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:

D7.5 MiCADO security modules reference implementations

Due date of deliverable: 31/12/2018 Actual submission date: 21/12/2018

WPL: Nicolae Paladi

Dissemination Level: PU

Version: 1.6

Status and Change History

Table 1 Status Change History

Status:	**Name:**	**Date:**	**Signature:**
Draft:	N. Paladi	11/12/2018	Nicolae Paladi
Reviewed:	B. Despotov	20/12/2018	Bogdan Despotov
Approved:	T. Kiss	21/12/2018	Tamas Kiss

Table 2 Document Change History

Version	**Date**	**Pages**	**Author**	**Modification**
V0.1	20/11	7	Nicolae Paladi	Document template
V0.2	22/11	18	Nicolae Paladi	Add sample description for Image Integrity Verifier (Section 3.1)
V0.3	23/11	35	Nicolae Paladi	Add sample description for the Crypto Engine security enabler (Section 3.2)
V0.4	03/12	34	Nicolae Paladi	Review Crypto Engine security enabler (Section 3.2)
V0.5	03/12	34	Nicolae Paladi	Add introduction to deliverable
V0.6	04/12	71	Antonis Michalas Amjad Ullah Hai-Van Dang	Add sample description for the Credential Engine and Credential Store security enablers
V0.7	04/12	71	Nicolae Paladi	Write conclusion (to be updated)
V0.8	04/12	71	Amjad Ullah	Changes to Section 2
V0.9	04/12	71	Peter Bauer	Changes to Section 2
V1.0	04/12	71	Hai-Van Dang	Changes to Section 2 and 3.4
V1.1	05/12	72	Nicolae Paladi	Add section 4 Artefact Traceability
V1.2	05/12	75	Amjad Ullah Hai-Van	Changes to Section 3.3 and 3.4
V1.3	10/12	83	Peter Bauer	Add description of Zorp, Security Policy Manager and Master-Worker Secure Communication security enablers
V1.4	11/12	83	Balint Kovacs	Minor wording and formatting fixes in all sections
V1.5	18/12	84	Nicolae Paladi	Address some issues raised by reviewer
V1.6	20/12	84	Balint Kovacs	Final edit

Glossary

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
CSP	Cloud Service Provider
MAC	Message Authentication Code
HMAC	Hash-based Message Authentication Code
DoS	Denial of Service Attack
PII	Personally Identifiable Information

Table 3 Glossary

List of Figures and Tables

Table of Contents

1 Introduction

This document focuses on describing the reference implementation of MiCADO security enablers. The *reference implementation of MiCADO security enablers* and the current *MiCADO security modules reference implementation description* constitute Deliverable D7.5.

The current document aims to describe the specific design and implementation decisions taken during the development of the MiCADO security modules within the COLA project.
To achieve that, we follow two approaches. First, we thoroughly describe the reference implementation of the MiCADO security modules delivered in the project. Second, we outline the traceability of the MiCADO security module implementations relative to the earlier relevant deliverables, namely D7.1 COLA security requirements, D7.2 MiCADO security architecture specification, D7.3 Design of application level security classification formats and principles and D7.4 Security policy formats specification.

The main objectives of this document are as follows:

- Describe the reference implementation of MiCADO security enablers.
- Document the technical decisions, implementation trade-offs and limitations of the reference implementations.
- Complement the technical implementation of the MiCADO security enablers.

The MiCADO security modules reference implementations will be used as input for D7.6 "***MiCADO security evaluation report***", the last deliverable in Work Package 7 of the COLA project. The current document explicitly excludes out of its scope the description of the *integration* of the MiCADO security enablers into the MiCADO orchestration system. A report describing the integration of the MiCADO security enablers is expected in the following deliverable of the project, D7.6 "***MiCADO security evaluation report***".

The remainder of this deliverable is structured as follows:

- Chapter 2 – Overview of MiCADO security modules.

  This chapter contains an overview of the MiCADO security modules in the context of the MiCADO platform.

- Chapter 3 – MiCADO security modules implementation description.

  This chapter contains the descriptions of the MiCADO security modules included in the deliverable. The descriptions focus on the specific implementation decisions and solutions, implementation trade-offs and limitations of the delivered reference implementations.

- Chapter 4 – Artefact Traceability

  This chapter describes the traceability of the MiCADO security enabler implementations to the requirements and design specifications described in the earlier deliverables.

- Chapter 5 – Summary and conclusion

  This chapter concludes this deliverable.

2 Overview of MiCADO security modules

This section provides an overview on implemented security modules and their interaction with existing core components of MiCADO. Further details on the implementation of security modules will be described in Section 3.

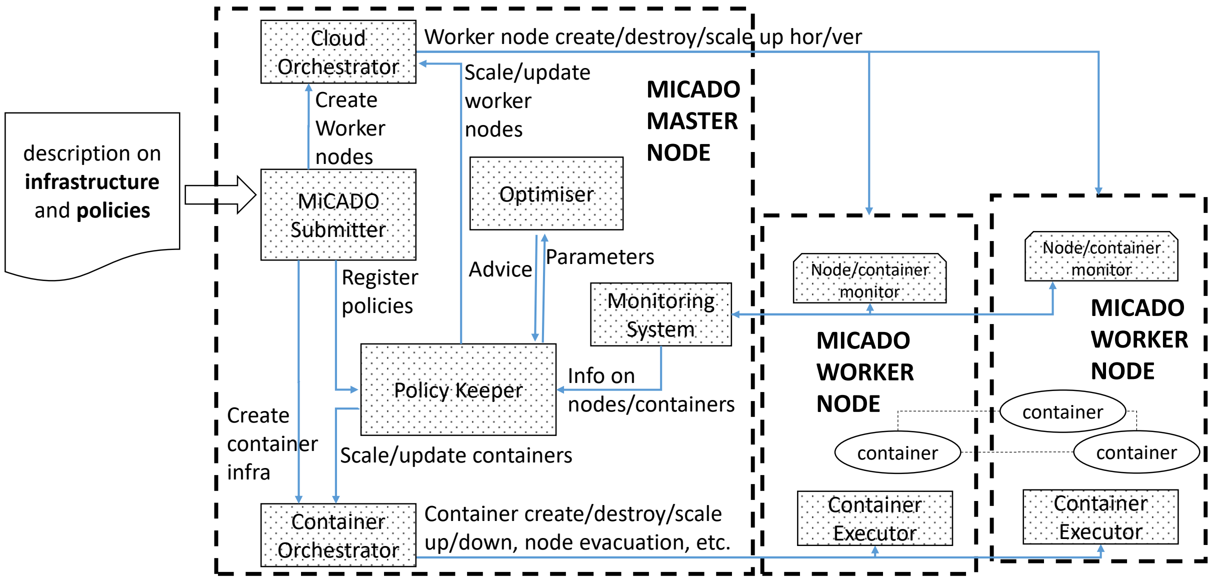


Figure 1 MiCADO infrastructure with core components [14]

Figure 1 displays MiCADO infrastructure, where it is composed of a master node and worker node(s). We mainly focus on MiCADO master node that consists of various components having different functionalities. The short descriptions of these components are provided below, except the Optimiser, which currently exists considering future extension.

- Submitter: It receives the Application Description Template (ADT) file from MiCADO users. The ADT file contains details on the application topology and the relevant policies, e.g. scaling and/or security policies. Please refer to deliverable D5.4 [15] for more details on the ADT file;
- Monitoring system: It collects monitoring data of virtual machines, microservices and containers from worker nodes;
- Policy Keeper: The purpose of the Policy Keeper is twofold. Firstly, it facilitates the definition of scaling policies in the ADT file. Secondly, it makes a scaling decision of virtual machines/containers based on the collected monitoring information;
- Cloud Orchestrator: It executes the scaling decision of virtual machines made by the Policy Keeper;
- Container Orchestrator: It performs the scaling of containers in worker nodes made by the Policy Keeper;

For further details on each component, please refer to deliverable D6.2 [14].

Figure 2 provides more secure version of MiCADO with additional security components.

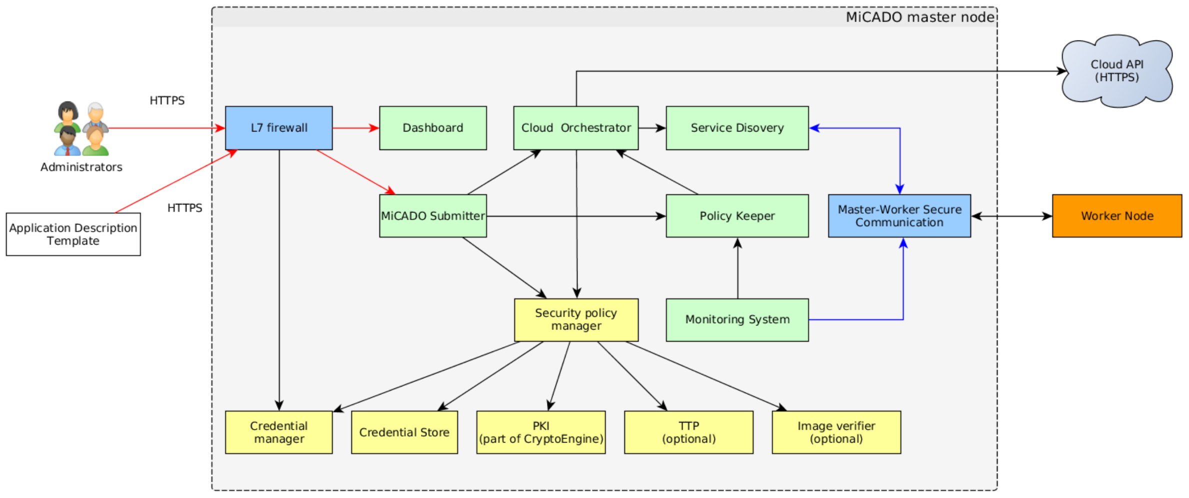


Figure 2 MiCADO infrastructure with core components and security components

The grey box in Figure 2 depicts the MiCADO master node along with all its components. The inner green boxes are the core components of MiCADO whereas the yellow boxes represent the security components. For the sake of simplicity, Figure 2 does not display all those core components that currently do not have direct interaction with security components. However, there are two additional components depicted in Figure 2 that are not core ones, i.e. Dashboard and Service Discovery. We show them as they also have interactions with the security components. The short description of all the security components are provided below except TTP, which is an extension for the future and therefore we skip it currently.

- Master Node L7 Zorp Firewall: It is an application level protocol firewall. It provides a secure TLS interface and adds authentication to the administrator's dashboard. The firewall protects the master node by blocking all outside communication but the management dashboard and the submitter;
- Master-Worker Secure Communication: It provides secure communication between master node management components and worker nodes. It identifies the endpoints and encrypts master-worker communication, ensuring authenticity and confidentiality;
- Security Policy Manager: It is a single point of access for MiCADO security components. The Security Policy Manager provides an aggregation of Restful API endpoints that serves different backends including Credential Store, Image Integrity Verifier, CryptoEngine, IPsec credentials and Kubernetes network join tokens;
- Credential Manager: It centrally manages all MiCADO users. It provides user verification for Zorp so that Zorp can perform authentication and access control. Besides that, it supplies the Security Policy Manager with functionalities for managing users such as creating, updating, deleting, etc.;
- Credential Store: It securely stores all sensitive information for MiCADO infrastructure. It provides the Security Policy Manager with functionalities to manage sensitive information such as creating, updating, deleting, etc.;
- PKI (part of CryptoEngine): It provides MiCADO with Public Key Infrastructure;
- Image Verifier: It ensures that the application images are not corrupted;

The following description provides an overview of the interaction between security components and core components. For communication among core components, please refer to deliverable D6.2 [14].

- Master Node Zorp Firewall → Dashboard: firewall provides secure communication, authentication and request routing to the different Dashboard components
- Master Node Zorp Firewall → MiCADO Submitter: firewall provides secure communication, authentication and request routing to the Submitter
- Master Node Zorp Firewall → Credential Manager: invokes a Restful API to the Credential Manager to verify the login credentials supplied by MiCADO users;
- MiCADO Submitter → Security Policy Manager: It invokes Restful APIs to Security Policy Manager (SPM) to enforce security policies defined in the ADT file;
- Security Policy Manager (SPM) → Credential Manager: For actions related to user management, SPM invokes APIs to Credential Manager;
- Security Policy Manager (SPM) → Credential Store: For actions related to sensitive information storage, SPM invokes calls to Credential Store;
- Security Policy Manager (SPM) → PKI (a part of CryptoEngine): For actions related to certificates issuing / signing / revocation/ etc., SPM calls PKI
- Security Policy Manager → Image Verifier: For actions related to application image verification, Verifier invokes calls to Image Verifier;

3 MiCADO Security Modules Implementation Description

3.1 Image Integrity Verifier

This section aims to describe the core functionality, requirements and implementation of the Integrity Image Verifier (IIV) as a security component in the MiCADO architecture. The IIV is responsible for providing integrity security guarantees to the MiCADO infrastructure. It does this through integrity verification of application images prior to deployment. The IIV provides a mechanism to detect corrupted images prior to their instantiation in the cloud.

3.1.1 Image Integrity Verifier Functionality

A privileged remote user provides the MiCADO infrastructure with a TOSCA file containing the list of images and VM's to deploy new topologies or services. The MiCADO infrastructure will delegate the integrity verification process of every image to the IIV, and it will continue with the deployment process only if the result of the integrity verification returns a valid response. IIV returns a positive response (TRUE) when the integrity of a supplied image is assured to have not been altered. Otherwise, the IIV returns a negative result (FALSE), which means the image integrity could not be confirmed.

Figure 3, illustrates the components interaction for the IIV and the sequence followed in the verification integrity of a valid image prior its instantiation. Within the remote attestation protocol, the Broadcaster - a component of the COLA architecture [1] - sends the image that is required to be verified plus the integrity quote of the enclave to the IIV (1). The integrity mechanism then validates the quote and if successful, proceeds to calculate a hash of the received image and compare the result against a hash stored in a list of well-known hashes (2). If both hashes match, the IIV mechanism returns the image and the result of the image integrity verification. Otherwise, only the result of the verification is returned (3). Upon a positive attestation result, the Broadcaster sends the image to the worker nodes for instantiation (4); otherwise, the image is rejected and the deployment aborted.

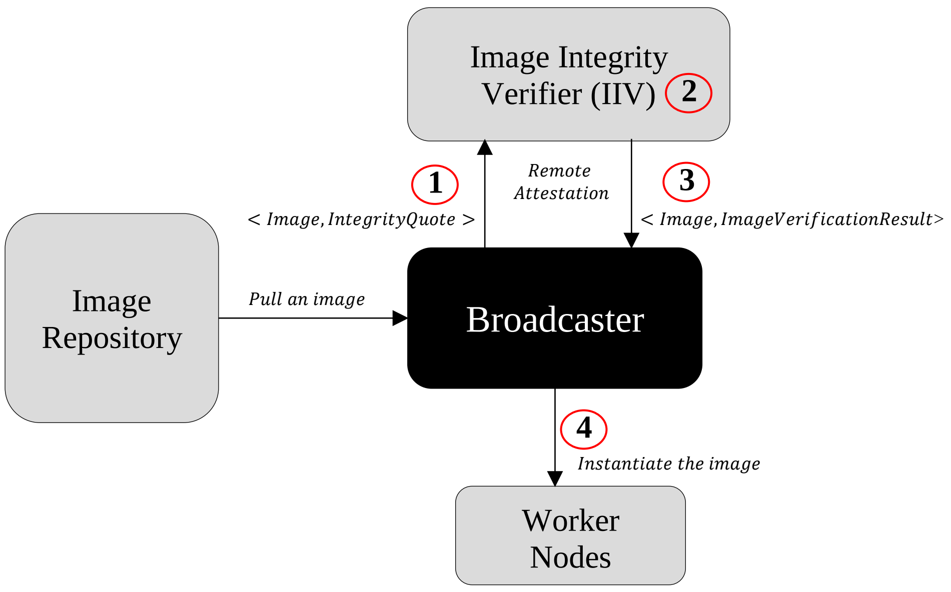


Figure 3 Component interaction for the image integrity verifier and image verification sequence

3.1.1.1 Image Integrity Verifier Requirements

To provide its core functionality, the IIV fulfils the following security requirements [1]:

1. Keep in a secure location a list of generated 256-byte hash (well-known measurements) of all images the system allows for deployment.
2. The integrity mechanism must verify the integrity of an image by comparing a fresh computing hash against a hash, for that image, in the list inside the IIV.

3.1.2 Image Integrity Verifier Design

The IIV core functionality is developed as an Intel SGX [2] dynamic library that can be embedded in a large system. The untrusted part exposes the API responsible for handling incoming requests with well-defined input parameters (i.e., an Image and the enclave's measurement) and returning the corresponding verification process result. The trusted part is in charge of the integrity mechanism itself. The integrity mechanism is invoked through a request via remote attestation, resulting in a quote with the result of the verification process. This quote is the basis of the decision to continue or halt image deployment. Figure 3, shows the IIV, developed as an SGX library, with its the respective components.

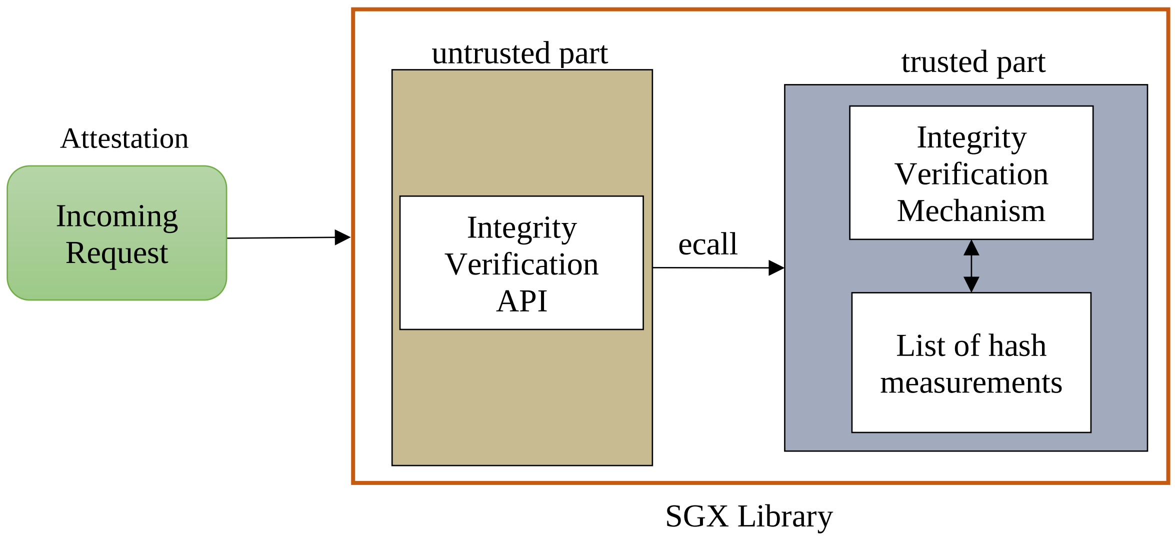

Attestation
Incoming Request
untrusted part
Integrity Verification API
ecall
trusted part
Integrity Verification Mechanism
List of hash measurements
SGX Library

Figure 4 The Image Integrity Verifier developed as an SGX library

The integrity verification API contains the function ImagVerify, which is responsible for the execution of the verification tasks within the enclave (trusted part) via an enclave call (ecall). The integrity verification mechanism is further split into sequential blocks, all of which are securely deployed inside the SGX enclave as depicted in figure 3.

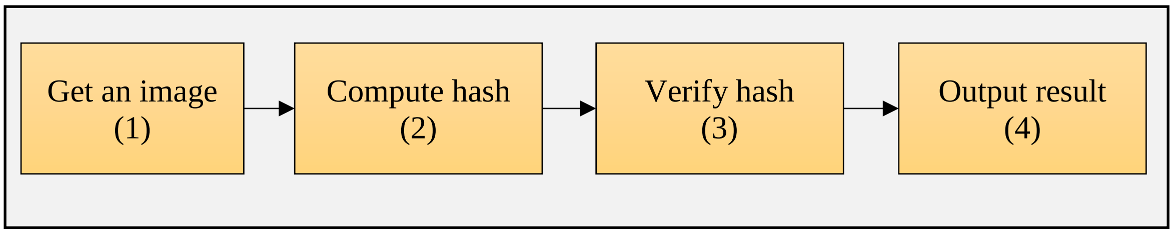


Figure 5 Integrity Verification Mechanism inside the IIV component

The first stage of the integrity verification mechanism is to get the image from the incoming request (1). Next, the mechanism computes a 256-byte hash for the image (2). After that, the verify hash block compares the calculated hash against a stored hash from the *hash_trusted_list* file (3). Finally, the output result block evaluates the result and based on that, returns an appropriate response to the requester (4).

The *hash_trusted_list* file, in trusted memory, contains a well-known list of previously computed image hashes. This implies that if a new image or a modification to an existing one is required, the IIV library must be updated to include the new measurement.

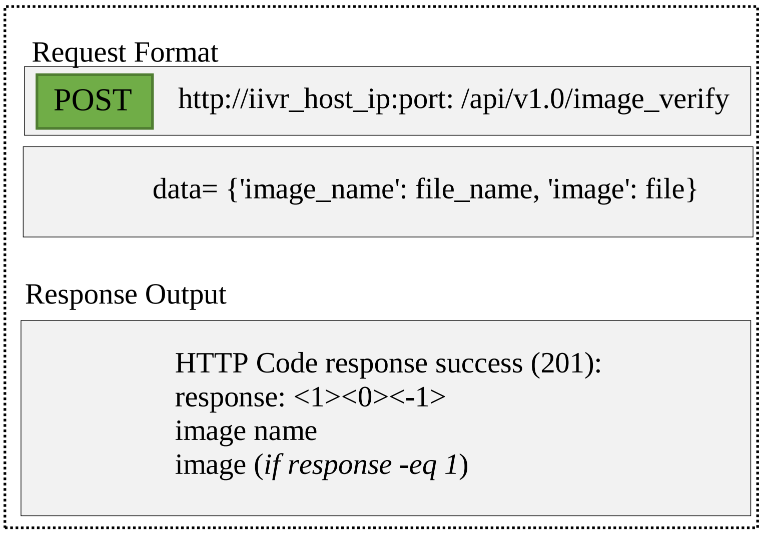
3.1.3 Image Integrity Verifier Implementation

The IIV is implemented as an SGX library named *libiivr*. In order to see the functionality of the integrity verification mechanism, the library is embedded in an application from which its service is requested. The application is deployed as a python flask app that embeds and calls the *libiivr* library and exposes a JSON REST API interface called *image verify*. Moreover, external entities invoke this *image verify* API to get images verified.

3.1.3.1 Image Integrity Verifier Main Application

As stated above, this application is responsible for providing an interface to the requester external entities and to call the services provided by the *libiivr* library. This application can be initialized in two different modes. First, the **complete mode** requires a list of hashes previously computed, to be passed. That list, is a file (semicolon separated values) with lines containing the exact name of the image followed by its corresponding measurement. The different hashes in the list are well-known measurements collected by a trusted administrator, who is also responsible for the compilation of the integrity verification mechanism. The second mode (**fast mode**) does not compile the entire *libiivr* library as in the case of the complete mode. Similar to the complete mode, in the fast mode, a hash list file needs to be provided. However, this file has to be a hash list previously sealed by the integrity mechanism. This indicates that the same material used in the key that seals the enclave is utilized in the derivation of the key that seals the hash list file (*hash_trusted_list).*

The main application hosts a JSON REST API, to validate the integrity of desired images. This is implemented in the API *image verifier*. A caller uses the POST HTTP method to invoke the *image verifi*er API, as illustrated in Figure 6. The body of the request includes the image name (*file_name*) and the contents of the image (*file*).



Request Format

POST http://iivr_host_ip:port: /api/v1.0/image_verify

data= {'image_name': file_name, 'image': file}

Response Output

HTTP Code response success (201):
response: <1><0><-1>
image name
image (*if response -eq 1*)

Figure 6 image verifier rest API invocation and response.

As a response, the API returns an HTTP code, indicating the results of the HTTP request:

- Result 1: success, the result of the integrity verification mechanism is returned. The the response returns TRUE (1) and the image name.
- Result 0: (Image not valid); the response includes the result and the image name.
- Result -1 (image not supported); the response includes the result and the image name.

The first task of the *image verify* API is to decode the binary content of the image from the JSON request. The recovered image is stored in a temporal location for its processing by the IIV. Next, the main application invokes a python *SGX_wrapper* library, which is a wrapper around the *libiivr* library. The *SGX_wrapper* library, is an interface created to be able to make use of the *libiivr* library created in C in the python flask microframework. It is implemented using the python extension Cython [2], which allows to combine C code into the python environment.

3.1.3.2 Initialization of the Image Integrity Mechanism

Depending on the chosen initialization mode, it is mandatory to provide a hash list and the corresponding signature (*IIV.signed.so*) required to initialized the enclave. This signature file must be kept secure by the administrator in a tamper-resistance place (e.g., TPM). The function *SGX*_init, defined in the untrusted part of *libiivr* library, is the responsible for the initialization of the IIV mechanism.

3.1.3.3 Image Integrity Verifier Mechanism

As illustrated in Figure 5, the IIV mechanism follows a sequential series of steps. The *image_verify* API requests the image verification by providing the image that was received from the caller to the IIVR mechanism. This image is passed by indirectly calling the SGX untrusted function *SGX_IIM* that takes the image name and the location or path of the image.

Once the IIV mechanism gets the image and its name, it proceeds to search if the image name is in the *hash_trusted_list*. If the image name is in the lists, the mechanism extracts the hash from the list and start the process of computing a fresh hash for that image. If the image name is not found, the function *SGX_IIM* returns with result -1 (image not supported).

To calculate the fresh hash of the incoming image, the IIV mechanism makes use of an outside enclave call (OCALL) to obtain the 256-byte measurement. The rationale for implementing this computation outside the enclave is for performance reason because of the high complexity when it comes to transfer large size images to trusted memory. The function *ocall_sha256_file* is the responsible for such computation in untrusted memory.

With the fresh hash value, the IIV proceeds to verify the integrity of the image by comparing both hashes. If both hashes match, the verification will return a 1 (Image OK), otherwise, a 0 (Image not OK) will be returned. The image verifier API receives the result of the IIV mechanism and proceeds to construct the final output to be given to the API caller. If the result is 1, the response includes the binary of the Image; in any of the other cases (0 or -1) the response only includes the result of the verification.

3.1.3.4 Summary of final flow implementation

The final flow of the IIV implementation is summarized in Figure 7. A privileged user runs the main application in the complete mode, providing a hash list of all images intended for instantiation (1). The main application invokes the python wrapper library to initialize the IIV mechanism, passing the plain hash file (2). The python wrapper on behalf of the main application invokes the function *SGX_init*, that initializes the SGX enclave and passes the plain hash list to the enclave for the corresponding sealing (3) and (4). The data sealing involves copying the content of the plain hash list to the *hash_trusted_list*, which is a file encrypted with a sealing key derived from the CPU. For I/O file manipulation, the trusted SGX protected file system libraries [3] are used. These libraries provide a set of I/O file handling functions similar as the ones provided by the *stdio* C library. To be able to incorporate these libraries in the *libiivr* library code, some requirements are necessary to be met as explain in [4]. Once the file has been created and sealed, the initialization of the IIV mechanism is successfully completed. If the main application must be restarted, it can be later re-initialized in fast mode, assuming that the *hash_trusted_list* is in place.

Following initialization, the IIV mechanism can accept incoming requests from external callers (5). The main application receives the HTTP POST requests invoked from its exposed rest API. This API receives as parameters the supplied image name and image file which is desired to be verified. Next, the python wrapper library is invoked to validate the integrity of the supplied image, by calling the function *SGX_IIM* (6). The untrusted part of the *libiivr* calls the method *ecall_ImagVerify* to delegate the verification process to the IIV mechanism (7). At this point the IIV mechanism is ready to execute the integrity verification of the image. To do so, the name of the image is first searched in the *hash_trusted_list* file and if found (9), the corresponding hash measurement is saved in a temporary variable for later comparison. If the IIV mechanism does not find the image name in the file, the mechanism returns a result of -1, indicating that the image is not supported for instantiation.

In case the hash for the supplied image is found in the sealed list, the IIV mechanism proceeds with the computation of a new fresh hash, by calling the untrusted function *ocall_sha256_file*, which makes use of the *openssl/sha* library (9). Once the IIV mechanism has obtained both hashes, the comparison takes places and the result of the verification is returned by the *ecall_Imagverify* function (10). If the result is successful a 1 is return, otherwise a 0 will be returned. Finally, the response is properly wrapped, by the main application, and returned to the caller.

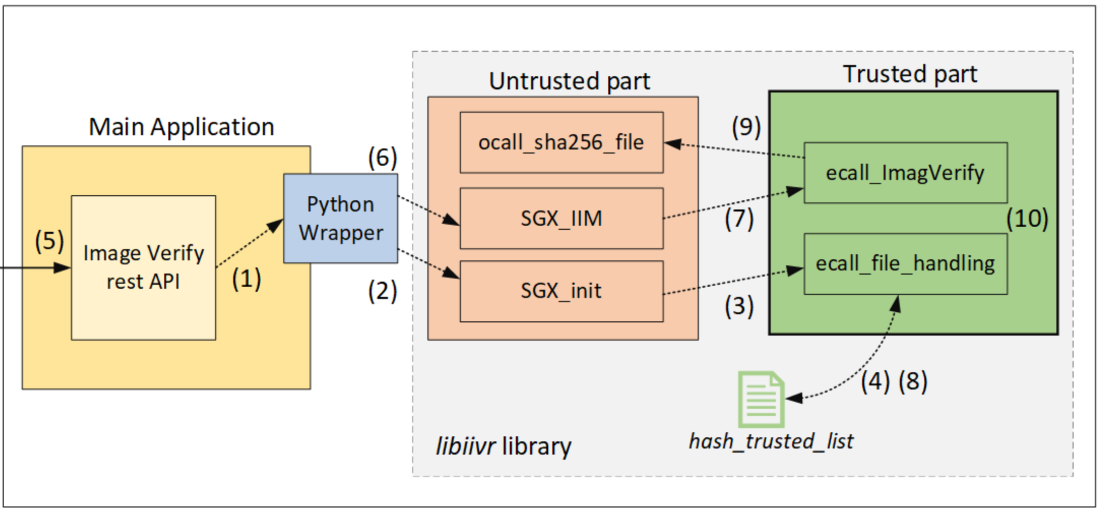


Figure 7 Summary of final flow implementation.

3.1.3.5 Limitations of the Proposed Solution.

The current implementation has some limitations, summarized as follows:

1. The IIV mechanism is attested locally instead of remotely. However, the mechanism can be extended to support it. The respective module can be integrated between the main application and Python wrapper, without altering the core functionality of the verification process.

2. The supplied images, are docker images, which binary files are compressed in a tar archive file. The compressed file, has to be named after the name of the image, otherwise, the verification process fails for that particular image. Once the tar file has been validated, the caller can instantiate it by simply execute the *docker load image.tar* command.

3. The sizes of the supplied images are limited to the available memory. The API in its data parameters expects to get the entire image 64byte-encoded, which sets a constraint in regard to the supplied image size and available memory.

4. To limit the number of ocalls to the minimum possible, it was decided to compute the fresh hash measurement in the untrusted part of the *libiivr* library. It is possible, however, to perform the entire computation in trusted memory, by splitting the image file in blocks of data. Every block is read in an ocall and consumed internally using the trusted crypto SGX library (*sgx_tcrypto*). This means that the total number of required ocalls will depend on both the size of the block chosen and the image size.

3.2 Crypto Engine

This subsection describes the functionality, design and implementation of the Crypto Engine, one of the security enablers of the COLA security architecture. The Crypto Engine aims to provide a set of cryptographic material and algorithms to enforce the security of the communication between the components of the MiCADO system.

3.2.1 Crypto Engine Functionality

As mentioned above, the Crypto Engine is responsible for the generation of cryptographic material and operations over the flowing data, as shown in Figure 8. It is designed and implemented as a microservice, providing the following functionalities [1]

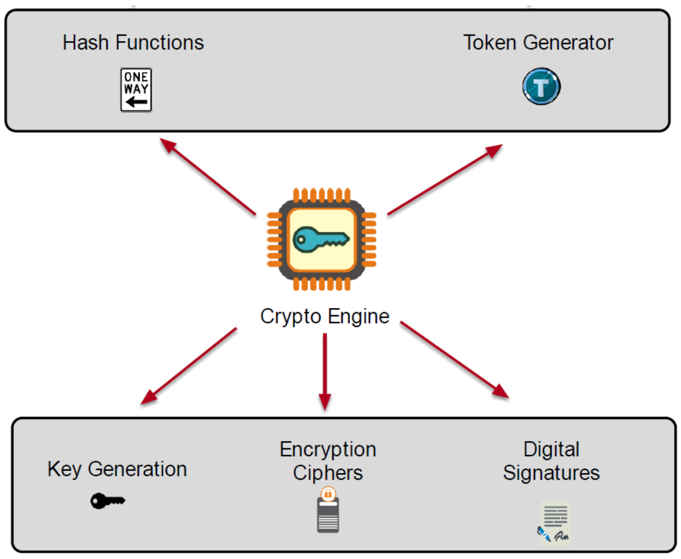


Figure 8 Crypto Engine Functionality

1. **Key Generation Orchestrator**: This function is responsible for the generation of both cryptographic symmetric and asymmetric keys. It takes as secure input parameters the size of the key $'\lambda'$ and the type of the key (asymmetric: $'asym'$ or symmetric: $'sym'$). This function will return the symmetric secret key $'k'$ or the pair of asymmetric keys, $^{p_k}/_{s_k}$ depending on the request being made.
2. **Symmetric Cipher Suite**: This function contains a collection of symmetric encryption algorithms to perform encryption/decryption operations. The symmetric algorithms will be supported in a variety of flavours; combining them with different encryption modes and key sizes. The supplied input parameters are verified by the Crypto Engine to make sure they meet the configured security policies. The definition of the involved symmetric operations is as follows:

$\boldsymbol{Definition\ 1}$ $(private - Key\ Encryption)$: 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 $\mathrm{K} \in \mathrm{K}$, where K is the available message space. The corresponding symmetric decryption operation is denoted by $m = Dec(\mathrm{K}, c)$.

3. **Asymmetric Ciphers Suite**: This function provides a library to perform asymmetric encryption/decryption operations. By default, the asymmetric Rivest-Shamir-Adleman (RSA) algorithm is used as the asymmetric encryption scheme. Similar to the Symmetric Cipher suite, the asymmetric algorithm supports different key sizes, depending on the security configuration of the Crypto Engine. The definition of asymmetric operations is specified as follows:

$\boldsymbol{Definition\ 2}$ $(Public - Key\ Encryption)$: We denote by $^{p_k}/_{s_k}$ a public/private key pair for an asymmetric encryption scheme. Encryption of a message m under the public key p_k is denoted by $c \leftarrow E_{pk}(\mathrm{m})$. While the corresponding decryption

$\boldsymbol{Definition\ 3}(Digital\ Signature)$: A digital signature over a message m signed with a private key s_k, is denoted by $\sigma = sign_{sk}(m)$. While the corresponding verification using a public key p_k over the signature σ, as $b = Verify_{pk}(m, \sigma)$ which equals to 1 if the signature is valid and 0 otherwise.

4. **Digital Signature:** a digital signature is an asymmetric encryption algorithm used to verify the integrity of a message and the actual identity of a sender. The signing of a message and the verification of a signature is defined as follows:

5. **Cryptographic Hash Functions:** This function provides a one-way fixed length compression of arbitrary-length messages. A cryptographic hash function contains special features that make it suitable for use in specific communication protocols. A hash function over a message m is denoted by $h_m = H(m)$.

   The requirement of a good secure cryptographic hash function requires ease in its computation but hard computation if the operation is reversed from a resultant hash. The result of the hash operation is known as a digest, and many are the algorithms that can be used for such computation. The Crypto Engine supports a variety of hash algorithms.

6. **Message Authentication Code (MAC)**: A MAC is a special type of hash function that uses a symmetric key to produce a fingerprint that is used to exchange messages in order to provide security integrity guarantees. A MAC of a message m with a secret key K is denoted by $\mu = MAC(K, m)$.

7. **Token Generator**: This function is responsible for the generation of secure strong random numbers, with sufficient entropy. This function can be denoted by $\tau = CSPRN(n)$, which is a random binary sequence of n bits generated by a Cryptographically Secure Pseudo-Random Number Generator (CSPRN).

3.2.2 Functional and Security Requirements

This section enumerates the high-level functional requirements and corresponding security considerations implemented by the Crypto Engine. Additionally, a specification of the different APIs implementing the different functions or services of the Crypto Engine are specified to give an insight into the final design and implementation [2].

3.2.2.1 High-Level Functional Requirements

These functional requirements are based on the requirements towards cryptographic security of ***(a)*** the primitive operations performed by the Crypto Engine and ***(b)*** the cryptographic primitives produce by the Crypto Engine. The functional requirements are described as follows:

1. The Crypto Engine should perform symmetric encryption/decryption operations with keys that are at least 128-bit long.
2. The Crypto Engine should perform asymmetric encryption/decryption operation with keys that are at least 2048-bit long.
3. The Crypto Engine should only accept combinations of parameters for both symmetric and asymmetric encryption schemes that are aligned with the security configuration set by the Crypto Engine.
4. The provided hash functions must produce preimage-resistant results.

3.2.2.2 API specifications

Table 4 contains a sample description of the API requirements, implementing the main functionalities of the Crypto Engine.

Table 4 API specification, for main functionalities offered by the Crypto Engine.

API	**Description**
Generate public-private key pair	a. Input parameters: 1. Function invocation –genKey 2. Parameters [crypto library, key type, Encryption algorithm] b. Output 1. Tuple list <public key, private key> c. Comment: The choice of the crypto library could be pre-defined by the administrator in the crypto security policy.
Generate X.509 Certificate	a. Input parameters: 1. Function invocation – genCert 2. Parameters [crypto library, encryption algorithm, validity period, certificate authority, certificate storage location] b. Output 1. X509 certificate c. Comment The choice of the crypto library, validity period and certificate authority could be pre-defined by the administrator in the crypto security policy.

Encrypt Content using a symmetric cipher suite	a. Input 1. Struct <Plaintext message, Encryption Key> 2. Parameters [crypto library, encryption algorithm, encryption mode] b. Output 1. Tuple list <result, ciphertext message> c. Comment N/A
Decrypt content using asymmetric cipher suite	a. Input 1. Struct <Ciphertext message, Decryption Key> 2. Parameters [crypto library, encryption algorithm, encryption mode] b. Output 1. Tuple list <Plaintext message> c. Comment d. N/A

3.2.2.3 Main Interactions

The following two use cases describe the main aspects that include most of the services provided by the Crypto Engine. They **do not** represent a complete list of uses cases.

Use case I: *Generation of a public key cryptography key pair*
The administrator, via the Security Policy Manager, makes a request to the Crypto Engine for the generation of a public-private key pair, specifying the key size and the asymmetric algorithm to be employed. For security considerations the key size must be greater than 2048 bits and the supported algorithms should include RSA [3] and ECDSA [4]. This kind of request can be used before making a request for the generation of an X.509 certificate or to request the encryption or decryption of a particular message.

Use case II: *Creation of an X.509 certificate*
The administrator, via the Security Policy Manager, makes a request to the Crypto Engine to create an X.509 certificate by proving a proper private key (as defined in use case I), the algorithm the key was generated with, the validity of the certificate and the Certificate Authority (CA) that signs the certificate. The Crypto Engine, via its configuration, decides the final validity and the CA that signs the certificate. By default, the Crypto Engine acts as the CA, unless specified in the configuration file of the Crypto Engine. The hashing algorithm used in the digital signature of the certificate is a default parameter configurable in the Crypto Engine configuration file.

3.2.3 Design and Implementation

The Crypto Engine is designed as a microservice using the Python microframework Flask. Functionality provided by the Crypto Engine is exposed as an API that takes some input parameters and returns the requested cryptographic information. Those input parameters are validated before any computation against the Crypto Engine's configuration file, in order to verify they are compliant with the security policies defined in the engine. The python cryptography module [5] (with the Open SSL library as the cryptographic backend) is used for provisioning cryptographic material and the execution of the cryptographic operations. This choice is motivated by the availability of safe recipes and low-level cryptography primitives.

3.2.3.1 Random Number Generator: genToken API

The Crypto Engine requires the generation of strong random numbers with sufficient entropy to be used as nonces, tokens and secret keys. This is achieved using the Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) functionality available in Python modules *os, secrets* and *uuid*.

The os module [4]
This module generates cryptographically secure random bytes with OS-specific randomness source, derived from the file */dev/urandom* (Unix) collected from device drivers and other sources. The function *os.urandom* is used for the generation of the random numbers, with the size number specified as the only input argument, no manual seeding is required.

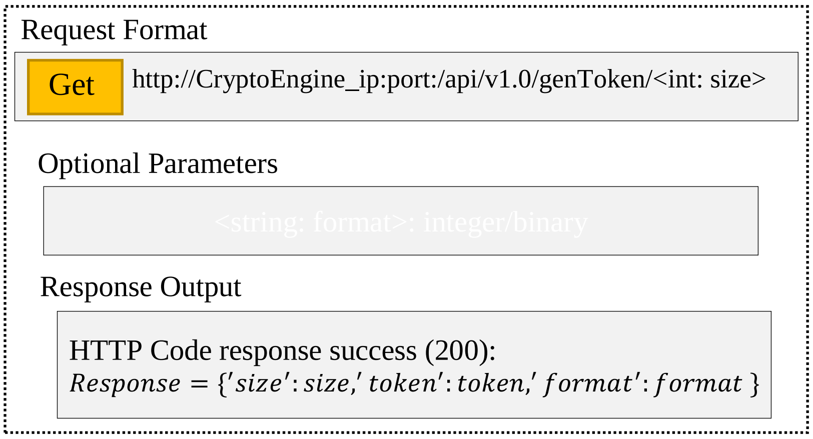
The secret module
New in Python v3.6, the secret module provides a user-friendly interface for the generation of strong random numbers and is suggested as the de facto module for secure random number generation. This module is a wrapper around the *os.urandom* method. In the secret module, tokens are generated passing the desired token size in bytes (default 16 bytes) using the functions *secrets.token_bytes* or *secrets.token_hex*.

The uuid module

Another option for generating random numbers as integers, bytes or hexadecimal objects is to use the function uuid4 from the uuid module. The disadvantage is that only 16-byte random numbers can be generated. There are other flavours of functions in the uuid library: *uuid1*, *uuid3* and *uuid4*. However, none of these meet the definition of randomness, as they take some form of input (seeding). uuid1 uses the machine's host ID and the current time, while uuid3 and uuid5 are based on the SHA-1 and MD5 hash respectively of both a namespace identifier and a name.

For the implementation of the genToken API, the os module is selected with a wrapper function for the generation of a token as an integer from the byte object returned. The function that takes care of the token generation is named after the genToken API and takes as an input argument the size (in bits) of the generated random number.

Figure 9, shows the structure of the genToken API request and response.


Request Format

Get http://CryptoEngine_ip:port:/api/v1.0/genToken/<int: size>

Optional Parameters

<string: format>: integer/binary

Response Output

HTTP Code response success (200):
$Response = \{'size': size, 'token': token, 'format': format\}$

Figure 9 genToken API format request

The genToken API expects to get the size of the token as a mandatory input and the format of the token (binary: as a sequence of bytes; or as an integer computed from the binary representation in big endian as default) as an optional parameter. The default format value is set to *binary*. If the requests returns with an HTTP code 200, the 'size' in bytes and the token in the chosen format are returned. Any other different HTTP code, means the request failed.

3.2.3.2 Generation of a public-private key pair: genKey API

To generate a public-private key pair, a GET request needs to be made specifying the size of the keys and optionally their desired encoding format. The default encoding configuration for the private key is PEM and SSH for the public key. The Crypto Engine checks the request, and verifies that the supplied parameters are aligned with its secure configuration (e.g., requests with keys sizes' < 2048 will not be processed). A processed request outputs a 200 HTTP code response, with the result of the operation being '1' if successful and '0', otherwise. If the result is '1' then the key pair is returned. However, if the result is '0', a status is returned as a way to inform the caller of the reason why the operation could not be completed.

The function KeyGenPair is responsible for the generation of the public-private key pair. This function takes as input arguments, the desired algorithm, the size of the keys and the encoding of every key. As a response, it returns a tuple consisting of the result of the operation; the private and public key; and the status of the operation. Internally, the keyGenPair function verifies the supplied parameters, returning immediately if one parameter is not compliant with the security of the engine.

Error! Reference source not found. displays the format of the request and the expected response of the genKey API. In case the operation returns a valid result ('1'), the private-public key pair are returned as a byte string base64 encoded. Table 5 shows some parameter requirements that need to be consider before using the genKeyAPI.

Request Format

```
Get   http://CryptoEngine_ip:port:/api/v1.0/genKey<string:algorithm>/<int:Size
```

Optional Parameters

```
<string:Eprivate>/<string:Epublic>
```

Response Output

```
HTTP Code response success (200):
 Response= {'result': 1, 'keypair':
                    {'private_key': {'key':key, 'encoding':enc },
                      'public_key': {'key':key, 'encoding ':enc }
             }
Response= {'result': 0, 'status':status}
```

Figure 10 genKey API format request

Table 5 Parameter requirements of genKey API usage

Parameter	Requirement
Algorithm	Choose any of the supported algortihms [RSA, ECDSA].
Key size	Keys should be at least 2048-bit
Key	Keys must be encoded as PEM or DER and should be base64 encoded before passing them to the Crypto Engine.

3.2.3.2.1 API error status

The API returns the following error statuses:

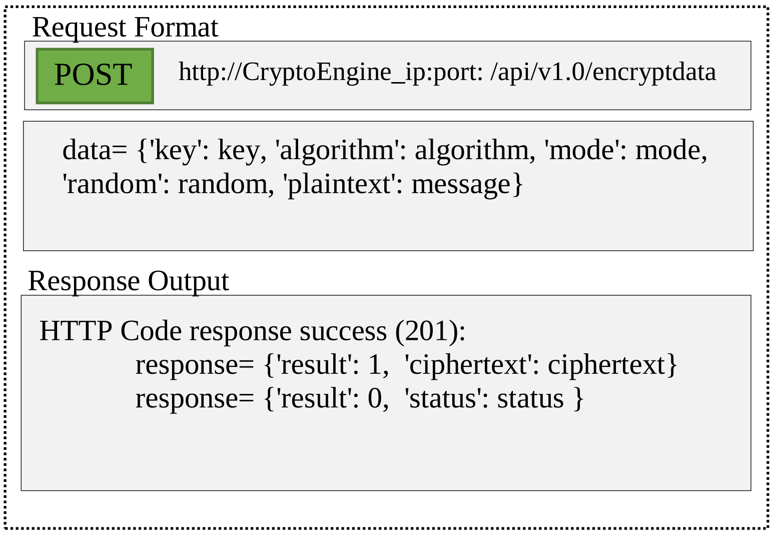
- *key-pair format not supported* status - if any of the key pairs provided is in an encoding format not supported by the Crypto Engine;
- *algorithm not supported* status, if the algorithm chosen is not any of the algorithms configured in the configuration file;
- *key size not supported* status, if the length in bits of any key is not compliant with the security policies of the engine.

3.2.3.3 Symmetric Encryption: encryptdata & decryptdata API

The Crypto Engine provides the service to encrypt messages and decrypt cipher texts using a secret private key following some security considerations. The Security Policy Manager - via an HTTP POST request - makes use of these symmetric services via invocation of the encryptdata and decryptdata API accordingly. In case the encryptdata API is invoked, the body of the request must contain the encryption algorithm, a mode, a random number with the same size as the encryption algorithm’s block size and the plaintext to be encrypted (base64 encoded). In case the decryptdata API is invoked, the body of the request must contain the decryption algorithm, the same mode and random number used in the encryption operation, the same secret key and the ciphertext (base64 encoded).

The encryption and decryption operations are executed by the functions **EncryptData** and **DecryptData**, respectively. The EncryptData function takes as input arguments the ciphertext, the secret key, the algorithm, mode, and a random number. Similar to the EncryptData, the DecryptData function takes the same arguments except on the plaintext, which is replaced by the corresponding ciphertext. Both functions respond, returning a tuple consisting of the result of the operation, the plaintext/ciphertext and the status. The security policies configured in the Crypto Engine's config file are enforced via internal functions that make sure the supplied parameters meet the security considerations.

Figure 11 shows the format request and response for the encryptdata API, while the decryptdata API format is shown in Figure 12. For the correct processing of the request, it is necessary that both plaintext and ciphertext be base64 encoded before invoking the respective API.
Table 6, shows a summary of both APIs requirements. It is important to mention that the Crypto Engine, ensures that messages get encrypted with the appropriate padding, i.e. padding the message so its size is a multiple of the algorithm's block size.


Request Format

POST http://CryptoEngine_ip:port: /api/v1.0/encryptdata

```
data= {'key': key, 'algorithm': algorithm, 'mode': mode,
'random': random, 'plaintext': message}
```

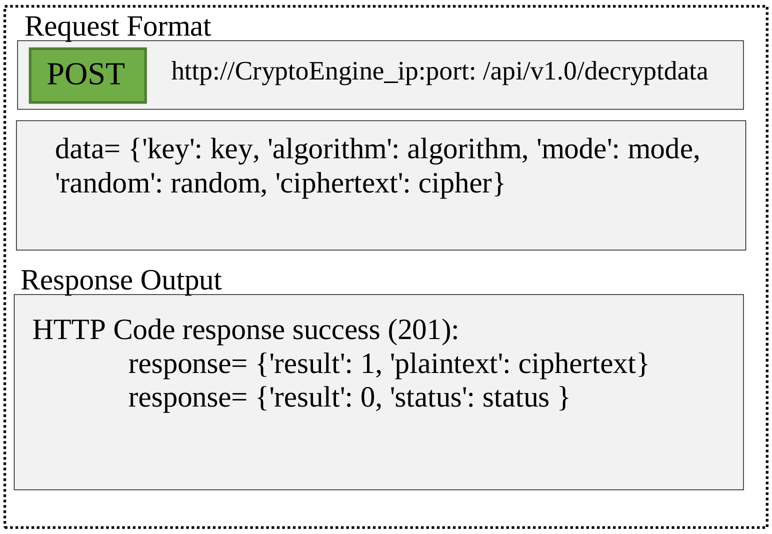
Response Output

```
HTTP Code response success (201):
        response= {'result': 1,  'ciphertext': ciphertext}
        response= {'result': 0,  'status': status }
```

Figure 11 encryptdata API format request

3.2.3.3.1 API error status

Most of the security verification of the supplied parameters in both encryption and decryption operations are performed by the internal function symmetric_check. This function can abort the operation and return error status due to unsupported algorithms, key sizes, modes and random numbers which sizes are not the same as the block sizes of the selected cryptographic algorithms. Finally, when the decryption API is invoked, the status wrong padding could be return when unpadding a message after using a random number or key different from the one used in the encryption operation.



Request Format

POST http://CryptoEngine_ip:port: /api/v1.0/decryptdata

```
data= {'key': key, 'algorithm': algorithm, 'mode': mode,
'random': random, 'ciphertext': cipher}
```

Response Output

```
HTTP Code response success (201):
        response= {'result': 1, 'plaintext': ciphertext}
        response= {'result': 0, 'status': status }
```

Figure 12 decryptdata request/response specification

Table 6 Parameter requirements for symmetric APIs usage

Parameter	Requirements
Secret key	The size of the key must be greater than 128 bits. Supported key sizes by default [192, 256] bits for AES and [192] bits for 3DES
Algorithm	Choose any of the configured algorithms. Supported by default: 3DES and AES
Mode	Choose any of the configured modes. Supported by default: CBC, CTR, OFB, CFB
Random Number	Depending on the mode selected, it is used as a nonce or as an initialization vector. The size of this random number has to be equal to the block size used by the selected algorithm.
Plain/cipher text	This text needs to be base64 encoded. The response from the API will also be base64 encoded.

3.2.3.4 Asymmetric Encryption: rsaencryptdata & rsadecryptdata APIs

The Crypto Engine provides functionality for asymmetric encryption and decryption using the RSA algorithm with some security considerations. Both asymmetric operations are implemented by the APIs rsaencryptdata and rsadecryptdata accordingly. The Crypto Engine makes sure only keys with sizes greater than 2048-bits are accepted in the involved asymmetric operations.

The encryption operation is performed by the function RSA_EncryptData that takes as input arguments the message to be encrypted, the public encryption key and the asymmetric algorithm (RSA). Similarly, the decryption operation is performed by the function RSA_DecryptData, that takes as input arguments the cipher text, the private decryption key and the algorithm (RSA). These two functions invoke the services of special methods that make sure the security policies defined in the Crypto Engine's config file are enforced. As a result, both functions return the result of the operation. If successful ('1') the function returns the cipher/plaintext. Otherwise, the function returns the status with the reason why the operation could not be completed if the result is '0'.

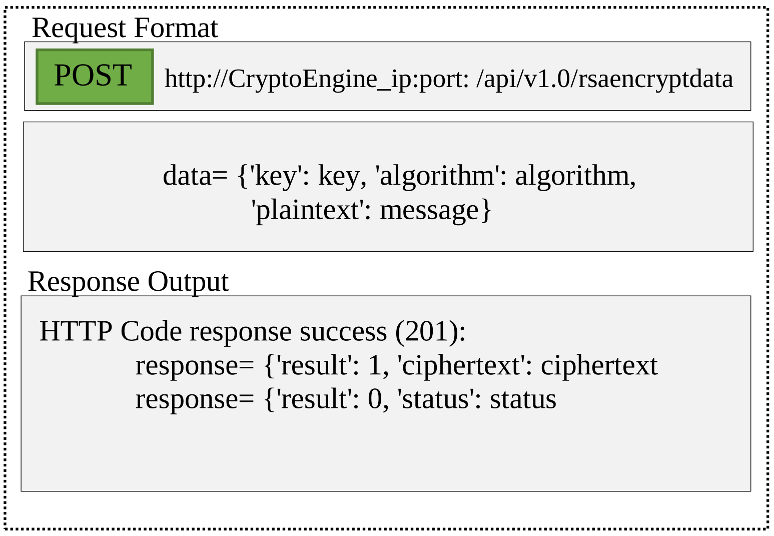
Figure 13 shows the request and response format for the invocation of the rsaencryptdata API, while Figure 14 shows it for the rsadecryptdata API. The crypto Engine supports by default RSA key sizes of 2048 and 4096 bits. Smaller sizes are discarded and the operation aborted. Table 7 show the API parameter requirements.

3.2.3.4.1 API error status

The possible reasons that can be returned in case any of the above functions cannot be completed are due to an algorithm not supported or to a key size lower than 2048 bits. Additionally, if the key provided cannot be loaded by the Engine due to a variety of reasons (e.g., wrong key or encoding/format not supported) the reason, key could not be loaded would be returned.

Table 7 Parameter requirements for Asymmetric API usage

Parameter	Requirement
Algorithm	RSA should be the supplied algorithm. The algorithm is specified for inclusion of other algorithms in future version of the Crypto Engine.
Key	The supplied key must be base64 encoded.
Key Size	Key sizes should be >2048 bits. By default, the Crypto Engine supports keys with sizes: [2048,4096] bits
Plaintext/ciphertext	Both texts must be base64 encoded.


Request Format

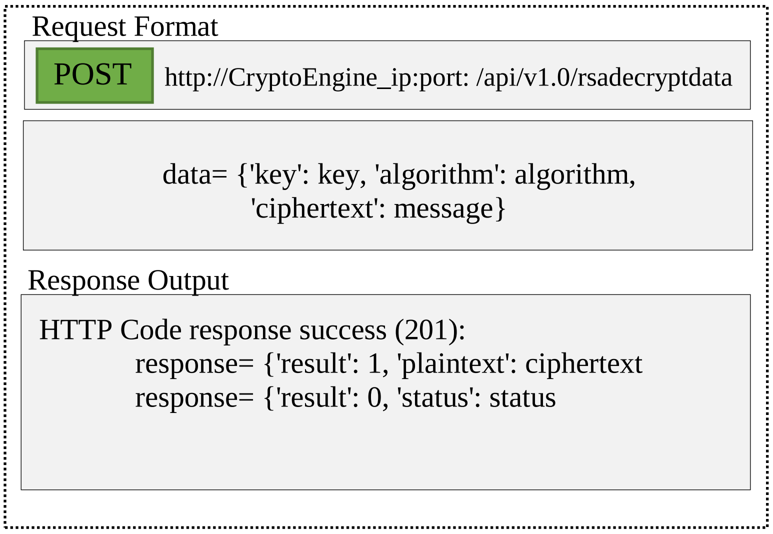
POST http://CryptoEngine_ip:port: /api/v1.0/rsaencryptdata

```
data= {'key': key, 'algorithm': algorithm,
        'plaintext': message}
```

Response Output

```
HTTP Code response success (201):
        response= {'result': 1, 'ciphertext': ciphertext
        response= {'result': 0, 'status': status
```

Figure 13 rsaencryptdata request/response specification



Request Format

POST http://CryptoEngine_ip:port: /api/v1.0/rsadecryptdata

```
data= {'key': key, 'algorithm': algorithm,
       'ciphertext': message}
```

Response Output

```
HTTP Code response success (201):
        response= {'result': 1, 'plaintext': ciphertext
        response= {'result': 0, 'status': status
```

Figure 14 rsadecryptdata request/response specification

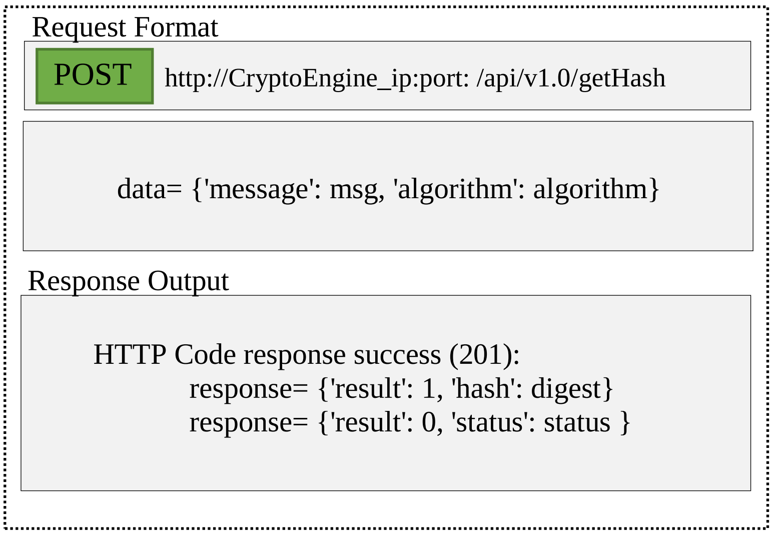
3.2.3.5 Hashing Algorithms: getHash API

The Crypto Engine provides the getHash API for the computation of a fixed-size bit string called hash (digest) from a block of data of any size using a one-way hashing algorithm. For security considerations the supported algorithms must have strong security properties, easy to compute but hard to reverse. However, weak hashing algorithms would still be considered, via configuration, for legacy applications only.

The hashing algorithms from the SHA-2 family are configured and supported in the Crypto Engine by default. If it is required to configure any other hashing algorithm, then the administrator can do it by simply adding the desired algorithm to the HASHCONF parameter, in the configuration file.

The Crypto Engine function for computing hash messages is called hash_message. This function takes as input arguments the message (in bytes) and the name of the supplied algorithm. The Crypto Engine searches in its configuration file if the supplied algorithm is supported and aborts the operation if not match is retrieved. As a successful outcome the function returns a tuple containing the results of the operation, the hash message and a status. A result ‘True’ indicates that the operation could be completed and ‘False’ that it failed. If the operation is completed, the resultant hash and the status OK are returned, otherwise, just the status is returned describing the reason why the operation failed. The getHash API, formats the response obtained from the hash_message function and returns to the caller the result of the operation. The result is ‘1’ for completion returning the hash message or ‘0’ for failed with its corresponding error status

Figure 15 shows the request and response format for the invocation of the getHash invocation API. A caller makes use of the API via a HTTP POST request with body containing a message and the desired hashing algorithm. Table 8 shows the parameter requirements to consider when using the API.


Request Format

POST http://CryptoEngine_ip:port: /api/v1.0/getHash

```
data= {'message': msg, 'algorithm': algorithm}
```

Response Output

```
HTTP Code response success (201):
        response= {'result': 1, 'hash': digest}
        response= {'result': 0, 'status': status }
```

Figure 15 getHash request/response specification

3.2.3.5.1 API error status

The error returned due to an incomplete hash operation is unsupported algorithm. If this is the case, the caller must make sure to use one of the secure hashing algorithms provided by the Crypto Engine.

Table 8 Parameter requirement for getHash API usage

Parameter	Requirement
Algorithm	Choose any of the default hashing algorithms from SHA-2 family ['SHA224','SHA256','SHA384','SHA512']. If another protocol is needed the Crypto Engine’s admin must configure it in the configuration file.
Message	This message needs to be base64 encoded, before passing it to the engine.

3.2.3.6 Certificate Generation: the genCert API

One fundamental function provided by the Crypto Engine is the generation of X.509 certificates which are used to authenticate clients and servers. To create a certificate is necessary to take a series of sequential steps, yielding the desired cryptographic material. The required steps are the following:

1. Generation of a private/public key pair
2. Creation of a Certificate Signing request (CSR), signed with the private key generated in step 1.
3. A Certificate Authority(CA) validates that the requester owns the resource claimed.

4. A CA signs with its private key the CSR, identifying the requester's public key and his/her domain.
5. The requester gets the certificate and can start using it to configure any server.

The genCert API, provides 3 functions related to the generation of certificates. First, it is possible to just generate a CSR. This could be the case when a caller wants a specific CA to sign his/her certificate. Second, the API, by default, generates an X.509 Certificate signed by the Crypto Engine as the CA. Third, sometimes it is necessary to generate self-signed certificates for testing-purposes only, this can also be possible via the genCert API.

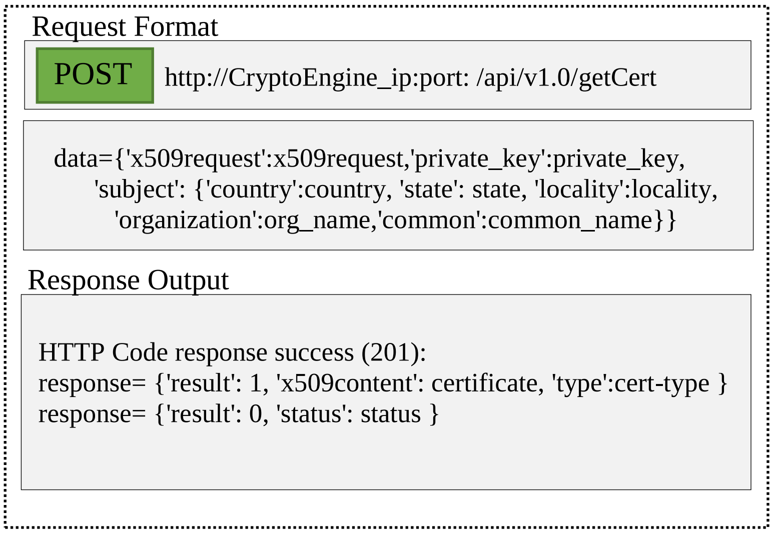
The function that does all the certificate work in the genAPI is called the gencert_content, which takes as input arguments the subject requesting the X.509 material, the request type, and the caller's private key. As a response the function returns a tuple with the result of the operation, the X.509 material and a status. If the operation is successfully completed (True), the X.509 material with the status OK is generated, otherwise, the function returns False with the status of the request, providing a reason to determine the cause of the failure. For the subject information, there are some mandatory fields that should be supplied by the caller (described below).

As explained above there are three types of requests that are supported: 'SELF' for self-signed certificates; 'SIGNED' for certificates signed by the Crypto Engine as CA; and 'CSR' for generation of X.509 certificate requests to be signed by a third-party CA. When certificates are signed by the Crypto Engine three configuration parameters must be set, the validity period in days; the Crypto Engine's private key; and details of the Crypto Engine acting as the issuer entity.

The private key supplied by the caller can be encoded as binary DER or as ASCII PEM, any other encoding will cause the operation to be aborted. The encoding verification is performed by the inner key_loader function. Figure 11 shows the request/response format specification for the genCert invocation. The caller makes a HTTP POST request providing all the required parameters in the body of the request and ensures a valid X.509 request type is specified. Table 9 shows the parameter requirements to consider when using genCert API.

3.2.3.6.1 API error status

There are many reasons why certificate operations cannot be completed. If the private key supplied by the caller cannot be loaded, the error status 'key could not be loaded' would be returned, which indicates that either the key is not large enough or the encoding is not supported. If the X509 request type is not valid, the invalid type of certificate would be returned. When the Crypto Engine is chosen to be the CA, the error indicating that the engine cannot sign the request can be returned if the configuration parameters are not set or if there is a field not specified correctly. Finally, the status certificate could not be generated can be returned if the backend selected and configured in the Crypto Engine does not support the functions required to generate certificate material.



Request Format

POST http://CryptoEngine_ip:port: /api/v1.0/getCert

```
data={'x509request':x509request,'private_key':private_key,
    'subject': {'country':country, 'state': state, 'locality':locality,
        'organization':org_name,'common':common_name}}
```

Response Output

```
HTTP Code response success (201):
response= {'result': 1, 'x509content': certificate, 'type':cert-type }
response= {'result': 0, 'status': status }
```

Figure 16 genCert request/response specification

Table 9 Parameter requirements for genCert API usage

Parameter	Requirement
X509request	The type of the request must be [’CSR’,’SIGNED’,SELF’] default is set to SIGNED.
private_key	Must be encoded as PEM or DER and passed base64 encoded.
Subject	The details of the subject of the certificate. Mandatory fields: country, state, organization, locality, common

3.2.3.7 Digital Signatures: genSignature and veriSignature APIs

The Crypto Engine provides the genSignature API for the generation of a digital signature and the veriSignature API for its verification. A digital signature is used to verify the integrity of a message, guaranteeing that it has not been tampered with during transit. A digital signature is produced by hashing a desired message with a secure algorithm and encrypting its output with a private key. To verify the signature is necessary to decrypt it using the public key corresponding to the encryption key and to compare the hash obtained from the signature with a fresh hash computed from the original message.

To produce the signature, the Crypto Engine makes use of the default SHA256 algorithm for the computation of the digest. The Crypto Engine supports keys obtained from the RSA and ECDSA algorithms for both the generation and validation of signatures. In the case a key obtained from the RSA algorithm is selected, a proper padding scheme must be put into place. In the Crypto Engine, the Probabilistic Signature Scheme (PSS) is the default padding scheme used, as it is suggested for any new protocols or applications.

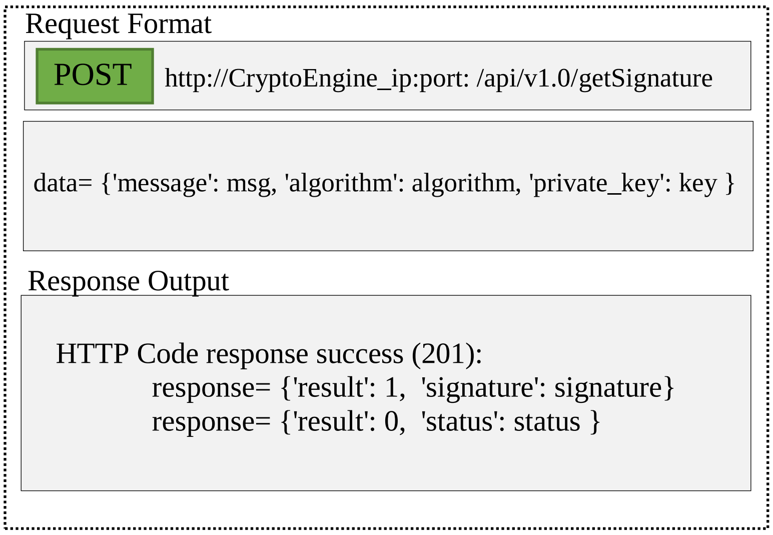
The function genDSA is the responsible for the generation of a digital signature. This function takes as input arguments a message, a private key and an algorithm. As a response the result of the operation is returned, which is True if completed or False otherwise. If the operation is successfully completed, then, the signature and the status OK are returned. On the other hand, if the operation is not completed, a status is returned to describe the reason of the failure.

The function veriDSA is the responsible for the verification of a digital signature. This function takes as input arguments the signature to verify, the message the signature is obtained from, the public key corresponding to the signing private key and the same key algorithm used. As a response, the function returns the result of the operation.

Figure 17 shows the request/response format for the getSignature API, while Figure 18 shows it for the veriSignature. Some security considerations should be met when constructing the body of the POST request for both APIs, Table 11 shows the parameter requirements.

Table 10 Parameter requirements for genSignature and veriSignature APIs

Parameter	Requirement
Message/Signature	Must be base64 encoded
Private/Public Key	Should be formatted with a valid encoding scheme (see genKey specs) and should be passed to the Crypto Engine base64 encoded
Algorithm	Choose any of the asymmetric encryption algorithm supported [RSA, ECDSA]


Request Format

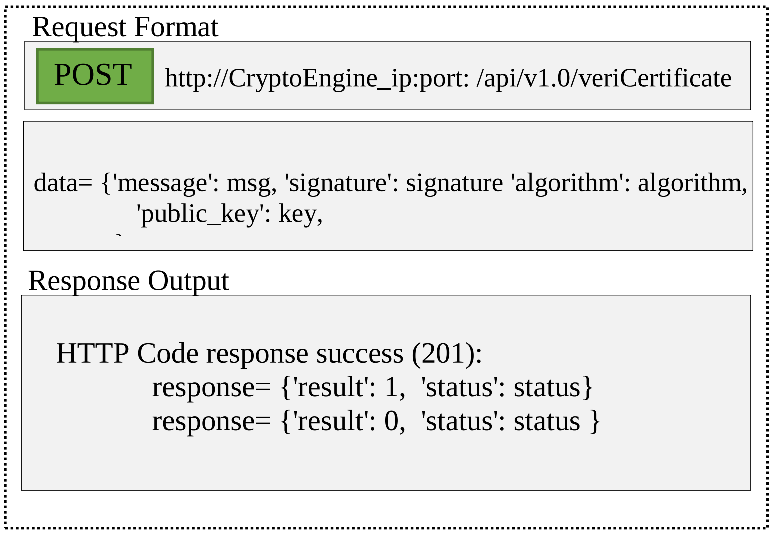
POST http://CryptoEngine_ip:port: /api/v1.0/getSignature

```
data= {'message': msg, 'algorithm': algorithm, 'private_key': key }
```

Response Output

```
HTTP Code response success (201):
        response= {'result': 1,  'signature': signature}
        response= {'result': 0,  'status': status }
```

Figure 17 genSignature request/response specification



Request Format

POST http://CryptoEngine_ip:port: /api/v1.0/veriCertificate

```
data= {'message': msg, 'signature': signature 'algorithm': algorithm,
            'public_key': key,
```

Response Output

```
HTTP Code response success (201):
            response= {'result': 1,  'status': status}
            response= {'result': 0,  'status': status }
```

Figure 18 veriSignature request/response specification

3.2.3.7.1 API error status

Both APIs can return the status algorithm not supported if the supplied algorithm is not any of the configured algorithms in the Crypto Engine config file. If any of the provided keys does not meet the security policies defined in the configuration file, then, the status key cannot be loaded is returned to the caller. Finally, if the validation of the signature is not successful, the verification API responds with a status verification failed.

3.2.3.8 Crypto Configuration Parameters.

The configuration file contains the global variables that define the overall security policies to be implemented by the Crypto Engine against the input parameters supplied by the different callers, especially the SPM. Table 11 shows the configuration parameters with their functions.

Table 11 Crypto Engine Configuration parameters

Parameter	Function
CONF_ASYMMETRIC	Defines the asymmetric algorithms, key sizes and encoding to be allowed.
CRYPTOCONF	Defines the symmetric algorithms, key sizes and modes to be supported.
ASYMCONF	Defines the asymmetric algorithms to be used in encryption/decryption operations
HASHCONF	Hashing algorithms supported
CA_PRIVATE_KEY_PATH	Location of Crypto Engine's private key
VALIDITY_PERIOD	Default validity duration of certificate
CA_ISSUER_CONF	Crypto Engine issuer information

3.3 Credential Store

3.3.1 *Credential Store Functionality*

This section aims to describe the core functionality of the Credential Store (CredStore) as a security component in the MiCADO architecture. The CredStore takes care of securely storing all types of sensitive information, which we will call infrastructure secrets or secrets, needed for running the MiCADO infrastructure. It protects infrastructure secrets by encrypting them and restricting access to them. The CredStore fulfils the following security requirements:

1. Infrastructure secrets must be stored in encrypted form;
2. Access to infrastructure secrets must be restricted to only Security Policy Manager (SPM) component in MiCADO. Other components are not allowed to access secrets stored in CredStore;
3. Secrets are only decrypted at the time of accessing.

3.3.2 *Terminology*

The CredStore implementation relies on the open source Hashicorp Vault [7] licensed under Mozilla Public License 2.0. As a consequence, we present a few terminologies related to Hashicorp Vault in the table below.

Terminology	Meaning
Vault or Hashicorp Vault	It is a tool for securely accessing secrets. It follows the client-server infrastructure.
Vault Server	It is the entity that interacts with the data storage and backends. In MiCADO, it is the Credential Store.
Vault Client	It is the entity that interacts with the vault server to access secrets. In MiCADO, it is the Security Policy Manager.
Secret	A piece of sensitive information that has a name and a value.
Encryption key	The key used for encrypting data.
Master key	The key used for encrypting encryption key. It could be recovered from a minimum number of unseal keys.
Unseal keys or shares	The keys used for unsealing. They are generated from master key.
Threshold	The minimum number of unseal keys required for reconstructing the master key.
#shares	The number of generated unseal keys from the master key.
Sealed state	The state of vault server in which vault server only knows the physical storage position of secrets but does not know how to decrypt them.
Sealing	The process to set the vault server into a sealed state.
Unsealing	The process of reconstructing the master key from unseal keys.

3.3.3 *Credential Store Interaction in MiCADO*

In MiCADO architecture, Credential Store (CredStore) component is designed as a Vault Server where Security Policy Manager (SPM) plays the role of Vault Client. Hence, SPM is the only component that directly interact with CredStore and all other components have to request secrets from SPM instead of direct interaction with CredStore. The CredStore can be used in the following two ways: (1) Those admins' users who have access to the Master Node through SSH to call Restful APIs to CredStore. The calls are to perform various functions such as insert, update, and delete secrets (2) In contrast to the admin interface, all other components that needed to interact with CredStore, has to communicate via SPM. E.g. Cloud Orchestrator (CO) component as an entity requests a secret from SPM. This interaction is illustrated in Figure 19. In this case, the CO request for the cloud user credentials, which are stored in CredStore, to SPM.

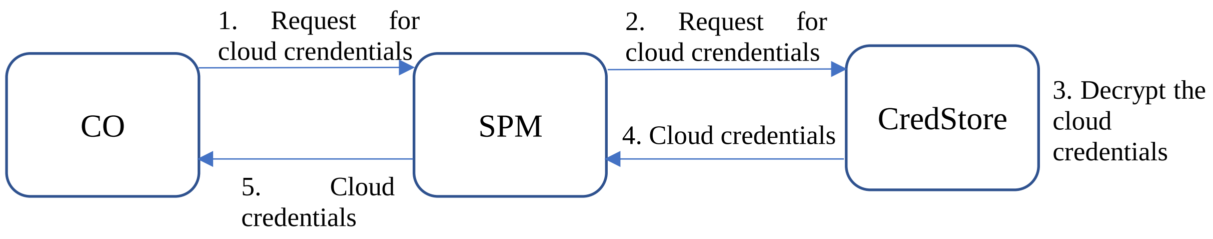

1. Request for cloud crendentials
2. Request for cloud crendentials
CO
SPM
CredStore
3. Decrypt the cloud credentials
4. Cloud credentials
5. Cloud credentials

Figure 19 Component interaction for the infrastructure secret request

The above-mentioned process applies to all other components of MiCADO to access infrastructure secrets.

3.3.4 *Credential Store Design and Implementation*

The CredStore is deployed as a Vault Server using Hashicorp Vault software. At first, we create a configuration file in which we could define type (e.g. file, Consul, etc.) and physical path of database backend, TLS/SSL enable/ disable option, TCP address to listen for API requests, log level, cache size, etc. After that, we deploy the Vault Server using the configuration file. It exposes Restful APIs for SPM use.

Apart from that, SPM plays the role of a Vault Client. It is implemented as a Python Flask web service that exposes Restful APIs for secret requests from other components. For its implementation, instead of interacting with CredStore through raw REST calls, we use a wrapping library, HashiCorp Vault API client for Python 2.7/3.x [8], i.e. HVAC.

3.3.4.1 Credential Store as Vault Server

The Credential Store is deployed as a Vault Server which is the main storage for all kind of infrastructure secrets. For the sake of simplicity, we configure the storage backend as a file stored inside CredStore component. In addition to that, based on current assumption that communication between components in the Master Node is secure, we disable TLS/SSL for communication between CredStore and SPM. Finally, CredStore is configured to listen on port 8200. All configurations are described in a file of HCL (HashiCorp Configuration Language) format [9]. Table 12 describes the above configuration in HCL.

```
{
    "backend": {"file": {"path": "/config/data"}},
    "listener": {"tcp": {"address": "0.0.0.0:8200", "tls_disable": 1}}
}
```

Table 12 Vault Server configuration in HCL format

Based on HCL file, the vault server configurations could be defined easily.

3.3.4.2 Security Policy Manager as Vault Client

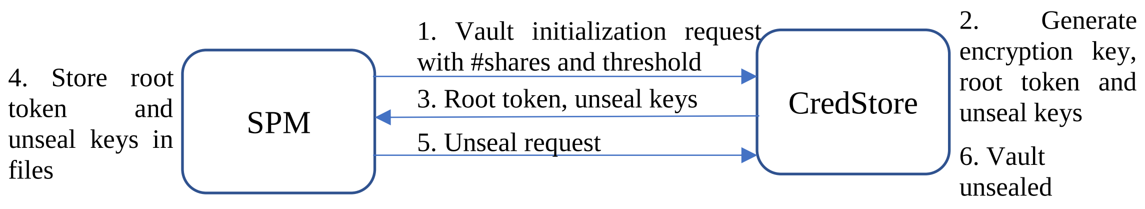
Security Policy Manager (SPM) plays the role of vault client. It is the only component in MiCADO master node that could access infrastructure secrets from CredStore. Any other component(s) would need to assess secrets through SPM as intermediary. SPM exposes APIs for bridging the calls to the Credential Manager:

- ***Secrets*** consists of APIs for inserting, accessing, updating and deleting secrets.

Apart from that, SPM keeps a token used for authenticating to CredStore and unseal key(s) for unsealing vault in CredStore. For the current version of implementation, the token is the root token without expiration.

3.3.4.3 Vault Initialization Mechanism

Prior to sending infrastructure secrets to CredStore for the very first time, SPM needs to send a request to initialize the vault in CredStore. Initialization is the process of setting up things for authentication and encryption. After initialization, SPM unseals the Vault so that upcoming secret management requests will be able to access it. Figure 20 depicts interaction between SPM and CredStore in that process.



4. Store root token and unseal keys in files
SPM
1. Vault initialization request with #shares and threshold
3. Root token, unseal keys
5. Unseal request
CredStore
2. Generate encryption key, root token and unseal keys
6. Vault unsealed

Figure 20 Vault Initialization Mechanism

More specifically, SPM sends the *number of shares* and *threshold value* to CredStore in the initialization request. Upon reception, CredStore generates an encryption key, a root token and a master key.

The encryption key would be used to encrypt secrets and the master key is used to encrypt the encryption key. By the end, ciphertext of the encryption key would be stored inside CredStore along with encrypted secrets. For the sake of security, the master key would be not stored anywhere. Instead, CredStore uses Shamir's Secret Sharing scheme to split the master key into multiple shares, i.e. unseal keys, in such a way that a minimum number of shares are required to re-construct the master key. Such minimum number is defined by *threshold value*.

The root token is a never-expiring token, generated at the time of vault initialisation. With the root token, the SPM is permitted to perform any operation in the vault of the CredStore.

The vault initialization is done automatically upon Security Policy Manager startup.

3.3.4.4 Infrastructure Secret Insertion Mechanism

The Admin user who has access to the Master Node can invoke API provided by SPM to insert infrastructure secrets into MiCADO. The secrets will be stored in CredStore.

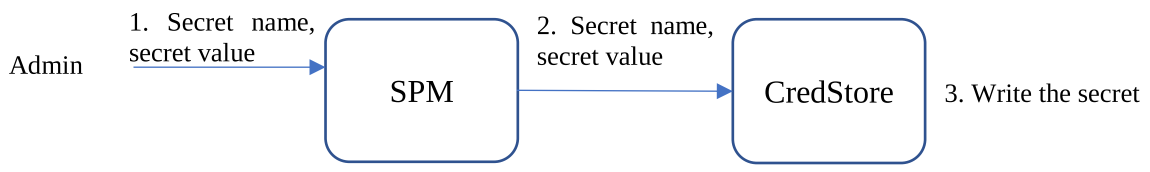
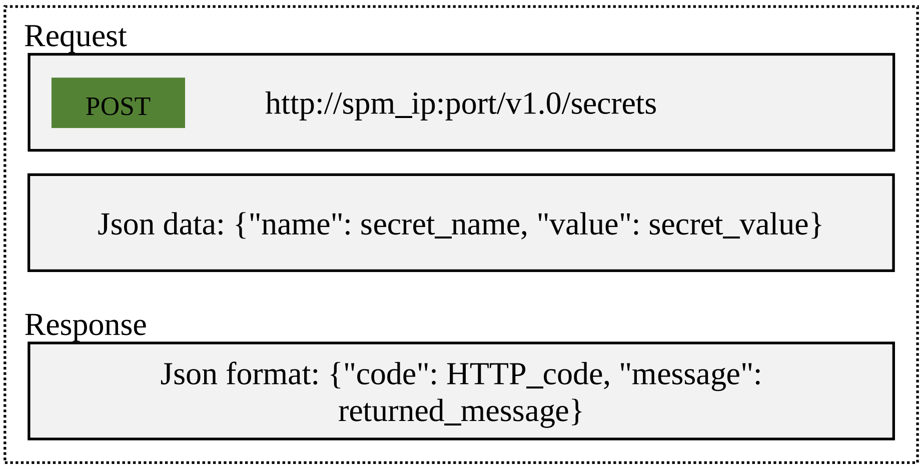


Figure 21 Secret Insertion Mechanism

The following API allows to write a secret to the initialized vault. If the secret already exists then it will be overwritten.



Request

POST http://spm_ip:port/v1.0/secrets

Json data: {"name": secret_name, "value": secret_value}

Response

Json format: {"code": HTTP_code, "message": returned_message}

Figure 22 Secret Insertion request and response

Request:

Name	In	Description
name	Json body	Name of infrastructure secret
value	Json body	Value of infrastructure secret

Example:

```
curl -H "Content-Type: application/json" -d
'{"name":"cloudsigma_username","value":"user1@mail.com"}' -X POST
http://127.0.0.1:5003/v1.0/secrets

curl -H "Content-Type: application/json" -d
'{"name":"cloudsigma_password","value":"1aB"}' -X POST
http://127.0.0.1:5003/v1.0/secrets
```

Response:

Name	In	Description
code	Json body	Status code
message	Json body	Returned message

Status codes:
Success:

- 201 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:
```
{
	"code": 201,
	"message": "Add/ Update secret successfully!"
}
```

3.3.4.5 Infrastructure Secret Retrieval/ Update/ Deletion Mechanism

MiCADO security components are designed to provide access to the CredStore only through the SPM. As a consequence, SPM stores necessary information for CredStore access, including token and unseal key(s).

Apart from that, SPM provides Restful APIs for other component(s) which needs to retrieve infrastructure secret(s) at running time. One example is Cloud Orchestrator which needs cloud user credentials for invoking Cloud Provider's APIs. The following describes APIs provided by SPM, their parameters and how to invoke them.

1. **Secret Retrieval Mechansim**

This mechanism illustrates how a secret will be retrieved. The mechanism consists of the interaction among three components i.e. CO, SPM and CredStore. The mechanism is started by CO which requires to get value of a secret.

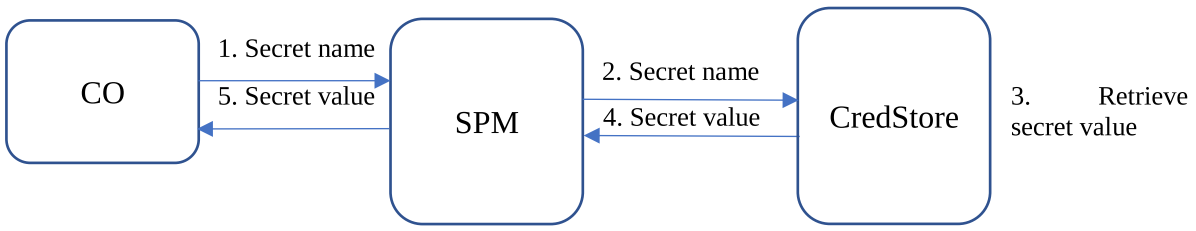
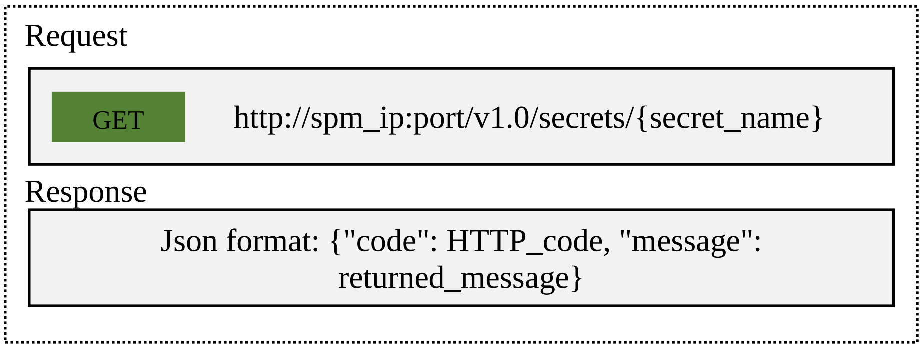


Figure 23 Secret Retrieval Mechanism

The following API allows to retrieve a secret value from CredStore.



Request

GET http://spm_ip:port/v1.0/secrets/{secret_name}

Response

Json format: {"code": HTTP_code, "message": returned_message}

Figure 24 Secret Retrieval request and response

Request:

Name	In	Description
secret_name	Path	Name of secret

Example:
```
curl -X GET http://127.0.0.1:5003/v1.0/secrets/cloudsigma_username
```

Response:

Name	In	Description
code	Json body	Status code
message	Json body	Returned message
data	Json object	
secret_value		Value of secret

Status codes:
Success:
- 200 - OK: Request was successful

Error:
- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 404 – Not found: Provided secret name does not exist
- 500 - Server error: Some error happened at server side

Example:
```
{
	"message": "Read secret successfully!",
	"code": 200,
	"data": {"secret_value": "user1@mail.com"},
}
```

2. Secret Deletion Mechanism

A secret should be removed from MiCADO when there is no longer need of it. In such case, the Admin user can invoke an API from SPM.

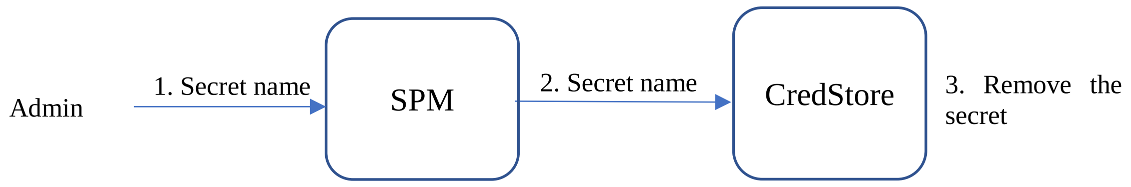
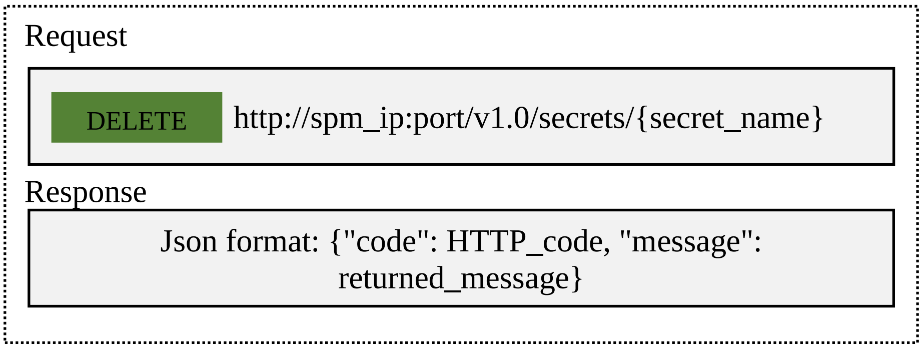


Figure 25 Secret Deletion Mechanism

The following API allows to remove a secret from CredStore.



Request

DELETE http://spm_ip:port/v1.0/secrets/{secret_name}

Response

Json format: {"code": HTTP_code, "message": returned_message}

Figure 26 Secret Deletion request and response

Request:

Name	In	Description
secret_name	Path	Name of secret

Example:

```
curl -X DELETE http://127.0.0.1:5003/v1.0/secrets/cloudsigma_username
curl -X DELETE http://127.0.0.1:5003/v1.0/secrets/cloudsigma_password
```

Response:

Name	In	Description
code	Json body	Status code
message	Json body	Returned message

Status codes:

Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:

```
{
        "code": 200,
        "message": "Deleted the secret successfully!"
}
```

3. Secret Update Mechanism

When MiCADO is running and there is a need to update some infrastructure secret, it could be done by invoking an API from SPM.

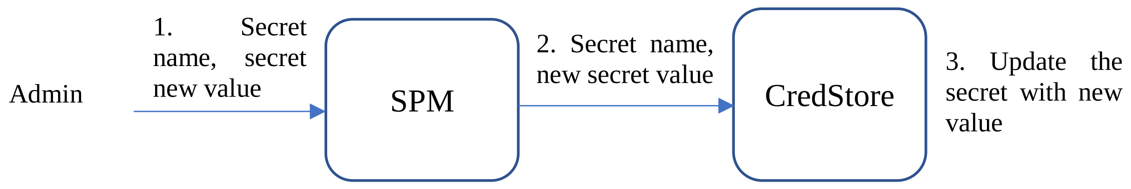
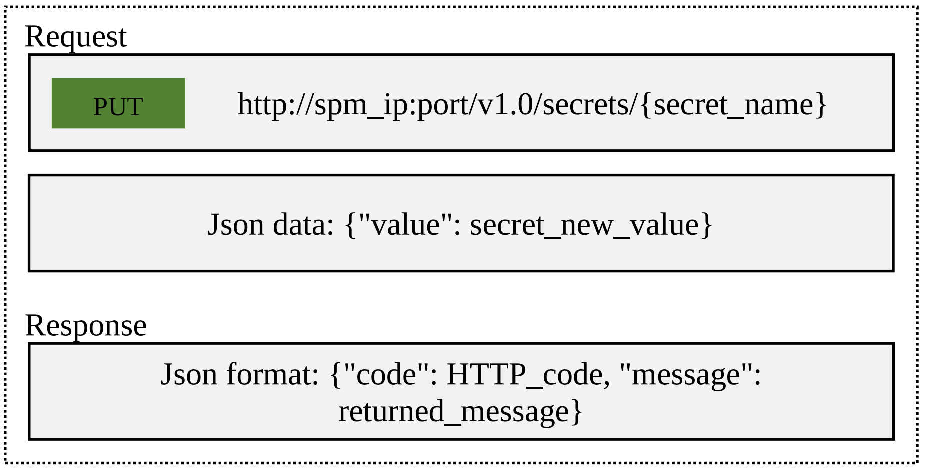

Admin
1. Secret name, secret new value
SPM
2. Secret name, new secret value
CredStore
3. Update the secret with new value

Figure 27 Secret Update Mechanism

The following API allows to update value of an infrastructure secret in CredStore.


Request

PUT http://spm_ip:port/v1.0/secrets/{secret_name}

Json data: {"value": secret_new_value}

Response

Json format: {"code": HTTP_code, "message": returned_message}

Figure 28 Secret Update request and response

Request:

Name	**In**	**Description**
secret_name	Path	Secret name

value	Json body	New value of the secret

Example:

```
curl -H "Content-Type: application/json" -d '{"value":"1aBc"}' -X PUT
http://127.0.0.1:5003/v1.0/secrets/cloudsigma_password
```

Response:

Name	In	Description
code	Json body	Status code
message	Json body	Returned message

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 404 – Not found: Provided secret name does not exist
- 500 - Server error: Some error happened at server side

Example:

```
{
      "code": 200,
      "message": " Update secret successful"
}
```

3.3.4.6 Summary of overall flow

The overall flow of Credential Store is summarized in

Figure 29. At launching time of MiCADO, Vault Initialization in SPM is invoked to set up secret storage in Credential Store (1b). Root token and unseal key(s) are stored as files inside SPM. After that, admin invokes Secret insertion API from SPM to add one or more infrastructure secrets into MiCADO (2a, 2b). Since then, a component inside the Master Node, such as CO, can invoke Secret retrieval API from SPM to access a secret value (3a, 3b). During the entire process, the admin user is allowed to invoke Secret update API to change a secret's value (4a, 4b), if required. The admin can also invoke the Secret deletion API to remove a secret from MiCADO (5a, 5b), when it is not required any longer.

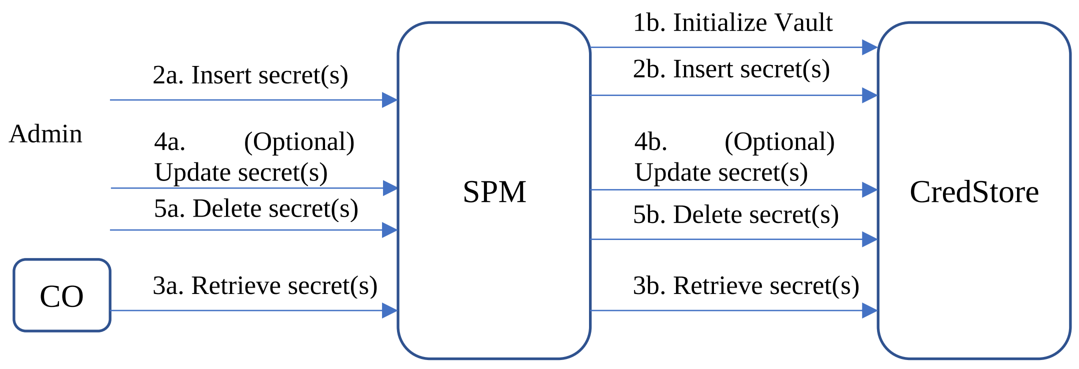


Figure 29 Credential Store Flow Implementation

3.3.4.7 Limitations and extensibility of the Proposed Solution

The Root token issued by CredStore is a token without expiration. In the current implementation, root token and all unseal keys are stored as files inside SPM.

In future, the root token should be exported to admin and then deleted from MiCADO. For SPM, admin should request to generate short-lived token(s) with restricted fine-grained control instead of root control [13]. In such case, SPM should be implemented to provide more APIs for generating/revoking/renewing token(s).

Due to simplicity, all the unseal key(s) are currently stored in SPM and any component in MiCADO could request to access secret through SPM. However, in order to restrict secret access, SPM should keep only one unseal key and distribute another key to an authorized component. The threshold value should be at least two. As soon as the component requests for a secret, it needs to send out its unseal key. Because it requires at least two (defined by the threshold value) unseal keys, the received key need to be used along with the stored unseal key inside SPM to access secret in CredStore. After the request is accomplished, SPM should remove the received unseal key. By this way, even SPM could not access the secret without the component's consent.

Finally, we may develop more functions related to keys and tokens. In addition to previously referred token functions (generating/revoking/renewing token), we could implement SPM with APIs for revoking unseal keys, renewing the master key (called as Rekey) and change the encryption key (called as Key Rotation) [12].

3.4 Credential Manager

3.4.1 Credential Manager Functionality

This section aims to describe the core functionality of Credential Manager (CredMan) as a security component in the MiCADO architecture. The CredMan stores and manages all users for MiCADO. It facilitates user verification through Zorp firewall to perform authentication and access control. The CredMan fulfils the following key security requirements:

1. Users' passwords are not stored in plaintext; instead, only their hash values are stored.

2. CredMan support strong password enforcement.
3. CredMan must supports the user access control based on roles.

3.4.2 *Credential Manager Interaction in MiCADO*

Credential Manager (CredMan) component is designed as a central user management for the MiCADO framework. The CredMan in MiCADO is used in the following two ways: (1) Those admins' users who have access to the Master Node through SSH are enabled to execute commands and/or call Restful APIs to CredMan. The commands include to perform the following functions, i.e. insert, retrieve, update, delete users and their roles. (2) Zorp firewall can access the CredMan API for the verification of users. The verification results are then used to perform authentication and access control.

Figure 30 illustrates the above-mentioned authentication process.

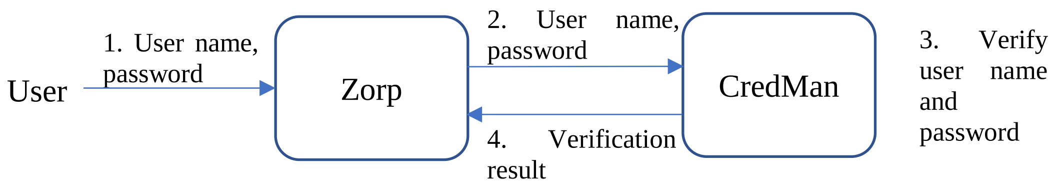


Figure 30 Component interaction for user authentication

3.4.3 *Credential Manager Design and Implementation*

Previously, the *CredMan version 1.0* was implemented from scratch as a Python based Flask Web Service that utilises *Flask-SQLAlchemy* [11] as a SQL tool to perform different data-base oriented operations, e.g. user insertion, deletion, etc. All functions of the component were personally developed. The key reason behind the development of CredMan from scratch is to have full control on the user management module implementation. While existing solutions also provide the functionality described in the requirements, their resource usage is much higher than that of a custom implementation. The CredMan version 1.0 module provides the following functionalities:

- User-related: This include the creation, deletion, retrieval and verification of users;
- Password-related: This allows the change and resetting of user password;
- Role-related: To retrieve and update users' roles;

Recently, we have re-analysed the implementation of CredMan *version 1.0* and found a number of necessary design changes and improvements. Furthermore, to make it more robust, flexible, and easily maintainable/extendible in future, we have also decided to utilise a very popular open source solution, known as Flask-User [10], for user management. This decision leads to the restructuring of the overall component that include major changes and therefore, it will result in the new version of CredMan, i.e. *version 2.0*.

Similarly, to *version 1.0*, the *CredMan version 2.0* is also implemented as a Python based Flask Web Service, however, it now integrates the open source packages including, *Flask-User* [10] for user management and *Flask-SQLAlchemy* [11] for SQL tool. Both these tools are used under MIT license. The key benefits of *Flask-User* include the following,

- Built-in database design for the user management,
- Easy to use utility classes to perform all the required functions of the CredMan module, i.e.
    - User management,
    - Password handling and maintenance,
    - Email confirmations and notifications, and
    - Roles management.
- Fully customisable and largely configurable
- Facilitates role-based authorization
- Facilitate multiple emails per user registration.

CredMan *version 2.0* is a Flask Web service and follows all the REST API standards. Currently it exposes the following resources:

- ***Users*** include APIs that manage MiCADO users. This consist of the following functions, i.e. Create, retrieve, update and delete Users,
- ***Roles*** include APIs that manage MiCADO roles and consist of the functions related to role management such as create, retrieve, update and delete roles,
- ***UserRoles*** include APIs that manage roles of MiCADO users such as retrieve user role, /grant and /revoke user roles,
- ***Password*** include APIs for users' passwords management such as verify, change and reset a given user password.

The email confirmation functionality is under consideration and could be implemented in future. Once such a functionality is included then the user will receive email notification for any event related to their account, i.e. registration, password change and reset, etc. Furthermore, we also intend to apply certain policies on password. This will restrict the password setting process must liaise with the active password policies. Hence making the system more secure overall.

3.4.3.1 Users Management Mechanism

1. User Creation Mechanism

After MiCADO master node is launched and admin is created, the admin can create other users by invoking CredMan Users API.

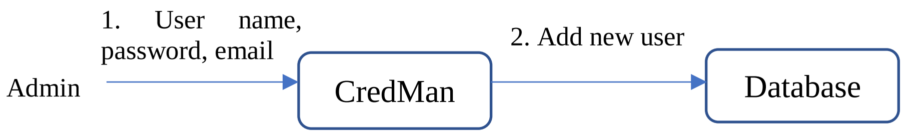
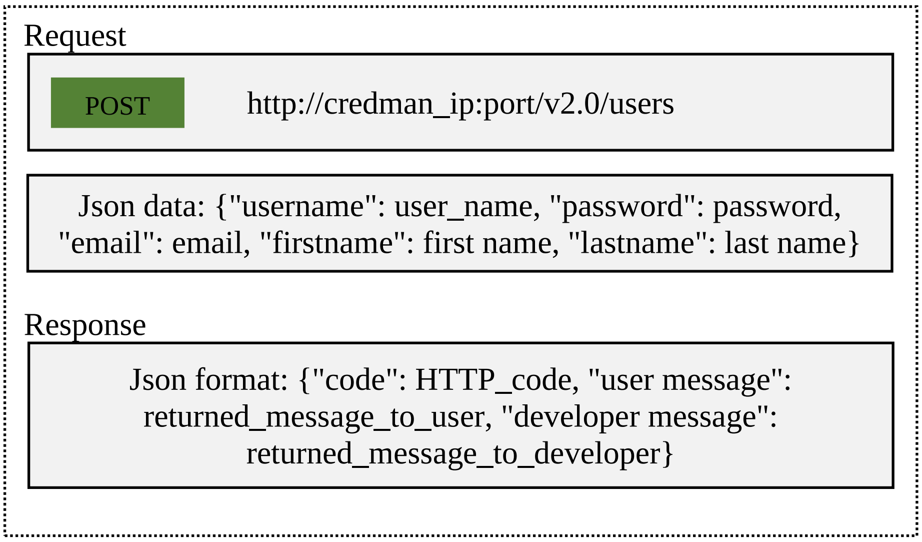


Figure 31 User Creation Mechanism

The following API facilitate the creation of a new MiCADO user.



Request

POST http://credman_ip:port/v2.0/users

Json data: {"username": user_name, "password": password, "email": email, "firstname": first name, "lastname": last name}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 32 New user creation API request and response

Request:

Name	**In**	**Description**
username	Json body	User name. This must be unique for a user.
password	Json body	Password
email	Json body	User’s email. This must be unique for a user.
firstname (optional)	Json body	User’s first name
lastname (optional)	Json body	User’s last name

Example:

```
curl -X POST \
  http://127.0.0.1:5001/v2.0/users \
  -H 'Content-Type: application/json' \
  -d '{
  "username": "user3",
  "password": "1aB",
  "email": "user03@a.com",
  "firstname": "user3fn",
  "lastname": "user3ln"
}'
```

Response:

Name	**In**	**Description**
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:

- 201 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error occured at server side

Example:
```
{
        "code": 201,
        "user message": "Add user successfully!",
        "developer message": "Add user successfully!"
}
```

2. User Retrieval Mechanism

At any time, admin can retrieve user's information by providing a user name.

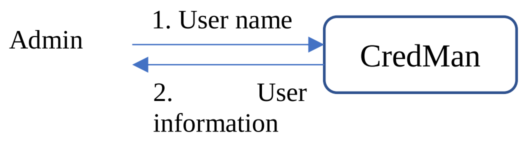
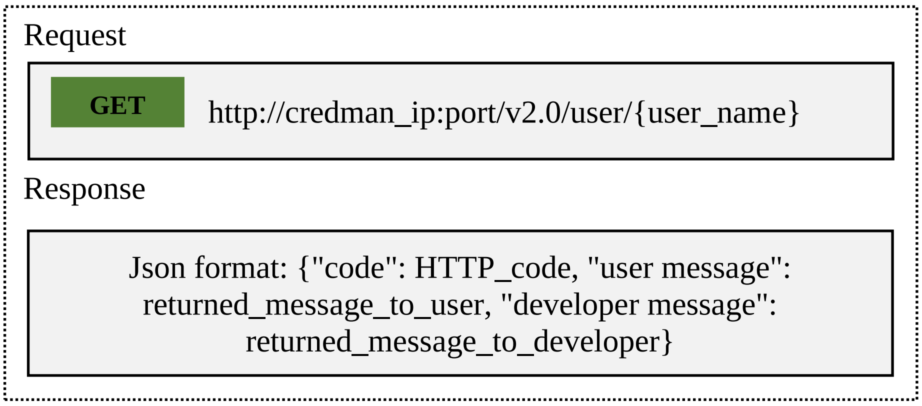


Figure 33 User Information retrieval mechanism

The following API facilitates the retrieval of user's information against a given user name.


Request

GET http://credman_ip:port/v2.0/user/{user_name}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 34 User information retrieval API request and response

Request:

Name	In	Description
user_name	Path	User name

Example:

```
curl -X GET http://127.0.0.1:5001/v2.0/user/user4
```

Response:

Name	In	Description
code	Json Body	Status code
User	Json object	(If user is retrieved successfully)
username		User name
email		Email
firstname		First name
lastname		Last name
active		Active status of user
User message	Json Body	(If failed to retrieve user) Message to user
Developer message	Json Body	(If failed to retrieve user) Message to developer

Status codes:

Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error occured at server side

Example:

```
{
    "code": 200,
    "User": {
        "username": "user4",
        "email": "user04@a.com",
        "first_name": "user4fn",
        "last_name": "user4ln",
        "active": true
    }
}
```

3. User Update Mechanism

Admin can also update users' information. The update function is associated with the users' information. Currently, the CredMan holds first name and last name of users. Therefore, the update API support the editing of these information only. Later on, this API must be further extended when user is defined with more information.

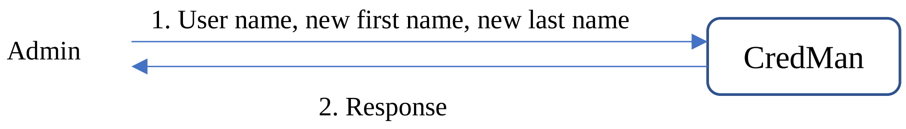
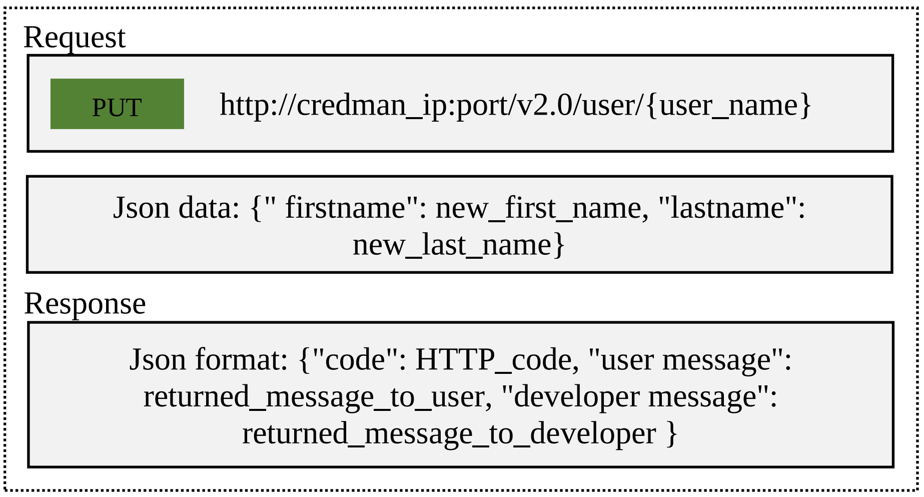


Figure 35 User update mechanism

The following API allows to update first and/or last name of a given user.



Request

PUT http://credman_ip:port/v2.0/user/{user_name}

Json data: {" firstname": new_first_name, "lastname": new_last_name}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer }

Figure 36 User update API request and response

Request:

Name	In	Description
user_name	Path	User name
firstname (optional)	Json body	New first name
lastname (optional)	Json body	New last name

Example:

```
curl -X PUT \
  http://127.0.0.1:5001/v2.0/user/user1 \
 -H 'Content-Type: application/json' \
 -d '{
  "lastname": "user1nln",
  "firstname": "user1nfn"
}'
```

Response:

Name	In	Description
code	Json Body	Status code
user message	Json Body	Message to user
developer message	Json Body	Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:

```
{
    "code": 200,
    "user message": "User's info is updated",
    "developer message": "User's info is updated"
}
```

4. User Deletion Mechanism

The Admin user can remove a user out of MiCADO by providing a user name.

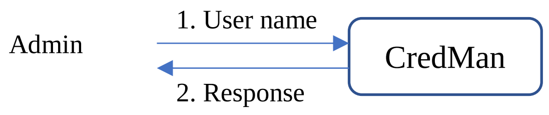
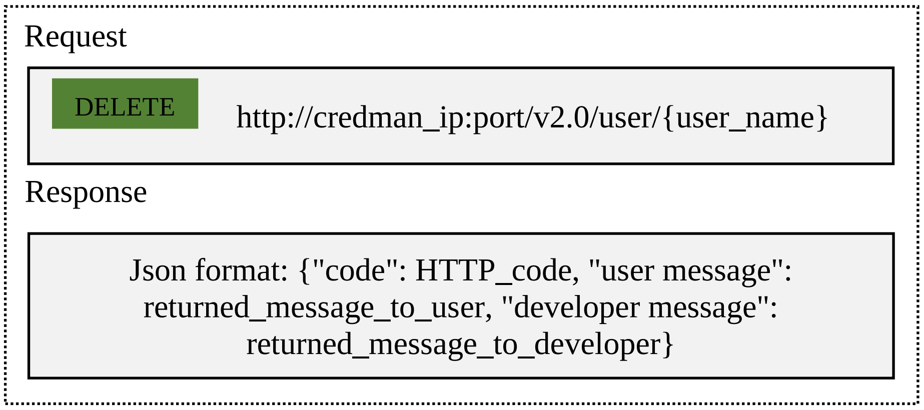


Figure 37 User deletion mechanism

The following API allows the deletion of a MICADO user.



Request

DELETE http://credman_ip:port/v2.0/user/{user_name}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 38 User delete API request and response

Request:

Name	**In**	**Description**
user_name	Path	User name

Example:

```
curl -X DELETE http://127.0.0.1:5001/v2.0/user/user4
```

Response:

Name	**In**	**Description**
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error occured at server side

Example:

```
{
    "code": 200,
    "user message": "Delete user successfully",
    "developer message": "Delete user successfully"
}
```

5. All User Retrieval Mechanism

The Admin user can retrieve all users at the same time. The number of MiCADO users are not expected to be large, therefore, it is assumed that this function will not cause any performance bottlenecks. However, if the number of users gets increase then this function should be altered to retrieve a limited number of users instead of all users to prevent performance overhead.

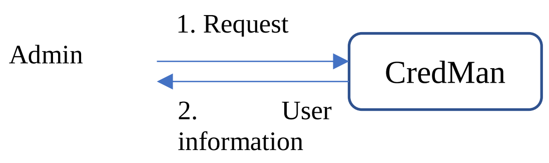
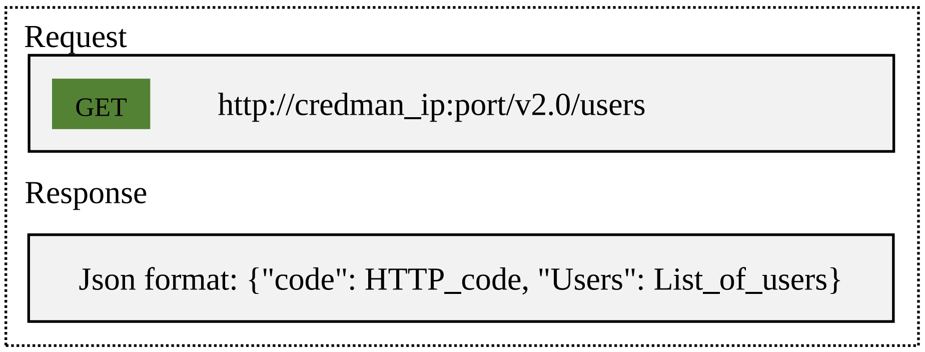


Figure 39 All users retrieval mechanism

The following API faciliates the retrieval of all users.



http://credman_ip:port/v2.0/users

Json format: {"code": HTTP_code, "Users": List_of_users}

Figure 40 All users retrieval API request and response

Request:
None

Example:

```
curl -X GET http://127.0.0.1:5001/v2.0/users
```

Response:

Name	**In**	**Description**
code	Json Body	Status code
Users	List of json objects	
username		User name
email		Email
firstname		First name
lastname		Last name
active		Active status of user

Status codes:

Success:
- 200 - OK: Request was successful

Error:
- 500 - Server error: Some error occured at server side

Example:

```
{
    "code": 200,
    "Users": [
        {
            "username": "user1",
            "email": "user01@a.com",
            "first_name": "",
            "last_name": "",
            "active": true
        },
        {
            "username": "user2",
            "email": "user02@a.com",
            "first_name": "user2fn",
            "last_name": "user2ln",
            "active": true
        },
    ]
}
```

3.4.3.2 Roles Management

1. **Role Insertion Mechanism**

Different roles are defined in order to provide fine-grained access control in MiCADO. Currently the following two roles, i.e. user and admin are defined. However, the design of CredMan is flexible and existing roles can be easily altered and more roles can be easily created. The Admin can easily create more role(s) by simply invoking an API to Credman.

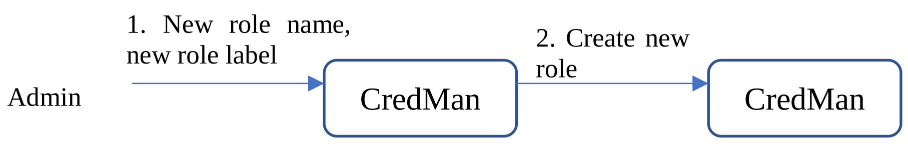
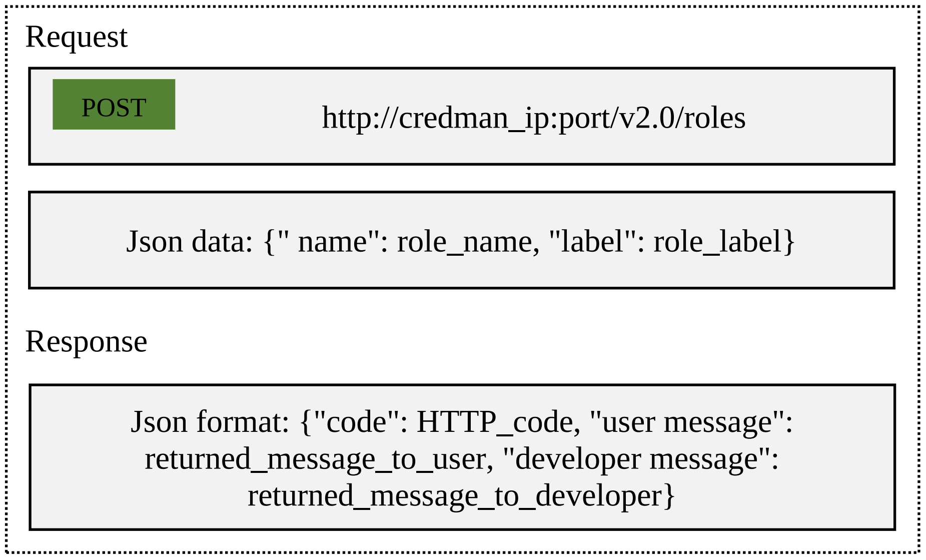


Figure 41 New role creation mechanism

The following API allows the creation of a new role in MiCADO.



Request

POST http://credman_ip:port/v2.0/roles

Json data: {" name": role_name, "label": role_label}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 42 New role creation API request and response formats

Request:

Name	In	Description
name	Json body	Name of role
label	Json body	Label of role

Example:

```
curl -X POST http://127.0.0.1:5001/v2.0/roles \
  -H 'Content-Type: application/json' \
  -d '{
      "name": "developer",
      "label": "Developer"
}'
```

Response:

Name	In	Description
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error occured at server side

Example:
```
{
    "code": 200,
    "user message": "New role added successfully.",
    "developer message": "New role added successfully."
}
```

2. All Roles Retrieval Mechanism

The Admin user can retrieve the list of defined roles in MiCADO.

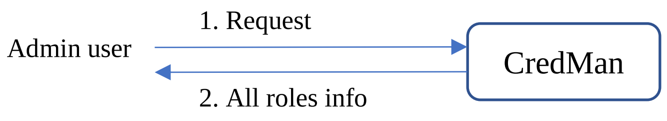
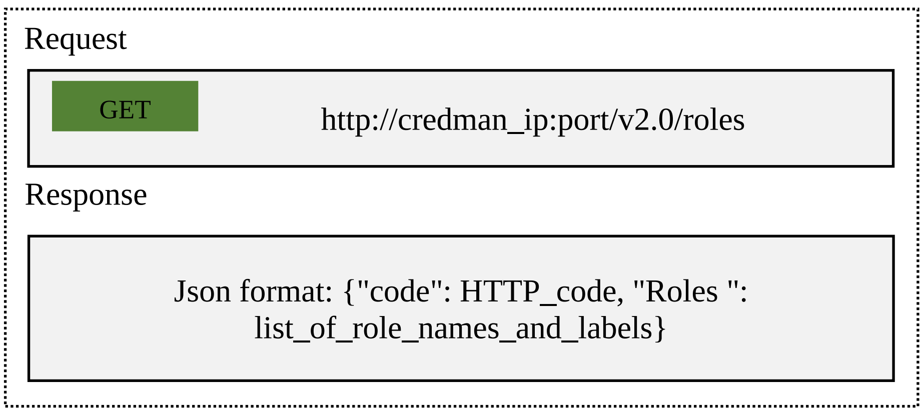


Figure 43 All roles retrieval mechanism

The following API allows to retrieve all roles of MiCADO framework.


Request

GET http://credman_ip:port/v2.0/roles

Response

Json format: {"code": HTTP_code, "Roles ": list_of_role_names_and_labels}

Figure 44 All roles retrieval API request and response formats

Request:
None

Example:

```
curl -X GET http://127.0.0.1:5001/v2.0/roles
```

Response:

Name	In	Description
code	body	Status code
Roles	List of json object	List of roles
name	body	Name of role
label	body	Label of role

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 500 - Server error: Some error happened at server side

Example:

```
{
    "code": 200,
    "Roles": [
       {
            "name": "admin",
            "label": "Admin"
        },
        {
            "name": "developer",
            "label": "Developer"
        }
    ]
}
```

3. Specific Role Retrieval Mechanism

The admin can also retrieve a specific role based on its name instead of retrieving all roles.

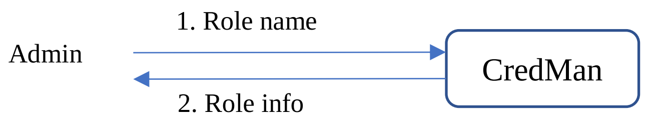
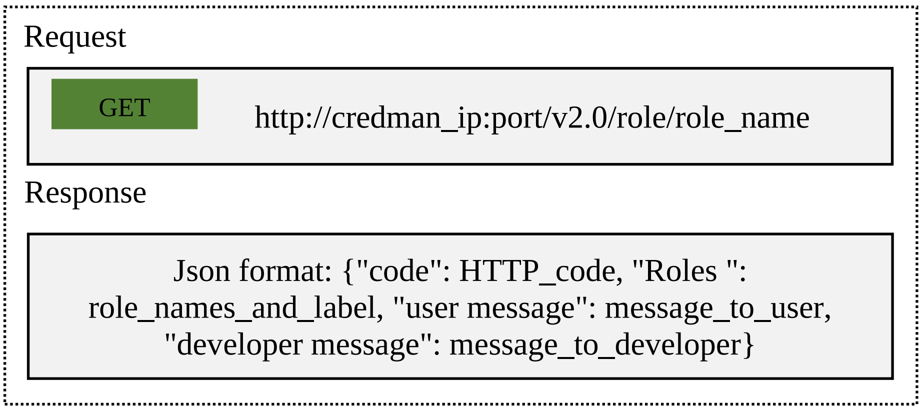


Figure 45 Role retrieval mechanism

The following API allows to retrieve a specific role information of MiCADO framework.


Request

GET http://credman_ip:port/v2.0/role/role_name

Response

Json format: {"code": HTTP_code, "Roles ": role_names_and_label, "user message": message_to_user, "developer message": message_to_developer}

Figure 46 Role retrieval API request and response formats

Request:

Name	In	Description
role_name	Path	Name of a role

Example:
```
curl -X GET http://127.0.0.1:5001/v2.0/role/admin
```

Response:

Name	In	Description
code	Json body	Status code
Roles	Json object	(If requested role_name exists)
name		Name of role
label		Label of role
user message	Json body	(If error) Message to user
developer message	Json body	(If error) Message to developer

Status codes:
Success:
- 200 - OK: Request was successful

Error:
- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error occured at server side

Example:
```
{
    "code": 200,
    "Roles": {
        "name": "admin",
        "label": "Admin"
    }
}
```

4. Role Label Update Mechanism

The admin user is allowed to update label of a specific role based on its name.

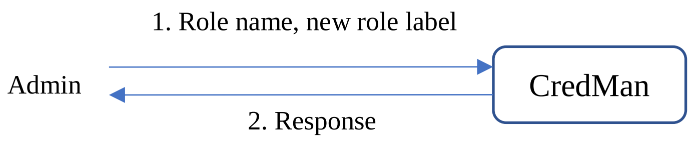
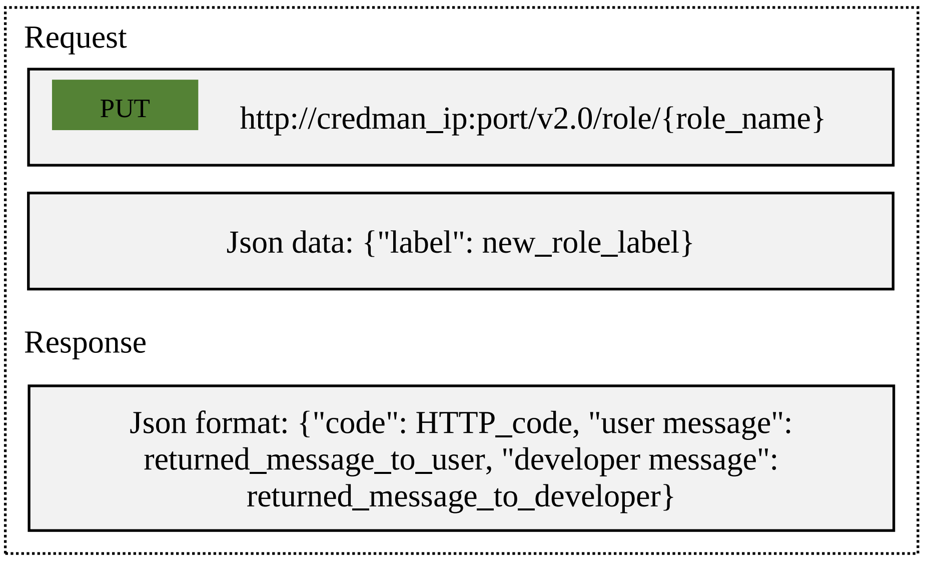


Figure 47 Role label update mechanism

The following API allows to update label of a role in MiCADO.



Request

PUT http://credman_ip:port/v2.0/role/{role_name}

Json data: {"label": new_role_label}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 48 Role label update API request and response formats

Request:

Name	In	Description
role_name	Path	Name of a role
label	Json body	New label of the role

Example:
```
curl -X PUT \
  http://127.0.0.1:5001/v2.0/role/user
  -H 'Content-Type: application/json' \
  -d '{
  "label": " normal_user"
}'
```

Response:

Name	In	Description
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:
```
{
	"code": 200,
	"user message": " Updated role label successfully!",
	"developer message": " Updated role label successfully!"
}
```

5. Role Deletion Mechanism

The Admin user can delete an existing role in MiCADO.

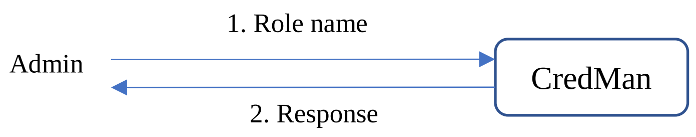
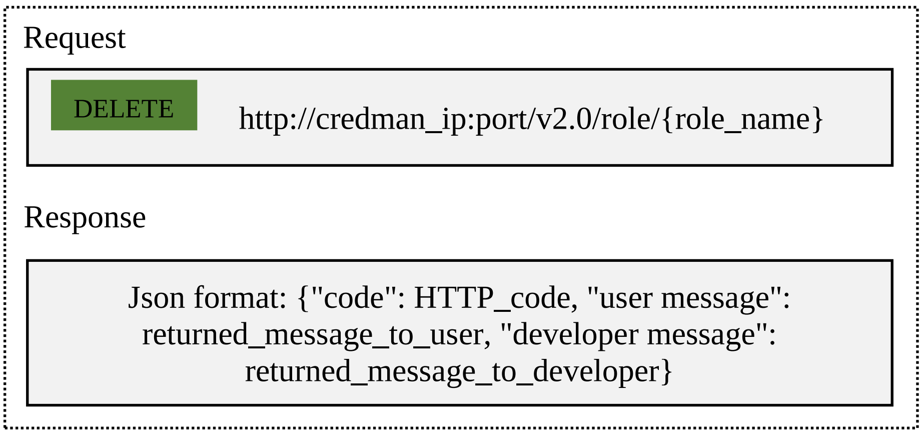


Figure 49 Role deletion mechanism

The following API allows the deletion of a role from MiCADO. This function also unassigned the deleted role from all MiCADO users.



Request

DELETE http://credman_ip:port/v2.0/role/{role_name}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 50 Role deletion API request and response formats

Request:

Name	In	Description
role_name	Path	Name of a role

Example:
```
curl -X DELETE http://127.0.0.1:5001/v2.0/role/developer
```

Response:

Name	In	Description
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:
```
{
    "code": 200,
    "user message": "Delete role successfully.",
    "developer message": "Delete role successfully."
}
```

3.4.3.3 User Role Management

1. **User Role Retrieval Mechanism**

The Admin user can retreive the assigned roles' information of a given user.

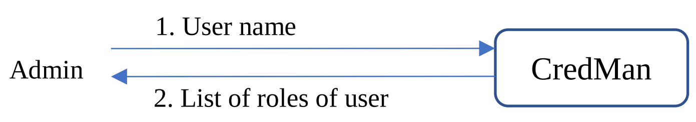
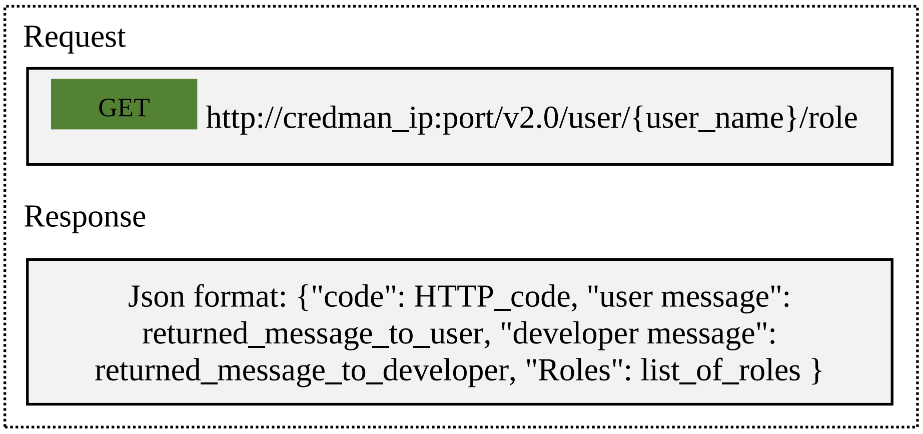


Figure 51 User role retrieval mechanism

The following API facilitate the above mentioned functionality.


Request

GET http://credman_ip:port/v2.0/user/{user_name}/role

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer, "Roles": list_of_roles }

Figure 52 User role retrieval API request and response formats

Request:

Name	In	Description
user_name	Path	User name

Example:
```
curl -X GET http://127.0.0.1:5001/v2.0/user/user1/role
```

Response:

Name	In	Description
code	Json body	Status code
Roles	Json body	(if success) List of roles
user message	Json body	(If error) Message to user
developer message	Json body	(If error) Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:
```
{
    "code": 200,
    "Roles": [
        "developer",
        "admin"
    ]
```

}

2. User Role Revocation Mechanism

The Admin user can revoke a particular role from a specific user. The following figure demonstrate this mechanism.

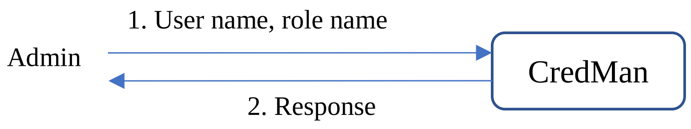
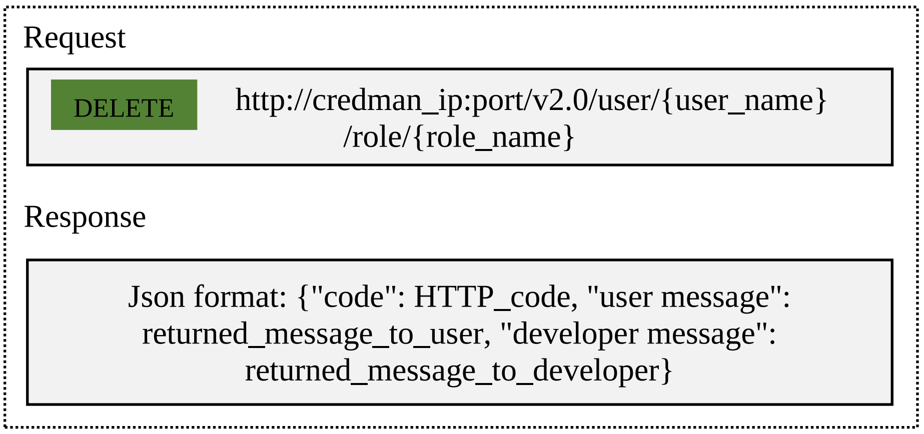


Figure 53 User role revocation mechanism

The following API allows to revoke a user's role.



```
DELETE http://credman_ip:port/v2.0/user/{user_name}/role/{role_name}
```

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 54 User role revocation API request and response formats

Request:

Name	In	Description
user_name	Path	User name
role_name	Path	Role name

Example:

```
curl -X DELETE http://127.0.0.1:5001/v2.0/user/user1/role/developer
```

Response:

Name	In	Description
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:

Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:

```
{
    "code": 200,
    "user message": " The specified role is revoked from the user!",
    "developer message": " The specified role is revoked from the user!"
}
```

3. User Role Grant Mechanism

The Admin user can grant a list of roles to an existing user as shown in the following figure.

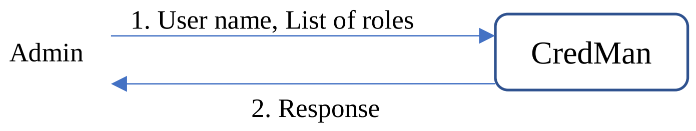


Figure 55 User role grant mechanism

The following API allows to assigned role(s) to a given user.

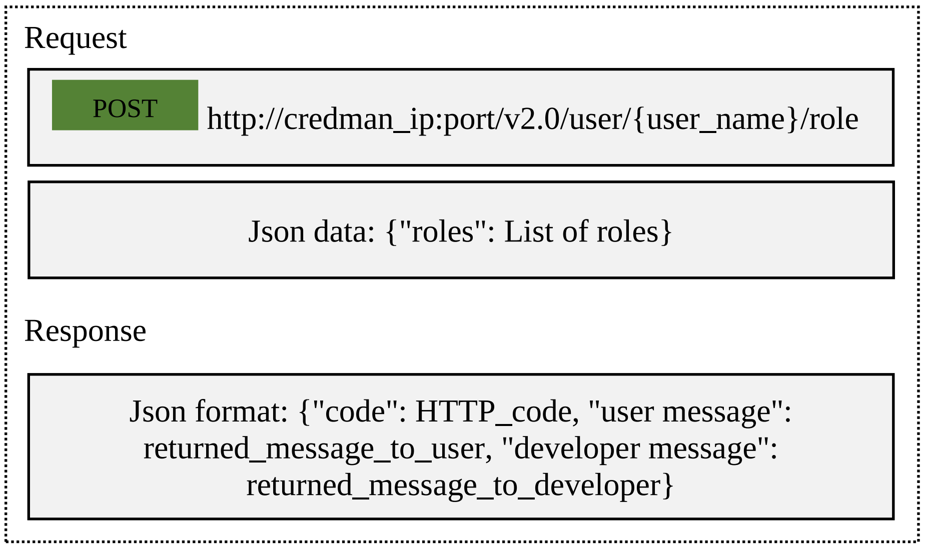

Request
POST
http://credman_ip:port/v2.0/user/{user_name}/role
Json data: {"roles": List of roles}
Response
Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer}

Figure 56 User role grant API request and response formats

Request:

Name	**In**	**Description**
user_name	Path	User name
roles	Json body	List of roles

Example:
```
curl -X POST http://127.0.0.1:5001/v2.0/user/user1/role \
  -H 'Content-Type: application/json' \
  -d '{
  "roles": ["developer","admin"]
}'
```

Response:

Name	In	Description
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:
```
{
    "code": 200,
    "user message": " The new role(s) is assigned to the specified user!",
    "developer message": " The new role(s) is assigned to the specified
user!"
}
```

3.4.3.4 Password

1. User Verification Mechanism

As user logs in with user name and password, Zorp invokes an API to CredMan to verify the user. Based on the verification result, Zorp performs access control to the user.

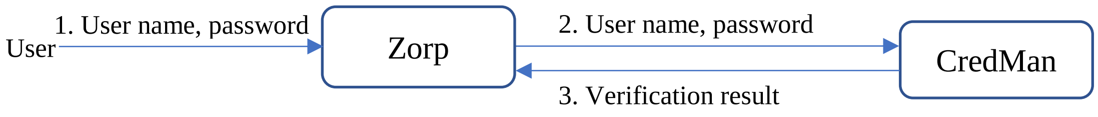


Figure 57 User verification mechanism

The following API allows to verify user's credential.

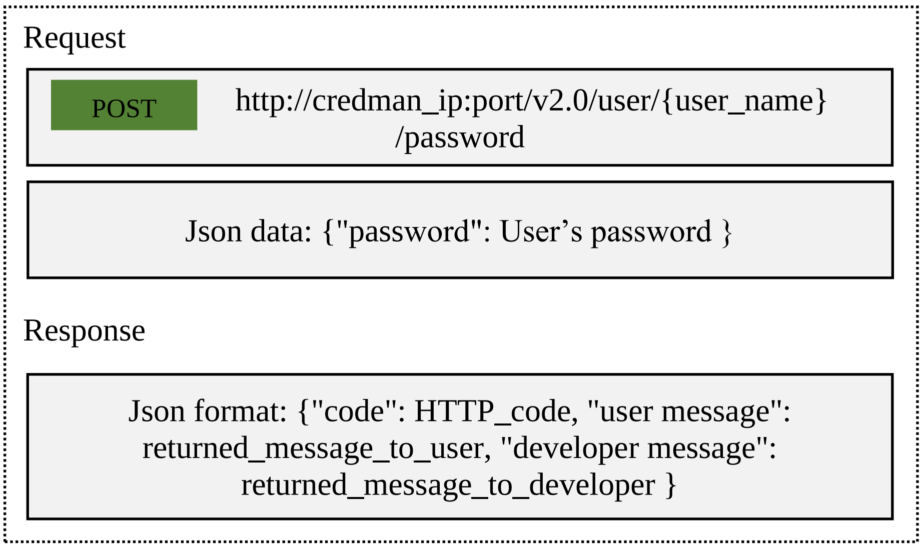

Request
POST
http://credman_ip:port/v2.0/user/{user_name}/password
Json data: {"password": User's password }
Response
Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer }

Figure 58 User verficiation API request and response formats

Request:

Name	In	Description
user_name	Path	User name
password	Json body	User's password

Example:
```
curl -X POST http://127.0.0.1:5001/v2.0/user/user1/password \
  -H 'Content-Type: application/json' \
  -d '{
      "password": "1234"
}'
```

Response:

Name	In	Description
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error occurred at server side

Example 1:

```
{
    "code": 200,
    "user message": "User is authenticated!",
    "developer message": "User is authenticated!"
}
```

Example 2:

```
{
    "code": 400,
    "user message": "User name or password is wrong!",
    "developer message": "Password does not match!"
}
```

2. User Password Change Mechanism

Zorp takes care of access control of users based on their roles. Users with both roles ‘user’ and ‘admin’ are allowed to change their passwords. In order to change password, a user needs to provide his/her user name, current password and new password. The new password must satisfy password policies defined by CredMan.

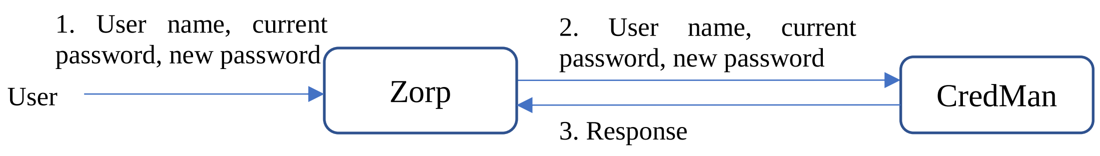
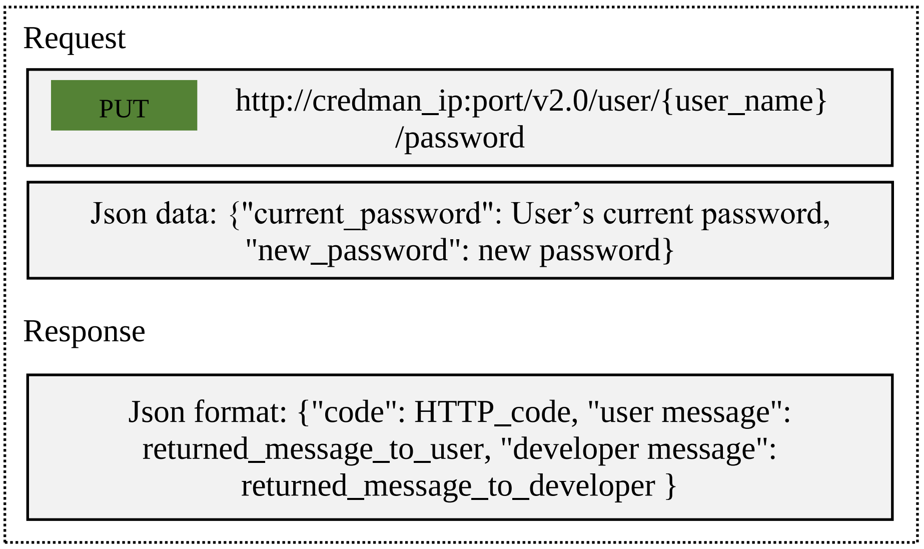

1. User name, current password, new password
User
Zorp
2. User name, current password, new password
CredMan
3. Response

Figure 59 User password change mechanism

The following API allows to change a user’s password


Request

PUT http://credman_ip:port/v2.0/user/{user_name}/password

Json data: {"current_password": User’s current password, "new_password": new password}

Response

Json format: {"code": HTTP_code, "user message": returned_message_to_user, "developer message": returned_message_to_developer }

Figure 60 User password change API request and response formats

Request:

Name	In	Description

user_name	Path	User name
current_password	Json body	User’s current password
new_password	Json body	New password

Example:
```
curl -X PUT http://127.0.0.1:5001/v2.0/user/user1/password \
  -H 'Content-Type: application/json' \
  -d '{
      "current_password": "1234",
      "new_password": "1aBc"
}'
```

Response:

Name	**In**	**Description**
code	Json body	Status code
user message	Json body	Message to user
developer message	Json body	Message to developer

Status codes:
Success:
- 200 - OK: Request was successful

Error:
- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error happened at server side

Example:
```
{
    "code": 200,
    "user message": "Your password is changed successfully!",
    "developer message": "Your password is changed successfully!"
}
```

3. User Password Reset Mechanism

The Admin user can reset users’ passwords. New password is generated randomly.

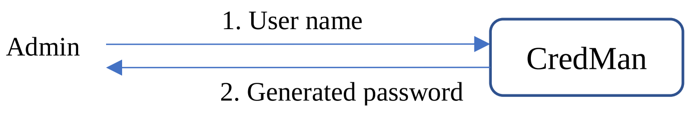


Figure 61 User password reset mechanism

The following API allows to reset a user’s password.

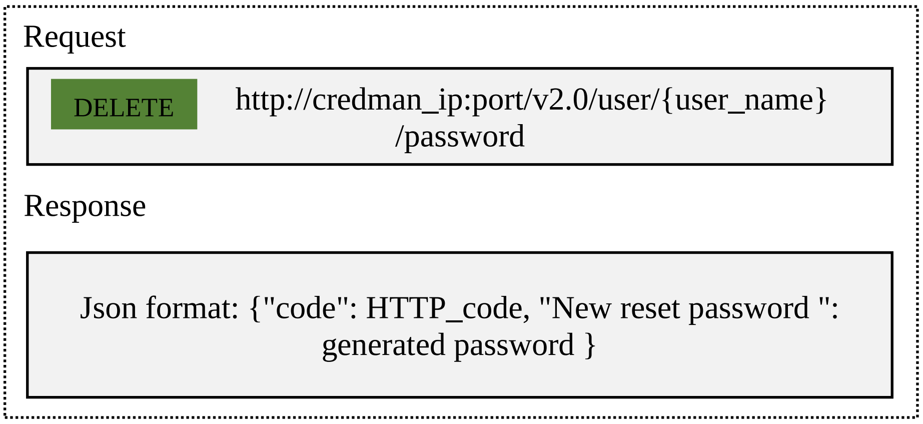

Request
DELETE
http://credman_ip:port/v2.0/user/{user_name}/password
Response
Json format: {"code": HTTP_code, "New reset password ": generated password }

Figure 62 User password reset API request and response formats

Request:

Name	In	Description
user_name	Path	User name

Example:
```
curl -X DELETE http://127.0.0.1:5001/v2.0/user/user1/password
```

Response:

Name	In	Description
code	Json body	Status code
new reset password	Json body	New password

Status codes:
Success:

- 200 - OK: Request was successful

Error:

- 400 - Bad Request: Some content in the request was invalid or missing a required parameter
- 500 - Server error: Some error occurred at server side

Example:
```
{
    "code": 200,
    "New reset password": "naspnMk"
}
```

3.4.3.5 Summary of overall flow

The final flow of Credential Manager is summarized in Figure 63. Assuming that MiCADO has been launched and admin has been created. At first, the admin creates role(s) (1), user(s) (2) and grant role(s) to user(s) (3). After that, as long as user logs into MiCADO (4), he/ she provides his/her user name and password. Upon reception, Zorp contacts to CredMan to verify the user (5) and retrieve his/ her role (6). Based on the verification result and the user's role, Zorp controls the user's access to MiCADO.

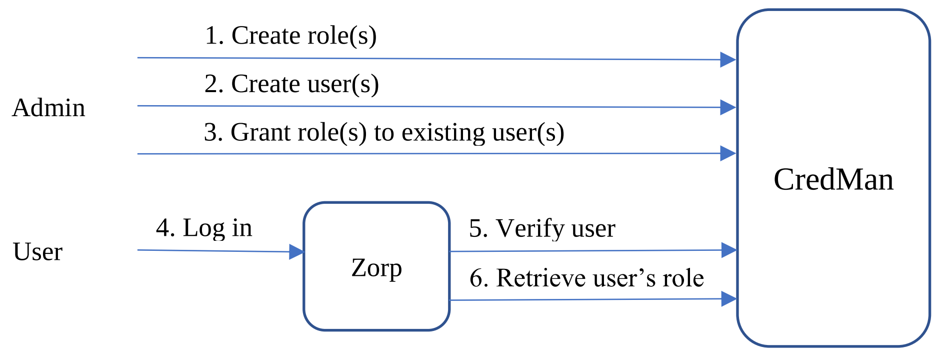


Figure 63 Credential Manager Flow Implementation

3.4.3.6 Limitations and extensibility of the Proposed Solution

The current implementation of Credential Manager provides basic functions for central user management. In future, we can extend CredMan to support more advanced functions such as:

- Email notification: Sending emails to notify users on actions such as password reset, password change, continuous failed log in, etc.;
- Password policy configuration: At present, password policy is defined by regular expression in source code of CredMan. Later, CredMan may be implemented to allow admin to configure password policy;
- Failed log in restriction: At present, user may try logging in as many time as possible. In the future, user account may be blocked for some time after a fixed number of continuous log in.

3.5 Master Node Zorp Firewall

3.5.1 *Master Node Zorp Firewall Functionality*

The "Layer 7" firewall on the master node is an application level firewall. Compared to the "Layer 4" (packet filter) firewall which filters out illegitimate traffic by its network source and destination the "Layer 7" firewall protects the components on the Master Node by inspecting their actual communication on protocol level. It enforces protocol compliance and acts as a user authentication and authorization point for accessing the components on the Master Node.

3.5.2 *Master Node Zorp Firewall Design*

The "Layer 7" firewall functionality is provided by Zorp. It acts as the network entry point for all externally available components on the Master Node:

- access to the Dashboard (and its sub-components);
- access to the TOSCA Submitter's API.

All external network connections are terminated on the firewall and recreated towards the internal components, effectively proxying the protocol traffic. This allows for a single point where traffic related security critical implementations can be placed. See Figure 64 for a network flow overview.

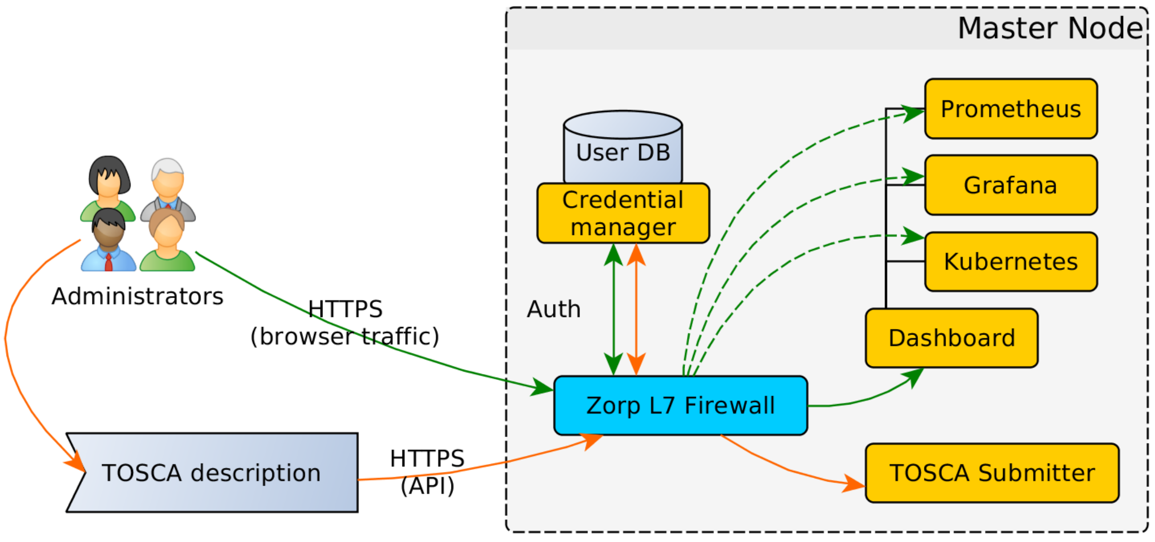


Figure 64 Network flow

Zorp firewall provides:

- TLS termination: it acts as a single point for handling TLS configuration and implementation. It also implements Strict Transport Security. Components internally are accessed through plain HTTP. This makes it easier to design, implement and configure the internal components.
- Protocol enforcement and filtering: The firewall enforces TLS and HTTP protocol compliance for incoming traffic. It also filters HTTP methods to those required by the components to further reduce any attack surface.
- User authentication: Zorp provides user authentication by verifying user's credentials through the Credential Manager component. It provides HTTP Basic Authentication for the TOSCA Submitter API and applies login form injection into browser traffic.
- Request routing: The firewall provides URL entry points for the internal components for the Dashboard. The Dashboard depends on these entry points to load the components' statuses into a unified web view.
- Request type filtering: Only request types that are valid for the selected endpoint are permitted to mitigate exploitation of possible security flaws.
- URL path filtering: URL paths within the specified endpoints can be denied to ensure that the configuration of the protected endpoints cannot be altered by MiCADO users.

3.5.3 Master Node Zorp Firewall Implementation

The following figures describe the communication flows through the Zorp firewall.
The firewall is responsible for TLS setup, user authentication and routing requests to the respective component. Figure 65 provides an overview of these possible call flows.

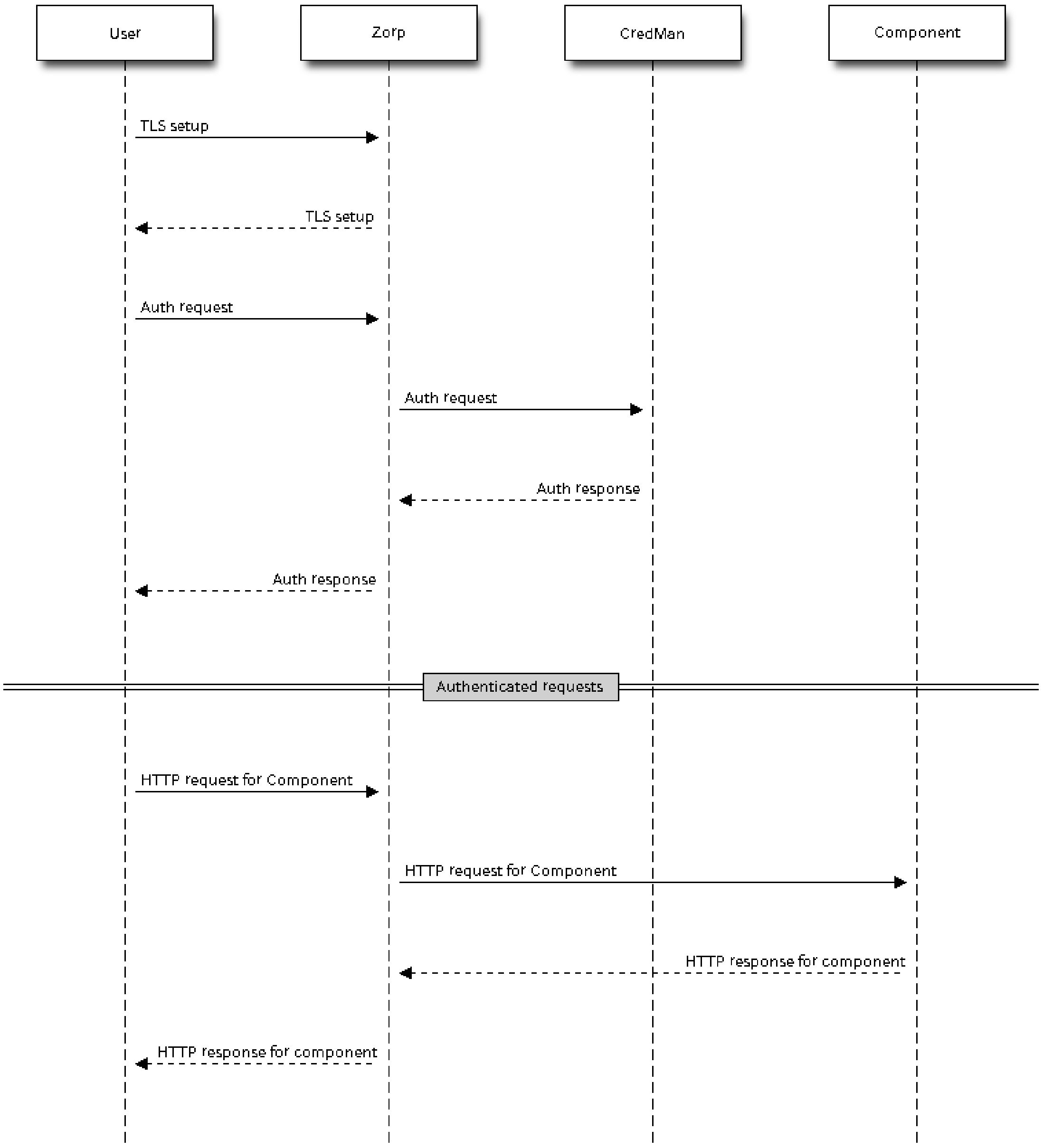


Figure 65 Firewall communication overview

The first step of any communication flow is setting up the TLS channel. Figure 66 gives a more detailed view of the standard TLS setup process.

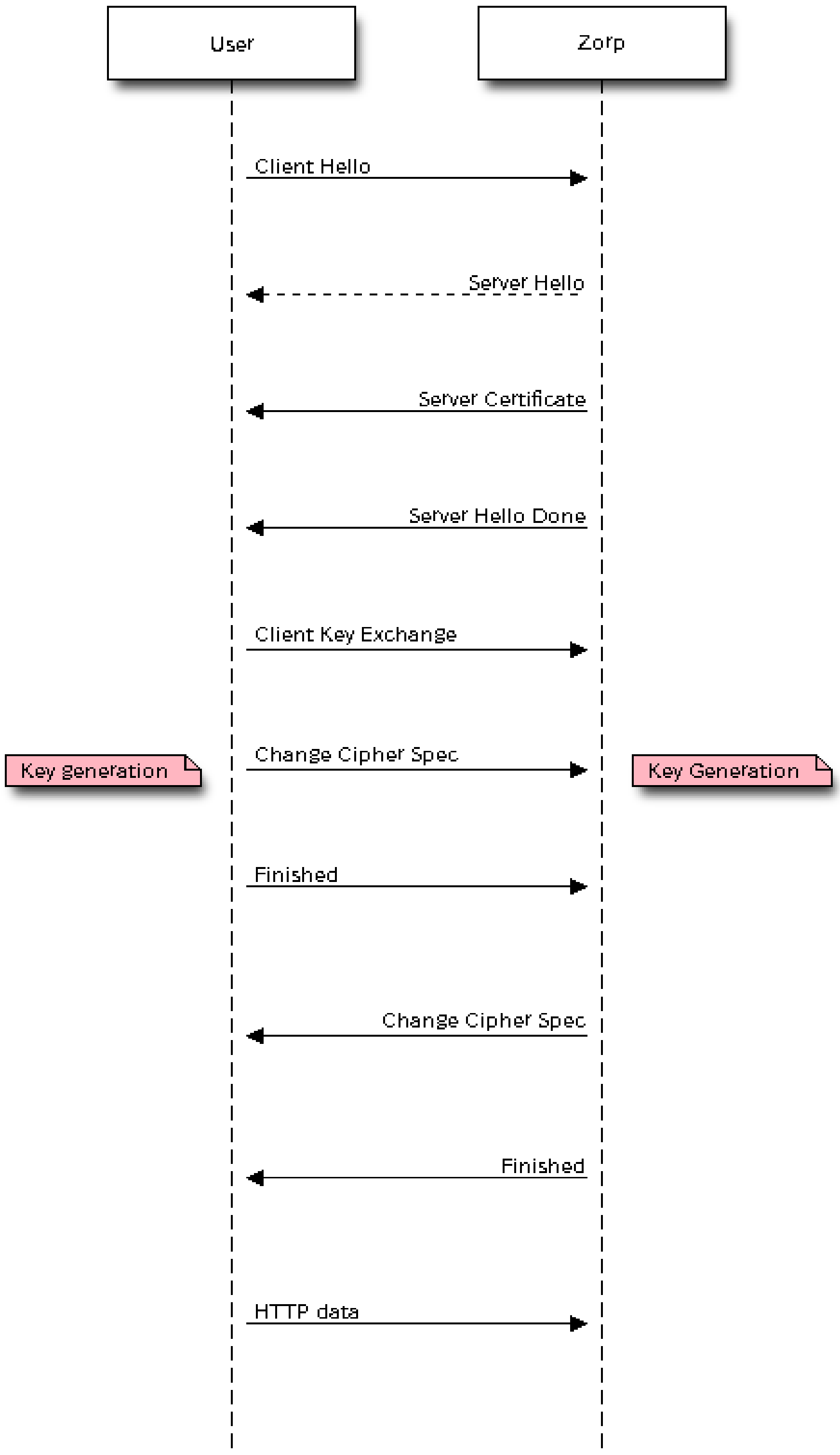


Figure 66 TLS setup process

Any request to the components of the Master Node must be authenticated. Depending on the client component two methods are supported: HTTP Basic authentication and Login form injection.
Basic authentication is used when accessing the API provided by TOSCA Submitter component, as described by Figure 67.

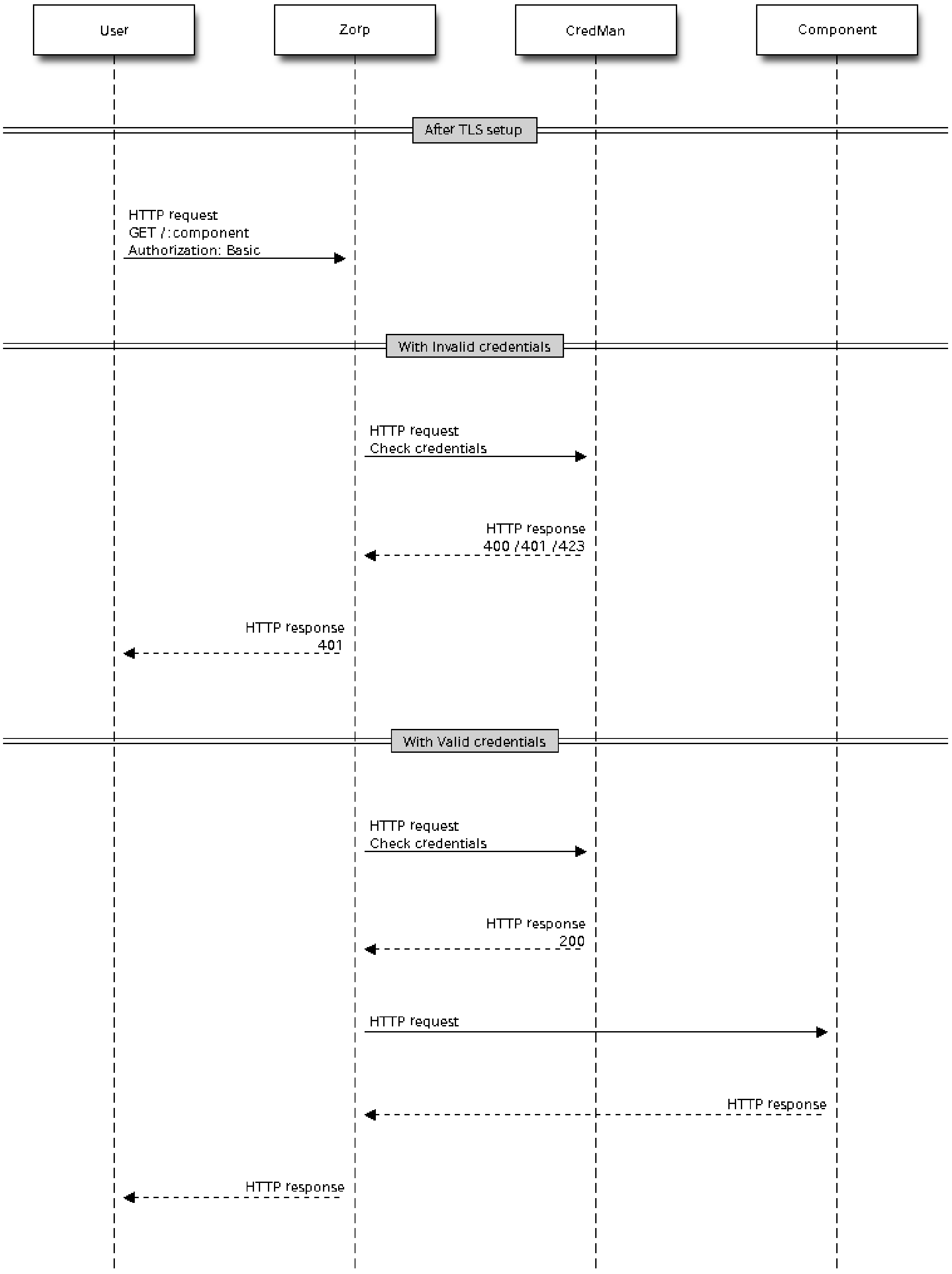


Figure 67 HTTP basic authentication

Browser based user sessions are authenticated by injecting an authentication form by the firewall. Zorp hands out ZorpSession cookies for authenticated sessions. For unauthenticated sessions it returns a custom HTML login form to the user and verifies the provided user credentials through the Credential Manager component as shown on Figure 68.

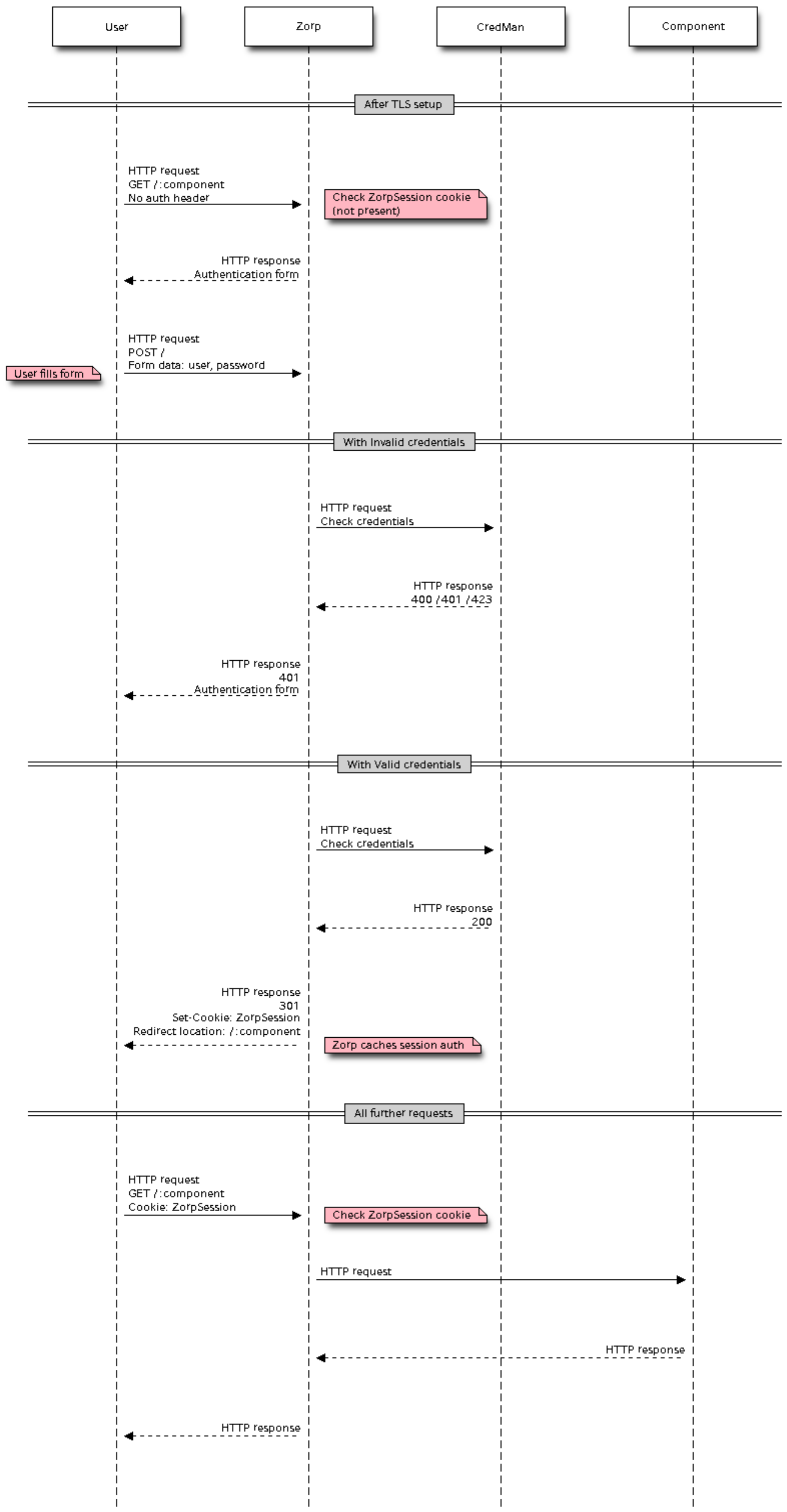


Figure 68 HTTP form authentication

The dashboard component provides embedded views for the individual dashboards provided by the other components. Figure 69 provides an overview of how requests are routed to the correct components.

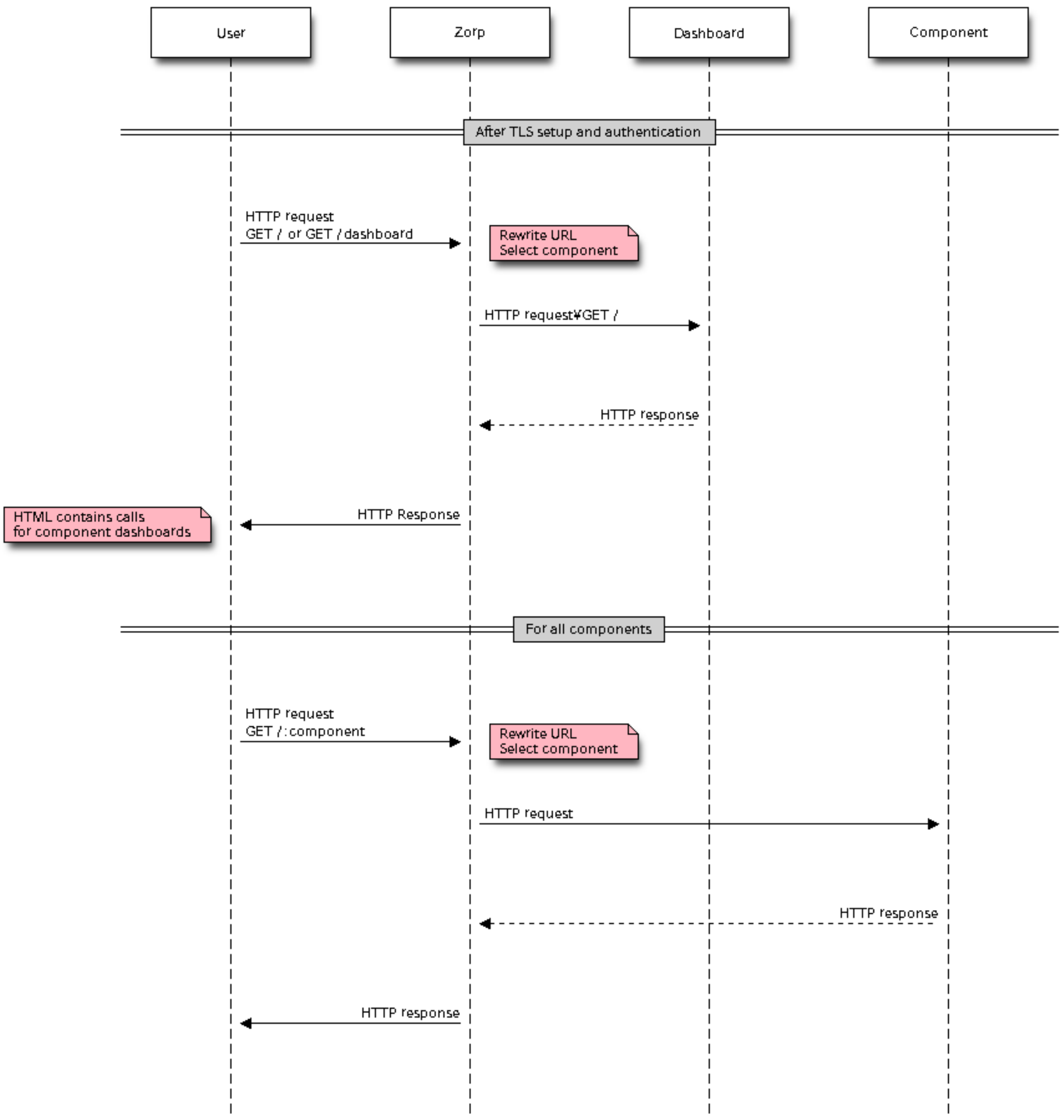


Figure 69 Request routing

3.6 Security Policy Manager

3.6.1 Security Policy Manager Functionality

Security Policy Manager is the single of point of access for MiCADO security components. SPM provides an aggregation of Restful API endpoints that serves different MiCADO Master Node components. It also acts as a workflow director that uses other security enablers to implement security-related business processes.

Security Policy Manager has the following endpoints:

- Credential Store for storing infrastructure secrets. See 3.3 for details.

- Worker Node Certificates for generating X.509 certificates used for Master-Worker Secure Communication. See 3.7 for details.
- Worker Node Certificate Revocation List.
- Worker Node Join Tokens for handling Kubernetes join tokens. See 3.7 for details.
- Crypto Engine to expose Crypto Engine’s functionality within MiCADO Master Node. See 3.2 for details.
- Image Verify to expose Image Verifier’s functionality within MiCADO Master Node. See 3.1 for details.

3.6.2 *Security Policy Manager Design*

Security Policy Manager is a Restful web service with the endpoints listed above. It is accessible via HTTP within the MiCADO master node and is not accessible from neither the public network or the Worker Nodes.

There are two types of security functionality served by SPM. Some of the functions are implemented by other MiCADO security enablers. These functions are exposed via an API similar to the enabler’s own API. Calls to these API’s are forwarded to the respective back end’s API.

The other type of business functionality is implemented directly in Security Policy Manager. SPM might still use external services as a backend.

Functionality implemented in other MiCADO security enablers, exposed by SPM:

- Crypto Engine
- Image Verifier

Functionality implemented in SPM:

- Credential Store
- Worker Node Certificates
- Worker Node Certificate Revocation List
- Worker Node Join Tokens

3.6.3 *Security Policy Manager Implementation*

Security Policy Manager is a Python application implementing a Flask-RESTful API. SPM uses HVAC (Python Hashicorp Vault client) to access Vault in the Credential Store implementation while Requests to access Vault PKI backend in Worker Node Certificates and Certificate Revocation List endpoints. SPM also uses Kubernetes Python API in the Worker Node Join Tokens endpoint’s implementation.

3.6.3.1 Security Policy Manager Endpoints

Security Policy Manager listens on TCP port 5003 for HTTP connections. It provides the following Restful web service endpoints:

- /v1.0/secrets for Credential Manager
- /v1.0/nodecerts for Worker Node Certificates
- /v1.0/nodecrl for Worker Node Certificate Revocation List
- /v1.0/nodetokens for Worker Node Kubernetes Join Tokens
- /v1.0/cryptoengine for Crypto Engine
- /v1.0/imageverify for Image Verifier

The Secrets endpoint exposes the Credential Store functionality as described in detail in 3.3.

Nodecerts and nodecrl endpoints are implemented in terms of Hashicorp Vault PKI secrets backend. A CA certificate is issued by Vault when MiCADO is set up, that is used for signing Worker Node certificates. We use these Worker Node certificates to authenticate the nodes when building IPsec tunnel to Master Node. Worker Node certificates are validated against the CA and the certificate revocation list by IPsec. This process is described in detail in 3.7.

Nodetokens endpoint lets Occopus get a Kubernetes cluster join token for the new Worker Node under provisioning. Join tokens are issued by the Kubernetes API, SPM calls this API to get a new join token and return it to Occopus. This process is described in detail in 3.7.

Cryptoengine endpoint exposes Crypto Enginer security enabler. This endpoint forwards calls to Crypto Engine and returns responses from it to the client application. No additional logic is implemented in this endpoint. See Crypto Engine’s detailed functionality in 3.2.

Imageverify endpoint exposes Image Verifier security enabler. This endpoint forwards calls to Image Verifier and returns responses from it to the client application. No additional logic is implemented in this endpoint. See Image Verifier’s detailed functionality in 3.1.

3.7 Master-Worker Secure Communication

3.7.1 Master-Worker Secure Communication Functionality

In MiCADO management traffic is flowing between the Master and the Worker Nodes. Since this communication takes place over an untrusted network the communication channel must be secured. The secured channel must provide confidentiality (including endpoint identification) and integrity for the transferred data.

This is ensured by encrypting all Master Node – Worker Node communication by:

- providing a secure network channel between the Master and the Worker nodes;
- providing worker node identification for the Kubernetes cluster;
- providing identification for the secure network channel where management traffic flows.

3.7.2 Master-Worker Secure Communication Design

Master-Worker Secure Communication requirements can be satisfied by building an encrypted communication channel between the Master Node and Worker Node with endpoint authentication.

In case of Kubernetes management traffic, additional authentication is enabled on the Kubernetes API by issuing a new join token with a short expiration for every provisioned Worker node.

For all MiCADO components requiring Master-Worker Secure Communication, an IPsec tunnel is set up between the Master Node and each Worker Node. IPsec provides the encrypted, secure communication channel thus guarantees management traffic confidentiality and integrity. Endpoint authentication is achieved by using X.509 certificates on both the Master and Worker Nodes.

In MiCADO, Security Policy Manager is responsible for generating and revoking client certificates as well as distributing Kubernetes join tokens.

3.7.3 Master-Worker Secure Communication Implementation

3.7.3.1 Setting up a new Worker Node

Setting the secure communication channel takes place when a new Worker Node is provisioned. Provisioning the new node is orchestrated by Occopus, while Occopus gets the required security tokens from Security Policy Manager.

Newly deployed Worker Nodes must be provisioned with identity tokens that can be verified by Master Node components. A Worker Node needs a Kubernetes join token so that kubelet component can join the Kubernetes cluster. A Worker also needs an X.509 certificate used by Ipsec to authenticate the node with Master.

Occopus receives these credentials from Security Policy Manager, that is responsible for lifecycle management of these tokens. SPM handles X.509 certificates with its Hashicorp Vault backend using the PKI secret engine. Kubernetes join tokens are only distributed by SPM while generated by the Kubernetes API.

Occopus incorporates the security tokens into the Cloud Init used for deploying the Worker Node. After the Worker Node is deployed, it joins the Kubernetes cluster with the join token and sets up an IPsec tunnel to the master node with the certificate. IPsec on the Master Node validates Worker's certificate with SPM.

Worker Node provisioning workflow in terms of secure communication is shown on Figure 70.

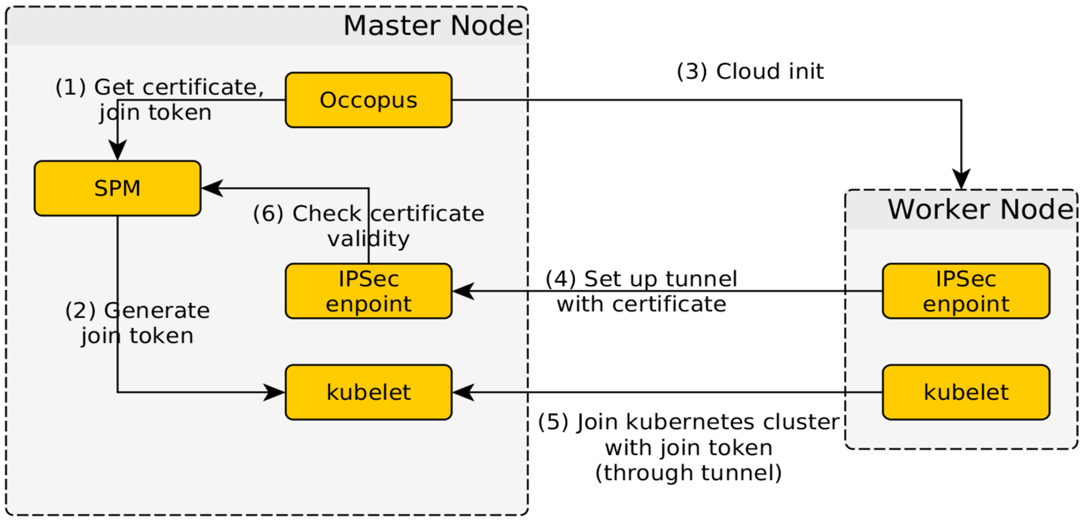

Master Node
(1) Get certificate, join token
Occopus
(3) Cloud init
SPM
(6) Check certificate validity
IPSec enpoint
(2) Generate join token
kubelet
(4) Set up tunnel with certificate
(5) Join kubernetes cluster with join token (through tunnel)
Worker Node
IPSec enpoint
kubelet

Figure 70 Master-Worker Secure Communication setup

3.7.3.2 Decommissioning a worker node

When decommissioning a worker node Occopus notifies SPM of this event. SPM then invalidates the IPsec certificate and Kubernetes cluster membership. Both IPsec endpoint and Kubernetes API is made aware of this invalidation.

4 Artefact Traceability

In this section we revisit the traceability chain of the MiCADO security modules – from security requirements to architecture objectives and to open specifications of security enablers.

4.1 Image Integrity Verifier

Security requirements traceability

The IIV addresses the following requirements outlined in D7.1 COLA security requirements: CNSR-2, CNSR-6

Architecture objectives traceability

The IIV addresses the following security architecture objectives outlined in D7.2 MiCADO security architecture specification: O4.1, O4.4, O6.2

Open Specifications traceability

The MiCADO Image Integrity Verifier Security Module corresponds to Open Specification 4.1 in D7.3 Design of application level security classifications formats in principles.

4.2 CryptoEngine

Security requirements traceability

The Crypto Engine directly addresses the following requirements outlined in D7.1 COLA security requirements: SR12, SR13, CNSR-3, CNSR-9, CSSR-1. Furthermore, the Crypto Engine supports a set of additional requirements outlined in D7.1 COLA security requirements: SR01, SR02, SR11, CNSR-7.

Architecture objectives traceability

The Crypto Engine directly addresses the following security architecture objectives outlined in D7.2 MiCADO security architecture specification: O3.1, O5.2. Furthermore, the Crypto Engine supports a set of additional security objectives outlined in D7.2 MiCADO security architecture specification: O1.1, O3.3, O2.2, O5.1, O4.1.

Open Specifications traceability

The MiCADO Crypto Engine Security Module corresponds to Open Specification 4.2 in D7.3 Design of application level security classifications formats in principles.

4.3 Credential Manager

Security requirements traceability

The CM addresses the following requirements outlined in D7.1 COLA security requirements: CNSR-1, CNSR-3.

Architecture objectives traceability

The CM addresses the following security architecture objective outlined in D7.2 MiCADO security architecture specification: O4.2, O5.1

Open Specifications traceability

The MiCADO Image Integrity Verifier Security Module corresponds to Open Specification 4.4 in D7.3 Design of application level security classifications formats in principles.

4.4 Credential Store

Security requirements traceability

The CM addresses an extension for the requirements outlined in D7.1 COLA security requirements.

Architecture objectives traceability

The CM addresses the following security architecture objective outlined in D7.2 MiCADO security architecture specification: O5.1

Open Specifications traceability

The MiCADO Image Integrity Verifier Security Module corresponds to Open Specification 4.5 in D7.3 Design of application level security classifications formats in principles.

4.5 Zorp Firewall

Security requirements traceability

Zorp Firewall addresses the following requirements outlined in D7.1 COLA security requirements: SR05, SR06, SR10, CNSR-1, CNSR-2, CNSR-3, CNSR-4, CNSR-5, CNSR-6, CNSR-7, CNSR-8, CNSR-9, CNSR-10

Architecture objectives traceability

The Zorp Firewall addresses the following security architecture objective outlined in D7.2 MiCADO security architecture specification: O1.1, O4.2, O4.3, O4.4, O6.1, O6.2

Open Specifications traceability

The MiCADO Image Integrity Verifier Security Module corresponds to Open Specification 4.6 in D7.3 Design of application level security classifications formats in principles.

4.6 Security Policy Manager

Security requirements traceability

The Security Policy Manager addresses the following requirements outlined in D7.1 COLA security requirements: SR05, SR06, SR10, CNSR-1, CNSR-2, CNSR-3, CNSR-4, CNSR-5, CNSR-6, CNSR-7, CNSR-8, CNSR-9, CNSR-10

Architecture objectives traceability

The SPM addresses the following security architecture objective outlined in D7.2 MiCADO security architecture specification: O1.1, O4.2, O4.3, O4.4, O6.1, O6.2

Open Specifications traceability

The MiCADO Security Policy Manager Enabler corresponds to Open Specification 4.3 in D7.3 Design of application level security classifications formats in principles.

4.7 Master-Worker Secure Communication

Security requirements traceability

The Master-Worker Secure Communication mechanism addresses the following requirements outlined in D7.1 COLA security requirements: SR05, SR06, SR10, CNSR-1, CNSR-2, CNSR-3, CNSR-4, CNSR-5, CNSR-6, CNSR-7, CNSR-8, CNSR-9, CNSR-10

Architecture objectives traceability

The Master-Worker Secure Communication implementation addresses the following security architecture objective outlined in D7.2 MiCADO security architecture specification: O1.1, O4.2, O4.3, O4.4, O6.1, O6.2

Open Specifications traceability

The MiCADO Master-Worker Secure Communication Enabler corresponds to Open Specification 4.7 in D7.3 Design of application level security classifications formats in principles.

5 Summary and Conclusions

This document contains the implementation documentation of the security enablers delivered within the COLA project. The enablers provide functionality addressing various aspects of cloud security, such as: integrity verification of container images; generation of cryptographic material for authentication and authorization, network security (Transport Layer Security Enforcement and Termination, Firewalling, etc.), management and storage of cryptographic material and credentials. The enabler descriptions collected in this document accompany several earlier artefacts delivered in project COLA, namely the security enablers reference implementation, the security enablers earlier specification as well as the MiCADO security architecture specification. Together with the earlier artefacts, the MiCADO Security Modules Reference Implementation allows for a complete and fundamental understanding of the MiCADO security components. This creates the preconditions for the successful subsequent development of security functionality in MiCADO.

The document contains the description of the following security enabler implementations:

1. Image integrity Verifier – provides integrity security guarantees to the MiCADO infrastructure, primarily through integrity verification of application images prior to deployment. This functionality allows to detect corrupted images prior to their instantiation in the cloud.
2. Cryptographic Engine – provides a set of cryptographic material and algorithms to enforce the security of the communication between the components of the MiCADO system. The module implements the common cryptographic algorithms widely used in cloud deployments.
3. Credential Store – stores and protects security sensitive data required for operating the the MiCADO infrastructure. The credential store protects infrastructure secrets by encrypting them and restricting access to them.
4. Credential Manager – stores and manages the MiCADO user identities. It provides user verification used by the components performing authentication and access control.
5. Master Node Zorp Firewall – application level protocol proxy firewall, provides TLS and authentication for MiCADO Dashboard and TOSCA Submitter.
6. Security Policy Manager – a set of restful web API's providing access to MiCADO security enablers for TOSCA Submitter and Occopus Cloud Orchestrator.
7. Master-Worker Secure Communication – a secure communication channel for management communication between the Master Node and Worker Nodes.

The descriptions of the MiCADO security module reference implementation reflect the design and implementation decisions, trade-offs, limitations and exposed application programming interfaces of the security enablers. The reference implementations of the security enablers follow the open specifications outlined in the earlier Deliverable D7.4 Security policy formats specification. However, the current implementation description fills in the remaining potential knowledge gaps regarding implementation details of the security enablers.

6 References

[1] Cloud Orchestration at the Level of Application (COLA), Project 731574; D7.2 MiCADO security architecture specification, October 2017

[2] McKeen, Frank, et al. "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave." Proceedings of the Hardware and Architectural Support for Security and Privacy 2016. ACM, 2016.

[3] Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." Communications of the ACM 21.2 (1978): 120-126.

[4] Johnson, Don, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)." International journal of information security 1.1 (2001): 36-63.

[5] Python Cryptography Toolkit (pycrypto) https://pypi.org/project/pycrypto/

[6] UUID objects according to RFC 4122 https://docs.python.org/3/library/uuid.html

[7] Hashicorp Vault, https://www.vaultproject.io

[8] Hashicorp Vault API client for Python, https://github.com/hvac/hvac

[9] Hashicorp Configuration Language, https://github.com/hashicorp/hcl

[10] Flask-User, https://flask-user.readthedocs.io/en/latest/

[11] Flask-SQLAlchemy, http://flask-sqlalchemy.pocoo.org/2.3/

[12] Hashicorp Vault Key Rotation, https://www.vaultproject.io/docs/internals/rotation.html

[13] Token Policy, https://www.vaultproject.io/docs/concepts/policies.html#default-policy

[14] D6.2 - Prototype and documentation of the monitoring service

[15] D5.4 - First Set of Templates and Services of Use Cases