QKD MACsec

Automated deployment of MACsec

Automation Team

Date: 2024-09-03

Document version: 3.0

Document Control

Document Information

	Information
Document Name	QKD MACsec
Document Owner	Juniper PS
Issue Date	2024-09-03
Last Saved Date	2024-09-03
File Name	QKD_MACsec

Document History

Version	Issue Date	Changes
3.0	2024-09-03	Document Draft

Document Approvals

Role	Name	Signature	Date
Programme Director	William Mead		
Programme Manager	Mark Tickner		
PS Consultant	Andrea Terren		

Table of contents

1. Introduction

1.1 MACsec overview

It's an industry standard protocol defined in IEEE 802.1AE (Ethertype 0x88E5, supporting both GCM/AES/128 and GCM/AES/256).
MACsec offers line-rate layer 2 hardware based encryption on a hop-by-hop basis.
MACsec is supported on MX, PTX, and ACX series routers. MACsec is also supported on some EX and QFX series switches.

Please refer to the Juniper MACsec support online documentation for current Juniper Networks hardware and JunoOS software supporting MACsec.

MACsec is best used to secure device-to-device connections in a colocation center or a data center environment, building-to-building links in a city, or even city-to-city links traversing long distances including undersea or terrestrial cables. You can also encrypt Layer 2 connections via a provider WAN with MACsec, performing "hop-by-hop" encryption/decryption across multiple tunnels.

MACsec is enabled on a "per-interface" basis. Once enabled on an interface, it drops all frames except MACsec encrypted packets.
However, you can configure MACsec Should-secure (Fail Open Mode, default) to allow for unprotected traffic if MACsec negotiation should fail.

1.2 PQC and MACsec integration

Post-quantum cryptography (PQC) is designed to be resistant against attacks from quantum computers. Quantum computers have the potential to break many of the current cryptographic systems that are widely used today, including RSA and ECC (Elliptic Curve Cryptography).

PQC aims to provide an alternative to these cryptographic systems that is secure against quantum attacks.

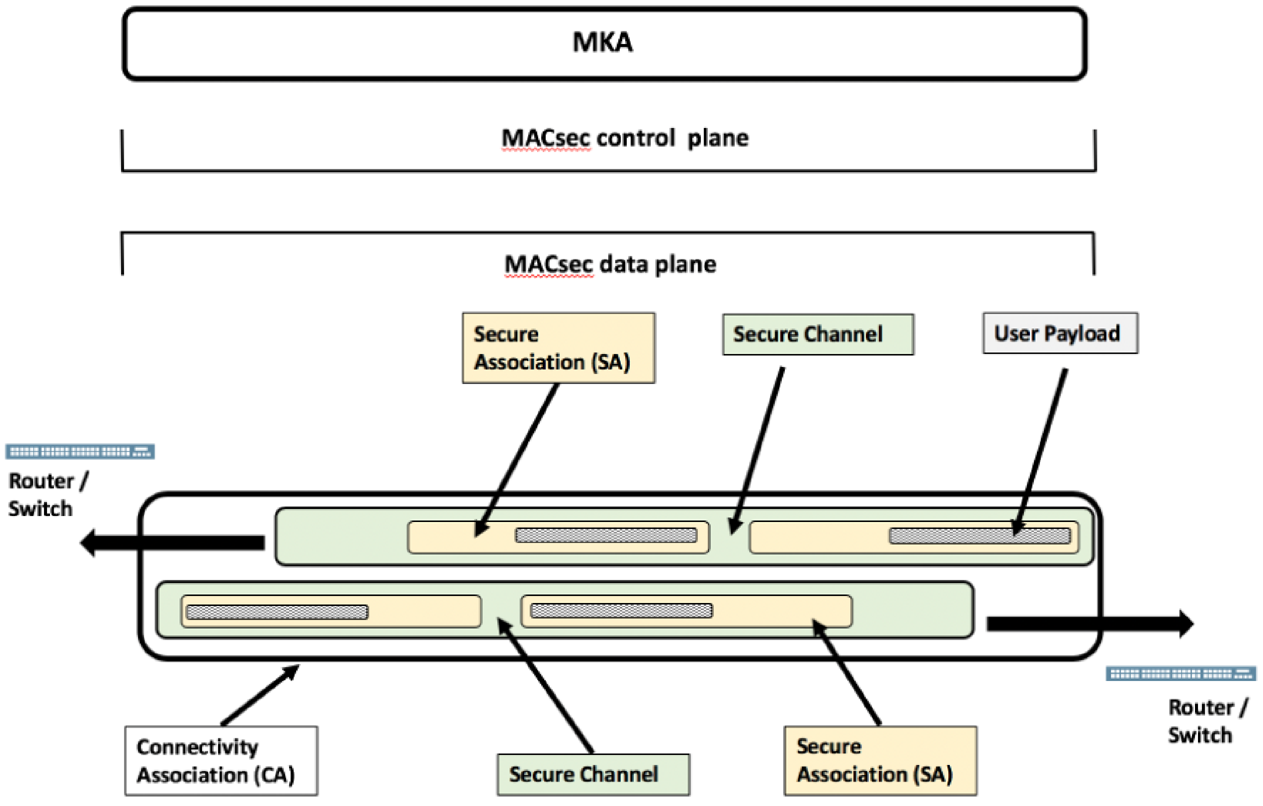
This documentation describes the implementation of MACsec together with quantum key distribution (QKD) networks.

Automated deployment of MACsec creates secure channel, generates keying material, and configures the routers/switches for each detected link between them. The script detects link changes and performs rekeying to provide a secure, configuration-free operation of MACsec.

1.3 Terminology and Entities involved

- **MACsec Key Agreement Protocol (MKA)**: Used to discover MACsec capable peers and used to negotiate encryption keys (for data encryption and for SAK encryption) - IEEE 802.1X-REV. MACsec security keys and sessions are managed by the MKA protocol software stack running on the Junos OS.This protocol is defined in IEEE standard 802.1X (specifically 802.1X-2010)
- **Connectivity Association (CA)**: Defines a secure relashionship between MACsec capable peers. It is a set of MACsec attributes that are used by interfaces to create inbound and outbound MACsec secure channels (or connections in simple terms) through which encrypted bi-directional traffic flows.
- **Programming the key** involves two items, CKN and CAK:
    - **Connectivity Association Key (CAK)**: static or dynamic key exchanged by MACsec speakers, this can be seen as a "primary key" that is used to derive all other session keys
    - **Connectivity Association Key Name (CKN)**: any name that defines a CAK
- **Primary and Fallback Keys**: primary key is used to negotiate an MKA; if this fails the fallback key is used
- **Security Association Key (SAK)**: derived from CAK used to encrypt data
- **Key Server**: generates SAK

A useful way to think of a MACsec-protected link is to split it into a control plane and a data plane.
The picture below illustrates the MKA control plane and the actual user data with MACsec wrappers, the data plane.



A connectivity association (CA) is a common MACsec relationship between all devices on a LAN who share a common key. In a MACsec association with two devices, a CA contains two unidirectional secure channels, one from each device. Finally, each secure channel has a series of SAs. In Junos OS, SAs last the shorter of either the time until the next key rollover in the chain or the wrap of the packet sequence counter. The operator configures the CAK, which secures the CA and from which Junos OS derives the SAK, which in turn secures each SA.

The initial IEEE MACsec standard (IEEE 802.1AE) describes the data plane functions and excluded key management.
An additional IEEE specification (802.1X 2010) provides the mechanism for devices to leverage the 802.1X specification to manage keys between devices.

1.4 Static CAK

You initially establish a MACsec secured link using a pre-shared key (PPK) when you are using static CAK security mode to enable MACsec.
A pre-shared key includes a connectivity association name (CKN) and its own CAK.
The CKN and CAK are configured by the user in the connectivity association and must match on both ends of the link to initially enable MACsec.

Once matching pre-shared keys are successfully exchanged, the MACsec Key Agreement (MKA) protocol is enabled.
The MKA protocol is responsible for maintaining MACsec on the link, and decides which switch on the point-to-point link becomes the key server. The key server then creates an SAK that is shared with the switch at the other end of the point-to-point link only, and that SAK is used to secure all data traffic

traversing the link. The key server will continue to periodically create and share a randomly-created SAK over the point-to-point link for as long as MACsec is enabled.

2. The core python MACsec/QKD script

The quantum key distribution (QKD) MACSEC script is written in python3.
The script allows the JUNOS devices to fetch key material from QKD/KME and update the MACsec static CAK accordingly.

2.1 Release Notes for MACSEC Configuration Script

2.1.1 Version 3.0.0

2.1.2 Overview

This release introduces significant improvements and additional functionality to the MACSEC Configuration Script. The script is designed for managing MACSEC configurations on Junos devices, fetching keys, and handling device settings with enhanced features, improved error handling, and more robust performance.

2.1.2.1 New Features

- **JSON/YAML inventories**: for parametric metadata, including:

1. certificates (pub/pri keys, RootCA)
2. QKD Keys/key-IDs
3. Provisioning of the basic MACsec configs applicable to different Juniper devices supporting MACsec, using config-set-commands and parametric data (i.e. keys, key-id, profile names, etc.)

- **REST API response**: calls are applied (instead of curl) commands for:

1. Verifications of HTTP/HTTPs return messages codes (4xx, 5xx)
2. syslog and trace messages (with timer profiles) upon return codes
3. Handling exceptions/errors
4. Success code verification and telemetry/monitoring after QKD successful setup

- **Enhanced Threading**: Improved the handling of concurrent device processing with dynamic thread management.
- **Configuration Validation**: Added validation for configuration files to ensure consistency and prevent errors.
- **Improved Error Handling**: Enhanced error handling and logging for better diagnostics and debugging.

- **Extended Logging Options**: Added additional logging options for detailed trace and debug information.

2.1.2.2 Features and Enhancements

- `get_args()`:
  - Added support for additional command-line arguments.
  - Improved argument parsing and validation.
- `initialize_logging(args)`:
  - Enhanced logging setup to support multiple log levels and output formats.
  - Added support for logging to both console and file outputs.
- `check_and_apply_initial_config(dev, targets_dict, log)`:
  - Improved the logic for applying initial configurations, including additional checks and optimizations.
  - Added more robust error handling and validation of configuration parameters.
- `process(dev, log)`:
  - Enhanced key fetching logic to handle different scenarios and improve performance.
  - Added detailed logging and debug output for better visibility into the processing steps.
- `fetch_kme_key(session, local_name, log, remote_mnmgt_add, kme_url, key_id=None)`:
  - Improved error handling for key fetching operations.
  - Added support for additional parameters and configurations.
- `save_key_ids(key_dict)`:
  - Added support for encrypted storage of key IDs.
  - Improved file handling and error management.

2.1.2.3 Configuration

- `targets_dict`: The dictionary that has to be updated will the following information consumed by the script:
  - device specific information like: ip, interfaces (that forms the MacSec channel), associated KME (dns name and ip)
  - CA Server information like: secrets, ip, CA certificate path, names
  - QKD/KME devices roles: master, slave and additional slave (assumed that the QKD/KME devices provider supports additional_slave_SAE_IDs parameter in the case of more than 2 devices used in the MacSec chains. -> please refer to the Quantum Key Distribution - Protocol and data format of REST-based key delivery API)
  - system information like:
    - maxthreads (in case of off-box multithreading execution)
    - event-options ... <start-time start time>

2.1.2.4 Usage

1. Offbox

1. **Command-Line Arguments**:

    - `targets`: List of target hosts to process.
    - `-t`, `--threads`: Number of threads to use for concurrent processing.
    - `-v`, `--verbose`: Increase verbosity level for logging.
    - `-tr`, `--trace`: Dump debug-level logs to a trace.log file.

2. **Manual ETSIA_v3.0.py Script Execution - Example Commands**:

    - **Run Script with Default Settings from the `targets_dict`**(see the section 2.1.2.3 Configuration):

    ```
    python3 ETSIA_v3.0.py
    ```

    - **Run Script with Specified Number of Threads**:

    ```
    python3 ETSIA_v3.0.py -t 4
    ```

    - **Run Script with Verbose Logging**:

    ```
    python3 ETSIA_v3.0.py --verbose
    ```

    - **Run Script with Trace Logging**:

    ```
    python3 ETSIA_v3.0.py --trace
    ```

3. **Scheduled ETSIA_v3.0.py Script Execution using Crontab**:

    Users' crontab files are named according to the user's name, and their location varies by operating systems. In Red Hat based distributions such as CentOS, crontab files are stored in the /var/spool/cron directory, while on Debian and Ubuntu files are stored in the /var/spool/cron/crontabs directory.
    Although you can edit the user crontab files manually, it is recommended to use the below crontab command (will run the script every 10 minutes):

    ```
    # crontab -u <user_name> -e
    */10 * * * * python3 ETSIA_v3.0.py
    ```

2. Onbox

    1. Import the ETSIA_v3.0.0.py and profiling Profile python scripts under /var/db/scripts/event/.py
    2. Manually run the script one time from the device and it will apply the needed even-options configuration or manually configure the event-options on the device (see 2.2.9.2 Event-options section) and the script will start running every <time-interval seconds> starting with <start-time start-time>.

2.1.2.5 Error Handling

- **Logging**: Improved logging for capturing detailed error and debug information.
- **Exceptions**: Enhanced exception handling for robustness and reliability during device processing and configuration.

2.1.2.6 Known/Potential Issues/Limitations

- **Scaling Performance**: While scaling tests have been implemented, performance optimizations for very large configurations may still be required.
- **renew_certificates function**: Current version of this function is not running with onbox execution of the script because pyOpenSSL is not part of the Junos python3 modules.

2.1.2.7 Contribution

For contributions, bug reports, or feature requests, please contact the development team at quantum_computing_team@juniper.net.

2.1.2.8 License

This script is licensed under the MIT License.

Thank you for using the MACSEC Configuration Script! For more information, please refer to the documentation.

2.2 Script details

The script is supposed to be kept at /var/db/scripts/event/.py, and it can be scheduled using configuration "event-options" in JunOS.
The current version of the script is 3.0 and the user is harcoded as "remote".

The script discriminates if it's going to be run "on-box" or "off-box", i.e. whether it's going to be running via "event python scripting" installed into the Juniper devices (e.g. an MX device) or if it's going to manage keys retrieval within an external device (likely to be installed within the KME itself or on a separate Linux target host).

The script sets a number of parameters to establish TLS connectivity with a RootCA Certification Authority.
Key material is exchanged via TLS between the KME and SAE using the KME available REST APIs (depending on the 3rd party QKD vendor compute appliance, different REST APIs might be available to

use).
The ETSI 014 QKD REST API is a specification developed by the European Telecommunications Standards Institute (ETSI) for the management of QKD networks. This API provides a standardized interface for communication between different QKD network components, such as QKD devices, key servers, and network components such as
routers, switches, and firewalls.

The REST APIs would typically be used to facilitate key management and distribution between a KME and an SAE.
These REST APIs are part of a higher-level Key Management System (KMS) that integrates with the QKD system.

The API is in the context of this document mainly used to manage the distribution of the keys and key IDs generated by the QKD devices.

In this QKD-enabled environment, these APIs would interact with the QKD system to ensure that the keys being managed and distributed are derived from quantum-safe processes.
The KME would serve as the intermediary between the QKD system and the application layer, ensuring that keys are securely generated, distributed, and managed.

2.2.1 Entities involved in the communication

- Key Management Entity (KME) is the entity that manages keys in a network (it is in cooperation with one or more other KMEs);
- Secure Application Entity (SAE) is the entity that requests one or more keys from a KME (e.g. a SAE is an MX Juniper router, an EX Juniper switch, etc.);
- QKD Entity (QKDE) is the entity providing key distribution functionality (e.g. a QKD Module - key manager implementing an underlying QKD protocol).

All SAEs and KMEs must possess an identifier that allows to uniquely identify them in the network.

2.2.2 REST APIs used in key exchange

Below, an outline of the main REST APIs that would be used for key and KeyID exchange in such a system:

1. Key Request API

   Endpoint: /api/v1/keys/request
   Method: POST
   Description: This API is used by the SAE to request a new key from the KME. The request may include parameters such as the desired key length, key type, and any specific requirements related to the application.

2. Key Retrieval API

   Endpoint: /api/v1/keys/{key_id}
   Method: GET

Description: This API allows the SAE to retrieve a specific key using the key_id provided during the key request or through other methods.

3. Key Deletion API

   Endpoint: /api/v1/keys/{key_id}
   Method: DELETE
   Description: This API allows the SAE or KME to delete a key that is no longer needed. This may be used to securely dispose of a key once it has been used or is no longer valid.

4. Key Status API

   Endpoint: /api/v1/keys/{key_id}/status
   Method: GET
   Description: This API allows the SAE to check the status of a specific key, such as whether it is active, expired, or revoked.

5. Key Exchange API

   Endpoint: /api/v1/keys/exchange
   Method: POST
   Description: This API facilitates the exchange of keys between different entities (e.g., from one SAE to another) under the supervision of the KME. This is useful in scenarios where keys need to be shared securely between different endpoints.

6. Key ID Management API

   Endpoint: /api/v1/keyids
   Method: POST, GET, DELETE
   Description: This API allows the creation, retrieval, and deletion of KeyIDs, which are used to reference specific keys within the KME system. KeyIDs are essential for tracking and managing the lifecycle of keys in the system. Tipically :

   - OPEN_CONNECT (in source, in destination, inout QOS, inout Key_stream_ID, out status)
   - GET_KEY (in Key_stream_ID, inout index, out Key_buffer, inout Metadata, out status)
   - CLOSE (in Key_stream_ID, out status)

7. Audit Logs API

   Endpoint: /api/v1/logs/audit
   Method: GET
   Description: This API provides access to audit logs related to key management activities, including key creation, retrieval, exchange, and deletion. This is critical for maintaining security and compliance.

2.2.3 QKD-MACsec key exchange automation

The python script implemented for QKD/MACsec integration is parametric and can be used for on-box (i.e. running as an event script inside a Juniper device, and scheduled using event/options in JUNOS), or off-box, on a remote server, and scheduled via Linux cron daemon.

Static CAK mode is implemented for links connecting switches or routers. Static CAK mode ensures security by frequently refreshing to a new random security key and by sharing only the security key between the two devices on the MACsec-secured point-to-point link.

When you enable MACsec using static CAK mode, two security keys are used to secure the link:

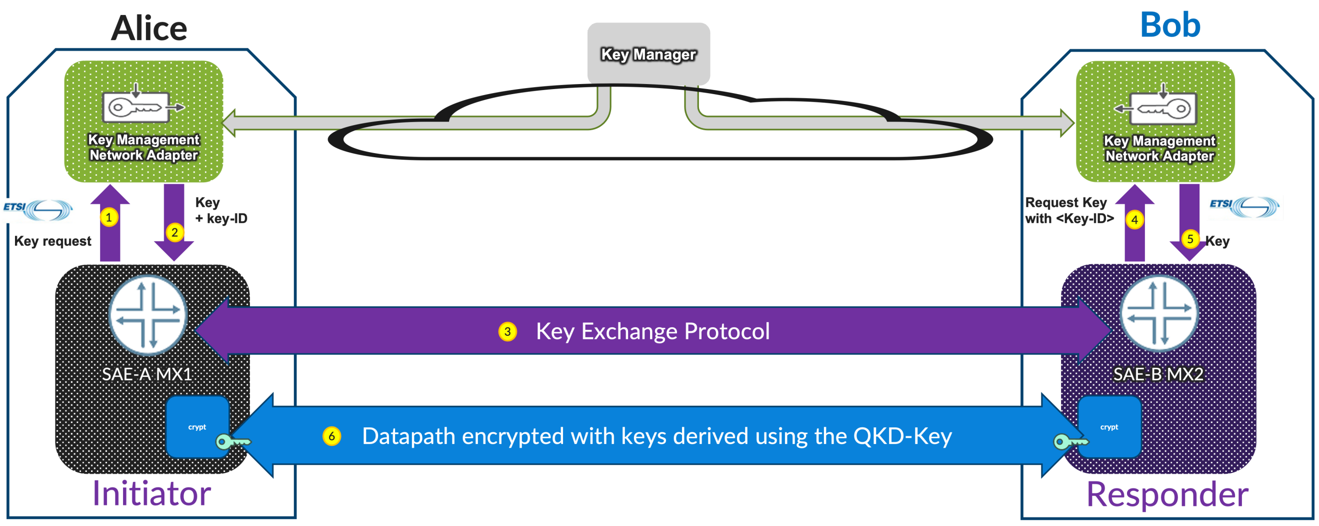
1. a connectivity association key (CAK) that secures control plane traffic and
2. a randomly-generated secure association key (SAK) that secures data plane traffic.

Both keys are regularly exchanged between both devices on each end of the p2p Ethernet link to ensure link security.

2.2.3.1 Key exchange mechanism between 2 Juniper devices

The ETSI standards setting organization defined a REST API to exchange keys between a Key Management Server [KMS] (typically included in the QKD system) and a security application entity [SAE] such as Juniper MX.

The API itself is secured by TLS (issuing client certificates via an available PKI CA Root Certification Authority), however this is not Quantum-safe.
For that reason, it is only allowed to be operated in a physically secure compartment. Such compartment can for example be a rack with locked doors or a locked room.
In practice this means that the QKD system and the secure application device are in proximity to each other and do not exchange key material over the Internet.



Alice
Key Manager
Bob
Key Management Network Adapter
Key Management Network Adapter
ETSI
1
Key request
2
Key + key-ID
Request Key with <Key-ID>
4
5
Key
ETSI
SAE-A MX1
SAE-B MX2
3 Key Exchange Protocol
crypt
crypt
6 Datapath encrypted with keys derived using the QKD-Key
Initiator
Responder

In a typical scenario, each SAE must first identify itself to the local KME to start setting up a quantum secure MACsec L2 tunnel.

```
MX1 = SAE-A
MX2 = SAE-B
QKD1 = KME-A
QKD2 = KME-B
```

1. The two SAEs need to elect an initiator and a responder. In the Juniper implementation this tie-breaking is part of the IKEv2 initialization process and is defined in the `targets_dict` (see the section 2.1.2.3 Configuration) from the ETSIA script. In the following we assume "SAE-A" is the initiator (master) role and "SAE-B" as responder (slave).
2. Certificates are exchanged between SAEs (the Juniper router or switch) and the KMEs (the QKD server generating the quantum key material) for TLS communication exchange between them.
3. The SAE indicates its preference to use QKD-keys to the MACsec peer (e.g.: MX1 being a primary device TLS queries the key and keyID pair from QKD1. MX1 will configure MACsec with the key fetched from QKD1).
4. If the peer also supports the use of QKD-keys, the Key exchange process gets scheduled to fetch keys.
5. SAE_A requests a `key` and `key-ID` from its local KME
6. SAE_A receives a `key` and `key-ID`
7. SAE_A communicates the `key-ID` to SAE_B: MX1 sets the keyID at the interface description. This information is available to MX2 either through LLDP (enabled on both devices) or via scp secure copy from remote QKD/KME device. It is assumed each QKD/KME shares the same key-id at anyone time.
   KeyID was previously carried over LLDP protocol, however keyID transport via OUI Ethernet type 0x88b7 maybe be supported in the future. A new methodology that can outperform LLDP in terms of key ID transport accross the non-quantum channel between SAE_A and SAE_B, is the SCP protocol, which is implemented in this v3.0.0.0 script. SCP guarantees encryption of the KEY-ID material and negligible delay/jitter if compared to LLDP timers. Also LLDP states updates are near-second, therefore SCP performances outlines LLDP in this case.
8. SAE_B requests the `key` corresponding to the received `key-ID`
9. The local KME delivers the `key` to SAE_B
10. SAE_B confirms having received the `key` corresponding to the `key-ID`: MX2 has used the keyID received from MX1 to fetch the corresponding key from KME2 and configures MACsec accordingly.
11. Now SAE_A and SAE_B are ready to use the shared key in their communication.
12. After both sides are in possession of their shared keys, those are mixed with the existing SA key materials to get Quantum safe keys.
13. When the MACsec keys are synced on both ends, a hitless rollover happens at regular intervals, to guarantee an automatic re-keying mechanism.

Although the QKD Key manager module is designed for Key exchange syncronization, both MX1 and MX2 SAE devices receive their keys asyncronously via a non-quantum channel.
There is in fact a deterministic timeframe where the configured keys will be out of sync on both Juniper devices.

To guarantee MACsec tunnel running during that period, the existing MACsec session will continue using the preceeding key.

The output below, shows the Display MACsec Key Agreement (MKA) session information to see the number of MKAs that are in progress, connectivity association key (CAK) type, CAK status, and MKA packet count activity.
See the security mka sessions summary for reference.

```
lab@SAE_A> show security mka sessions summary
Interface    Member-ID                  Type       Status        Tx
Rx         CAK Name
ge-0/0/1     1F51BB9778BCB24D17F4BE68   primary    live          34
3
ABCD1234ABCD5678ABCD1234ABCD56786C00260045EBAF1EBB0D41FBDD9FF375
ge-0/0/1     D0AC8A5F4D5A39B5DD9F649B   preceding  active        97
63
ABCD1234ABCD5678ABCD1234ABCD5678B4C9C6A1426AB71B1D53479F3D795B0D
```

2.2.4 Script main functions

The initial monolithic Python script has been refactored into a modular structure with separate functions, to improve readability, maintainability, and reusability. The section below details each function and its use.

2.2.4.1 Main KME/SAE interaction

The `main` function in this script is the central component that coordinates the execution of different parts of the program.
It handles setting up logging, defining target devices and their respective information (ex: interfaces, kme, connectivity-association etc.) forming a 121 tunnel for a number of channelled devices, and determining whether to execute the code on-box or off-box.
It also manages threading to process multiple devices concurrently (used only on the off-box implementation).

2.2.4.2 Detailed View

1. **Global Variables**:
    - `global prof`
        - Declares that `prof` is a global variable, which might be used for performance profiling or logging purposes.
2. **Parse Arguments**:
    - `args = get_args()`
        - Calls the `get_args` function to parse command-line arguments and stores the result in `args`.

3. **Initialize Logging**:
    - `log = initialize_logging(args)`
        - Initializes logging based on the parsed command-line arguments and assigns it to `log`.

4. **Define Target Devices**:
    - `targets_dict`:
        - A dictionary that contains information about various target devices. Each device entry includes details like IP addresses, interfaces, and KME (Key Management Entity) information.
        - **Example Entries**:
            - **CA_server**: Configuration for the CA server with details such as IP address, path, and credentials.
            - **mx001**, **mx002**, **mx003**, **mx004**: Various devices with their respective IP addresses, interfaces, and KME information.

5. **Process Off-Box**:
    - Checks if `onbox` is `False` (indicating an off-box approach):
        - Logs and prints that an off-box approach is being taken.
        - **Thread Management**:
            - Creates a list of master, slave and additional slave devices.
            - Divides the devices among threads based on the number of available threads (`maxthreads`).
            - Creates and starts threads to process these devices using `req_thread`.
            - Waits for all threads to complete using `join`.

6. **Process On-Box**:
    - Checks if `onbox` is `True` (indicating an on-box approach):
        - Logs and prints that an on-box approach is being taken.
        - **Device Processing**:
            - Opens a device connection using the `Device()` context manager.
            - Checks and renews certificates if necessary.
            - Applies initial MACsec and event-driven configuration and processes the device.

7. **Error Handling**:
    - Both in off-box and on-box approaches, exceptions are caught and logged if any errors occur during the device processing.

8. **Script Entry Point**:
    - `if __name__ == '__main__':`
        - Ensures that the `main` function is called only when the script is executed directly, not when imported as a module.

    - `main()`
        - Executes the `main` function.
    - `prof.close()`
        - Closes the profiling or logging instance (assuming `prof` is a profiler or logger that needs to be closed).

The `process` function interacts with a JUNOS device to fetch keys from a Key Management Entity (KME) and update the MACsec CAK (Connectivity Association Key) configuration accordingly. Below is a detailed breakdown of what the function does:

2.2.4.3 Function Purpose

- **Purpose**: To fetch keys from a KME and apply or update the MACsec CAK configuration on a JUNOS device. This includes checking existing configurations, retrieving necessary information, and applying new configurations.

2.2.4.4 Parameters

- `dev`: The JUNOS device object used to interact with the device's configuration.
- `targets_dict`: A dictionary that contains information about various target devices. Each device entry includes details like IP addresses, interfaces, and KME (Key Management Entity) information.
- `log`: A logging object used to record informational messages and errors.

2.2.4.5 Function Steps

1. **Define Configuration Filter**:
    - Creates an XML filter to retrieve the MACsec configuration and interfaces from the JUNOS device.
2. **Fetch and Log Current Configuration**:
    - Retrieves the current configuration using the defined filter.
    - Logs the configuration in a human-readable format.
3. **Parse MACsec Configuration**:
    - Extracts the MACsec configuration from the retrieved XML.
    - Finds interfaces that have the `qkd` apply-macro.
4. **Handle Missing Interfaces**:
    - Logs a critical error and exits if no interfaces with `qkd` apply-macro are found.
5. **Initialize Key Retrieval**:
    - Creates a session for HTTPs requests.
    - Retrieves previously saved key-IDs, or if key-ID is not provided, sends a POST request to the KME API endpoint for encryption keys (enc_keys). The request includes additional slave

SAE IDs if specified.
   - Initializes placeholders for new MACsec and interface configurations.

6. **Process Each Device**:

   - Iterates through all the devices extracted from the `targets_dict` (in the case of off-box) or actual device (in the case off on-box) and depending on the roles handle it respectively:
     - **Master Device Handling**:
       - Fetches keys from the KME if the local device is the master.
       - Updates the MACsec configuration with the received key.
     - **Slave Device Handling**:
       - Attempts to retrieve key-IDs from master, using SCP, if the local device is a slave.
       - Validates and fetches keys from the KME if necessary.
     - **Saves the key IDs**:
       - Get the Key_ID from the master device.
       - Key-ID are saved to a JSON file if present.
       - Retrieves the previous key IDs from a JSON file.

7. **Update MACsec Configuration**:

   - Updates the connectivity association name and key information in the MACsec configuration XML.
   - Adds the updated configurations to the list of configurations to be committed.

8. **Commit Configuration**:

   - Commits the updated configuration to the JUNOS device if there are changes.
   - Handles errors and logs success or failure.

9. **Save Key IDs**:

   - Saves the new key IDs to a persistent JSON file.

10. **Logging and Completion**:

    - Logs the success of the script execution.

2.2.5 Multi threading function

The `request_thread` function designed to manage Juniper network devices, performing tasks such as renewing certificates and applying initial MACsec configurations, assuming an environments where multiple Juniper network devices need to be configured or updated simultaneously, reducing the time needed to manage large networks.
Each instance of this function is intended to run in its own thread, allowing for parallel processing of multiple Juniper devices at once. It connects to each device via SSH, checks local and Root CA certificates, and applies configurations as needed.
A configuration function checks the device's MACsec initial configuration and applies any necessary changes, using the `targets_dict` dictionary which contains specific JunOS MX initial MACsec configuration parameters for each device. It's also profiled for performance measurement, and logs its

progress for any errors encountered, ensuring that each step can be tracked and debugged if necessary.

2.2.6 Certificate life cycle management (LCM)

In case a RootCA isn't provided, a `generate_certificate` function automates the process of generating a self-signed CA certificate and its associated private key. It sets various attributes like the subject name, certificate version, validity period, and extensions. The resulting certificate and key are then saved to specified file paths for later use in a Public Key Infrastructure (PKI).
Note: this function can be currently used only on the off-box implementation of the script (PyOpenSSL not being part of the Junos python3 modules).

2.2.6.1 Generate client certificate function

It automates the process of generating a client certificate signed by the aforementioned (or any existing) Root CA certificate. It handles the generation of the client's private key, the creation of the client certificate with necessary details and extensions, and then signs the certificate using the CA's private key. Finally, it saves both the client certificate and the private key to the specified file paths.

2.2.6.2 Is certificate valid function

It checks whether a given X.509 certificate is valid for at least a specified number of days. This function helps ensure that a certificate is not about to expire and provides a way to proactively manage certificate validity.

2.2.6.3 Create SSH client function

It is used to establish an SSH (Secure Shell) connection to a remote device. It leverages the paramiko library, which is a Python implementation of the SSH protocol, to manage the connection. This function is essential for automating interactions with remote devices over SSH.

2.2.6.4 Get certificates function

It is used to obtain certificate files either locally from the device where it is running (e.g. a Juniper MX) or remotely from the device over SSH connection. It constructs file paths based on the environment and handles the retrieval process accordingly. If running on the device (**onbox** is **true**), it simply constructs and returns local file paths. If running externally (**onbox** is **false**), it connects to the device via SSH, downloads the certificate files, and then returns the local paths for these files.

2.2.6.5 Upload certificates function

It handles the process of uploading certificate files from a local machine to a remote device using SCP over SSH. It starts by creating an SSH client connection to the device, then uses SCP to transfer the certificate and key files to the device's file system. If any issues arise during the upload process, they are caught and reported. This function is crucial for scenarios where certificates need to be deployed or updated on remote devices.

2.2.6.6 Fetch CA certificate function

It is used to connect to a remote CA server via SSH, download the CA certificate and key files, and save them locally. The function handles establishing the SSH connection, performing the SCP file transfer, and handling any errors that might occur during the process. It returns the local file paths where the downloaded certificate and key are stored.
The local path where the certificates will be stored is determined based on whether the code is running on the device (onbox) or not.

```
* CERTS_DIR: Directory used if running on the device.
* OFFBOX_CERTS_DIR: Directory used if running off the device.
```

2.2.6.7 Renew certificates function

It is responsible for managing the lifecycle of certificates used by devices. It checks if the existing certificates are valid or if they are missing. If necessary, it fetches the CA certificate, generates new client certificates, and uploads the updated certificates to the device. This ensures that devices have up-to-date certificates for secure communication. If the onbox flag is False, indicating that the script is not running on the device, the function retrieves the current paths to the CA and client certificates and keys using the get_certificates function. This function provides the paths to these certificates that may need to be renewed.

2.2.6.8 Should check certs function

It determines whether it is time to check the validity of certificates based on the current date and time. The function determines if it's time to perform a certificate validity check based on:

```
- Whether the current time is between midnight and 1 AM.
- Whether the current day of the month is a specific day (i.e., the 1st,
6th, 11th, 16th, or 21st).
```

This scheduling mechanism ensures that certificate validity checks occur at regular intervals during specific hours of the day, helping to maintain the validity and security of certificates without manual intervention.

2.2.7 Logging and tracing

2.2.7.1 Initialize logging function

It sets up logging with various configurations:

```
* Defines a custom log level (NOTICE).
* Adds methods to the Logger class for this custom level.
* Configures logging based on user verbosity and other options (trace
file, on-box logging).
```

```
* Sets up both console and file handlers, with different levels and
formats as required.
* Provides feedback on successful initialization and returns the
configured logger instance.
```

2.2.7.2 Get Args function

It is designed to define and parse command-line arguments for a Python script. It uses the `argparse` module to handle command-line arguments, allowing users to specify various options when running the script. Here's a breakdown of the function:

2.2.8 Initial configuration

2.2.8.1 Check and apply initial config function

It ensures that an initial MACsec (Media Access Control Security) configuration is applied to a network device if it hasn't been already. Here's a detailed explanation of how the function works:

2.2.8.1.1 Function Purpose

- **Purpose**: To check if the initial MACsec configuration is already applied to a device and, if not, to apply it.

2.2.8.1.2 Parameters

- `dev`: An object representing the network device, which provides methods for interacting with the device's configuration.
- `targets_dict`: A dictionary containing target device details, including IP addresses, interfaces, and KME (Key Management Entity) details.
- `log`: A logging object used to log informational messages and errors.

2.2.8.1.3 Function Steps

1. **Extract Device and Configuration Details**:
    - **`device_name`**: The hostname of the device from the `dev` object.
    - **`device_ip`**: The IP address of the device from `targets_dict`.
    - **`c_a`**: The connectivity association name used in the MACsec configuration.
    - **`interfaces`**: A list of interfaces on the device that will be configured.
    - **`kme_name`**: The name of the KME.
    - **`kme_ip`**: The IP address of the KME.
2. **Check Existing Configuration**:
    - Uses the device's RPC (Remote Procedure Call) to retrieve the current configuration related to the MACsec connectivity association.

    - Checks if the configuration is already applied by looking for the presence of a `<name>` element for a specific configuration filtered by security.macsec.connectivity-association[name='c_a']. If found, logs a message and exits the function early.

3. **Define Initial MACsec Commands**:
    - **Basic Commands**: Defines a set of commands for setting up the MACsec connectivity association, including cipher suite, security mode, and pre-shared keys.
    - **Interface-Specific Commands**: Extends the list of commands to configure each interface with MACsec settings, static host mappings, SCP, and event options.

4. **Apply Configuration**:
    - **Logging**: Logs a message indicating the start of the configuration process.
    - **Apply Commands**:
        - Sets a timeout for the device interaction.
        - Uses a context manager (`Config`) to lock the configuration, load the commands, and commit the changes.
        - Logs success if the configuration is applied successfully, or logs an error if there is an issue.

5. **Cleanup and Delay**:
    - Unlocks the configuration to allow further changes.
    - Introduces a 60-second sleep to ensure that the configuration is applied consistently across all devices.

2.2.9 Configurations

2.2.9.1 Macsec configuration function

A list of interfaces are considered as input values to start the MACsec tunnel.

It's recommended that CKN length be 64 hex digits (32 octets) irrespective of the cipher suite used (128 bits or 256 bits).
The length of CKN is flexible although there is a requirement that it has to be between one to 32 octets long, which would mean two to 64 hex digits. An odd number of hex digits is not permitted.

We recommend using full-length CAKs. This means 32 hex digits (16 Octets, 128 bits) for 128-bit AES, and 64 hex digits (32 octets, 256 bits) for 256-bit AES.
CAK length is extremely important for interoperability with third party MACsec devices because a smaller-than-required CAK could result in incompatible padding of zeroes by a third-party vendor when compared to Juniper.

The following CAK and CKN values are given as an example reference, to represnet the correct string format type&lenght when configured on a Junier device.

`ckn_example` = abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678

`cak_example` = abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678

The kme_a-qkd-remote-server-FQDN is the FQDN of the remote KME_A.

The following code, specifies protocols whose packets are not secured using Media Access Control Security (MACsec) when MACsec is enabled on a link using static connectivity association key (CAK) security mode. When this option is enabled in a connectivity association that is attached to an interface, MACsec is not enabled for all packets of the specified protocols that are sent and received on the link.

See the Juniper MACsec exclude-protocol reference documentation online.

```
#set security macsec connectivity-association CA_basic exclude-protocol
lldp
```

```
set security macsec connectivity-association {c_a} cipher-suite gcm-aes-
xpn-256
set security macsec connectivity-association {c_a} security-mode static-
cak
set security macsec connectivity-association {c_a} pre-shared-key ckn
<ckn_example>
set security macsec connectivity-association {c_a} pre-shared-key cak
<cak_example>

set security macsec interfaces {interface} apply-macro qkd kme-ca false
set security macsec interfaces {interface} apply-macro qkd kme-host
{kme_a-qkd-remote-server-FQDN}
set security macsec interfaces {interface} apply-macro qkd kme-port 443
set security macsec interfaces {interface} connectivity-association
{c_a}
set security macsec interfaces {interface} apply-macro qkd kme-keyid-
check true
set system static-host-mapping {kme_name} inet {kme_ip}
set system static-host-mapping {device_name} inet {device_ip}
```

2.2.9.2 Event-options

```
# devices must be in sync with NTP
set event-options generate-event every10mins time-interval 600 <start-
time start-time>
set event-options policy qkd events every10mins
set event-options policy qkd then event-script onbox.py
set event-options event-script file onbox.py python-script-user remote
set event-options traceoptions file script.log
set event-options traceoptions file size 10m
```

2.2.9.3 Other necessary config

```
#set system host-name SAE_A
set system static-host-mapping [kme_a-qkd-remote-server-FQDN] inet [qkd-
management-IP-addr]
set system static-host-mapping [DEVICE_A] inet [DEVICE_A_IP]
```

2.2.9.4 Description of the macros

apply-macro	Use
kme-ca (true, false)	When set to true, uses HTTPS(SSL/TLS) to verify connection to KME
kme-host	Hostname or IP address of the KME connected to the SAE. Make sure you add a static host-name Junos config if you use hostname
kme-method (get, post)	Specifies the method used to fetch key with keyID in secondary device
kme-port	Port used to talk to KME. Should be set to 443, unless you need to experiment
kme-keyid-check (true, false)	If set to false, get/post call to KME will be made even if KeyId is stale. This macro is ignored for Primary SAE

2.2.10 Key Management functions

2.2.10.1 Fetch KME key function

It is responsible for fetching keys from a Key Management Entity (KME) using HTTP requests.

1. **Function Parameters**:
    - `session`: An active `requests.Session` object used for making HTTP requests.
    - `local_name`: The local identifier used to derive paths for client certificate and key.
    - `log`: A logger instance for logging messages and errors.
    - `remote_mnmgt_add`: The remote management address to be used in the KME URL.
    - `kme_url`: The base URL of the KME.
    - `key_id`: An optional parameter specifying the ID of the key to fetch.
    - `additional_slave_SAE_IDs`: an optional parameter used for specifying two or more slave SAEs to share identical keys.
2. **Certificate and Key Paths**:
    - The function constructs paths for the client certificate (`client_crt`) and client key (`client_key`) based on the `local_name` and the directory `CERTS_DIR`.
    - `CLIENT_CERT` is a tuple containing the paths to these certificate and key files.
3. **Fetching Keys**:

    - If `key_id` is provided, the function constructs a URL to request decryption keys (`dec_keys`) from the KME, appending the `key_id` to the base URL.
    - If `key_id` is not provided, the function constructs a URL to request encryption keys (`enc_keys`).
    - The constructed URL includes the `remote_mnmgt_add` for the remote management address (peer SAE).
    - The `requests.Session.get()` method is used to perform the HTTP GET request. The request includes the `verify` parameter set to `CA_CERT` to specify the CA certificate for SSL verification and `cert` set to `CLIENT_CERT` to provide client-side SSL credentials.

4. **Error Handling**:

    - If an exception occurs during the request (such as connection errors or invalid responses), the function logs an error message and returns `None`.

5. **Return Value**:

    - If the request is successful, the function returns the JSON response from the KME, which contains the requested keys.

2.2.10.2 Get previous key ids function

It performs the following tasks:

- Attempts to open and read a JSON file to retrieve previous key IDs.
- Handles scenarios where the file is missing or contains invalid data by logging appropriate messages.
- Returns an empty dictionary if the file is not found or if it contains no valid key IDs.

This approach ensures that the function is resilient to common issues like file absence or data corruption, while providing informative logs to help diagnose such issues.

2.2.10.3 Save key ids function

It is designed to save a dictionary of key IDs to a JSON file.

3. Appendix

3.1 Glossary of MACsec Concepts

3.1.1 AN - Association Number

A two-bit value concatenated with a Secure Channel Identifier to identify a Secure Association. Used to easily identify if a Secure Association Key (SAK) has rolled over.

3.1.2 CA - Connectivity Association

A long-term relationship between endpoints that share common MACsec parameters, such as a key. On a router-to-router link, a CA would contain two unidirectional secure channels.

3.1.3 CAK - Connectivity Association Key

A long-term key either configured statically or assigned dynamically via RADIUS.

3.1.4 CKN - Connectivity Association Key Name

A hexadecimal Key Name value that identifies a CAK. It can be a short (but an even number of characters) set of values shared by participating endpoints when agreeing on which CAK to use.

3.1.5 ICV - Integrity Check Value

A field similar to the CRC, which protects the destination and source MAC addresses of the Ethernet packet and everything in between. The Integrity Check Value (ICV) validates that the MACsec packet is correct in both encrypted and unencrypted MACsec sessions.

3.1.6 MKA - MACsec Key Agreement

A protocol used between MACsec peers to synchronize keys. MKA securely encrypts and distributes the SAK between peers, handles the rollover of keys, and includes a built-in polling interval to allow for key server redundancy. MKA uses the EAP over LAN EtherType of 0x888e.

3.1.7 SA - Secure Association

A secure channel has a succession of secure associations (SAs), each with its own Secure Association Number (AN). A Secure Association Key (SAK) is unique for each SA and is derived from the long-term CAK. The Junos OS rolls over to the next SA either during hitless rollovers and keychain usage or when packet sequence numbers would roll over.

3.1.8 SAK - Secure Association Key

A key derived from the Connectivity Association Key (CAK) and relevant only for the duration of the Secure Association between two peers. The SAK is securely distributed between MACsec peers.

3.1.9 SC - Secure Channel

A unidirectional channel from one host to one or more hosts on the same broadcast domain. Typically, on point-to-point links, an SC has one recipient and includes a succession of secure associations (SAs).

3.1.10 SCI - Secure Channel Identifier

A concatenation of the Secure Channel sender's MAC address and a port ID. The SCI remains constant through secure association rollovers within a secure channel.

3.1.11 SecTAG - MAC Security Tag

The MACsec header, starting with the MACsec Ethertype of 0x88e5, follows the standard destination and source MAC fields on any Ethernet packet.

3.1.12 XPN - Packet Number and eXtended Packet Number

A sequence number increasing by one for each packet sent within an SA. The initial MACsec specification used a 32-bit identifier, necessitating frequent SA renegotiation on high-speed links. eXtended Packet Numbering (XPN) increases the packet sequence number space to 64-bits.

3.2 Imported Packages and Modules Synthesis

3.2.1 `requests`

- **Purpose**: A popular HTTP library in Python used to send HTTP requests (GET, POST, etc.) to web servers and handle the responses.
- **Common Usage**: Making REST API calls, downloading content from the web, and interacting with web services.

3.2.2 `jnpr.junos (Device, Config)`

- **Purpose**: Part of the `PyEZ` library provided by Juniper Networks for automating and managing Junos devices (e.g., routers, switches).
- **Common Usage**:
    - `Device`: Used to establish a connection to a Junos device.
    - `Config`: Used to manage the configuration on a Junos device (loading, committing changes).

3.2.3 `paramiko (SSHException)`

- **Purpose**: A Python library used to handle SSH connections and execute commands on remote machines securely.
- **Common Usage**: Establishing SSH connections, executing remote commands, handling SSH-related exceptions.

3.2.4 `scp (SCPClient, SCPException)`

- **Purpose**: A module that works with `paramiko` to copy files to and from a remote machine using the SCP protocol.
- **Common Usage**: Securely transferring files over SSH between a local and a remote machine.

3.2.5 `lxml (etree, E)`

- **Purpose**: A library for processing XML and HTML documents.
- **Common Usage**:
    - `etree`: Used to parse and interact with XML data structures.
    - `E`: A convenient way to build XML documents using a more Pythonic syntax.

3.2.6 `argparse`

- **Purpose**: A module for parsing command-line arguments passed to a Python script.
- **Common Usage**: Define and handle command-line arguments for scripts, making them more flexible and user-friendly.

3.2.7 `threading (Thread, Lock)`

- **Purpose**: The `threading` module provides tools for working with threads in Python, allowing for concurrent execution.
- **Common Usage**:
    - `Thread`: Create and manage threads.
    - `Lock`: Synchronize threads to prevent race conditions.

3.2.8 `logging (handlers)`

- **Purpose**: A standard module for logging events, errors, and debug information in Python applications.
- **Common Usage**:
    - Setting up logging configurations.
    - `handlers`: Manage log file rotation, logging to different destinations, etc.

3.2.9 `json`

- **Purpose**: A module for working with JSON data, a common data interchange format.
- **Common Usage**: Serializing Python objects to JSON format and deserializing JSON strings back into Python objects.

3.2.10 `copy`

- **Purpose**: Provides functions to create shallow or deep copies of Python objects.
- **Common Usage**: Duplicating objects when you need to modify them without altering the original.

3.2.11 `base64`

- **Purpose**: A module for encoding and decoding data in Base64, a binary-to-text encoding scheme.
- **Common Usage**: Encoding binary data for transmission over media that are designed to handle textual data.

3.2.12 `uuid`

- **Purpose**: A module for generating universally unique identifiers (UUIDs).
- **Common Usage**: Creating unique identifiers for objects, such as in distributed systems.

3.2.13 `re`

- **Purpose**: A module for working with regular expressions in Python.
- **Common Usage**: Searching, matching, and manipulating strings based on patterns.

3.2.14 `subprocess`

- **Purpose**: A module for spawning new processes, connecting to their input/output/error pipes, and obtaining their return codes.
- **Common Usage**: Running shell commands from within a Python script.

3.2.15 `os`

- **Purpose**: A module providing a way of interacting with the operating system, including file and directory management.
- **Common Usage**: Working with the file system, environment variables, and executing system commands.

3.2.16 `OpenSSL (crypto)`

- **Purpose**: A wrapper around a subset of the OpenSSL library, which provides tools for cryptography, including managing certificates.
- **Common Usage**: Generating and managing SSL/TLS certificates, encryption/decryption.

3.2.17 `datetime`

- **Purpose**: A module for working with dates and times in Python.
- **Common Usage**: Creating, manipulating, and formatting date/time objects.

3.2.18 `schedule`

- **Purpose**: A lightweight job scheduling library for Python, which allows running tasks at specific intervals.
- **Common Usage**: Scheduling and managing periodic tasks in a Python application.

3.2.19 `Profile`

- **Purpose**: A module for profiling and analyzing the performance of Python code.
- **Common Usage**: Measuring and optimizing the execution time of Python functions and scripts.

3.2.20 `time`

- **Purpose**: A module providing various time-related functions.
- **Common Usage**: Pausing execution, getting the current time, and measuring intervals.

3.3 Logs

3.3.1 Sample logs

3.3.1.1 Offbox parallel execution (4 devices under test: mx304-11, mx304-12, mx10008-23 and mx10008-24)

```
administrator@jump:~$ python3 ETSIA_v3.0.py
onbox False
Logging modules initialized successfully
Offbox approach taken
start check_and_apply_initial_config
start check_and_apply_initial_config
start check_and_apply_initial_config
start check_and_apply_initial_config
stop check_and_apply_initial_config
start process
stop check_and_apply_initial_config
```

```
start process
local_name: mx304-11
mx304-11 is Master
local_name: mx304-12
mx304-12 is Slave
local_name: mx304-12 last_key_dict: {'mx304-12': 'c107281c-9d04-4900-
b731-896ab8f3a6c2'}
local_name: mx304-12 master_key_dict: {'mx304-11': 'c107281c-9d04-4900-
b731-896ab8f3a6c2'}
local_name: mx304-12 Same KeyId: c107281c-9d04-4900-b731-896ab8f3a6c2,
Retrying 1
stop check_and_apply_initial_config
start process
stop check_and_apply_initial_config
start process
local_name: mx10008-24
mx10008-24 is Slave
local_name: mx10008-24 last_key_dict: {'mx10008-24': 'c107281c-9d04-
4900-b731-896ab8f3a6c2'}
local_name: mx10008-24 master_key_dict: {'mx304-11': 'c107281c-9d04-
4900-b731-896ab8f3a6c2'}
local_name: mx10008-24 Same KeyId: c107281c-9d04-4900-b731-896ab8f3a6c2,
Retrying 1
local_name: mx10008-23
mx10008-23 is Slave
local_name: mx10008-23 last_key_dict: {'mx10008-23': 'c107281c-9d04-
4900-b731-896ab8f3a6c2'}
local_name: mx10008-23 master_key_dict: {'mx304-11': 'c107281c-9d04-
4900-b731-896ab8f3a6c2'}
local_name: mx10008-23 Same KeyId: c107281c-9d04-4900-b731-896ab8f3a6c2,
Retrying 1
local_name: mx304-11: {'keys': [{'key_ID': 'd9b6fa63-dd88-47d2-8afe-
94fcecf40d19', 'key':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xO
p0qOiDLH9ldIb4Q=='}]}
KME: [GET] Get Keys API: https://kme-5.acct-1286.etsi-qkd-
api.qukaydee.com/api/v1/keys/mx304-12/enc_keys
KME: [GET] Get Keys API: for mx304-12 {'keys': [{'key_ID': 'd9b6fa63-
dd88-47d2-8afe-94fcecf40d19', 'key':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xO
p0qOiDLH9ldIb4Q=='}]}
Received KeyId:d9b6fa63-dd88-47d2-8afe-94fcecf40d19
ckn abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19
cak 9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f
local_name: mx304-11 commit True
Configuration <configuration>
  <security>
    <macsec>
      <connectivity-association>
        <name>CA_basic</name>
        <cipher-suite>gcm-aes-xpn-256</cipher-suite>
        <security-mode>static-cak</security-mode>
        <pre-shared-key>
```

```
<ckn>abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19</c
kn>

<cak>9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f</c
ak>
        </pre-shared-key>
      </connectivity-association>
    </macsec>
  </security>
</configuration>

========================== script run SUCCESS
=============================
{'mx304-11': 'd9b6fa63-dd88-47d2-8afe-94fcecf40d19'}
Saved the key IDs to the JSON file: /home/administrator/mx304-
11_last_key_test.json
local_name: mx304-12 last_key_dict: {'mx304-12': 'c107281c-9d04-4900-
b731-896ab8f3a6c2'}
local_name: mx304-12 master_key_dict: {'mx304-11': 'd9b6fa63-dd88-47d2-
8afe-94fcecf40d19'}
local_name: mx304-12 Previous keyID: c107281c-9d04-4900-b731-
896ab8f3a6c2
local_name: mx10008-24 last_key_dict: {'mx10008-24': 'c107281c-9d04-
4900-b731-896ab8f3a6c2'}
local_name: mx10008-24 master_key_dict: {'mx304-11': 'd9b6fa63-dd88-
47d2-8afe-94fcecf40d19'}
local_name: mx10008-24 Previous keyID: c107281c-9d04-4900-b731-
896ab8f3a6c2
local_name: mx10008-23 last_key_dict: {'mx10008-23': 'c107281c-9d04-
4900-b731-896ab8f3a6c2'}
local_name: mx10008-23 master_key_dict: {'mx304-11': 'd9b6fa63-dd88-
47d2-8afe-94fcecf40d19'}
local_name: mx10008-23 Previous keyID: c107281c-9d04-4900-b731-
896ab8f3a6c2
local_name: mx304-12: {'keys': [{'key_ID': 'd9b6fa63-dd88-47d2-8afe-
94fcecf40d19', 'key':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xO
p0qOiDLH9ldIb4Q=='}]}
ckn abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19
cak 9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f
local_name: mx304-12 commit True
Configuration <configuration>
  <security>
    <macsec>
      <connectivity-association>
        <name>CA_basic</name>
        <cipher-suite>gcm-aes-xpn-256</cipher-suite>
        <security-mode>static-cak</security-mode>
        <pre-shared-key>

<ckn>abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19</c
kn>
```

```
<cak>9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f</c
ak>
        </pre-shared-key>
      </connectivity-association>
    </macsec>
  </security>
</configuration>

local_name: mx10008-24: {'keys': [{'key_ID': 'd9b6fa63-dd88-47d2-8afe-
94fcecf40d19', 'key':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xO
p0qOiDLH9ldIb4Q=='}]}
ckn abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19
cak 9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f
local_name: mx10008-24 commit True
Configuration <configuration>
  <security>
    <macsec>
      <connectivity-association>
        <name>CA_basic</name>
        <cipher-suite>gcm-aes-xpn-256</cipher-suite>
        <security-mode>static-cak</security-mode>
        <pre-shared-key>

<ckn>abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19</c
kn>

<cak>9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f</c
ak>
        </pre-shared-key>
      </connectivity-association>
    </macsec>
  </security>
</configuration>

local_name: mx10008-23: {'keys': [{'key_ID': 'd9b6fa63-dd88-47d2-8afe-
94fcecf40d19', 'key':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xO
p0qOiDLH9ldIb4Q=='}]}
ckn abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19
cak 9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f
local_name: mx10008-23 commit True
Configuration <configuration>
  <security>
    <macsec>
      <connectivity-association>
        <name>CA_basic</name>
        <cipher-suite>gcm-aes-xpn-256</cipher-suite>
        <security-mode>static-cak</security-mode>
        <pre-shared-key>

<ckn>abcd1234abcd5678abcd1234abcd5678d9b6fa63dd8847d28afe94fcecf40d19</c
```

```
kn>

<cak>9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f</c
ak>
        </pre-shared-key>
      </connectivity-association>
    </macsec>
  </security>
</configuration>

========================== script run SUCCESS
=============================
{'mx10008-23': 'd9b6fa63-dd88-47d2-8afe-94fcecf40d19'}
Saved the key IDs to the JSON file: /home/administrator/mx10008-
23_last_key_test.json
========================== script run SUCCESS
=============================
{'mx10008-24': 'd9b6fa63-dd88-47d2-8afe-94fcecf40d19'}
Saved the key IDs to the JSON file: /home/administrator/mx10008-
24_last_key_test.json
========================== script run SUCCESS
=============================
{'mx304-12': 'd9b6fa63-dd88-47d2-8afe-94fcecf40d19'}
Saved the key IDs to the JSON file: /home/administrator/mx304-
12_last_key_test.json
----------------------mx304-11-----------------------------
<output>
Interface    Member-ID                  Type        Status        Tx
Rx        CAK Name
et-0/0/9     A2FC7A578AC21E943BF48DBC   primary     live          8
4
ABCD1234ABCD5678ABCD1234ABCD5678D9B6FA63DD8847D28AFE94FCECF40D19
xe-0/0/1:0   F6AC6383308556191AAFB283   primary     live          9
7
ABCD1234ABCD5678ABCD1234ABCD5678D9B6FA63DD8847D28AFE94FCECF40D19
</output>

stop process
----------------------mx10008-23-----------------------------
<output>
Interface    Member-ID                  Type        Status        Tx
Rx        CAK Name
et-0/0/1     7EA175A5FCE62846080918AD   primary     live          9
8
ABCD1234ABCD5678ABCD1234ABCD5678D9B6FA63DD8847D28AFE94FCECF40D19
xe-0/0/2:0   3E1BF61913EDC22757C9F013   primary     live          9
9
ABCD1234ABCD5678ABCD1234ABCD5678D9B6FA63DD8847D28AFE94FCECF40D19
</output>

stop process
----------------------mx10008-24-----------------------------
<output>
```

```
Interface    Member-ID                   Type        Status        Tx
Rx         CAK Name
et-0/0/1     75146F34B0738685954F1E48   primary     live          9
8
ABCD1234ABCD5678ABCD1234ABCD5678D9B6FA63DD8847D28AFE94FCECF40D19
</output>

stop process
----------------------mx304-12-----------------------------
<output>
Interface    Member-ID                   Type        Status        Tx
Rx         CAK Name
et-0/0/9     535246D1CF595CCD1BA8B067   primary     live          9
10
ABCD1234ABCD5678ABCD1234ABCD5678D9B6FA63DD8847D28AFE94FCECF40D19
</output>

stop process
```

3.3.1.2 Onbox execution (master device under test: mx304-11)

```
ahanciu@mx304-11-re0> op ETSIA_v3.0.py
onbox True
Logging modules initialized successfully
Onbox approach taken
local_name: mx304-11
mx304-11 is Master
/var/home/remote/mx304-11.crt
/var/home/remote/mx304-11.key
/var/home/remote/account-1286-server-ca-qukaydee-com.crt
local_name: mx304-11: {'keys': [{'key_ID': 'a310923e-9de9-4ab1-bafe-
806bcc2e62f9', 'key':
'E/n0dB6HY4PfpoYvJOuaEHba9PhB1/hz+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8I
xkDnDP/8Ofkw1rA=='}]}
inside try
KME: [GET] Get Keys API: https://kme-5.acct-1286.etsi-qkd-
api.qukaydee.com/api/v1/keys/mx304-12/enc_keys
KME: [GET] Get Keys API: for mx304-12 {'keys': [{'key_ID': 'a310923e-
9de9-4ab1-bafe-806bcc2e62f9', 'key':
'E/n0dB6HY4PfpoYvJOuaEHba9PhB1/hz+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8I
xkDnDP/8Ofkw1rA=='}]}
Received KeyId:a310923e-9de9-4ab1-bafe-806bcc2e62f9
ckn abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9
cak 13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58
local_name: mx304-11 commit True
Configuration <configuration>
  <security>
    <macsec><connectivity-association>
                <name>CA_basic</name>
                <cipher-suite>gcm-aes-xpn-256</cipher-suite>
                <security-mode>static-cak</security-mode>
                <pre-shared-key>

<ckn>abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9</c
kn>

<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</c
ak>
                </pre-shared-key>
            </connectivity-association>
            </macsec>
  </security>
</configuration>
{'mx304-11': 'a310923e-9de9-4ab1-bafe-806bcc2e62f9'}
Saved the key IDs to the JSON file: /var/home/remote/mx304-
11_last_key_test.json
----------------------mx304-11-----------------------------
<output>
Interface    Member-ID                  Type       Status        Tx
Rx        CAK Name
et-0/0/9     F3793517620D59C1F98119D6   primary    live          43
39
ABCD1234ABCD5678ABCD1234ABCD5678FF6A6454A59B4A11A3F267F397398BEF
et-0/0/9     98DD8DE595C05146185D6C85   primary    in-progress   2
0
ABCD1234ABCD5678ABCD1234ABCD5678A310923E9DE94AB1BAFE806BCC2E62F9
xe-0/0/1:0   08A726358F84ACB1F0C99FE0   primary    live          43
36
ABCD1234ABCD5678ABCD1234ABCD5678FF6A6454A59B4A11A3F267F397398BEF
xe-0/0/1:0   E6388930FE1A773944C09EBA   primary    in-progress   2
0
ABCD1234ABCD5678ABCD1234ABCD5678A310923E9DE94AB1BAFE806BCC2E62F9
</output>
========================== script run SUCCESS
=============================
```

3.3.1.3 Onbox execution (slave device under test: mx304-12)

```
ahanciu@mx304-12-re0> op ETSIA_v3.0.py
onbox True
Logging modules initialized successfully
Onbox approach taken
local_name: mx304-12
mx304-12 is Slave
local_name: mx304-12 last_key_dict: {'mx304-12': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 master_key_dict: {'mx304-11': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 1
```

```
local_name: mx304-12 last_key_dict: {'mx304-12': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 master_key_dict: {'mx304-11': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 2
local_name: mx304-12 last_key_dict: {'mx304-12': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 master_key_dict: {'mx304-11': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 3
local_name: mx304-12 last_key_dict: {'mx304-12': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 master_key_dict: {'mx304-11': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 4
local_name: mx304-12 last_key_dict: {'mx304-12': 'ff6a6454-a59b-4a11-
a3f2-67f397398bef'}
local_name: mx304-12 master_key_dict: {'mx304-11': 'a310923e-9de9-4ab1-
bafe-806bcc2e62f9'}
local_name: mx304-12 Previous keyID: ff6a6454-a59b-4a11-a3f2-
67f397398bef
local_name: mx304-12: {'keys': [{'key_ID': 'a310923e-9de9-4ab1-bafe-
806bcc2e62f9', 'key':
'E/n0dB6HY4PfpoYvJOuaEHba9PhB1/hz+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8I
xkDnDP/8Ofkw1rA=='}]}
ckn abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9
cak 13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58
local_name: mx304-12 commit True
Configuration <configuration>
  <security>
    <macsec><connectivity-association>
                <name>CA_basic</name>
                <cipher-suite>gcm-aes-xpn-256</cipher-suite>
                <security-mode>static-cak</security-mode>
                <pre-shared-key>

<ckn>abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9</c
kn>

<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</c
ak>
                </pre-shared-key>
            </connectivity-association>
            </macsec>
  </security>
</configuration>
========================== script run SUCCESS
==============================
{'mx304-12': 'a310923e-9de9-4ab1-bafe-806bcc2e62f9'}
Saved the key IDs to the JSON file: /var/home/remote/mx304-
```

```
12_last_key_test.json
----------------------mx304-12-----------------------------
<output>
Interface     Member-ID                   Type        Status        Tx
Rx        CAK Name
et-0/0/9      FCC2076AA69DCBAE2011CC38    primary     live          44
46
ABCD1234ABCD5678ABCD1234ABCD5678FF6A6454A59B4A11A3F267F397398BEF
et-0/0/9      5DD52C660EA70BD7DE81F439    primary     in-progress   2
0
ABCD1234ABCD5678ABCD1234ABCD5678A310923E9DE94AB1BAFE806BCC2E62F9
</output>
```

3.3.1.4 Onbox execution (additional slave device under test: mx10008-23)

```
ahanciu@mx10008-23-re0> op ETSIA_v3.0.py
onbox True
Logging modules initialized successfully
Onbox approach taken
local_name: mx10008-23
mx10008-23 is Slave
mx10008-23
{'mx10008-23': 'ff6a6454-a59b-4a11-a3f2-67f397398bef'}
local_name: mx10008-23 last_key_dict: {'mx10008-23': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-23 master_key_dict: {'mx304-11': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-23 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 1
mx10008-23
{'mx10008-23': 'ff6a6454-a59b-4a11-a3f2-67f397398bef'}
local_name: mx10008-23 last_key_dict: {'mx10008-23': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-23 master_key_dict: {'mx304-11': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-23 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 2
mx10008-23
{'mx10008-23': 'ff6a6454-a59b-4a11-a3f2-67f397398bef'}
local_name: mx10008-23 last_key_dict: {'mx10008-23': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-23 master_key_dict: {'mx304-11': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-23 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 3
mx10008-23
{'mx10008-23': 'ff6a6454-a59b-4a11-a3f2-67f397398bef'}
local_name: mx10008-23 last_key_dict: {'mx10008-23': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-23 master_key_dict: {'mx304-11': 'a310923e-9de9-
4ab1-bafe-806bcc2e62f9'}
local_name: mx10008-23 Previous keyID: ff6a6454-a59b-4a11-a3f2-
67f397398bef
local_name: mx10008-23: {'keys': [{'key_ID': 'a310923e-9de9-4ab1-bafe-
806bcc2e62f9', 'key':
'E/n0dB6HY4PfpoYvJOuaEHba9PhB1/hz+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8I
xkDnDP/8Ofkw1rA=='}]}
ckn abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9
cak 13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58
local_name: mx10008-23 commit True
Configuration <configuration>
  <security>
    <macsec><connectivity-association>
                <name>CA_basic</name>
                <cipher-suite>gcm-aes-xpn-256</cipher-suite>
                <security-mode>static-cak</security-mode>
                <pre-shared-key>

<ckn>abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9</c
kn>

<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</c
ak>
                </pre-shared-key>
            </connectivity-association>
            </macsec>
  </security>
</configuration>
========================== script run SUCCESS
=============================
{'mx10008-23': 'a310923e-9de9-4ab1-bafe-806bcc2e62f9'}
Saved the key IDs to the JSON file: /var/home/remote/mx10008-
23_last_key_test.json
----------------------mx10008-23-----------------------------
<output>
Interface    Member-ID                   Type       Status        Tx
Rx        CAK Name
et-0/0/1     9D553DC27696177B2E5649A4   primary    live          38
23
ABCD1234ABCD5678ABCD1234ABCD5678FF6A6454A59B4A11A3F267F397398BEF
et-0/0/1     E5F4FC6970435F42E1AA8924   primary    in-progress   2
0
ABCD1234ABCD5678ABCD1234ABCD5678A310923E9DE94AB1BAFE806BCC2E62F9
xe-0/0/2:0   9D0DA77F1FFC68249674AA7C   primary    live          38
43
ABCD1234ABCD5678ABCD1234ABCD5678FF6A6454A59B4A11A3F267F397398BEF
xe-0/0/2:0   BCC4E0D4221BC1B34ADB2ABB   primary    in-progress   2
0
ABCD1234ABCD5678ABCD1234ABCD5678A310923E9DE94AB1BAFE806BCC2E62F9
</output>
```

3.3.1.5 Onbox execution (additional slave device under test: mx10008-24)

```
ahanciu@mx10008-24> op ETSIA_v3.0.py
onbox True
Logging modules initialized successfully
Onbox approach taken
local_name: mx10008-24
mx10008-24 is Slave
local_name: mx10008-24 last_key_dict: {'mx10008-24': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-24 master_key_dict: {'mx304-11': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-24 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 1
local_name: mx10008-24 last_key_dict: {'mx10008-24': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-24 master_key_dict: {'mx304-11': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-24 Same KeyId: ff6a6454-a59b-4a11-a3f2-67f397398bef,
Retrying 2
local_name: mx10008-24 last_key_dict: {'mx10008-24': 'ff6a6454-a59b-
4a11-a3f2-67f397398bef'}
local_name: mx10008-24 master_key_dict: {'mx304-11': 'a310923e-9de9-
4ab1-bafe-806bcc2e62f9'}
local_name: mx10008-24 Previous keyID: ff6a6454-a59b-4a11-a3f2-
67f397398bef
local_name: mx10008-24: {'keys': [{'key_ID': 'a310923e-9de9-4ab1-bafe-
806bcc2e62f9', 'key':
'E/n0dB6HY4PfpoYvJOuaEHba9PhB1/hz+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8I
xkDnDP/8Ofkw1rA=='}]}
ckn abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9
cak 13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58
local_name: mx10008-24 commit True
Configuration <configuration>
  <security>
    <macsec><connectivity-association>
                <name>CA_basic</name>
                <cipher-suite>gcm-aes-xpn-256</cipher-suite>
                <security-mode>static-cak</security-mode>
                <pre-shared-key>

<ckn>abcd1234abcd5678abcd1234abcd5678a310923e9de94ab1bafe806bcc2e62f9</c
kn>

<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</c
ak>
                </pre-shared-key>
            </connectivity-association>
            </macsec>
  </security>
</configuration>
```

```
========================== script run SUCCESS
=============================
{'mx10008-24': 'a310923e-9de9-4ab1-bafe-806bcc2e62f9'}
Saved the key IDs to the JSON file: /var/home/remote/mx10008-
24_last_key_test.json
----------------------mx10008-24-----------------------------
<output>
Interface    Member-ID                   Type        Status        Tx
Rx         CAK Name
et-0/0/1     D5C45816AF10660C0DBF9F04   primary     live          23
36
ABCD1234ABCD5678ABCD1234ABCD5678FF6A6454A59B4A11A3F267F397398BEF
et-0/0/1     3E0D5101F36A32C97ABE2CD9   primary     in-progress   2
0
ABCD1234ABCD5678ABCD1234ABCD5678A310923E9DE94AB1BAFE806BCC2E62F9
</output>
```