Cloud Orchestration at the Level of Application

Project Acronym: **COLA**

Project Number: **731574**

Programme: **Information and Communication Technologies Advanced Computing and Cloud Computing**

Topic: **ICT-06-2016 Cloud Computing**

Call Identifier: **H2020-ICT-2016-1**
Funding Scheme: **Innovation Action**

Start date of project: 01/01/2017 Duration: 30 months

Deliverable:

D7.2 MiCADO security architecture specification

Due date of deliverable: 31/10/2017 Actual submission date: 31/10/2017

WPL: SICS

Dissemination Level: PU

Version: 2.0

Status and Change History

Table 1 Status Change History

Status:	Name:	Date:	Signature:
Draft:	N. Paladi and A. Michalas	07/10/2017	N. Paladi
Reviewed:	Jozsef Kovacs	26/10/2017	J. Kovacs
Approved:	Tamas Kiss	31/10/2017	T. Kiss

Table 2 Document Change History

Version	Date	Pages	Author	Modification
V0.1.1	16/06	5	N. Paladi	Document template
V0.1.2	04/07	8	A. Michalas	Notes on related work
V0.1.3	07/08	10	N. Paladi	First version with outline and introduction
V.01.4	25/08	13	N. Paladi	Defined objectives of the document
V0.1.5	07/09	14	N. Paladi	Add security landscape
V0.2	08/09	15	A. Michalas	Add an introduction for the Security Architecture
V0.2	09/09	16	A. Michalas	Add a description for the Crypto Engine component
V0.3	11/09	18	A. Michalas	Enhanced the Crypto Engine component by describing all the main functions supported by the component
V0.4	14/09	19	A. Michalas	Described the Credential Manager Component
V0.5	16/09	19	A. Michalas	Described the Image Verifier component
V0.6	22/09	20	N. Paladi	Add threat model and regulatory aspects
V0.7	22/09	21	N. Paladi	Add cloud orchestration security objectives
V0.7	23/09	28	A. Michalas	Completed the Security Architecture section
V0.8	25/09	31	A. Michalas	Created and added images for the Crypto Engine, Credential Manager and Image Verifier components
V0.8	25/09	32	A. Michalas	Add Summary and Conclusions Section
V0.9	26/09	32	N. Paladi	Review introduction to better match focus
V0.91	03/10	32	N. Paladi	Add section 5
V1.0	04/10	32	N. Paladi	Review complete document
V1.1	18/10	33	N. Paladi	Add section regarding architecture implementation

V1.2	23/10	38	Balasys	Update section 5
V1.29	26/10	39	N. Paladi	Address reviewer comments
V 1.30	27/10	39	Balasys	Address reviewer comments
V1.31	30/10	41	N. Paladi	Update description of Figure 4 and text related to Zorp, address other reviewer comments
V2.0	31/10	39	N. Paladi	Update formatting, ready for submission.
V2.0	31/10	all	T.Kiss	Final check, minor corrections

Acronyms

Table 3 List of acronyms

API	Application Programming Interface
AWS	Amazon Web Services
COLA	Cloud Orchestration at the Level of Application
UML	Unified Modelling Language
MiCADO	Microservice-based Cloud Application-level Dynamic Orchestrator
CM	Credential Manager
PM	Policy Manager
TTP	Trusted Third Party
SPEL	Security Policy Enforcement Layer
TPM	Trusted Platform Module
CSP	Cloud Service Provider
ADV	Malicious Adversary
MAC	Message Authentication Code
PRF	Pseudorandom Function
DoS	Denial of Service Attack
IaaS	Infrastructure-as-a-Service
PII	Personally Identifiable Information
TPM	Trusted Platform Module
SGX	Software Guard Extensions
IaaS	Infrastructure-as-a-Service
CM	Credential Manager

ABE	Attribute-Based Encryption
KP-ABE	Key-Policy Attribute-Based Encryption
PEL	Policy Enforcement Layer
TCP	Transport Control Protocol
IP	Internet Protocol
CS	Credential Store
CSP	Cloud Service Provider
CSC	Central Security Component

Definitions

Attestation protocol: a cryptographic protocol involving a target, an attester, an appraiser, and possibly other principals serving as trust proxies. The purpose of an attestation protocol is to supply evidence that will be considered authoritative by the appraiser, while respecting privacy goals of the target (or its owner).

Control plane: router architecture hardware and software components for handling packets destined to the device itself as well as building and sending packets originated locally on the device.

Forwarding plane: router architecture hardware and software components responsible for receiving a packet on an incoming interface, performing a lookup to identify the packet's IP next hop and determine the best outgoing interface towards the destination, and forwarding the packet out through the appropriate outgoing interface.

Integrity measurement records (also *integrity measurements*): a list of hashes recording some sequence of events, such as e.g. installation of sets of binaries or opening files for reading or writing. The list is expanded on each event and cannot be forged assuming that the underlying cryptographic primitives are secure.

List of Figures and Tables

Table of Contents

1 Introduction

Throughout the two decades since the cloud-computing paradigm was introduced, cloud infrastructure deployments have grown in both absolute numbers and in their complexity.
The increase in complexity was primarily induced by the growth in the number of inter-communicating components, variety of supported application programming interfaces (APIs), and additional mechanisms that enable flexible scalability and computation efficiency.

As a result of the increase in the complexity of cloud deployments, *cloud orchestration* has become increasingly important at all stages of the cloud infrastructure lifecycle. The role of cloud administrators has gradually shifted from manually deploying, configuring and monitoring the compute, storage and networking infrastructure to configuring the orchestrator systems. This includes writing either configuration policies that describe *what* the orchestrator should do to realize the cloud infrastructure, or expressing high-level imperative *intents* describing *how* the infrastructure should look like and leaving the implementation details to the orchestrator.
Beyond the deployment stage, modern orchestrator systems also include monitoring, load balancing and continuous availability functionality throughout the lifecycle of the cloud infrastructure components. Such automated orchestrator functionality allows maintaining a stable, continuous, and highly available set of cloud services with minimal or no human interference – based solely on the defined configuration policies or formulated intents.

However, while cloud orchestration systems have the potential to act as a leverage of operator capabilities, *misconfigured* or *maliciously modified* orchestrators can likewise act as a leverage of adversary capabilities. For example, adversaries can exploit orchestrator vulnerabilities in order to intercept or control the generation credentials provisioned to the virtual machine instances; insert backdoors in virtual machine images to automate the deployment of maliciously modified virtual machine instances; or take control over the placement policy of virtual resources, either to benefit a chosen service provider (in case of federated deployments) or deploy virtual components on hardware resources with compromised physical security.

To address such risks, the COLA project aims to develop a secure and open cloud orchestration platform. The goal is to allow efficient and secure deployment of arbitrary applications as well as advanced security policy management and enforcement at the orchestration level. To achieved the proposed objectives, the COLA project has collected a set of use cases and defined the security requirements towards a cloud orchestration platform, based on the *Microservices-based Cloud Application-level Dynamic Orchestrator* (MiCADO) framework, which stands at the core of the COLA project.

Based on earlier work within the project, this document describes a *security architecture* for a secure and open cloud orchestration platform. We expect that this security architecture will be applied to implement the security features of the MiCADO framework.

1.1 Scope

This document describes the MiCADO security architecture specification, the guiding document for the design of the structure, functionality and interactions of the security components in MiCADO.

The purpose of the document is to specify the overall security design and functions supported in COLA. The purpose of the document *does not* include detailed specifications of the security components. Such detailed specifications will be done in deliverable D7.3 and will be based on the security architecture specifications described in the current document. At the end of the project, the final COLA security design choices will be evaluated against the requirements described in D7.2 and the security architecture specifications described this document.

1.2 Objectives

The primary objectives of this document are to:

- Define and describe the security architecture rationale;
- Describe the principles behind the design of the MiCADO security components;
- Describe the architecture and interactions among the MiCADO security components;
- List the state-of-the-art security techniques to be used in the MiCADO framework.

1.3 Relation with other WPs and Deliverables

The security architecture of the MiCADO framework is central for creating a secure and reliable orchestration platform and is developed with consideration to the overall MiCADO architecture. This deliverable must be read in conjunction with D5.1 and D6.2:

1. D5.1 - *"Analysis of existing application description approaches"* - touches upon the management of resource scaling security policies.
2. D6.1 – "*Prototype and documentation of the cloud deployment orchestrator service"* – contains the technical and user documentation of the MiCADO platform.

The COLA Security Architecture will be used as input for D7.3 "*MiCADO application security classification specification"*, as well as subsequent deliverables in WP7.

1.4 Organization

This document begins with an introduction describing the current landscape for cloud deployments and motivates the need for advanced security features to be provided by COLA. Chapter 2 describes the principles and security considerations for the COLA security architecture. Chapter 3 describes the state of the art in the area of cloud orchestration security. Chapter 4 introduces the COLA security architecture and describes on a high level the security components of COLA. Chapter 5 introduces the approach to the implementation of the security architecture. Chapter 6 reviews the relation of the security requirements collected in D7.1 [11] with the security architecture presented in this document. Finally, Chapter 7 concludes the document.

2 Security Landscape of Cloud Orchestration

In complex systems, such as clouds and software-defined network environments, services are deployed dynamically using *orchestrators* - logically centralized entities that manage and coordinate the lifecycles of the components that make up the service. For services within one administrative domain, one orchestrator may be responsible for the end-to-end service setup.

2.1 Emergence of dynamic cloud applications

The importance of orchestration in cloud computing has been steadily increasing as monolithic *legacy applications* ported to cloud platforms have been replaced by microservice-based *dynamic applications*. Such legacy applications use fixed layering and inter-layer invocation through well-defined layer-specific interfaces. Likewise, they are tied to infrastructure resources, such that the same entity owns service features, functionality and physical resources. In contrast, dynamic applications are built from micro-services providing a small number of primitive features. They use recursion rather than layering, such that functionality is accessed using generic service interfaces that are the same at all layers of the stack. Furthermore, features are decoupled from resources – owners of the features and functionality may be different than the owners of the physical infrastructure.
An orchestrator can decide at each step in the decomposition process whether to manage a component itself, or hand off responsibility for that component to a different orchestrator in a different domain, that may in turn use recursive decomposition to deploy that component on its own resources.

2.2 Orchestration Security

From a **security** point of view, the shift from legacy applications to dynamic applications (both deployed in the cloud) requires novel approaches to aspects that can affect system security. Such aspects include *key provisioning*, *dynamic integrity verification of hosts* and system components, as well as *protecting the integrity* (and potentially the confidentiality) *of system configuration data*. Furthermore, cloud tenants that own and operate the dynamic cloud applications require verifiable guarantees that systems have been orchestrated according to the specified templates or intents.

A survey by Weerasiri et al. lists security as one of the cross-cutting concerns in cloud orchestration implementations, along with service level agreements and negotiations, portability, interoperability, standardization, resource demand profiling, resource pricing, profit maximizing and other runtime issues [3]. Weerasiri et al. found that such cross-cutting concerns are addressed both by research initiatives, and to a larger extent by enterprise-ready orchestration techniques, such as AWS OpsWorks[1], AWS CloudFormation[2], VMWare vSphere[3], Heroku[4], Puppet[5], Juju[6], Docker[7] and CFEngine[8]. The authors attribute this to the fact that the utilization of orchestration techniques requires effectively addressing security concerns in orchestrated cloud environments – this provides a suitable opportunity to build up on the existing techniques with a more nuanced approach.

[1] OpsWorks product page: https://aws.amazon.com/opsworks/
[2] Cloudformation product page: https://aws.amazon.com/cloudformation/
[3] vSphere product page: https://www.vmware.com/ca/products/vsphere.html
[4] Heroku product page: https://www.heroku.com/
[5] Puppet product page: https://puppet.com/
[6] Juju product page: https://www.ubuntu.com/cloud/juju
[7] Docker product page: https://www.docker.com/
[8] CFEngine product page: https://cfengine.com/

The current COLA security architecture specification builds on the security principles shared with the above enterprise-ready orchestration techniques. However, it is also extendable to make use of the recent evolution in the field of micro-services, trusted computing and lightweight virtualization. In particular, with regard to lightweight virtualization, the emerging Unikernels [4] may present a more secure alternative to container-based (hypervisor-free) approaches, as application developers have explicit control over core security areas.

Finally, logical centralization of orchestrator systems is another important aspect at the crossroads between functional and security architectures: decentralized orchestration needs careful consideration with regard to discovery, synchronization, coordination and security aspects of cloud application agents.

2.3 Threat Model

The threat model of COLA aims to describe the threats towards a cloud orchestration system and aims to capture both the *external* and the *internal* threats towards an orchestrator.
The goal of this description is to present a *generic* overview of the security landscape for cloud orchestration, without focusing on any specific attacks. The threat model will serve as input for the *COLA security architecture specification* described in this document and later on for the *COLA security design specification* to be delivered in D7.3.

2.3.1 Threat Model Assumptions

We base the threat model of the COLA orchestration platform on the threat model earlier defined in [6]. The remote *Adversary* can intercept, drop, inject or otherwise interfere with all network communication. We start by defining the *assumptions* on which we base the threat model for the COLA orchestration security module.

- **Hardware Integrity:** Media revelations have raised the issue of hardware tampering en route to deployment sites. We assume that the cloud provider has taken necessary technical and non-technical measures to prevent such hardware tampering.
- **Physical Security**: We assume physical security of the data centres where the Infrastructure-as-a-Service (IaaS) resources are deployed. This assumption holds both when the IaaS provider owns and manages the data center and when the provider utilizes third party capacity, since physical security can be observed, enforced and verified through known best practices by audit organizations. This assumption is important to build higher-level hardware and software security guarantees for the components of the IaaS.
- **Low-Level Software Stack**: we assume that at installation time, IaaS providers reliably record integrity measurements of the low-level software stack: the Core Root of Trust for measurement; BIOS and host extensions; host platform configuration; Option ROM code, configuration and data; Initial Platform Loader code and configuration; state transitions and wake events, and a minimal hypervisor. We assume the record is kept on protected storage with read-only access and the adversary cannot tamper with it.
- **Network Infrastructure**: IaaS providers have physical and administrative control of the network. The *Adversary* is in full control of the network configuration, can overhear, create, replay and destroy all messages communicated between the *Tenant* and their resources (VMs, virtual routers, storage abstraction components) and may attempt to gain access to other domains or learn confidential information.

- **Cryptographic Security**: we assume encryption schemes are semantically secure and the *Adversary* cannot obtain the plain text of encrypted messages. We also assume the signature scheme is unforgeable, i.e. the *Adversary* cannot forge the signature of the *Tenant* and that the MAC algorithm correctly verifies message integrity and authenticity. We assume that the *Adversary*, with a high probability, cannot predict the output of a pseudorandom function.
- **Availability:** we explicitly exclude denial-of-service (DoS) attacks that aim to disrupt service availability of the infrastructure deployed by the orchestrator platform. This is because denial-of-service can be caused by a wide variety of approaches and is especially difficult to prevent in distributed environments that rely on components in different administrative and trust domains.

2.3.2 Risks

The *Adversary* can use information intercepted from network communication to:

- **Impersonate the *Tenant***: deploy arbitrary components; start, stop, migrate and otherwise interact with deployed components; and obtain remote access to the infrastructure operated by the *Tenant*.
- **Impersonate the *Cloud Service Provider*** in the communication with the Tenant;

2.3.3 High level attacks

This subsection describes a set of high-level attacks that the *Adversary* can launch by exploiting vulnerabilities in the cloud orchestration platform. The following set of high-level attacks is based both on earlier research [6] as well as several reviews of the state of the start in cloud security [3, 12].

2.3.3.1 VM Substitution Attack

In a VM substitution attack, the *Adversary* induces the orchestrator to launch a maliciously modified VM instance that contains hidden vulnerabilities, instead of a virtual machine instantiated from the VM image selected by the *Tenant*. In case of a successful attack, the *Adversary* can extract sensitive information from the virtual machine instance or monitor the activity of the Tenant on the virtual machine instance.

2.3.3.2 Host Substitution Attack

In a host substitution attack, the *Adversary* induces the orchestrator to ignore placement policies with regard to host selection in order to instantiate the virtual machine on a compromised or otherwise vulnerable virtualization host. In case of a successful attack, the *Adversary* can extract sensitive information from the virtual machine instance or monitor the activity of the Tenant on the virtual machine instance.

2.3.3.3 Storage Host Substitution Attack

In a storage host substitution Attack, the *Adversary* induces the orchestrator to ignore placement policies with regard to storage selection in order to attach data storage with exploitable vulnerabilities. In case of a successful attack, the *Adversary* can extract sensitive information from the stored data.

2.3.3.4 Resource Parasite Attack

In a resource parasite attack, the *Adversary* induces the orchestrator to modify the configuration of the infrastructure requested by the *Tenant*. In case of a successful attack, the

Adversary can execute hidden parasite processes (such as e.g. crypto currency mining) on the *Tenant* infrastructure.

2.3.3.5 Placement Bias Attack

In a placement bias attack, the *Adversary* induces the orchestrator to ignore placement policies in federated cloud deployments and to favour a certain deployment target. In case of a successful attack, the *Adversary* can increase the utilization – and implicitly the profit – of a chosen infrastructure service provider.

2.4 Regulatory Aspects

A growing number of audit, certification and regulatory frameworks address cloud computing. The COLA security architecture specification *does not* aim to enforce compliance to any specific framework. However, it *does* aim to be generic and extensible to be used by an orchestration platform compliant to such frameworks. Several cloud certification and compliance frameworks are described below, based on [13, 14]. See [13] for a more extensive description.

2.4.1 ISO/IEC 27017

Cloud Security clarifies the division of responsibilities for protecting data in the cloud environment between the cloud service provider and cloud users; it describes controls regarding sharing information security roles, management of customer assets in case of service termination, isolation of virtual computing and network environments, monitoring, etc.

2.4.2 ISO/IEC 27018

Cloud Privacy contains controls for protection of Personally Identifiable Information (PII) in cloud environments, aimed towards customer control over PII, transparency with regard to data collection, PII transfer to third parties and data breach disclosure procedures.

2.4.3 COBIT 5

COBIT, originally ‘Control Objectives for Information and related Technology’ but used in acronym only since 2009, was first released in 1996 by ISACA. The current version, COBIT 5, was published in 2012. COBIT 5 helps enterprises create optimal value from information and related technology (IT) for their stakeholders by maintaining a balance between realizing benefits and optimizing risk levels and resource use. The framework addresses both business and IT functional areas across an enterprise and considers the IT-related interests of internal and external stakeholders. Enterprises of all sizes, whether commercial, not-for- profit or in the public sector, can benefit from COBIT 5.

2.4.4 Federal Information Security Management Act (FISMA)

FISMA was passed as Title III of the E-Government Act (Public Law 107-347) in the United States in 2002 and sets high-level security requirements. FISMA requires each US federal agency to develop, document, and implement an agency-wide program to provide information security for the information systems that support the operations and assets of the agency, including those provided or managed by another agency, contractor, or other source. Federal Information Processing Standards (FIPS) are developed by NIST and approved by the US Secretary of Commerce. FIPS standards are also used by other organizations around the world, as a best practice for security requirements, especially in regulated industry sectors (such as financial and health-care institutions) that process Sensitive But Unclassified (SBU) information. FIPS does not apply to national security systems (as defined in Title III, Information Security, of FISMA).

Federal agencies must adhere to FIPS 200, which "specifies minimum security requirements for federal information and information systems and a risk-based process for selecting the security controls necessary to satisfy the minimum requirements." In essence, the high-level security requirements mandated by FISMA are further specialized through FIPS and other more detailed standards, such as NIST SCAP. The interested reader can find a description of the FIPS framework in the appendix.

2.4.5 Health Insurance Portability and Accountability Act (HIPAA)

The USA's Health Insurance Portability and Accountability Act of 1996 (HIPAA), Public Law 104-191, mandates the US Dept. of Health and Human Services (HHS) to adopt and enforce national standards for electronic health care transactions, health identifiers, security and privacy (Part 164).

The HIPAA evaluation standard (§ 164.308(a)(8)) also requires covered entities to perform a periodic (technical and non-technical) evaluation to assess if security policies and procedures meet the security requirements. This evaluation can be performed internally or by a third-party auditor.

3 MiCADO Security Architecture Principles and Considerations

Orchestration heavily relies on automation tools and “rules”. While orchestration and automation platforms have the potential to streamline cloud operations, they can introduce security risks if misused. Such security risks include malicious commands, service disruption and file/system/app modification and are caused by both gaps in the security features of the orchestration tools and by the insufficient risk awareness of the orchestrator administrators. However, if managed well, orchestrator systems can improve security.

3.1 Cloud Orchestration Security Problem Definition

Here we aim to capture the core essence of *cloud orchestration security* as problem statement, summarizing the security landscape and its threat situation as described above. This leads us to the following **problem statement**:

Create a highly scalable, flexible and efficient security infrastructure and security protocol design that fulfils the security needs of a cloud orchestration platform. Such needs include the traditional cloud service stakeholders such as cloud service providers and operators, tenants, regulators, as well as the needs of emerging new stakeholder such as cloud brokers and telecom-cloud infrastructure providers. Allow this multitude of cloud orchestration user categories to securely share federated cloud platforms.

In the next section, we break down this problem definition into finer-grained *security objectives*.

3.2 Cloud Orchestration Security Objectives

The derived cloud orchestration security objectives fall into the following categories:

3.2.1 General

O1.1 Where possible, cloud orchestration security should be decoupled from specific physical deployments, focusing on defence-in-depth, in particular self-protection of assets, limiting dependency on protection at network, site, or node perimeter**.**

O1.2 In a logical or physical part of the cloud deployment, governed by a specific security policy, further fine-grained security policy enforcement should be possible based on mechanism such as e.g. virtualization and domain segmentation.

3.2.2 Security-relation to Legacy Systems

O2.1 Cloud orchestration must provide a security level higher or at least equal to the security and privacy level of individual system components.

O2.2 Cloud orchestration security should not be negatively affected by the security of legacy systems with which it interworks.

O2.3 Cloud orchestration must enable seamless interworking of different cloud resources without exposing the security level of each of these resources to new threats.

3.2.3 Secure Virtualization

O3.1 Cloud orchestration must enable a secure, reliable, and traceable sharing of cloud resources (i.e., compute, storage and network) between the various tenants having vastly different requirements.

O3.2 Cloud orchestration platforms must be dynamically scalable in order to easily and securely enable the changes required to ensure any new use cases, new trust models and new service delivery approaches.

O3.3 Cloud orchestration platforms should support necessary root-of-trust functionality.

O3.4 Cloud orchestration platforms must support strong isolation of (virtual) resources allocated to different tenants.

O3.5 Slices must support configurable security and be able to provide tenant-unique security services as required by specific services and applications and specified by tenant policies.

3.2.4 Security Management

O4.1 Cloud orchestration platforms must support security monitoring capable of detecting advanced cyber security threats and support coordinated monitoring between different cloud service providers.

O4.2 Cloud orchestration platforms must support strong mutual authentication and authorization.

O4.3 Cloud orchestration platforms must provide functionality to mutually assess the trustworthiness before, and during interactions.

O4.4 Cloud orchestration platforms' interactions must be auditable and produce evidence of liabilities.

3.2.5 New Business and Use-cases

O5.1 Cloud orchestration platforms must be able to deliver and maintain SLA to tenants in terms of: availability, security, latency, bandwidth, and access control from an end-to-end perspective.

O5.2 Cloud orchestration platforms systems must allow secure federation of cloud resources, e.g. resources provided by multiple tenants or external cloud service providers.

3.2.6 Regulatory Aspects

O6.1 Cloud orchestration platforms must be extensible to comply with regulatory aspects.

O6.2 If required by regulation, the cloud orchestrator operator must have means to demonstrate their provided level of security.

Note that the COLA security architecture specification ***does not*** aim to address all of the above outlined security objectives for cloud orchestration systems. Rather, only the security architecture will only address the objectives that are aligned with the security requirements described in D7.1 [11]. See Section 6 for a mapping between the security requirements, security objectives and the architectural domains.

3.3 MiCADO architectural principles

The cloud orchestrator system developed in the COLA project builds on the existing knowledge and best practices. The architectural principles presented below are based on the security requirements outlined in D7.1 [11] and aim to lay the foundation of a secure, reliable and scalable cloud orchestration platform. The security architecture of MiCADO is based on the following principles:

- Authenticated access to cloud APIs;
- Verification of computing resource properties (cloud platform attestation);
- Verification and enforcement of data encryption policies and properties;
- Verification of access control properties.

We next describe each of the principles in more detail.

Authenticated access to cloud APIs aims to exclusively allow only authorized access to the cloud resources, as well as to information about the utilization of cloud resources by a tenant (so-called *metadata*). This principle is in line with the current cloud security best practices [5]. The MiCADO security architecture adopts a modular approach to support both current and upcoming authentication mechanisms.

Verification of computing resource properties (cloud platform attestation) aims to ensure that the software and configuration that is part of the trusted computing base of the cloud host providing a specific cloud resource matches the software and configuration expected by the cloud tenant. This approach is based on earlier work such as [7], where discrete security hardware was used to provide cloud user security guarantees and [8], where hardware-enabled execution isolation features were used to protect cloud network infrastructure.
The MiCADO security architecture builds on the rich experience of using hardware-based integrity verification and execution isolation mechanism in order to provide security guarantees for the cloud tenants.

Verification and enforcement of data encryption policies and properties aims to ensure on the application level that production data, configuration data as well as metadata are never stored in plain text. Rather, such data are stored encrypted, are processed in isolated secure execution environments or even processed and queried encrypted when appropriate. The MiCADO security architecture builds on earlier experience in cloud data protection [7], [8] to ensure the confidentiality and integrity of data used by and produced by the cloud orchestration software.

Verification of access control policies and properties aims to ensure that orchestration-related data is only accessible by authorized users, according to access control policies defined by the cloud tenant. This aspect of the MiCADO security architecture also aims to ensure the compartmentalization and strong isolation of data. This includes tenant authentication data, configuration data and policies defining the operation of the cloud orchestration software, as well as data produced by the cloud orchestration software (such as logs and other metadata). The MiCADO security architecture builds upon earlier work on data access control in cloud infrastructure [9] and data isolation in cloud network infrastructure [10].

4 COLA security architecture components

In this section, we describe the main security-related abstract components[9] that will be incorporated into the MiCADO architecture and will be deployed as micro-services.
While we share the components described in D7.1 [11], we further extend this description by describing in detail the functionality of each abstract component and by introducing new abstract components that play a key role in the overall security of MiCADO. Finally, we describe the relations between the security components and we provide several details regarding the protocols that each component will be responsible to run.
This description is of paramount importance for the project as it will be used as a guide during the development of the security-related components. Before describing each of the security components, we need to provide some initial assumptions that are vital for the proper function of the components. At this point we note that the assumptions are based on standard procedures described in the literature and are considered realistic.

AS1. Each security component generates a unique public/private key pair of sufficient length. The private key (sk) will remain private while the public key will be shared with the rest of the components.

AS2. The communication channel between all the security components is considered as secure. More precisely, we assume that all the communication is taking place over a TLS-enabled channel.

AS3. While we can assume that a component can be corrupted, none of the components will be able to impersonate another, legitimate component. This is a valid assumption since we have assumed cryptographic security. In other words, by knowing the public key of an entity it is impossible to extract any valuable information about the corresponding key. Hence, an impersonation attack will be impossible.

AS4. All the messages exchanged between two or more parties are using proper mechanisms to ensure the freshness of the messages. More precisely, we assume that the corresponding function from the Crypto Engine component for generating unique tokens will be always used before sending a new message. Thus, a malicious adversary will be unable to successfully launch a replay attack.

4.1 Crypto Engine

Crypto Engine, an abstract component, is one of the key-components for the security of MiCADO and is implemented as a collection of cryptographic algorithms. More precisely, this component is responsible for generating certain cryptographic keys used by several entities to securely interact with other entities or components within the MiCADO framework. Furthermore, Crypto Engine is responsible for issuing new credentials based on a typical challenge-response protocol between the requestor and the issuer. Crypto Engine will contain an explicit list of cryptographic functions that will be available to users who wish to perform certain cryptographic operations (such as encryption, decryption, hashing etc). Crypto Engine will be available as a separate microservice and will be considered as one of the standard microservices offered by COLA. The Crypto Engine will support the following list of functions (also see Figure 1).

[9] *Abstract components* do not refer to a specific implementation; rather, they are a collection of algorithms or protocols that fulfill a certain specific feature set and can be implemented by third-party vendors.

1. **Key Generation Orchestration:** This function will be responsible for generating several cryptographic keys (both symmetric and asymmetric). The key generation Orchestration is a probabilistic algorithm that takes as input a security parameter λ, which will define the length of the key, and an identifier for the type of key that needs to be generated (i.e. sym for symmetric and $asym$ for asymmetric) and outputs either a symmetric secret key K or a public/private key pair pk/sk.

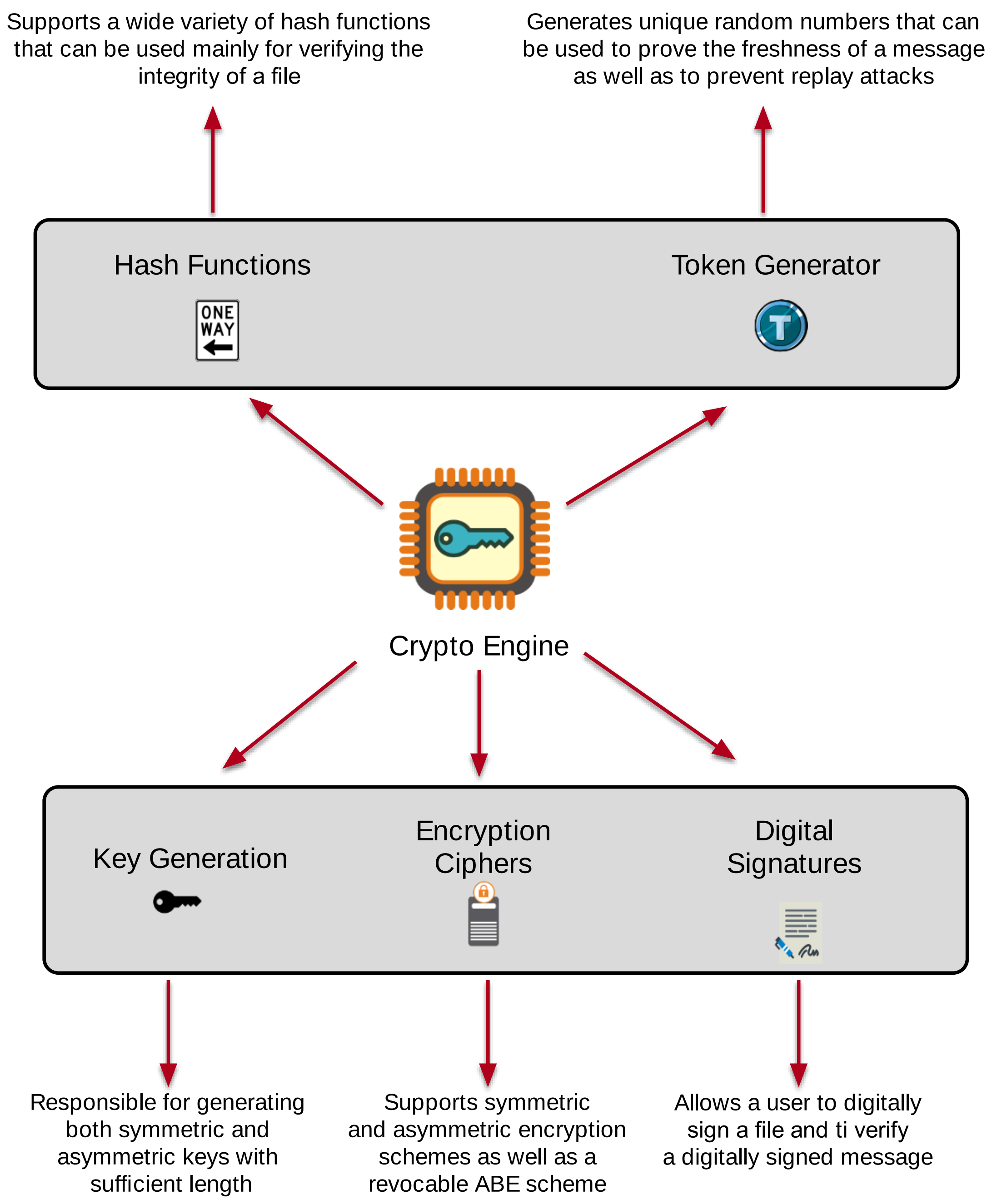
Supports a wide variety of hash functions that can be used mainly for verifying the integrity of a file
Generates unique random numbers that can be used to prove the freshness of a message as well as to prevent replay attacks
Hash Functions
ONE WAY
Token Generator
T
Crypto Engine
Key Generation
Encryption Ciphers
Digital Signatures
Responsible for generating both symmetric and asymmetric keys with sufficient length
Supports symmetric and asymmetric encryption schemes as well as a revocable ABE scheme
Allows a user to digitally sign a file and ti verify a digitally signed message

Figure 1: Main Functions offered by the Crypto Engine Component

2. **Symmetric Ciphers Suite:** This is as a library where several symmetric encryption algorithms can be used. This list will contain by default the most popular symmetric ciphers such as Advanced Encryption Standard (AES), Data Encryption Standard

(DES) and Triple DES. While the use of DES and TripleDES is widely discouraged from use, we consider it is important to maintain compatibility considering their widespread deployment in legacy applications. All the aforementioned algorithms will be offered in all possible versions (i.e. different key size). A legitimate user/microservice will be able to interact with Crypto Engine and use any of the supported symmetric ciphers. In general, a symmetric encryption cipher will follow the following specifications:

Definition 1 (Private-Key Encryption): For an arbitrary message $m \in \{0,1\}$, we denoted by $c = Enc(K, m)$ a symmetric encryption of m using a symmetric secret key $K \in K$, where K is the available message space. The corresponding symmetric decryption operation is denoted by $m = Dec(K, c)$.

3. **Asymmetric Ciphers Suite:** similar to the Symmetric Ciphers suite, this is a library that contains all the necessary functions to properly run a typical asymmetric encryption scheme. More precisely, the default scheme that will be supported from this suite is the Rivest-Shamir-Adleman scheme which is also known as RSA. However, a user with specific access rights, will be able to enhance the Asymmetric Ciphers Suite by adding several other ciphers.

Definition 2 (Public-Key Encryption): We denote by pk/sk a public/private key pair for an asymmetric encryption scheme. Encryption of a message m under the public key pk is denoted by $c \leftarrow E_{pk}(m)$ while the corresponding decryption operation is denoted by $m = Dec_{sk}(c)$.

Apart from the traditional asymmetric encryption schemes that will be offered to the users, the Asymmetric Ciphers Suite will also support a revocable Attribute-Based Encryption (ABE) scheme. This will allow users to create ciphertexts that will be protected by certain policies. In addition to that, this kind of encryption will allow users to share encrypted files without having to share a unique secret key for the decryption. The ABE scheme that will be supported by default is the so-called revocable Key-Policy Attribute-Based Encryption (KP-ABE) scheme described in [15] and further used in other works [16]. We proceed by defining the main steps of a KP-ABE scheme.

Definition 3 (Revocable Key-Policy ABE): A revocable KP-ABE scheme is a tuple of the following five algorithms:

1. $Setup$ is a probabilistic algorithm that takes as input a security parameter λ and outputs a public key pk and a master key MSK. We denote this by $(pk, MSK \leftarrow Setup(1^{\lambda})$.
2. Gen is a probabilistic algorithm that takes as input a master key, a policy $P \in P$ and the unique identifier of a user/entity and outputs a secret key which is bind both to the corresponding policy and the corresponding unique identifier. We denote this by $(sk_{P,ID}, ID) \leftarrow Gen(MSK, P, ID)$.
3. Enc is a probabilistic algorithm that takes as input a public key, a message m, a set of attributes Ω and a timestamp t. After a proper run, the algorithm outputs a ciphertext $c_{s,t}$ which is bind both to the set of attributes and the time. We

denote this by $c_{S,t} \leftarrow Enc(pk, m, S, t)$.

4. $KeyUpdate$ is a probabilistic algorithm that takes as input a master key, a revocation list rl and a timestamp t and outputs a key update information for time t. We denote this by $(K_t) \leftarrow KeyUpdate(MSK, rl, t)$.
5. Dec is a deterministic algorithm that takes as input a secret key, a key update $K_{t'}$ and a ciphertext and outputs the original message m iff the set of attributes that are bind to the ciphertext satisfy the policy $P, t' \geq t$ and the ID of the corresponding user was not revoked at time t. We denote this by $Dec(sk_{P,ID}, K_{t'}, c_{S,t}) \rightarrow m$.

4. **Digital Signature:** A digital signature is an asymmetric encryption algorithm that will be used in order to verify the integrity of a message as well as the actual identity of the sender. A digital signature over a message m is denoted by $\sigma = Sign_{sk}(m)$, where sk is the private key of the signer (as described in the asymmetric encryption scheme earlier). Apart from the signing algorithm, there is also a signature verification scheme that allows anyone who has access to the public key of the signer to verify the validity of the signature. The verification algorithm is denoted by $b = Verify_{pk}(m, \sigma)$, where pk is the corresponding public key, m is the message that has been signed and σ is the actual signature over m. Finally, the output of $Verify$ is a single bit $b = 1$ if the signature is valid and $b = 0$ otherwise.

5. **Cryptographic Hash Functions:** This will be a list of available cryptographic hash functions that will be supported by COLA. A hash cryptographic function is a special class of hash function that have certain properties that makes it suitable for use in specific communication protocols. More precisely, a cryptographic hash function is easy to compute but it should be impossible to invert. In addition to that, cryptographic hash functions can be also seen as one-way compression functions since they reduce the size of any message to a standard length. Furthermore, for the needs of our project the following cryptographic hash functions will be supported by default:
   - SHA1, for legacy applications only;
   - SHA2;
   - SHA3.

   While users will have the option to add more hash functions to the Crypto Engine, there should be specific mechanisms that will prevent the addition of specific hash functions that are considered as insecure (e.g. MD5, SHA1, except for legacy applications). A hash function over a message m is denoted by $h_m = H(m)$. Finally, cryptographic hash functions will be used to successfully and efficiently complete many phases of our operations (e.g. calculate and easily verify a signature will require from the user to first hash the message that wishes to sign).

6. **Message Authentication Code (MAC):** A MAC is a special case of a hash function that uses a symmetric secret key K and is used over a message m to calculate a unique fingerprint that can be used to exchange messages that are integrity protected. The MAC of a message m with a secrete key K is denoted by $\mu = MAC(K, m)$.

7. **Token Generator:** A token generator is a function that is responsible for generating random numbers that can be used to prove and verify the freshness of a message. We denote by $\tau = RAND(n)$ a random binary sequence of length n where $RAND(n)$ represents a random function that takes as input a binary length argument n and outputs a a random sequence of the same length.

4.2 Credential Manager (CM)

The Credential manager, an abstract component, is responsible for securely storing the credentials of entities that can access the MiCADO service. Credential manager can receive requests from any entity and is responsible for realizing the corresponding credential in a secure and privacy-preserving way. In addition to that, all credentials that are managed by the CM should be stored in such a way that the CM would be unable to reveal any valuable information about the content of the credential except the fact that there are valid. Hence, CM is not considered as a trusted entity. However, we explicitly assume that it will follow protocol specifications correctly. In addition to that, CM needs to communicate with the Crypto Engine micro-service to reveal the identity of a misbehaving user. Figure 2 shows a high-level overview with the main functions supported by the credential manager. We proceed by giving a formal description of the protocols that can be run by the CM.

Definition 4 (COLA.CM): The functionality supported by the credential manager in COLA, is defined by a tuple of the following three algorithms:

1. $CM.CredReceive$ is a protocol that takes place between an entity that wishes to login or to store a fresh credential that owns at a local database hosted by the CM. This algorithm takes as input a credential $cred$, the public key pk of the sender, the public key pk_{CM} of the CM as well as a signature of $(cred \vee pk)$ with sk. The generated message is encrypted with pk_{CM} and it is sent to CM. We denote this by $CredReceive = cred, pk, \sigma >$.
2. $CM.CredVerification$ is the process through which CM verifies the validity of the credential received in the previous step. It takes as input $CredReceive$, which is the output of $CM.CredReceive$ and outputs a bit b=1 if the credential is valid and b=0 otherwise. The verification process includes the following steps:
   1. Upon reception of $CredReceive$, CM uses pk_{CM} to decrypt the message.
   2. Calculates the hash of the message and verifies the received signature.
   3. Verifies that the credential is valid (i.e. has not expired, revoked, etc.) and gives or refuses access to the requestor.
3. $CM.Revoke$ is the process through which CM revokes access to a credential. The algorithm, takes as input a credential $cred$, a revocation list rl and outputs a single bit b=1 if the process has been completed successfully and b=0 otherwise.

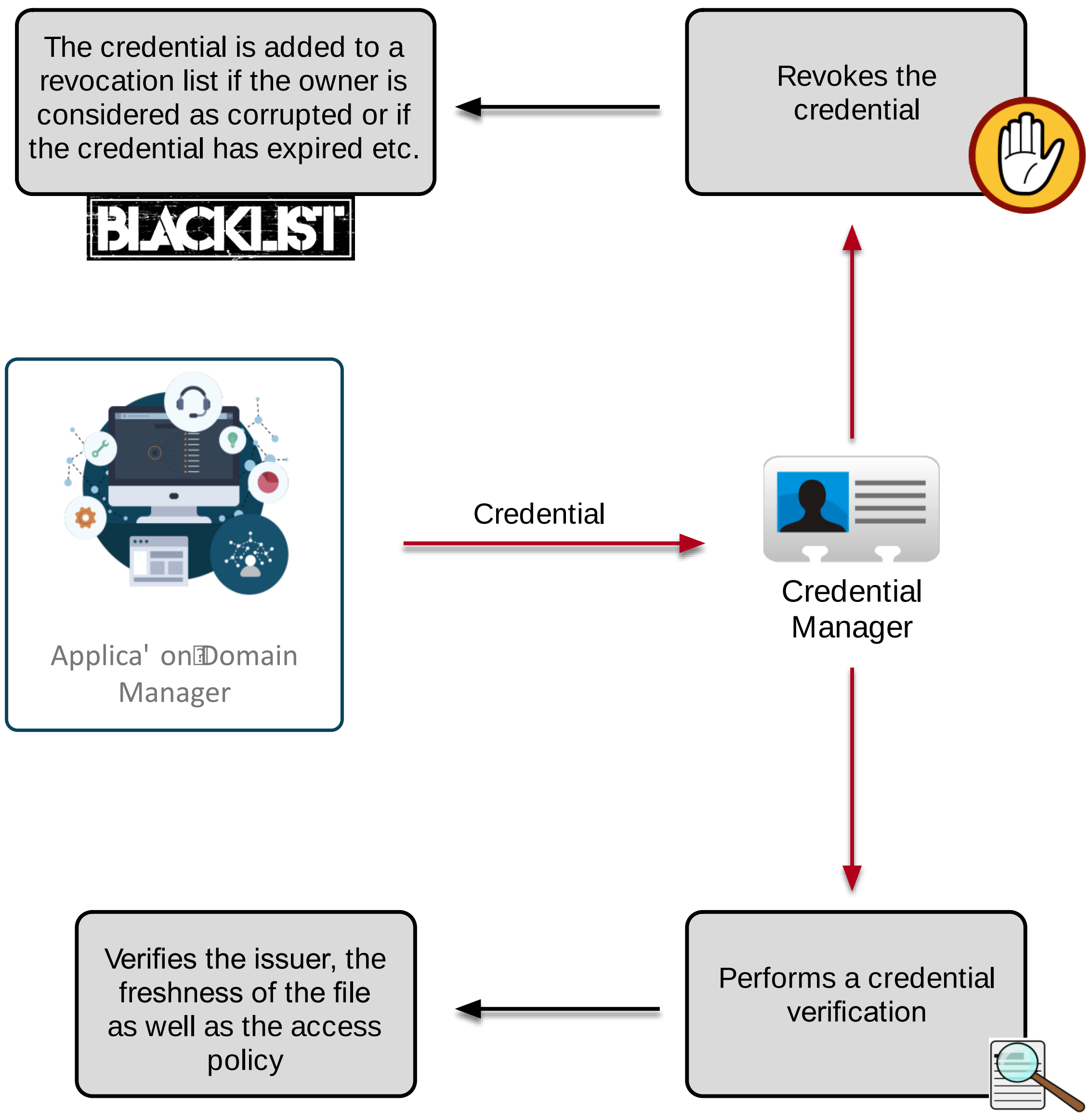
The credential is added to a revocation list if the owner is considered as corrupted or if the credential has expired etc.
BLACKLIST
Revokes the credential
Applica' on Domain Manager
Credential
Credential Manager
Verifies the issuer, the freshness of the file as well as the access policy
Performs a credential verification

Figure 2: Main Functionality Offered by the Credential Manager

4.3 Trusted Third Party

COLA uses a "trusted third party", an abstract component, with a key role in the overall framework and is trusted by the rest of the components. The role of TTP is of paramount importance for the security of COLA since it will be responsible for generating certain security guarantees about the trustworthiness of the cloud infrastructure that will be connected to MiCADO through the cloud access API. We rely on the commonly supported proposition that a large code base normally contains a proportionally large number of vulnerabilities. To reduce the code base, it is important that the TTP only supports the minimal necessary functionality. A TTP can communicate with components deployed on compute hosts to exchange integrity attestation information, authentication tokens and cryptographic keys. In addition to that, the TTP can verify the integrity of a pre-defined set of security-sensitive code and data executing or stored on the compute hosts. This can be done over an *attestation protocol* assuming that the hosts are equipped with common, commodity hardware-based isolation components or features – such as a Trusted Platform Module, or

Software Guard Extensions. In addition to that, the TTP can *seal* data to the correct configuration of a compute host, such that the data is only available if certain pre-defined code and data on the host have not been modified. For the needs of COLA, TTP will be communicating with Security Policy Enforcement Layer in order to provide certain information that is needed for the successful verification of a cloud resource. Moreover, TTP can verify the authenticity of a client as well as perform necessary cryptographic operations.

The main operation that TTP can perform is what we call a remote attestation. This process will prove that a cloud host is running under a trusted state. While this protocol is quite complex, it is a work that has been already designed and implemented in the past by members from the consortium of the project. To this end, the remote attestation protocol will be solely based on the work presented in [6, 9].

4.4 Image Integrity Verifier

This abstract component is responsible for determining whether one or more of image files are corrupted. Many attacks are focused on modification of critical files or configuration parameters. Especially, corrupted image files are considered as a substantial threat for cloud environment since a corrupted image can result in being unable to perform operations on a virtual machine/container. These operations include powering it on, taking a snapshot, and even modifying the virtual disks.

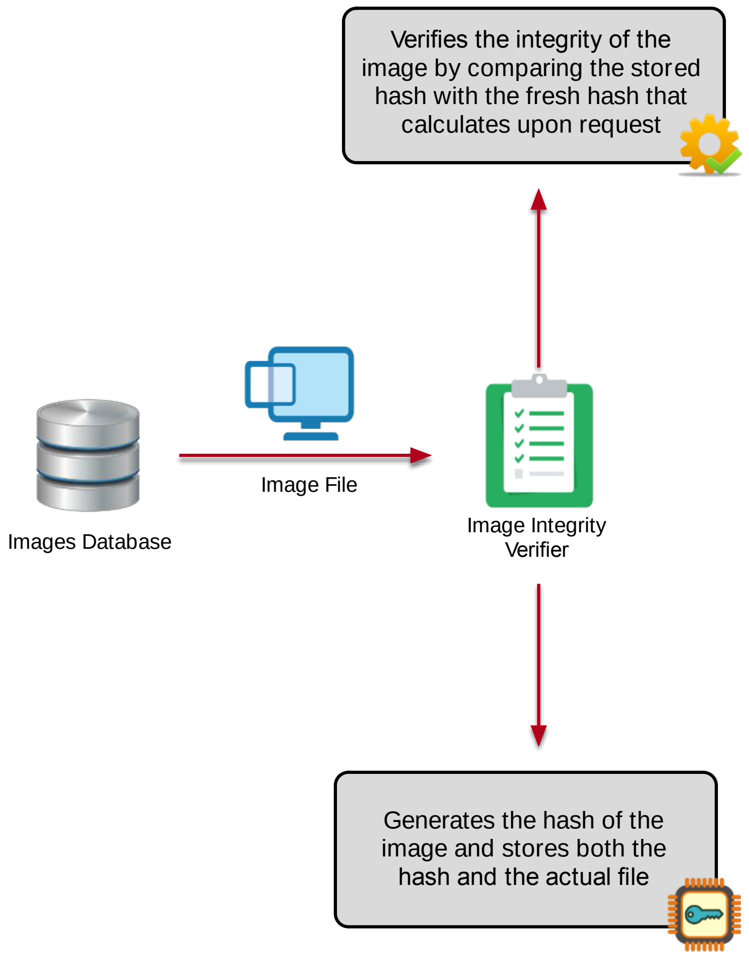
Verifies the integrity of the image by comparing the stored hash with the fresh hash that calculates upon request
Image File
Images Database
Image Integrity Verifier
Generates the hash of the image and stores both the hash and the actual file

Figure 3: Image Verifier

The image verifier will rely on a protocol where the integrity of the image will be verified. Figure 3 shows a high-level overview of the two main functions supported by the image verifier component. For the proper execution of the protocol, certain functions from the Crypto Engine will be used by the image integrity component. We now proceed by giving a formal description of the underlying protocol.

Definition 5 (COLA.ImageIntegrity): The image integrity protocol, is defined by the following two algorithms:

1. $ImageIntegiry.Store$ takes as input a credential an image file img and the public key pk of the user/entity that is initiating the process. The output of this algorithm is a hash value of the current state of the image along with a token and a hash of the public key of the user who stores the image file. We denote this by $I_{img} = h_{image}, h_{pk}, \tau >$.
2. $ImageIntegiry.Verification$ is the process through which the integrity of an image is verified. It takes as input I_{img} as well as the actual image img that we need to verify its integrity and outputs a bit b=1 if the image is not corrupted and vald and b=0 otherwise. We denote this by $b = ImgVerify(I_{img}, img)$. The verification is a simple process where the image integrity verifier calculates the hash of img and then compares the generated hash with the one contained in I_{img}. If the two hashes are identical, then the freshness of the file is checked by verifying that τ is not a token that has been used in the past. This process, ensures that the integrity of the corresponding image has not been corrupted. However,

```
this process does not check if the image is following a specific
security profile. This is done by other components as well as from
the attestation process that is supported by the TTP.
```

In the next deliverables we plan to evaluate whether file integrity monitoring tools can run on the orchestration management platform. This will allow us not only to provide a wide variety of image integrity functionality but also to compare several approaches regarding both their efficiency and their robustness. We hope that this work will give valuable insights to protocol designers who wish to conduct further research in this emerging problem as well as to many companies who are building certain cloud-based services.

4.5 Policy Manager and Security Policy Enforcement Layer

Policy manager (PM) is an abstract component that provides a scalable way to manage the security of numerous applications and orchestration functions. More precisely, PM can define and distribute security policies, allow the installation of certain software to local or remote systems and monitor the activities of all systems in the COLA framework to ensure compliance with corporate policies and centralized control. Having a centralized policy management helps users to use same access policies in multiple applications and CSPs. PM is communicating directly with the application domain manager. Hence, through certain monitoring processes it is easy to verify that the entire domain is protected as well as to modify the security settings when necessary. In addition to that, policy manager is responsible for creating policies that will be used to allow certain users and microservices to decrypt and use image files. More precisely, this will be done using the revocable ABE scheme described earlier and is offered by the Crypto Engine. Every time that a new image is created, PM will be generating a set of attributes that will be given as input to the ABE scheme. This will generate an encrypted instance of the image that will be bind to the attributes generated earlier by PM. The policy enforcement layer will later use these attributes to make sure that only users with certain access rights can access/decrypt the encrypted images.

The policy enforcements layer is responsible for ensuring that certain policies will be followed. Moreover, this layer will be responsible for giving access to certain image files only to users that have certain access rights. More precisely, every time that a user wishes to access an image will have to first decrypt it. The decryption will be based on the ABE scheme we defined earlier. Each user will have to communicate with the Crypto Engine to generate an ABE public/private key pair. This process will also include the cooperation of the policy manager who will be responsible for creating a set of policies that will be added to these keys. Then, every time when a user wishes to decrypt an image, will have to use the generated key private key that owns. The decryption process will be successful if and only if the attributes that are bind to the corresponding image satisfies the policy that is connected to the user's private key. With this way, we make sure that a policy will be always enforced, otherwise, the user will not be able to decrypt the image file. Hence, only users with certain permissions will be allowed to use and/or make changes to the stored image files. In addition to that, the security policy enforcement layer in the MiCADO architecture is responsible for the verification of both external and internal cloud resources for the application domain. Such verifications typically require interaction with external verification resources (in our case this will be the trusted third party described earlier). For clarity, such connections are omitted sin the overall architecture picture.

5 Security Architecture Implementation

This section introduces the implementation approaches to address the COLA security objectives outlined in Section 3.2 and the COLA security requirements described in Deliverable *D7.1 COLA security requirements.*
This approach closely follows the MiCADO architectural principles described in Section 3.3. It explicitly implements several of the security components described in Section 4. Its modular structure allows to further extend it to implement additional security components as necessary.

The Security Policy Enforcement Layer implementation aims to be cloud provider agnostic in order to provide a uniform security level and usage experience regardless of the platform used. In practice, this means application of provider specific security functions *SHOULD* be minimized to provide meaningful defaults (that can be considered as part of the specific cloud provider's integration); dynamic configuration updates of provider specific security functions based on user configuration *MUST NOT* take place.

The Security Policy Enforcement layer follows a matrix model to mitigate network related threats ranging from low level network layers to specific application protocols across the different assets of the platform.

5.1 Security assets

The assets that need to be protected are as follows:

Master Node: The Virtual Machine that hosts most of MiCADO's Internal Service Components.

Worker Node: An instance of a Virtual Machine that is created on demand to host an application.

Internal Service Components: Components that provide the internal services required to operate MiCADO. Most of these are hosted on the Master Node, but some are present on the Worker Node typically acting as an agent to one of the central internal services hosted on the master node, in order to supervise the worker node or to provide some Cloud Provider agnostic service on the Worker Node (such as *dockerd* to run application containers).

Application Service Components: Components that run user applications, deployed as (a set of) Docker containers on worker nodes.

Internal Communication Traffic among Internal Service Components: Communication flows that take place inside a Master / Worker Node among the Internal Service Components. Depending on the level of trust placed in the Cloud Service Provider such traffic can be considered to happen on a trusted network and as such do not require a secured channel of communication to mitigate the risk of an *Adversary* eavesdropping on it. Best practices still dictate that these channels *SHOULD* be encrypted unless a specific, noteworthy benefit (for e.g. in performance of a highly saturated interface) comes from using unencrypted channels. Such communication *MUST* still be authenticated.

External Communication Traffic among Internal Service Components: Communication flows that flow over among the Internal Service Components over untrusted networks, typically among Nodes. Such traffic *MUST* always be encrypted and authenticated.

Management Communication Traffic: Communication that flows from MiCADO's administrative user to an Internal Service Component

Internal Application Communication Traffic: Communication that flows among an Application's Service Components, either inside a Worker Node or among worker nodes if the Application has components on multiple nodes.

External Application Communication Traffic: Communication that flows to (such as user requests) or from (such as communication towards external backend systems) of an Application Service Component. External dependencies might limit the available security controls that can be applied to such traffic.

Authentication Credentials: Credentials that are needed inside the system, they MUST be protected both in transit and when at rest.

5.2 Security enforcement points

Enforcement of security policies happen at several places, listed below:

Cloud Service Provider (CSP): Cloud services have various means to provide security measures. While MiCADO's security Policy Enforcement Layer aims to be provider agnostic and thus limits to a minimum the usage of such interfaces (in some cases it may explicitly open them to avoid interference with its enforcement points), certain limitations can only be applied on such.

Docker network driver: *Docker*[10] and *Docker swarm*[11] use an overlay network that emulates VLANs and provide routing external traffic to them. This can be used for Network Segregation. It is possible to encapsulate node external traffic in IPSEC which can be used to secure node external traffic.

Zorp: In order to *demonstrate* the utilization of such security services, MiCADO will be extended by supporting a policy management software component, *Zorp*. Zorp[12] is a GPL licensed firewall solution that can act both as a simple Network Level access control tool and as an Application Protocol enforcer for certain protocols (TLS/SSL, HTTP, FTP, SMTP, POP3, etc.). It can also provide authentication for those protocols prior to letting traffic through. These features combined into a single software stack make it feasible to be applied with smaller impact on architectural complexity than deploying distinct software for these features. It can be deployed directly both as a firewall on Virtual Machines (Master and Worker nodes) as well as a Docker container.

[10] Docker product page: https://www.docker.com/
[11] Docker Swarm documentation page: https://docs.docker.com/engine/swarm/
[12] https://github.com/Balasys/zorp

The analysis of the alternative approaches to policy management is out of the scope of this document. However, given the modular architecture of MiCADO, Zorp may be replaced by other alternative policy management components. Within the framework of the COLA project, Zorp was prioritized over other proprietary or open source solutions available on the market considering that Zorp is a product of COLA project partner Balabit. Therefore, the necessary expertise regarding the possible tailoring of Zorp to MiCADO is readily available among project partners. Note that Zorp is a specific instance of a software component necessary to demonstrate that an external security solution can be integrated with MiCADO.

Credential Store (CS): An abstract software component that acts as a central storage of Authentication Credentials that is used to securely handle passwords, keys, tokens, other secrets in the system. It stores and serves credentials for other components in a secure manner.

Application code and configuration: Business logic in each application should ensure it is protected against threats. Their configuration should also limit attack surfaces by narrowing access, for example by binding only to localhost or to a local network when applicable.

The following table summarizes enforcement points:

	Network segregation	**Network level access control and security**	**Application protocol enforcement**	**Access authentication and authorization**
Master Node	CSP	Zorp@Master	N/A	N/A
Worker Node	CSP	Zorp@Worker	N/A	N/A
Internal Service Components	App, docker	Zorp@Node	Zorp@Node	Zorp@Node / App, CS
Application Service Components	App, docker	Zorp@Container / docker	Zorp@Container	Zorp@Container / App
Internal Communication Traffic among Internal Service Components	App, docker	App, docker	Zorp@Node	App, CS
External Communication Traffic among Internal Service Components	App, docker, Zorp@Nodes	Zorp@Node, docker	Zorp@Node	App, CS

Management Communication Traffic	CSP	Zorp@Node	Zorp@Node	Zorp@Node / App / CS
Internal Application Communication Traffic	docker	Zorp@Container / docker	Zorp@Container	Zorp@Container / App / CS
External Application Communication Traffic	docker	Zorp@Container, docker	Zorp@Container	Zorp@Container / App / CS
Authentication Credentials	N/A	Zorp	N/A	Zorp

5.3 Component Implementation and Interactions

This section describes the implementation and interactions of components described in Section 4 with the components of the MiCADO architecture. In particular, Figure 4 illustrates the communication between the *security components* and *MiCADO components*. This high-level overview identifies the main operations of the security components. For clarity, all security components are drawn in red colour. For details regarding the MiCADO architecture, its components and functionalities please refer to COLA deliverable D.6.2 [17].

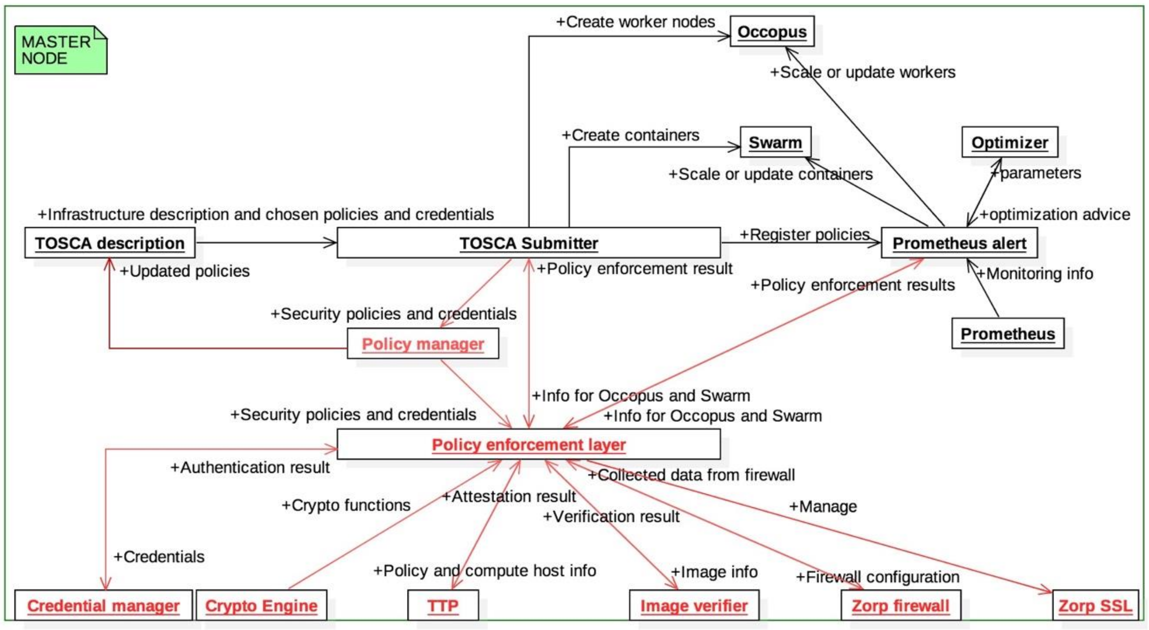

Figure 4 MiCADO master node with security components

The main communication between MiCADO and the designed security components are between the *TOSCA submitter*, *Prometheus alerting* and *Policy enforcement layer (PEL)*. PEL is a gateway to protect the MiCADO system from insecure command execution. All commands that consist of *creating*, *scaling* and *updating* workers or containers should be verified in order to guarantee its security prior execution. As a consequence, the TOSCA submitter and Prometheus alerting need to send configuration data to PEL and only after receiving enforcement results from PEL, they can transmit execution commands to Occopus/Swarm.

Furthermore, the PEL needs to be aware of the security policies to be enforced. Security policies are given as input to MiCADO by utilizing a TOSCA description. Security policies are transmitted to the TOSCA submitter along with other operation policies, credentials and configuration data. Next, the TOSCA submitter extracts security policies with credentials, and transfers them to Policy Manager (PM). The PM is in charge of storing and distributing policies and credentials to PEL. In addition to that, all credentials are transformed before being sent to the Credential Manager (CM) to enable the CM to reveal any valuable information about the private data contained in a credential. Apart from that, there is a reverse communication from the PM to the TOSCA description which is used when the PM generates new policies or updates the existing ones.

Additionally, the PEL sends requests to cryptographic components such as the Crypto Engine, Trusted Third Party (TTP), Image Verifier to execute cryptographic operations and protocols such as revocable ABE encryption, host attestation, image verification, etc. The communication between CM, TTP, Image Verifier and Crypto Engine (CE) is required in order to use functions provided by CE. For clarity, we omit such communication in Figure 4.

Finally, in order to provide an additional layer of security, a firewall will be installed to prevent possible malicious or suspicious access both to the master node and the worker nodes. Furthermore, all underlying communication between the different instances will be protected using TLS. This includes the communication between Occopus and worker node, Swarm and Docker CE, Prometheus and Node or Container monitor. The implementation of this functionality will be based on *Zorp firewall* and *Zorp SSL*, described in Section 5.5. These components are controlled by PEL that is also responsible for generating and sending the corresponding configuration files. On each worker node, there is a PEL with reduced functionality, to control the Zorp firewall and SSL. This entity is also managed by the PEL in the master node as shown in Figure 5. Finally, the master node's PEL is responsible for generating and sending configuration files and retrieving and processing collected data from the worker nodes' PEL. Communication between the PEL on the different nodes must be done in secure manner and privacy-preserving way.

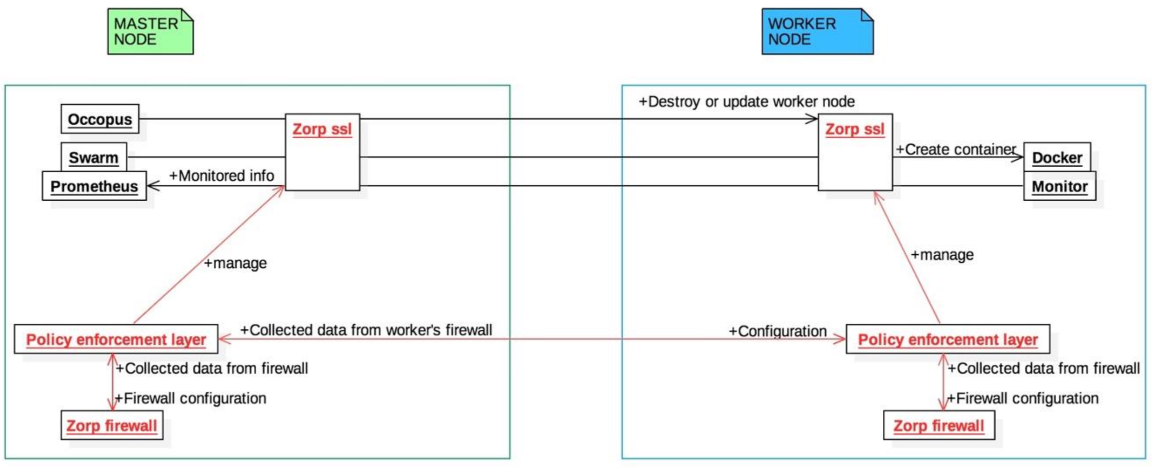

Figure 5 Communication between master node and worker node with security components

5.4 Security Layers

This subsection describes the layers where protection is applied, with their most relevant functions:

Network segregation: Security domains – such as a given application hosted on the platform or services that belong together on the Master Node – *MUST* be separated on the level of the Network Interface Layer of the TCP/IP model (the quasi-physical network), in order to avoid eavesdropping and to force all security related decision onto the upper layers where enforcement points are more conveniently applied through routing.

Network level access control: Network traffic *SHOULD* also be filtered on the Host-to-Host Transport Layer. The main goal of such filtering is to reduce the attack surface by:

- denying traffic from known unwanted sources
- denying traffic to non-existent destinations
- detecting malicious sources by analysing network behaviour
- allowing access to certain services only from trusted network sources

Such actions narrow the surface that has to be protected by Application protocol enforcement.

Application protocol enforcement: Network traffic passing to/from hosted application code *SHOULD* also be protected by denying protocol non-compliant elements and traffic with malicious intentions. While such enforcement *SHOULD* happen in the application itself for well-defined protocols, it is also possible to enforce rules by application level firewalling. This adds several benefits by providing an extra layer of application enforcement, making sure malicious traffic never reaches application code and reducing the load on application services.

Access authentication and authorization: By general rule, any agent sending or receiving sensitive data *MUST* authenticate the other party in the communication and make sure it has authorization to given data. Therefore, client access to services *MUST* be authenticated and authorized except for services explicitly published to the general public. In cases where relevant information is flowing from the client – typically in traffic among Internal Service

Components – services should also be authenticated. Bi-directional channels thus *MUST* be mutually authenticated. Authentication credentials *MUST* be securely handled:

- Machine to machine communication *SHOULD* use certificate based authentication whenever possible. Every service *SHOULD* have its own individual set of credentials. Access to private keys must be limited to the using application. Private keys *SHOULD* be protected by a pass-phrase provided application initialization.
- When other password based solutions are needed credentials *MUST* be stored securely in the CS and retrieved upon need. Access to the CS must be individually authenticated for each user of the credential by their own.
- Passwords used internally for authentication *SHOULD NOT* be persistently stored in a retrievable manner (plain-text or symmetric encryption), instead cryptographically proper hash functions of the password should be stored.

Authentication *MAY* be supplemented by User Behaviour Analysis for further securing the system.

Session Audit: Privileged access *MAY* be recorded for audit purposes.

5.5 Security policy management

The MiCADO security architecture will enable and support the utilization of various security solutions/products based on user defined policies. Security policies are applied through several channels:

- Static configuration is built into the default configuration of each component.
- Dynamic configuration is based
  - mainly on infrastructure description through user input by TOSCA
  - any dynamic polices from Policy Keeper
  - administrative configuration through the Dashboard might be necessary in the future
- Dynamic configuration is maintained by the Central Security Component (CSC). The CSC will
  - receive TOSCA updates through MICADO submitter
  - may receive policy keeper updates
  - gather actual infrastructure details from Docker Swarm
  - calculate and maintain dynamic security policies
  - distribute these policies as configuration to the enforcement points

Figure 6 illustrates the envisioned integration of Zorp with the other software components of the COLA architecture. In particular, Zorp will be deployed on both the master node as well as the MiCADO worker nodes. Zorp will enforce the security policy implementation – including security of the communication between MiCADO nodes and components – according to the description of infrastructure and policies in the TOSCA format.

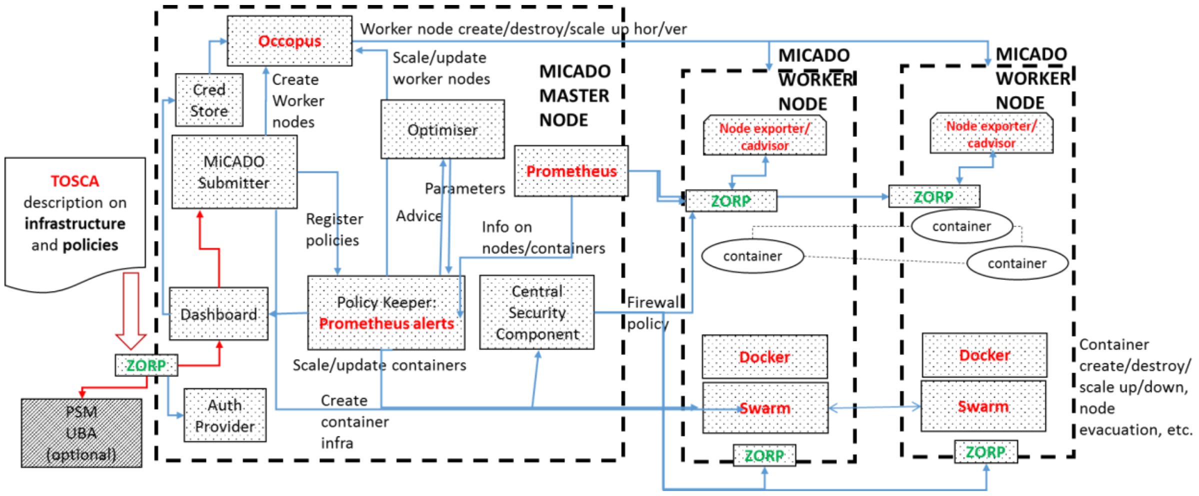

Figure 6 Relations of the Security Policy Implementation with the components of the COLA architecture

Beyond this high-level description, further details regarding the integration of Zorp with the MiCADO components, as well as detailed description of the communication protocols employed in the communication among the Zorp agents are out of the scope of this document and will be described in deliverable *D7.3 MiCADO application security classification specification*.

6 Security Requirements Traceability

This section describes the vertical relationship between the documents, namely between requirements, architectural objectives and security enforcement components.

The architectural objectives outlined in section 3.2 are based on a selection of the security requirements outlined in deliverable *D7.1 COLA security requirements*. The selected requirements were addressed by the architectural objectives and subsequently reflected in the architectural components described in section 4. We have chosen such requirements on an analysis of the current state-of-the-art with regards to cloud orchestration security as well as their applicability to the MiCADO cloud orchestration platform.

Table 4 contains a three-party mapping between the security requirements enumerated in D7.1, the architectural objectives described in section 3.2 and the architectural components described in section 4. This mapping is necessary in order to ensure the traceability of requirements throughout the project, from requirements elicitation to final prototype.

In several cases, security requirements are relevant to more than one objective and are addressed by a combination of several architectural components. It is important to note that we assume the architectural components to interact as a unified system in order to effectively address the relevant architectural objectives.

This mapping will be complemented and expanded in the upcoming deliverables, in order to continue the traceability of requirements, objectives and components. We expect that such traceability will communicate the rationale behind the security architecture and will help developers take better design decisions.

Table 4 Mapping of Security Requirements, Architectural Objectives and Architectural Components

Requirement(s)	**Objective(s)**	**Architectural Component(s)**
SR01, SR02, SR04, SR08, CNSR-3	O1.1, O3.3, O4.3	Trusted Third Party
SR05, SR06, SR07, SR09, SR10, CCSR-1, CNSR-2, CNSR-6, CSSR-1	O1.2, O3.5, O4.4, O5.1	Policy Manager
SR06, SR09, SR10, CCSR-2, CNSR-5, CNSR-6, CNSR-7	O2.1, O2.2, O2.3, O3.2,	Security Enforcement Layer
SR03, SR11, CNSR-3, CNSR-8	O4.3	Credential Manager
SR12, SR13, CNSR-3, CNSR-9, CSSR-1	O3.1, O5.2,	Crypto Engine
CNSR-2, CNSR-6	O4.1, O4.4, O6.2	Image Integrity Verifier

7 Summary and Conclusions

The scope of this deliverable was to twofold. First, we provided a detailed analysis of the security landscape of cloud orchestration. This work is something that is currently missing from the existing literature and it can give valuable insights to organizations as well as researchers who are using services based on the cloud orchestration principles. During this study, we extensively analysed the security of current orchestration approaches and we provided a concrete list of possible risks that need to be considered while one builds cloud orchestration-based services. Furthermore, by analysing existing security issues and identifying possible risks, we managed to define an adversarial model that fits squarely to the specific requirements of an orchestration platform. The defined threat model is based on a set of realistic assumptions regarding the capabilities of the attacker. These assumptions combined with the security issues we identified by analysing the existing literature, resulted in a concrete threat model that targets cloud orchestration environments. To the best of our knowledge, this is something that is currently missing from the existing works in the area. Hence, it is considered as an important contribution of this work. Furthermore, defining the exact capabilities of the attacker, we managed to describe the actual attacking surface. This, helped us to proceed with the actual design of relevant security and/or privacy-preserving mechanisms.

The second main contribution of this work, is the design of the actual security architecture that will enhance MiCADO. This architecture is based on the design of several security-related components that aim to satisfy the different types of requirements that have been formulated during the requirements analysis and the definition of the adversarial model. More precisely, D7.1 [11] highlighted specific functional and security requirements based on a concrete list of considered use-cases. The overall goal was to identify all stakeholders and as many as possible functionalities that would be required towards the formulation of a secure orchestration platform. As a result, all these parameters were taken into consideration in this deliverable while building the necessary security components.

During the design of the security architecture, we defined in a concrete way all the underlying security components that cover the functional aspects of the requirements. More precisely, we extended the high-level description of the security components that was described in D1.1 by properly describing all the functions that each component supports. In addition to that, we further elaborated on each component by providing a usage walkthrough. Even though the described architecture is considered as “reference”, since it can be subjected to multiple “instantiations”, it provides a solid overview of how the security components are orchestrated in order to support the MiCADO framework.

Finally, we provided a detailed discussion regarding current regulatory aspects related to cloud computing. While the COLA security architecture specification does not aim to enforce compliance to any specific framework, it does aim to be generic and extensible to be used by an orchestration platform compliant to such frameworks. To this end, several cloud certification and compliance frameworks were described.

8 References

[1] A. Michalas and K. Y. Yigzaw, "LocLess: Do you Really Care Where Your Cloud Files Are?". *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Luxembourg City, 2016, pp. 515-520, 12-15 Dec, 2016.

[2] A. Michalas, "Sharing in the rain: Secure and efficient data sharing for the Cloud," *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 182-187, 5-7 Dec, Barcelona, Spain, 2016.

[3] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. 2017. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Comput. Surv.* 50, 2, Article 26 (May 2017), 41 pages. DOI: https://doi.org/10.1145/3054177

[4] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (ASPLOS '13). ACM, New York, NY, USA, 461-472. DOI=http://dx.doi.org/10.1145/2451116.2451167

[5] Badger, Lee, et al. "Draft cloud computing synopsis and recommendations." *NIST special publication* 800 (2011): 146.

[6] N. Paladi, C. Gehrmann and A. Michalas, "Providing User Security Guarantees in Public Infrastructure Clouds," in *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 405-419, July-Sept. 1 2017. doi: 10.1109/TCC.2016.2525991

[7] Nicolae Paladi and Linus Karlsson. 2017. Safeguarding VNF Credentials with Intel SGX. In *Proceedings of the SIGCOMM Posters and Demos* (SIGCOMM Posters and Demos '17). ACM, New York, NY, USA, 144-146. DOI: https://doi.org/10.1145/3123878.3132016

[8] Dowsley, Rafael, et al. "A survey on design and implementation of protected searchable data in the cloud." *Computer Science Review* (2017).

[9] Paladi, Nicolae, Antonis Michalas, and Christian Gehrmann. "Domain based storage protection with secure access control for the cloud." *Proceedings of the 2nd international workshop on Security in cloud computing*. ACM, 2014.

[10] Paladi, Nicolae, and Christian Gehrmann. "TruSDN: Bootstrapping Trust in Cloud Network Infrastructure." *International Conference on Security and Privacy in Communication Systems*. Springer, Cham, 2016.

[11] A. Michalas, N. Paladi and C. Gehrmann, "D7.1 COLA Security Requirements", in COLA – Cloud Orchestration at the Level of Applications, (2017)

[12] Claudio A. Ardagna, Rasool Asal, Ernesto Damiani, and Quang Hieu Vu. 2015. From Security to Assurance in the Cloud: A Survey. *ACM Comput. Surv.* 48, 1, Article 2 (July 2015), 50 pages. DOI: https://doi.org/10.1145/2767005

[13] M. Dekker, C. Karsberg, M. Lakka, and D. Liveri, "Auditing Security Measures, An Overview of schemes for auditing security measures," Tech. Rep. TP-03-13-551-EN-N, European Union Agency for Network and Information Security, September 2013.

[14] Nicolae Paladi. 2017. Trust but Verify: Trust Establishment Mechanisms in Infrastructure Clouds. Ph.D. Dissertation. Lund University.

[15] A. Sahai and H. Seyalioglu, "Dynamic credentials and ciphertext delegation

for attribute-based encryption," *In Proceedings of the 32nd Annual International Cryptology Conference: Advances in Cryptology (CRYPTO2012)*, pp. 199–217, Springer, 2012.

[16] Antonis Michalas and Noam Weingarten. "HealthShare: Using Attribute-Based Encryption for Secure Data Sharing Between Multiple Clouds". *Proceedings of the 30th IEEE International Symposium on Computer-Based Medical Systems (CBMS'17)*, Thessaloniki, Greece, 2017.

[17] Botond Rakoczi, Jozsef Kovacs, Tamas Kiss, Peter Kacsuk, Gabor Terstyanszky: "D6.2 Prototype and documentation of the monitoring service", in COLA – Cloud Orchestration at the Level of Applications, (2017)