parsed and included in the documentation of the TREDISEC Framework. In order to do so please add at the end of the file overview.rst the following line:

```
include:: architecture.rst
```

In order to modify/complete these files you can use the tool Online Sphinx editor. This editor provides tools for creating different structures in the text (e.g. sections, sub-sections, bullet points, etc.), insert figures, etc. Also, being online makes easy to work with the original text and see directly the result after compiling (what you see is what you get).

Then, users could copy the original text and paste directly in the .rst files we described previously. You can find the tool at the following address:

https://livesphinx.herokuapp.com/
7. We access now the folder interface

The file codename.txt must include the name of the security primitive implementation. It is used for the naming of the solution in the documentation although it is not mandatory for the creation of the package. This means if it is not provided the generation of the documentation would success but it could have some problems for labelling some elements of the documentation (e.g. titles, references, etc.).

The file categories.json contains the description of the interfaces of the security primitive implementation. This element helps defining the interfaces from the point of view of the calls with their arguments and output. Below we show the structure of this element:
8. Include application data in the folder Application

This folder supports data in any form so it can be included as .zip, scripts, raw data, source files, etc. Following, we show an example of the security primitive implementation EPICA with the data provided as zip files.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA API-v1.06.zip</td>
<td>04/09/2017</td>
<td>Compressed (zip)</td>
<td>21.888 KB</td>
</tr>
<tr>
<td>PEP-v0.2.zip</td>
<td>04/09/2017</td>
<td>Compressed (zip)</td>
<td>5.605 KB</td>
</tr>
<tr>
<td>Token Manager.zip</td>
<td>04/09/2017</td>
<td>Compressed (zip)</td>
<td>24.989 KB</td>
</tr>
</tbody>
</table>

9. Regarding the Deployment folder, it needs to contain information for how to integrate the application in the target environment.
10. **The folder Testing provides optional testing information of the security primitive implementation.**

- As with the deployment, the testing data can be provided in three different ways: automatic, semi-automatic and manual. Automatic implies to provide in the file `run_tests` scripts for automatic testing (which can be supported with additional files in this folder such as APIs); semi-automatic is similar to automatic but requiring some input from the user (such as working directory of a file, url, creation of an account in the database, etc.); manual means to provide information of how to perform the testing by means of documentation, APIs, code, etc.
- Although the information can be provided in these three different forms the file `run_tests` must always exist (provided in the template).

Following we present an example of the folder `Testing` of the security primitive implementation of EPICA:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiTenancyTester.zip</td>
<td>04/09/2017 14:41</td>
<td>Compressed (zip)...</td>
</tr>
<tr>
<td>PeApiTester.zip</td>
<td>04/09/2017 14:40</td>
<td>Compressed (zip)...</td>
</tr>
<tr>
<td>run_tests</td>
<td>07/06/2017 18:39</td>
<td>File</td>
</tr>
<tr>
<td>TokenManagerTester.zip</td>
<td>04/09/2017 14:40</td>
<td>Compressed (zip)...</td>
</tr>
</tbody>
</table>

11. **After all files have been completed we create a .zip file of the folders shown in the figure of Step 2.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>05/09/2017 9:22</td>
<td>File folder</td>
</tr>
<tr>
<td>Deployment</td>
<td>27/09/2017 15:05</td>
<td>File folder</td>
</tr>
<tr>
<td>Documentation</td>
<td>27/09/2017 15:05</td>
<td>File folder</td>
</tr>
<tr>
<td>Testing</td>
<td>27/09/2017 15:05</td>
<td>File folder</td>
</tr>
<tr>
<td>Template - Security Primitive Implementation.zip</td>
<td>27/09/2017 15:07</td>
<td>Compressed (zip)...</td>
</tr>
</tbody>
</table>

12. **We now access the TREDISEC Framework and click in the option Security Primitive Implementation.**
13. We click in “Create Security Primitive Implementation”

14. We complete now the information of the security primitive implementation
15. We search using the browser for the .zip we just created

16. We click in Submit

17. We select the access level for the security primitive implementation
18. After selecting our preferences we click submit and the security primitive implementation will be correctly created in the TREDISEC Framework.
Annex V: Creation process of a TREDISEC Recipe

19. Download the template of the TREDISEC Recipe
You can obtain the template from the TREDISEC Framework in the Section TREDISEC Resources

20. Extract the .zip file. You will then have four different folders

21. Modify the mandatory files and add information of the overview, architecture, requirements, and nodes and references. For a TREDISEC Recipe it is mandatory to complete all folders.

22. Access to Documentation (folder) -> source (folder). The first set of documents to complete are: author.txt, doc_version.txt, title.txt, version.txt

23. Now we have to complete the information in the folders files and interface

24. We access first the folder files
The description of each file can be found in the section “Security Primitive Implementation package structure”.

The output of these files, after being processed by the tool of the TREDISEC Framework, is as follows:

<table>
<thead>
<tr>
<th>File</th>
<th>Section in the TREDISEC Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>overview.rst</td>
<td>Overview tab in the TREDISEC Recipe. It is also shown in the documentation of the TREDISEC Recipe.</td>
</tr>
<tr>
<td>requirements.rst</td>
<td>Requirements tab in the TREDISEC Recipe</td>
</tr>
<tr>
<td>architecture.rst</td>
<td>Overview section in the documentation of the TREDISEC Recipe. It is not shown in the tab of Overview.</td>
</tr>
<tr>
<td>notes_and_references.rst</td>
<td>Notes and references section in the documentation of the TREDISEC Recipe.</td>
</tr>
</tbody>
</table>

The file architecture.rst must be referenced in the file overview.rst in order to be parsed and included in the documentation of the TREDISEC Framework. In order to do so please add at the end of the file overview.rst the following line:

```
include:: architecture.rst
```

In order to modify/complete these files you can use the tool Online Sphinx editor. This editor provides tools for creating different structures in the text (e.g. sections, sub-sections, bullet points, etc.), insert figures, etc. Also, being online makes easy to work with the original text and see directly the result after compiling (what you see is what you get).

Then, users could copy the original text and paste directly in the .rst files we described previously. You can find the tool at the following address:

https://livesphinx.herokuapp.com/
25. We access now the folder interface

The file codename.txt must include the name of the TREDISEC Recipe. It is used for the naming of the solution in the documentation although it is not mandatory for the creation of the package. This means if it is not provided the generation of the documentation would success but it could have some problems for labelling some elements of the documentation (e.g. titles, references, etc.)

The file categories.json contains the description of the interfaces of the TREDISEC Recipe. This element helps defining the interfaces from the point of view of the calls with their arguments and output. Below we show the structure of this element:
An example of it is shown below.

It must be edited using any editor (e.g. notepad+). Note it requires a minimum level of knowledge in order to define the interfaces in this format but should be achievable by the security technology expert.

26. Include application data in the folder Application

This folder supports data in any form so it can be included as .zip, scripts, raw data, source files, etc. Following, we show an example of the TREDISEC Recipe EPICA with the data provided as zip files.

27. Regarding the Deployment folder, it needs to contain information for how to integrate the application in the target environment.
- Depending on the type of deployment (automatic, semi-automatic or manual) it should provide the information as a script (automatic or semi-automatic), using for installation the files in this folder or the Application folder, or documentation (manual) about how to integrate the solution.
- The file deploy is mandatory in this folder (provided in the template). It doesn’t matter if it includes code or is empty.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>deploy</td>
<td>29/09/2017</td>
<td>File</td>
</tr>
<tr>
<td>docker-compose.</td>
<td>14/09/2017</td>
<td>YML File</td>
</tr>
<tr>
<td>yml</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example of deploy file

```
#!/bin/bash
BASEDIR=$(dirname "$0")
L echo "$BASEDIR"
docker-compose -f "$BASEDIR/docke-compose.yml" build
docker-compose -f "$BASEDIR/docker-compose.yml" up
```

28. The folder Testing provides optional testing information of the security primitive implementation.

- As with the deployment, the testing data can be provided in three different ways: automatic, semi-automatic and manual. Automatic implies to provide in the file run_tests scripts for automatic testing (which can be supported with additional files in this folder such as APIs); semi-automatic is similar to automatic but requiring some input from the user (such as working directory of a file, url, creation of an account in the database, etc.); manual means to provide information of how to perform the testing by means of documentation, APIs, code, etc.
- Although the information can be provided in these three different forms the file run_tests must always exist (provided in the template).

Following we present an example of the folder Testing of the TREDISEC Recipe of EPICA

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiTenancyTest</td>
<td>18/09/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>API</td>
<td>18/09/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>TokenManagerTest</td>
<td>18/09/2017</td>
<td>File folder</td>
</tr>
<tr>
<td>docker-compose</td>
<td>14/09/2017</td>
<td>SH File</td>
</tr>
<tr>
<td>dockerfile</td>
<td>14/09/2017</td>
<td></td>
</tr>
<tr>
<td>run_tests</td>
<td>14/09/2017</td>
<td></td>
</tr>
</tbody>
</table>

An example of the file run_tests is as follows:

```
#!/bin/bash
docker rm -f epica_test_suite
docker rmi -f epica-test
docker build -t epica_test_suite .
docker run --network=deployment_epica-nw -itd --name=epica-test epica_test_suite
docker logs -f epica-test
```

93
29. After all files have been completed we create a .zip file of the folders shown in the figure of Step 2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>28/09/2017 14:06</td>
<td>File folder</td>
</tr>
<tr>
<td>Deployment</td>
<td>29/09/2017 12:51</td>
<td>File folder</td>
</tr>
<tr>
<td>Documentation</td>
<td>14/09/2017 14:22</td>
<td>File folder</td>
</tr>
<tr>
<td>Testing</td>
<td>28/09/2017 14:07</td>
<td>File folder</td>
</tr>
<tr>
<td>epica_TREDISEC_recipe.zip</td>
<td>29/09/2017 12:52</td>
<td>Compressed zip...</td>
</tr>
</tbody>
</table>

30. We now access the TREDISEC Framework and click in the option TREDISEC Recipes

31. We click in “Create TREDISEC Recipe”
32. We complete now the information of the TREDISEC Recipe

33. We search using the browser for the .zip we just created
34. We click in Submit

35. We select the access level for the TREDISEC Recipe
36. After selecting our preferences we click submit and the TREDISEC Recipe will be correctly created in the TREDISEC Framework

Test_template - TREDISEC Recipe

Overview

EPICA (Efficient and Privacy-respectful Interoperable Cloud-based Authorization) is a software component that implements the primitive pattern Multi-tenancy Attribute-based Access Control Primitive. The primitive pattern addresses an Attribute-Based Access Control (ABAC) model that satisfies a set of requirements that are crucial in order to maintain the cost reductions and high performance that cloud infrastructures provide.

EPICA builds upon version 3 of the XACML standard and advances the WS02 Banaa Open Source implementation by extending it with new functionalities, improving existing ones and addressing the most important challenges that multi-tenancy poses to traditional authorization schemes (i.e. trust management, common vocabulary, tenants policy isolation and shared resources), while remaining fully compatible with most popular storage efficiency implementation techniques.

Besides, EPICA supports high availability and performance deployments, implementing an efficient policy retrieval approach with scalable policy stores, which has been validated with the requirements of a real use case scenario that have a load balancing scheme in place. The architecture of EPICA has been designed taking into account interoperability and privacy concerns, so the information exchanged between the cloud provider and the user, required to perform authorization, remains minimal.