QKD Quantum-Safe MACsec

Automated MACsec deployment

Automation Team

Date: 2025-12-01

Document version: 3.2

Document Control

Document Information

	Information
Document Name	QKD MACsec
Document Owner	Juniper Professional Services
Issue Date	2025-12-01
Last Saved Date	2025-11-27
File Name	QKD_MACsec

Document History

Version	**Issue Date**	**Changes**
3.2	2025-12-01	Document Main

Document Approvals

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

Contribution

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

License

Acknowledgements

Thank you for using the MACSEC Configuration Script! For more information, please refer to the documentation. This software has been developed in the projects ...

Table of contents

Introduction

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.

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.

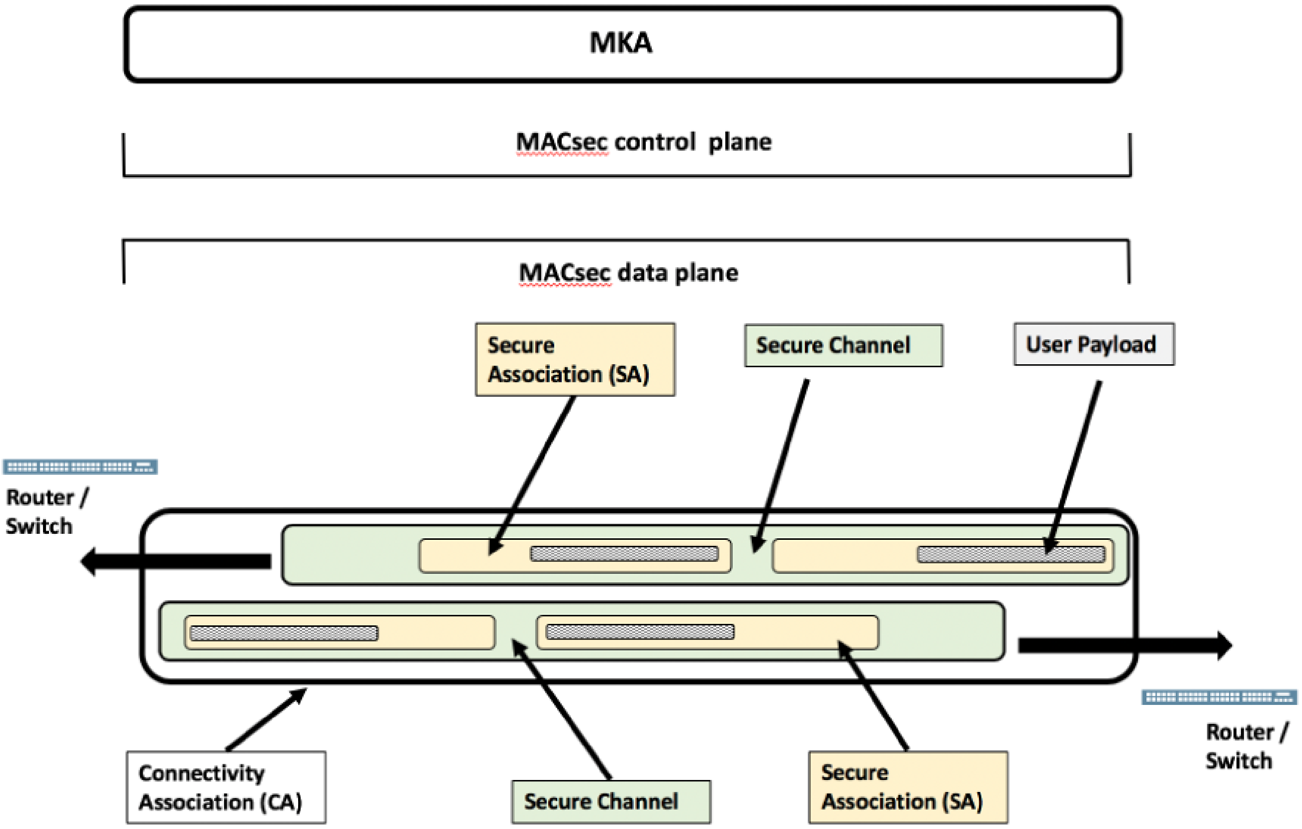
Automated deployment of MACsec creates secure channel, generates keying material, and configures the routers/switches for each detected link between them. The script detects link changes and performs rekeying to provide a secure, configuration-free operation of MACsec.

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.

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.

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.

The script makes use of the libraries in the ETSI GS QKD 014 v1.1.1 - Reference Implementation public repository.

Release Notes for MACSEC Configuration Script

Script version 3.2.0

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.

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.
- **Password Encryption**: Incorporated encryption for sensitive information such as passwords in the configuration dictionary.
- **Extended Logging Options**: Added additional logging options for detailed trace and debug information.
- **Scaling Tests**: Implemented scaling tests to handle multiple interfaces more efficiently.

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.

Script Configuration

- `targets_dict`: The dictionary that has to be updated will the following information consumed by the script:
  - Updated dictionary containing device-specific configurations with encrypted passwords.
  - 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>

- `config.ini`: Optional configuration file for additional settings and parameters.

Script Usage

- Offbox

  - **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.
    - `-t`, `--trace`: Dump debug-level logs to a trace.log file.
    - `--config-file`: Specify a configuration file for additional settings.

  - **Manual qkd_v3.2.0.py Script Execution - Example Commands**:

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

      ```
      python3 qkd_v3.2.0.py
      ```

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

      ```
      python3 qkd_v3.2.0.py -t 4
      ```

    - **Run Script with Verbose Logging**:

      ```
      python3 qkd_v3.2.0.py --verbose
      ```

    - **Run Script with Trace Logging**:

      ```
      python3 qkd_v3.2.0.py --trace
      ```

  - **Scheduled qkd_v3.2.0.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 qkd_v3.2.0.py
```

- Onbox
    - Import the qkd_v3.2.0.py and profiling Profile python scripts under /var/db/scripts/event/qkd_v3.2.0.py
    - 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 Event-options section) and the script will start running every <time-interval seconds> starting with <start-time start-time>.

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.

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.

Future Improvements

- **Advanced Configuration Management**: Implement more advanced configuration management features.

- **Extended Encryption Support**: Enhance support for encryption and security features.

- **Enhanced User Interface**: Develop a user-friendly interface for easier configuration and management.

- **OUI Ethertype 88B7 Key-ID tranport**: tranport key-ids through OUI Ethernet-type instead of LLDP or scp protocol (given the key-id remains the same for the whole chain of MACSEC tunnels).

- **Enhanced PPK**: Static vs dynamic Pre-Shared Key (PPK), represented with two strings separated by spaces. The first string is PPK_ID and the second one is a path to another file which the PPKs will be taken from. This file should also have only two strings in it. The first one would be offset and the second one a long string that will be used as a one-time pad.

- **Vault container(SSSD)**: Use of Vault container (e.g. SSSD) for key material received from KME quantum servers. This requires extra server or VM (close to SAE device) between end SAE and KME itself. If extra server isn’t an option, this should be directly embedded into Junos Code using Junos to trigger API request/response from KME to SAE network device

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 3.2.0 and the user is harcoded as "lab".

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.

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.

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:

- 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.

- 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.

- 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.

- 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.

- 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.

- 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)

- 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.

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:

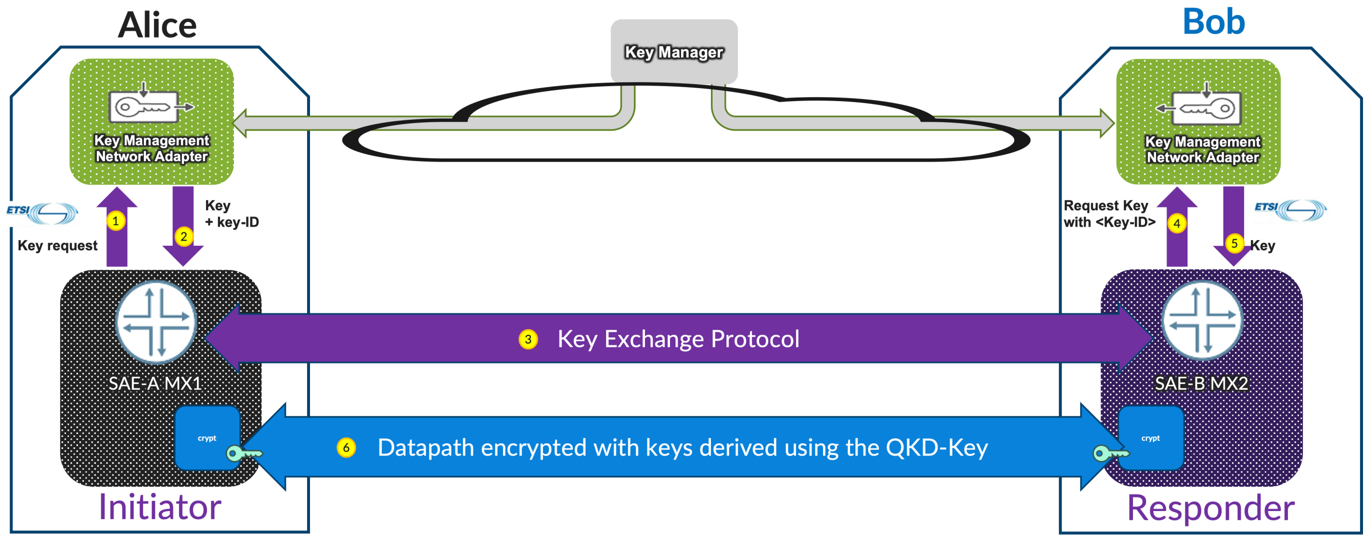
1. a connectivity association key (CAK) that secures control plane traffic and
2. a randomly-generated secure association key (SAK) that secures data plane traffic.

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

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
Bob
Key Manager
Key Management Network Adapter
Key Management Network Adapter
ETSI
1
Key request
Key + key-ID
2
Request Key with <Key-ID>
4
ETSI
5 Key
3 Key Exchange Protocol
SAE-A MX1
SAE-B MX2
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
```

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.

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).

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:

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.

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.

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 Iterates through interfaces that have the `qkd` apply-macro (in the case of 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.

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.

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).

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.

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.

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.

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.

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.

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.
```

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.

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.

Logging and tracing

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.
```

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:

Initial configuration

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:

Function Purpose

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

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.

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.

MACsec configuration script main functions

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}
```

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
```

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]
```

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

Key Management functions

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.

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.

Save key ids function

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

QKD MACsec Automation Script — functions python deep dive

Purpose of the python script

This script automates the full lifecycle required to configure **QKD-backed MACsec** on network devices. It handles:

- Secure SSH connectivity
- Certificate generation, renewal, and upload
- Retrieval of QKD keys from a KME
- Distribution of key IDs between devices
- Applying initial and periodic MACsec configuration
- Multithreaded off-box management of many devices
- Profiling and logging for debugging and traceability

It supports two execution modes:

- **Offbox mode** — Run from a management host and configure multiple devices in parallel.
- **Onbox mode** — Run directly on a device, performing the full workflow locally.

The script is separated into logical components (threads, certificates, SSH, KME interactions, MACsec config, etc.), with each function contributing toward completing the workflow.

Below is a description of every function, grouped by subsystem, showing how they work together.

Threading / Parallel Execution

`def background(func):`

A decorator that runs the wrapped function asynchronously in a background thread.

- Creates a `Thread`
- Starts it as a daemon
- Tracks it in the global `threads` list

Used primarily to parallelize device operations in **offbox mode**.

`def req_thread(tnum, reqs, targets_dict, log):`

Worker function executed in background threads.

- Iterates over a subset (`reqs`) of devices assigned to this thread.
- For each device:
  - Creates a connection (`Device`)
  - Applies initial config

  - Fetches and pushes keys
  - Applies final MACsec configuration

Contributes to scaling the script for multi-device deployments.

Certificate Management

`def generate_ca_certificate(ca_cert_path, ca_key_path, ca_subject):`

Creates a self-signed **CA certificate** and private key, used for signing client certificates.

Used when the CA certificate is missing or expired.

`def generate_client_certificate(client_cert_path, client_key_path, ca_cert_path, ca_key_path, client_subject):`

Generates a client certificate/key pair signed by the CA.

Uploaded later to the device for secure KME communication.

`def is_certificate_valid(cert_path, min_valid_days=10):`

Checks if a certificate exists and whether it is still valid for at least `min_valid_days` days.

Used before renewing certificates.

`def get_certificates(dev, log, targets_dict):`

Downloads the device's **existing** client certificate and key over SSH.

Used to validate and determine whether renewal is required.

`def upload_certificates(dev, cert_path, key_path, log, targets_dict):`

Uploads new or renewed certificates back onto the device via SCP.

`def fetch_ca_certificate(targets_dict):`

Retrieves (or generates) the CA certificate on the management host.

May be used to distribute a trusted CA to devices.

`def renew_certificates(dev, log, targets_dict):`

Central certificate workflow on a device:

1. Check existing cert validity
2. If invalid or missing:
    - Generate new client cert
    - Upload it to the device
3. Log success/failure

Used within the main processing flow.

`def should_check_certs():`

Tells the script if certificate validation should be performed. Triggered via command-line flags or environment conditions.

Logging / CLI

`def initialize_logging(args):`

Configures logging:

- Verbosity (`-v`)
- Trace mode (`--trace`)
- Logging to file or console

Essential for debugging distributed device operations.

`def get_args():`

Defines command-line parameters for:

- Thread count
- Verbose output
- Trace logs

Used by `main()` to interpret user input.

Key ID Tracking

`def get_previous_key_ids(log, name):`

Reads previously stored key IDs from the device's JSON file.

Used to determine if a new key has been fetched and prevent unnecessary work.

`def save_key_ids(key_dict, local_name):`

Writes the key IDs into a JSON file for persistent tracking.

KME Key Retrieval

`def fetch_kme_key(session, local_name, log, remote_mnmgt_add, kme_host, key_id, additional_slave_SAE_IDs=None):`

Contacts the **KME** and retrieves a fresh QKD key for the specified device.

- Negotiates session keys
- Stores the returned key material locally
- Updates key IDs for master/slave/additional devices

Core function for QKD key retrieval.

Device MACsec Configuration

`def check_and_apply_initial_config(dev, targets_dict, log):`

Runs initial device configuration:

- Basic MACsec setup
- CA/client certificate preparation
- Upload of trust anchors
- Validation checks

Executed once per device unless the device already has MACsec config.

`def get_key_id_from_master(dev, log, targets_dict):`

Retrieves the **master's Key ID**:

- Offbox: Reads from master's JSON file
- Onbox: SCP fetches key ID file

Used so slaves can request the correct KME key.

`def process(dev, targets_dict, log):`

The **central device workflow**:

1. Renew certificates (if needed)
2. Fetch key ID from master (if slave)
3. Retrieve new KME key
4. Apply MACsec key + policy configuration
5. Save the new key ID locally

This is what each thread or onbox execution ultimately performs.

Main Program Flow

`def main():`

Entry point of the script.

Responsible for:

1. Parsing CLI arguments
2. Initializing logging
3. Loading configuration (`targets_dict`)
4. Deciding Onbox vs Offbox execution
5. Offbox:
   - Split device list into thread groups
   - Call `req_thread()` for each
   - Wait for all threads to finish
6. Onbox:
   - Run full `process()` directly and sequentially

This function connects everything into a complete automated workflow.

End-to-End Workflow Summary

The entire script implements this high-level pipeline:

```
Parse Args → Init Logging → Load Targets → (Offbox: Threading)
     ↓
Connect to Device
     ↓
Certificate Handling (validate, renew, upload)
     ↓
Retrieve Key ID
     ↓
Fetch QKD Key from KME
     ↓
Apply MACsec Configuration
     ↓
Save Key IDs → Done
```

This structure ensures **automated, scalable, secure MACsec provisioning** with QKD key material across many devices.

Environment configuration block.

- Defines a base directory (CUR_DIR) path and certificate path
- Builds all file paths relative to that directory.
- Sets up database connection configuration settings.
- Specifies TLS certificates.
- Defines API URLs.
- Creates logging locations and profiling locations.

CUR_DIR means “current working directory for this application” It is simply the base folder from which the program reads/writes files like:

- certs
- logs
- JSON output
- profiling data

OFFBOX_CERTS_DIR stores certificates:

- TLS certificate
- TLS private key
- Root CA

```
DATABASE_PORT = '10000'
DATABASE_HOST = '9.173.9.102'
DATABASE_USER = 'db_user'
DATABASE_PASSWORD = 'db_password'
DATABASE_URL = f'postgres://{DATABASE_USER}:
{DATABASE_PASSWORD}@{DATABASE_HOST}:{DATABASE_PORT}/key_store'
```

These define how the ETSI-014 reference implementation connects to PostgreSQL. Final URL produced:

```
postgres://db_user:db_password@9.173.9.102:10000/key_store
```

```
CERTS_DIR = f'{CUR_DIR}/certs/'
Same as OFFBOX_CERTS_DIR, used to store:
CA_CERT = f'{CERTS_DIR}/root.crt'
```

This is the path to the CA certificate.

```
ETSI_014_REF_IMPL_TLS_CERT = f'{OFFBOX_CERTS_DIR}/kme_001.crt'
ETSI_014_REF_IMPL_TLS_PRIVATE_KEY = f'{OFFBOX_CERTS_DIR}/kme_001.key'
ETSI_014_REF_IMPL_TLS_ROOT_CRT = f'{OFFBOX_CERTS_DIR}/root.crt'
```

These are TLS files for a Key Management Entity (KME) implementation.

Managing multithreading

The threads[] list will hold every thread that the decorator creates.This list will hold every thread the program starts. It allows the program to keep track of all the work happening in parallel. Useful if you later want to:

- wait for them
- check their status
- debug how many threads are running

The decorator starts that function in a separate background thread. The thread is set as a daemon thread, which means it does not block the program from exiting. The thread starts immediately. The thread object is added to the threads list for tracking. The original function runs asynchronously (in parallel), without waiting for it to finish. In simple terms: Putting @background above a function makes it run in the background.

Understanding the Threading Code: a list to track background threads

```
threads = []
```

This list will hold every thread the program starts. It allows the program to keep track of all the work happening in parallel.

A decorator that runs functions in the background

```
def background(func):
    def bg_func(*args, **kwargs):
        t = Thread(target=func, args=args, kwargs=kwargs)
        t.setDaemon(True)
        t.start()
        threads.append(t)
    return bg_func
```

This is a **decorator**. Decorators wrap a function and change how it behaves.

What this decorator does:

- When you call a decorated function, it **starts that function in a separate background thread**.
- The thread is set as a **daemon thread**, which means it does not block the program from exiting.
- The thread starts immediately.
- The thread object is added to the `threads` list for tracking.
- The original function runs *asynchronously* (in parallel), without waiting for it to finish.

In simple terms:

Putting `@background` above a function makes it run in the background.

A threaded function that processes devices

```
@background
def req_thread(tnum, reqs, targets_dict, log):
```

By adding `@background`, this function now runs in its **own thread** whenever it's called.

```
req_thread() # Explained
```

This function is designed to run on one thread and process a list of devices.

Here’s what it does step-by-step:

Start profiling for the whole thread

```
prof.start("Thread-{0}".format(tnum))
```

This logs performance information for the thread.

Loop through each device in the list

```
for device in reqs:
    print(device)
```

It prints each device name as it works on it.

Connect to the network device

```
with Device(host=targets_dict[device]['ip'],
            user=targets_dict["secrets"]["username"],
            password=targets_dict["secrets"]["password"],
            port=22) as dev:
```

For each device:

- It opens an SSH session to the Juniper device.
- The connection automatically closes when done.

Optionally renew certificates

```
if should_check_certs():
    renew_certificates(dev, log, targets_dict=targets_dict)
```

If the program decides certificates need checking, it:

- runs certificate renewal
- logs the activity
- profiles how long it took

Apply initial configuration

```
check_and_apply_initial_config(dev, targets_dict, log)
```

This step ensures that the device has the correct initial configuration before performing further tasks.

Profiling is used around this step to measure performance.

Run the main processing logic

```
process(dev, targets_dict, log)
```

This is the main work done for each device. It could be anything: pushing config, collecting data, etc.

Again, profiling measures how long it takes.

Stop profiling at the end of the thread

```
prof.stop("Thread-{0}".format(tnum))
```

This finalizes all timing measurements for the thread.

Summary

✔ The `@background` decorator makes any function run in a **background thread**.

✔ `req_thread()` is a function that:

- receives a list of network devices
- connects to each device
- optionally renews certificates
- applies initial configuration
- performs the main processing logic
- logs and profiles everything
- does each device in order
- but runs the *entire function* in its own thread

✔ Multiple `req_thread()` calls allow the program to work on many devices **in parallel**, each handled by a separate thread.

All python functions

`generate_ca_certificate` Explained

This function creates a **self-signed Certificate Authority (CA) certificate** and its private key, and saves them as files.

Function Signature

```
def generate_ca_certificate(ca_cert_path, ca_key_path, ca_subject):
```

Parameters:

- `ca_cert_path` → file path where the CA certificate will be saved (PEM format)
- `ca_key_path` → file path where the CA private key will be saved (PEM format)
- `ca_subject` → the “common name” (CN) for the CA (e.g., `"My Root CA"`)

Returns:

- Tuple `(ca_cert_path, ca_key_path)` → paths of the saved certificate and key files

1: Generate the CA private key**

```
ca_key = crypto.PKey()
ca_key.generate_key(crypto.TYPE_RSA, 2048)
```

- Creates a **2048-bit RSA key**, which will be used to sign the CA certificate.

- This key is the secret that proves the CA's identity.

2: Create the CA certificate object**

```
ca_cert = crypto.X509()
ca_cert.set_version(2)
ca_cert.set_serial_number(int(uuid.uuid4()))
ca_cert.get_subject().CN = ca_subject
ca_cert.set_issuer(ca_cert.get_subject())
ca_cert.set_pubkey(ca_key)
```

- `X509()` → creates a new certificate object.
- `set_version(2)` → X.509 version 3 certificate (0-based counting in OpenSSL: version 2 = v3).
- `set_serial_number(int(uuid.uuid4()))` → unique serial number using a UUID.
- `get_subject().CN = ca_subject` → sets the **Common Name (CN)** field of the certificate.
- `set_issuer(ca_cert.get_subject())` → issuer is the same as subject → **self-signed**.
- `set_pubkey(ca_key)` → associate the public key with the certificate.

3: Set validity period**

```
ca_cert.gmtime_adj_notBefore(0)
ca_cert.gmtime_adj_notAfter(5 * 365 * 24 * 60 * 60)  # 5 years
```

- Certificate is valid **from now** (`notBefore`) until 5 years (`notAfter`).
- Time is expressed in **seconds**.

4: Add X.509 extensions**

```
ca_cert.add_extensions([
    crypto.X509Extension(b"subjectKeyIdentifier", False, b"hash",
subject=ca_cert),
    crypto.X509Extension(b"basicConstraints", False, b"CA:TRUE"),
])
```

- `subjectKeyIdentifier` → allows others to identify this certificate's public key.
- `basicConstraints` → marks this certificate as a **Certificate Authority** (`CA:TRUE`).
- Some other extensions are commented out (authorityKeyIdentifier, keyUsage), but could be added later.

5: Sign the certificate

```
ca_cert.sign(ca_key, 'sha256')
```

- Uses the CA private key to **sign the certificate** with SHA-256.
- This makes it a valid self-signed CA certificate.

6: Save the certificate and key to files

```
with open(ca_cert_path, "wb") as f:
    f.write(crypto.dump_certificate(crypto.FILETYPE_PEM, ca_cert))

with open(ca_key_path, "wb") as f:
    f.write(crypto.dump_privatekey(crypto.FILETYPE_PEM, ca_key))
```

- Writes the certificate and private key in **PEM format**.
- These files can be used later to sign server or client certificates.

7: Return the paths

```
return ca_cert_path, ca_key_path
```

- Returns the file paths for reference, so the caller knows where they were saved.

Summary

1. Create a new RSA key (private key for the CA).
2. Create a new X.509 certificate object.
3. Fill in the certificate info (subject, issuer, serial number, public key).
4. Set how long the certificate is valid (5 years).
5. Add CA-specific extensions (marks it as a certificate authority).
6. Sign the certificate using the CA private key (self-signed).
7. Save both certificate and private key to files.
8. Return the file paths.

Here's a **plain-language explanation in Markdown** of your `generate_client_certificate` function, step by step.

`generate_client_certificate` Explained

This function creates a **client certificate** that is signed by an existing **CA (Certificate Authority)**. It also generates a private key for the client and saves both files.

Function Signature

```python
def generate_client_certificate(client_cert_path, client_key_path, ca_cert_path, ca_key_path, client_subject):
```

Parameters:

- `client_cert_path` → where the client certificate will be saved
- `client_key_path` → where the client private key will be saved
- `ca_cert_path` → path to the CA certificate (used to sign the client cert)
- `ca_key_path` → path to the CA private key
- `client_subject` → the “Common Name (CN)” for the client certificate

Returns:

- Tuple `(client_cert_path, client_key_path)` → paths of saved certificate and key files

1: Load the CA certificate and key**

```python
ca_key = crypto.load_privatekey(crypto.FILETYPE_PEM, open(ca_key_path, 'rb').read())
ca_cert = crypto.load_certificate(crypto.FILETYPE_PEM, open(ca_cert_path, 'rb').read())
```

- Reads the **CA private key** and **CA certificate** from disk.
- These are used to **sign the client certificate**, proving trust.

2: Generate a new client key

```python
client_key = crypto.PKey()
client_key.generate_key(crypto.TYPE_RSA, 2048)
```

- Creates a **2048-bit RSA key** for the client.
- This key is secret and will be stored in a file for the client to use.

3: Create the client certificate object

```python
client_cert = crypto.X509()
client_cert.set_version(2)
client_cert.set_serial_number(int(uuid.uuid4()))
client_cert.get_subject().CN = client_subject
client_cert.set_issuer(ca_cert.get_subject())
client_cert.set_pubkey(client_key)
client_cert.gmtime_adj_notBefore(0)
client_cert.gmtime_adj_notAfter(10*365*24*60*60)
```

- `X509()` → create a new certificate object
- `set_version(2)` → X.509 version 3 certificate
- `set_serial_number(int(uuid.uuid4()))` → unique serial number
- `get_subject().CN = client_subject` → sets the client's name
- `set_issuer(ca_cert.get_subject())` → issuer is the CA
- `set_pubkey(client_key)` → attach client's public key
- `gmtime_adj_notBefore(0)` → valid from now
- `gmtime_adj_notAfter(...)` → valid for 10 years

4: Add X.509 extensions

```
client_cert.add_extensions([
    crypto.X509Extension(b"basicConstraints", False, b"CA:FALSE"),
    crypto.X509Extension(b"authorityKeyIdentifier", False, b"keyid:always", issuer=ca_cert),
    crypto.X509Extension(b"keyUsage", False, b"Digital Signature, Non Repudiation, Key Encipherment, Data Encipherment"),
])
```

- `basicConstraints` → marks this certificate as **not a CA**
- `authorityKeyIdentifier` → links this cert to the CA that signed it
- `keyUsage` → defines what the certificate can be used for (signing, encryption, etc.)
- Other extensions like `subjectKeyIdentifier` and `extendedKeyUsage` are commented out but could be added for client authentication.

5: Sign the client certificate

```
client_cert.sign(ca_key, 'sha256')
```

- Uses the **CA private key** to sign the client certificate.
- This makes it **trusted by anyone who trusts the CA**.

6: Save the client certificate and key

```
with open(client_cert_path, "wb+") as f:
    f.write(crypto.dump_certificate(crypto.FILETYPE_PEM, client_cert))
```

```
with open(client_key_path, "wb+") as f:
    f.write(crypto.dump_privatekey(crypto.FILETYPE_PEM, client_key))
```

- Saves both files in **PEM format**.
- These files can be used by the client for authentication.

7: Return the paths

```
return client_cert_path, client_key_path
```

- Returns the file paths so the caller knows where the certificate and key were saved.

Summary

1. Load the CA certificate and private key (the signer).
2. Generate a new RSA key for the client.
3. Create a new certificate object for the client.
4. Fill in client info (name, issuer, public key, validity).
5. Add extensions (not a CA, key usage, link to CA).
6. Sign the certificate with the CA's key.
7. Save the certificate and private key to files.
8. Return the file paths.

`is_certificate_valid` Explained

This function checks whether a **certificate file** is still valid for at least a given number of days.

Function Signature

```
def is_certificate_valid(cert_path, min_valid_days=10):
```

Parameters:

- `cert_path` → path to the certificate file (PEM format)
- `min_valid_days` → the minimum number of days the certificate should still be valid (default is 10)

Returns:

- `True` → certificate is valid for at least `min_valid_days`
- `False` → certificate expires sooner than `min_valid_days`

---

1. Read the certificate file

```python
with open(cert_path, 'rb') as cert_file:
    cert_data = cert_file.read()
```

- Opens the certificate file in **binary mode** (`rb`)
- Reads the contents into `cert_data`

2: Load the certificate

```python
cert = crypto.load_certificate(crypto.FILETYPE_PEM, cert_data)
```

- Converts the PEM data into an **OpenSSL X.509 certificate object**
- This allows access to the certificate fields like `notAfter` (expiration date)

3: Parse the expiration date

```python
not_after = datetime.datetime.strptime(cert.get_notAfter().decode('ascii'), '%Y%m%d%H%M%SZ')
```

- `cert.get_notAfter()` → returns the expiration date in ASN.1 format (bytes like `b'20251127120000Z'`)
- `.decode('ascii')` → convert bytes to string
- `datetime.strptime(..., '%Y%m%d%H%M%SZ')` → convert string to a Python `datetime` object

Now `not_after` is the **exact expiration date and time** of the certificate.

4: Calculate remaining validity

```python
remaining_days = (not_after - datetime.datetime.now()).days
```

- Subtracts **current date/time** from the expiration date
- Gets the difference in **days**
- This tells us how many days are left until the certificate expires

---

5: Check against minimum required days

```python
return remaining_days >= min_valid_days
```

- Returns `True` if the certificate is valid for at least `min_valid_days`
- Returns `False` if it will expire sooner

Summary

1. Open the certificate file.
2. Load it into a usable certificate object.
3. Read its expiration date.
4. Calculate how many days remain until it expires.
5. Return `True` if it’s valid for at least the minimum days, otherwise `False`.

This function is useful for:

- **Automated certificate renewal checks**
- **Preventing expired certificate errors** in TLS/SSL connections
- **Monitoring certificate health** in network devices

`createSSHClient` Explained

This version of the function is designed to **safely connect to a remote device over SSH** with automatic **retries** and **timeouts**. It’s more robust than the previous version.

---

1. Function signature

```
def createSSHClient(device, username, password, port=22, retries=3, delay=5, timeout=10):
```

- `device` → IP or hostname of the device
- `username` → SSH username
- `password` → SSH password
- `port` → SSH port, default 22
- `retries` → how many times to try connecting if it fails
- `delay` → seconds to wait between retries
- `timeout` → maximum time (seconds) to wait for SSH connection

---

2. Retry loop

```
for attempt in range(1, retries + 1):
```

- Tries connecting **multiple times** if the first attempt fails.
- `attempt` counts the current try (1, 2, 3, …).

---

3. Create and configure SSH client

```
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

- `SSHClient()` → creates a new SSH client object.
- `set_missing_host_key_policy(AutoAddPolicy())` → automatically trusts unknown hosts instead of failing.

---

4. Connect to the device

```
client.connect(
    hostname=device,
    port=port,
    username=username,
    password=password,
    timeout=timeout,
    banner_timeout=timeout,
    auth_timeout=timeout
)
```

- Attempts the actual SSH connection.
- Uses **timeouts** for connection, banner, and authentication to prevent hanging.

---

5. Success message

```
print(f"Connected to {device} on attempt {attempt}")
return client
```

- If connection succeeds, prints a confirmation and **returns the connected SSH client**.

---

6. Handle connection errors

```
except (SSHException, AuthenticationException, TimeoutError) as e:
    print(f"Attempt {attempt} failed to connect to {device}: {e}")
```

- Catches **SSH errors, authentication failures, and timeout errors**.
- Prints the error for visibility.

---

7. Retry or fail

```
if attempt < retries:
    print(f"Retrying in {delay} seconds...")
    time.sleep(delay)
else:
    print(f"Failed to connect to {device} after {retries} attempts.")
    return None
```

- If there are retries left, waits for `delay` seconds and tries again.
- If this was the last attempt, prints a failure message and **returns `None`**.

Summary

1. Try to connect to the device via SSH.
2. If it fails, retry up to `retries` times, waiting `delay` seconds each time.
3. If connection succeeds, return the SSH client object.
4. If all attempts fail, return `None`.
5. Uses timeouts to prevent the program from hanging indefinitely.
6. Automatically accepts unknown host keys.

This version is **safe for looping over many devices** because failed connections don't crash the program, and successful connections can still be used for commands.

`get_certificates` Explained

This function retrieves **TLS/SSL certificates** for a given device, either from **local files** (on the device itself) or via **SCP from a remote device**.

Function Signature

```
def get_certificates(dev, log, targets_dict):
```

Parameters:

- `dev` → a device object (likely from `jnpr.junos.Device`)
- `log` → logging object (for writing logs, not used here in the snippet)
- `targets_dict` → dictionary containing device info and credentials

Returns:

- `ca_cert_path` → path to the CA certificate
- `ca_key_path` → path to the CA private key
- `client_cert_path` → path to the client certificate
- `client_key_path` → path to the client private key

1: Determine if running on the device (onbox)**

```
if onbox:
    local_name = dev.facts['hostname']
    ca_cert_path = os.path.join(CERTS_DIR, 'client-root-ca.crt')
    ca_key_path = os.path.join(CERTS_DIR, 'client-root-ca.key')
    client_cert_path = os.path.join(CERTS_DIR, f'{local_name}.crt')
    client_key_path = os.path.join(CERTS_DIR, f'{local_name}.key')
```

- If the code is **running directly on the device**:
  - Use the device’s hostname as `local_name`.
  - Certificates are expected to be in `CERTS_DIR`.
  - Build paths for the CA certificate, CA key, client certificate, and client key.

2: If running off-device (offbox)**

```
else:
    local_name = dev.facts['hostname'].split('-re')[0]
    client = createSSHClient(local_name, username=targets_dict["secrets"]
["username"], password=targets_dict["secrets"]["password"], port=22)
```

- If the code runs elsewhere:
  - Derive `local_name` from the hostname (remove `-re` suffix).
  - Create an SSH connection to the remote device using credentials in `targets_dict`.

2a: Copy certificates via SCP**

```
with SCPClient(client.get_transport()) as scp:
    scp.get(remote_path=CERTS_DIR, local_path=OFFBOX_CERTS_DIR,
recursive=True)
```

- Uses SCP (Secure Copy) to **download all files from the device’s `CERTS_DIR` to local `OFFBOX_CERTS_DIR`**.
- `recursive=True` ensures directories are copied fully.

2b: Build certificate paths**

```
ca_cert_path = os.path.join(OFFBOX_CERTS_DIR, 'account-1286-server-ca-
qukaydee-com.crt')
ca_key_path = os.path.join(OFFBOX_CERTS_DIR, 'account-1286-server-ca-
qukaydee-com.crt')
client_cert_path = os.path.join(OFFBOX_CERTS_DIR, f'{local_name}.crt')
client_key_path = os.path.join(OFFBOX_CERTS_DIR, f'{local_name}.key')
```

- Sets paths to CA certificate and key (for testing they have the same file).
- Sets paths for client certificate and key using `local_name`.

2c: Handle SCP errors**

```
except SCPException as e:
    print(f'SCP get exception error: {e}')
```

- Prints an error if **SCP fails**, e.g., network problem or missing files.

3: Return certificate paths**

```
return ca_cert_path, ca_key_path, client_cert_path, client_key_path
```

- Returns all four paths for use elsewhere, e.g., generating client certificates, validation, or TLS connections.

Summary

1. Determine if the script is **running directly on the device** (`onbox`) or elsewhere.

2. If onbox, certificates are read locally.

3. If offbox:

   - Connect via SSH to the device
   - Copy certificate files with SCP to a local folder
   - Build paths to CA certificate, CA key, client certificate, and client key

4. Return the file paths for use by other functions.

5. Errors during SCP are caught and printed.

This function is useful for **centralizing certificate management** across multiple network devices, ensuring both **on-device and off-device workflows** are supported.

`upload_certificates` Explained

This function **uploads certificates to a remote device** using SSH and SCP.

Function Signature

```python
def upload_certificates(dev, cert_path, key_path, log, targets_dict):
```

Parameters:

- `dev` → device object (likely from `jnpr.junos.Device`)
- `cert_path` → local path to the certificate file
- `key_path` → local path to the private key file
- `log` → logging object (not used in this snippet)
- `targets_dict` → dictionary containing device info and credentials

Returns:

- Nothing; it **uploads files** to the remote device.

1: Determine the device IP**

```python
local_name = dev.facts['hostname'].split('-re')[0]
# TODO: line below used for testing
local_name = targets_dict[local_name]['ip']
```

- Uses the device hostname to look up the **IP address** from `targets_dict`.
- The `split('-re')[0]` removes any `-re` suffix from hostnames (device naming convention).
- For testing, it directly assigns the IP from `targets_dict`.

2: Create an SSH connection**

```python
client = createSSHClient(local_name, username=targets_dict["secrets"]["username"], password=targets_dict["secrets"]["password"], port=22)
```

- Calls your improved `createSSHClient` function to establish a **secure SSH session** to the device.

3: Upload files via SCP**

```python
with SCPClient(client.get_transport()) as scp:
    cert_files = [cert_path, key_path]
    scp.put(files=cert_files, remote_path=CERTS_DIR)
```

- Opens an SCP session over the SSH connection.
- Defines `cert_files` → a list of the certificate and key files to upload.

- Uploads the files to the device's `CERTS_DIR` directory.

4: Handle errors**

```
except SCPException as e:
    print(f'SCP put exception error: {e}')
```

- Catches **SCP exceptions**, e.g., network errors, permissions issues.
- Prints a message instead of crashing the program.

Summary

1. Determine the IP of the device from its hostname.
2. Connect to the device via SSH.
3. Upload the certificate and private key files to the device using SCP.
4. Catch and report any errors.

💡 Notes / Suggestions

- You could add **return values** or logging to indicate success/failure.
- Consider **checking if the files exist locally** before attempting upload.
- Using the `retries` and `timeout` from your SSH client function can make uploads more robust.

`fetch_ca_certificate` Explained

This function **retrieves the CA (Certificate Authority) certificate and key** from a remote CA server.

Function Signature

```
def fetch_ca_certificate(targets_dict):
```

Parameters:

- `targets_dict` → dictionary containing information about the CA server and credentials

Returns:

- `ca_cert_path` → local path to the downloaded CA certificate
- `ca_key_path` → local path to the downloaded CA private key

1: Get CA server details**

```
ca_server_ip = targets_dict["CA_server"]["ca_server_ip"]
ca_user = targets_dict["CA_server"]["ca_user"]
ca_pass = targets_dict["CA_server"]["ca_pass"]
ca_path = targets_dict["CA_server"]["ca_path"]
ca_cert_name = targets_dict["CA_server"]["ca_cert_name"]
ca_key_name = targets_dict["CA_server"]["ca_key_name"]
```

- Reads IP, username, password, remote path, certificate name, and key name for the CA server from `targets_dict`.
- These are used to **connect to the server and locate the files**.

2: Create an SSH connection to the CA server**

```
client = createSSHClient(ca_server_ip, username=ca_user, password=ca_pass,
port=22)
```

- Uses the `createSSHClient` function to establish a secure SSH connection.
- This allows the script to access files on the remote CA server.

3: Determine local download path**

```
LOCAL_PATH = CERTS_DIR if onbox else OFFBOX_CERTS_DIR
```

- If running **on the device** (`onbox`), save certificates to `CERTS_DIR`.
- If running **off-device**, save to `OFFBOX_CERTS_DIR`.

4: Copy CA certificate and key via SCP**

```
with SCPClient(client.get_transport()) as scp:
    scp.get(remote_path=ca_path, local_path=LOCAL_PATH, recursive=True)
    ca_cert_path = os.path.join(LOCAL_PATH, ca_cert_name)
    ca_key_path = os.path.join(LOCAL_PATH, ca_key_name)
```

- Opens an SCP session over SSH.
- Downloads **all files in `ca_path`** from the CA server to the local folder.
- Constructs **local paths** for the certificate and private key.

5: Handle errors**

```
except SCPException as e:
    print(f'SCP exception error: {e}')
```

- Catches errors like **network failure, permissions issues, or missing files**.
- Prints an error message instead of crashing.

Summary

1. Read CA server connection details from `targets_dict`.
2. Connect to the CA server via SSH.
3. Determine where to save the downloaded files locally.
4. Use SCP to download the CA certificate and key from the server.
5. Return the paths of the local CA certificate and key.
6. If SCP fails, print an error message.

💡 Notes / Suggestions

- You could add **checks** to verify that the files actually exist after download.
- Using your **retry-enabled SSH client** here would make it more robust.
- Optionally, log successful downloads for auditing.

`renew_certificates` Explained

This function **ensures that the CA and client certificates are valid** on a device. It **renews or generates certificates** if they are missing or about to expire, and uploads them to the device if needed.

Function Signature

```
def renew_certificates(dev, log, targets_dict):
```

Parameters:

- `dev` → device object (from `jnpr.junos.Device`)
- `log` → logging object for info messages
- `targets_dict` → dictionary with device info, CA server settings, and credentials

Returns:

- Nothing; performs actions (generates/renews/upload certificates) on the device.

1: Get current certificate paths**

```
if not onbox:
    ca_cert_path, ca_key_path, client_cert_path, client_key_path =
get_certificates(dev, log, targets_dict=targets_dict)
```

- If **running off-device**, use `get_certificates` to download certificate paths from the device.
- If running on-device (`onbox`), paths are assumed to be local.

2: Check if renewal is needed**

```
if not os.path.isfile(client_cert_path) and not
os.path.isfile(ca_cert_path):
    renew = True
elif not is_certificate_valid(client_cert_path):
    renew = True
else:
    renew = False
```

- **Renew if**:
  1. Certificates don't exist yet, or
  2. The client certificate is about to expire (`is_certificate_valid` returns `False`)
- Otherwise, no renewal is needed.

3: Renew or generate certificates**

```
if renew:
    if targets_dict["CA_server"]["CA_cert"]["fetch"]:
        ca_cert_path, ca_key_path = fetch_ca_certificate(targets_dict)
    elif targets_dict["CA_server"]["CA_cert"]["generate"]:
        ca_cert_path, ca_key_path = generate_ca_certificate(ca_cert_path,
ca_key_path, 'Juniper CA')
    else:
        log.info('The CA certificate was manually generated and uploaded')
```

- If renewal is needed:
  - **Fetch CA certificate** from a remote CA server if `fetch` is True.
  - **Generate a new CA certificate** locally if `generate` is True.
  - Otherwise, assume the CA certificate was **manually uploaded**.

4: Generate a client certificate**

```
generate_client_certificate(client_cert_path, client_key_path,
ca_cert_path, ca_key_path, 'client')
```

- Creates a **client certificate signed by the CA**.
- Saves it to `client_cert_path` and `client_key_path`.

5: Upload certificates to device (if offbox)**

```
if not onbox:
    upload_certificates(dev, client_cert_path, client_key_path, log,
targets_dict=targets_dict)
    if targets_dict["CA_server"]["CA_cert"]["fetch"] or
targets_dict["CA_server"]["CA_cert"]["generate"]:
        upload_certificates(dev, ca_cert_path, ca_key_path, log,
targets_dict=targets_dict)
```

- Uploads **client certificate and key** to the device using SCP.
- Uploads **CA certificate and key** only if it was fetched or generated automatically.
- Assumes manually uploaded CA certificates do not need re-uploading.

Summary

1. Check if certificates exist on the device.

2. Check if the client certificate is about to expire.

3. If certificates are missing or expiring:

   - Fetch or generate the CA certificate if needed.
   - Generate a new client certificate signed by the CA.

4. Upload the client certificate and key to the device.

5. Upload the CA certificate and key only if they were fetched/generated automatically.

This function essentially **automates certificate lifecycle management**: it detects missing/expiring certificates, renews them, and ensures the devices always have valid credentials.

`should_check_certs` Explained

This function determines **whether it's time to check certificates** based on a schedule: every 5 days, and only during a certain time window.

Function Signature

```
def should_check_certs():
```

Parameters: None **Returns:** `True` if certificates should be checked now, `False` otherwise.

1: Get the current date and time**

```
now = datetime.datetime.now()
current_time = now.strftime("%H:%M")
```

- `now` → the current date and time
- `current_time` → formatted as `"HH:MM"` (24-hour clock)

2: Check if current time is in the allowed window**

```
if "00:00" <= current_time <= "01:00":
```

- Only proceed if the current time is **between midnight (00:00) and 1 AM (01:00)**.
- This ensures certificate checks happen during off-peak hours.

3: Calculate the day of the month**

```
days_passed = (now - now.replace(day=1)).days + 1
```

- Calculates the **number of days since the start of the month**.
- Adds 1 because `replace(day=1)` makes `days_passed = 0` on the first day.

4: Check if it’s a scheduled day**

```
if days_passed % 5 == 1:
    return True
```

- Checks if the **current day is “day 1 + multiples of 5”** (1, 6, 11, 16, 21, 26, 31).
- Only these days are eligible for a certificate check.

5: Return False otherwise**

```python
return False
```

- If the time is outside 00:00–01:00 or it’s not the scheduled day, the function returns `False`.

Summary

1. Get the current date and time.
2. Only allow checks between **midnight and 1 AM**.
3. Calculate how many days have passed since the start of the month.
4. Only proceed if the current day is **day 1, 6, 11, 16, 21, 26, or 31**.
5. Return `True` if both conditions are met, otherwise `False`.

💡 **Note:**

- This schedule ensures that certificate checks happen **periodically every 5 days** and **during a safe time window** to avoid interfering with production operations.
- The window is **narrow** (1 hour), so if this script is not running at that exact time, the check will be skipped.

`initialize_logging` Explained

This function sets up **Python logging** for your application, including **custom log levels, output to console and files**, and **verbosity control**.

Function Signature

```python
def initialize_logging(args):
```

Parameters:

- `args` → arguments object (e.g., from `argparse`)
  - `args.verbose` → controls logging verbosity
  - `args.trace` → if `True`, creates a detailed debug trace file

Returns:

- `log` → the initialized `logging.Logger` object

1: Define a custom log level**

```
```

```
LOG_NOTICE = 25
logging.addLevelName(LOG_NOTICE, "NOTICE")
```

- Adds a custom log level called `NOTICE` between `WARNING (30)` and `INFO (20)`.
- Allows logging messages with medium importance.

2: Add a `notice` method to Logger**

```
def log_notice(self, message, *args, **kwargs):
    if self.isEnabledFor(LOG_NOTICE):
        self._log(LOG_NOTICE, message, args, **kwargs)
logging.Logger.notice = log_notice
```

- Adds a new `Logger.notice()` method to log at the `NOTICE` level.

3: Define log levels list and capture warnings**

```
LOG_LEVELS = [logging.ERROR, logging.WARNING, LOG_NOTICE, logging.INFO,
logging.DEBUG]
logging.captureWarnings(True)
```

- List of possible log levels in increasing verbosity order.
- Converts Python **warnings** to logging messages automatically.

4: Determine logging level based on verbosity**

```
verbosity = min(args.verbose, len(LOG_LEVELS) - 1)
log_level = LOG_LEVELS[verbosity]
```

- Uses `args.verbose` to choose the logging level from `LOG_LEVELS`.
- Ensures it stays within valid range.

5: Create logger and formatter**

```
log = logging.getLogger()
formatter = logging.Formatter('%(asctime)s %(threadName)-10s %(name)s %
(levelname)-8s %(message)s')
stderr = logging.StreamHandler()
```

```
stderr.setFormatter(formatter)
log.addHandler(stderr)
```

- Creates the root logger.
- Configures a formatter with timestamp, thread name, logger name, log level, and message.
- Adds a **console handler** (`stderr`) so logs appear on the terminal.

6: Configure file logging**

```
if args.trace:
    fh = logging.FileHandler('trace.log')
    fh.setLevel(logging.DEBUG)
    fh.setFormatter(formatter)
    log.setLevel(logging.DEBUG)
    log.addHandler(fh)
    stderr.setLevel(log_level)
elif onbox:
    fh = logging.handlers.RotatingFileHandler(LOG_FILENAME,
maxBytes=10000000, backupCount=5)
    fh.setLevel(logging.INFO)
    fh.setFormatter(formatter)
    log.setLevel(logging.INFO)
    log.addHandler(fh)
    stderr.setLevel(log_level)
else:
    log.setLevel(log_level)
```

- **Trace mode**:
    - Writes all debug logs to `trace.log`.
    - Console shows logs according to verbosity.
- **On-device (`onbox`) mode**:
    - Writes logs to a rotating file (`LOG_FILENAME`).
    - Each file max 10 MB, keeps 5 backups.
- **Off-device mode**:
    - Logs only go to console, at the selected verbosity.

7: Final info message**

```
log.info('Logging modules initialized successfully')
print('Logging modules initialized successfully')
```

- Logs and prints a confirmation message to indicate logging is ready.

Summary

1. Adds a **custom `NOTICE` log level**.
2. Captures Python warnings in the logger.
3. Chooses **log level** based on verbosity argument.
4. Sets up **console logging** and optionally **file logging**:
    - Trace file if `args.trace`
    - Rotating log file if `onbox`
5. Prints/logs a message indicating logging is initialized.
6. Returns the configured `Logger` object for use elsewhere.

💡 Notes / Suggestions

- Using `RotatingFileHandler` prevents logs from growing indefinitely.
- The custom `NOTICE` level is useful for mid-importance messages between `INFO` and `WARNING`.
- Could extend to **per-module logging** for finer control.

`get_previous_key_ids` Explained

This function **reads previously stored key IDs** from a JSON file. It is useful for tracking which cryptographic keys have already been used or generated.

Function Signature

```
def get_previous_key_ids(log, name):
```

Parameters:

- `log` → logging object to write informational messages
- `name` → string used to format the JSON filename

Returns:

- A dictionary containing previous key IDs, or an empty dictionary `{}` if none exist.

1: Try to read the JSON file**

```
```

```
with open(KEYID_JSON_FILENAME.format(name), 'r') as openfile:
    return json.load(openfile)
```

- `KEYID_JSON_FILENAME` is a **template path**, e.g., `'/var/home/admin/{}last_key.json'`.
- `format(name)` replaces `{}` with the provided `name`.
- Reads the JSON file and returns its contents as a Python dictionary.

2: Handle file not found**

```
except FileNotFoundError:
    log.info(f"File to read previous keyId(s) not found")
    return {}
```

- If the JSON file **doesn't exist**, likely because this is the **first time the script runs**:
    - Log an informational message.
    - Return an **empty dictionary**.

3: Handle empty or invalid file**

```
except ValueError:
    log.info(f"No previous keyId(s) found in file")
    return {}
```

- If the file exists but is **empty or contains invalid JSON**, `json.load` raises a `ValueError`.
- Logs a message and returns an empty dictionary.

Summary

1. Construct the JSON file path using `name`.
2. Try to read the file and load previous key IDs into a dictionary.
3. If the file **doesn't exist**, log that it's missing and return `{}`.
4. If the file exists but is **empty or invalid**, log that no key IDs are found and return `{}`.

💡 **Notes / Suggestions**

- This function ensures the script **doesn't crash on first run**.
- Returning `{}` is safe for subsequent operations that rely on previous keys.
- Could extend with a **file lock** if multiple processes might write to the same JSON file.

`save_key_ids` Explained

This function **saves cryptographic key IDs** to a JSON file for later retrieval. It complements the `get_previous_key_ids` function.

Function Signature

```
def save_key_ids(key_dict, local_name):
```

Parameters:

- `key_dict` → dictionary containing key IDs to save
- `local_name` → string used to format the JSON filename

Returns: None (performs a file write)

1: Construct the file path**

```
KEYID_JSON_FILENAME.format(local_name)
```

- `KEYID_JSON_FILENAME` is a template path, e.g., `'/var/home/admin/{}last_key.json'`.
- Replaces `{}` with `local_name` to create a unique filename for each device or context.

2: Write key IDs to the file**

```
with open(KEYID_JSON_FILENAME.format(local_name), 'w+') as outfile:
    outfile.write(json.dumps(key_dict))
```

- Opens the file in **write mode (`w+`)**, creating it if it doesn't exist.
- Serializes the dictionary (`key_dict`) to JSON format using `json.dumps`.
- Writes the JSON data to the file.
- Overwrites any existing file content.

3: Print confirmation**

```
print(f'Saved the key IDs to the JSON file:
{KEYID_JSON_FILENAME.format(local_name)}')
```

- Prints a message confirming that the key IDs were successfully saved.
- Useful for debugging or tracking the workflow.

Summary

1. Construct the JSON filename using `local_name`.
2. Open the file (create it if it doesn't exist).
3. Convert the key dictionary into JSON and write it to the file.
4. Print a confirmation message.

💡 Notes / Suggestions

- You could **add logging** instead of `print` for consistency with `get_previous_key_ids`.
- Consider **atomic writes** (writing to a temp file then renaming) to prevent data corruption if the script is interrupted.
- Using JSON ensures the file can easily be read back with `get_previous_key_ids`.

`fetch_kme_key` Explained

This function **fetches cryptographic keys from a Key Management Entity (KME)** using HTTPS requests with client and CA certificates for authentication.

Function Signature

```python
def fetch_kme_key(session, local_name, log, remote_mnmgt_add, kme_host, 
key_id, additional_slave_SAE_IDs=None):
```

Parameters:

- `session` → a `requests.Session()` object used for making HTTP requests
- `local_name` → the local device name (used to find its certificates)
- `log` → logging object
- `remote_mnmgt_add` → the master device or remote address for key retrieval
- `kme_host` → base URL of the KME server
- `key_id` → optional; if provided, fetch a specific key, otherwise fetch new keys
- `additional_slave_SAE_IDs` → optional list of additional devices for which to fetch keys

Returns:

- JSON object containing keys if the request succeeds, or `None` if it fails

---

1: Determine certificate paths**

```python
if onbox:
    client_crt = CERTS_DIR + local_name + '.crt'
    client_key = CERTS_DIR + local_name + '.key'
    CLIENT_CERT = (client_crt, client_key)
    CA_CERT = CERTS_DIR + 'client-root-ca.crt'
```

```
else:
    client_crt = OFFBOX_CERTS_DIR + '/' + local_name + '.crt'
    client_key = OFFBOX_CERTS_DIR + '/' + local_name + '.key'
    CLIENT_CERT = (client_crt, client_key)
    CA_CERT = OFFBOX_CERTS_DIR + '/' + 'root.crt'
```

- Determines where to find **client certificate, client key, and CA certificate** depending on whether the script is running on-device (`onbox`) or off-device.
- These certificates are used for **mutual TLS authentication**.

2: Make an HTTPS request to the KME**

```
if key_id:
    response = session.get(f"
{kme_host}/api/v1/keys/{remote_mnmgt_add}/dec_keys?key_ID={key_id}",
verify=CA_CERT, cert=CLIENT_CERT, headers={"Content-Type":
"application/json"})
else:
    response = session.get(f"
{kme_host}/api/v1/keys/{remote_mnmgt_add}/enc_keys", verify=CA_CERT,
cert=CLIENT_CERT, headers={"Content-Type": "application/json"})
```

- If `key_id` is provided → fetch a **specific decryption key**.
- If not → fetch **new encryption keys** from the KME.
- Uses `CLIENT_CERT` for client authentication and `CA_CERT` to verify the KME server.
- Sets `Content-Type` header to `"application/json"`.

3: Check the response**

```
if response.status_code == 200:
    response_json = response.json()
    return response_json
else:
    print(f'Request failed with status code {response.status_code}')
    print(response.text)
```

- If the server responds with `HTTP 200 OK` → parse the JSON and return it.
- Otherwise → print the status code and response content for debugging.

4: Handle exceptions**

```
except requests.RequestException as e:
    log.error(f"KME request failed: {e.response.text}")
    print(f"KME request failed: {e.response.text}")
    return None
```

- If the HTTP request fails due to **network issues, SSL errors, or server errors**, log the error and return `None`.

Summary

1. Determine the **client and CA certificate paths** depending on whether the script is running on the device or off-device.
2. Make a **mutual TLS HTTPS request** to the KME:
   - If `key_id` is given → fetch an existing key.
   - Otherwise → fetch new keys for the master device.
3. Parse and return the JSON response if successful.
4. Log and print errors if the request fails.

💡 Notes / Suggestions

- `CLIENT_CERT` and `CA_CERT` ensure **mutual authentication** between the client and the KME.
- The function currently prints a lot of debug info; consider using `log.debug()` instead of `print` for cleaner production logs.
- `additional_slave_SAE_IDs` is defined but not used; you could extend the request payload to include it if needed.

`check_and_apply_initial_config` Explained

This function **checks whether the initial MACsec configuration is applied** on a network device and applies it if it isn't. It is intended to run **only once** on the first execution of the script.

Function Signature

```python
def check_and_apply_initial_config(dev, targets_dict, log):
```

Parameters:

- `dev` → a `jnpr.junos.Device` object representing the network device
- `targets_dict` → dictionary containing device details, interfaces, KME info, and CA server info
- `log` → logging object for informational and error messages

Returns: None (applies configuration directly to the device)

1: Extract device information**

```
device_name = dev.facts['hostname'].split('-re')[0]
device_ip = targets_dict[device_name]["ip"]
c_a = targets_dict["CA_server"]["c_a"]
interfaces = targets_dict[device_name]["interfaces"]
kme_name = targets_dict[device_name]["kme"]["kme_name"]
kme_ip = targets_dict[device_name]["kme"]["kme_ip"]
start_time = targets_dict["system"]["event_options"]["start_time"]
```

- Retrieves device name, IP, **connectivity-association (CA) name**, list of interfaces, KME host info, and event start time from `targets_dict`.

2: Check if MACsec configuration already exists**

```
config_check = 
dev.rpc.get_config(filter_xml=E.configuration(E.security(E.macsec(E('connectivity-association', E.name(c_a))))))
if config_check.find('.//name') is not None:
    log.info("Initial macsec configuration already applied on the device: {}.".format(device_name))
    return
```

- Uses Juniper RPC to check the device's configuration.
- If the **connectivity-association already exists**, it logs a message and exits the function.

3: Build the MACsec commands**

```
initial_macsec_commands = [
    f"set security macsec connectivity-association {c_a} cipher-suite gcm-aes-xpn-256",
    f"set security macsec connectivity-association {c_a} security-mode static-cak",
    f"set security macsec connectivity-association {c_a} pre-shared-key ckn abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678",
    f"set security macsec connectivity-association {c_a} pre-shared-key cak abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678abcd1234abcd5678"
]
```

- Defines the **basic MACsec CA configuration**, including cipher, security mode, and pre-shared keys.

3a: Add interface-specific commands**

```
for interface in interfaces:
    initial_macsec_commands.extend([
        f"set security macsec interfaces {interface} apply-macro qkd kme-ca false",
        f"set security macsec interfaces {interface} apply-macro qkd kme-host {kme_name}",
        f"set security macsec interfaces {interface} apply-macro qkd kme-port 443",
        f"set security macsec interfaces {interface} connectivity-association {c_a}",
        f"set security macsec interfaces {interface} apply-macro qkd kme-keyid-check true",
        f"set system static-host-mapping {device_name} inet {device_ip}"
    ])
```

- Configures each network interface to use MACsec, apply QKD macros, associate with KME, and map device IP.

3b: Add event options for on-device (`onbox`) mode**

```
if onbox:
    initial_macsec_commands.extend([
        f"set event-options generate-event every10mins time-interval 600 start-time {start_time}",
        f"set event-options policy qkd events every10mins",
        f"set event-options policy qkd then event-script ETSIA_v3.1.0_Turkcell_Phase2_v1.py",
        f"set event-options event-script file ETSIA_v3.1.0_Turkcell_Phase2_v1.py python-script-user admin",
        f"set event-options traceoptions file script.log",
        f"set event-options traceoptions file size 10m",
    ])
```

- Sets up **automated event generation** every 10 minutes to run a QKD Python script.
- Configures **logging and trace options** for the event script.

4: Apply the configuration**

```
try:
    dev.timeout = 300
    with Config(dev) as cu:
        cu.lock()
        for command in initial_macsec_commands:
            cu.load(command, format='set')
        cu.commit()
        log.info("Initial macsec configuration applied successfully on the
device: {}.".format(device_name))
except Exception as e:
    log.error(f'Initial macsec configuration commit failed: {e}')
finally:
    cu.unlock()
    time.sleep(60)
```

- Locks the device configuration.
- Loads all MACsec commands using **set-style configuration**.
- Commits the configuration.
- Unlocks the configuration and waits **60 seconds** to allow changes to propagate.
- Logs success or error messages.

Summary

1. Extracts device details and KME info.
2. Checks if MACsec configuration already exists; skips if present.
3. Builds a list of **set commands** for MACsec and interface configuration.
4. Adds event scheduling for running the QKD script if on the device.
5. Locks the device configuration, loads all commands, commits, unlocks.
6. Waits for 60 seconds to ensure configuration is applied.
7. Logs success or failure.

💡 Notes / Suggestions

- Ensures the initial MACsec configuration is **idempotent** (applied only once).
- Uses **QKD macros** and static keys for secure communication.
- Event scripts automate recurring QKD key operations.
- Consider **error handling for individual commands** in large deployments.

`get_key_id_from_master` Explained

This function **retrieves the Key ID file from the master device** in a QKD (Quantum Key Distribution) setup. The Key ID file keeps track of which encryption keys have been generated or used by the master.

Function Signature

```
def get_key_id_from_master(dev, log, targets_dict):
```

Parameters:

- `dev` → device object (Juniper Device) — not directly used here but passed for consistency
- `log` → logging object for informational messages (currently `print` is used)
- `targets_dict` → dictionary containing device information, roles, secrets, and paths

Returns:

- Path to the Key ID file on the local system, or `None` if it cannot be retrieved.

1: Determine the master device**

```
master_name = targets_dict["qkd_roles"]['master']
master_key_id_file = KEYID_JSON_FILENAME.format(master_name)
```

- Looks up the **master device name** from `targets_dict`.
- Constructs the Key ID filename for the master device using `KEYID_JSON_FILENAME` template.

2: Handle off-device scenario (`not onbox`)**

```
if not onbox:
    if os.path.exists(master_key_id_file):
        return master_key_id_file
    else:
        print('the master key id file does not exist')
        return None
```

- If running off-device:
  - Checks if the Key ID file already exists locally.
  - If yes → return its path.
  - If no → log a message and return `None`.

3: Handle on-device scenario (`onbox`)**

```
client = createSSHClient(master_name, username=targets_dict["secrets"]
["username"], password=targets_dict["secrets"]["password"], port=22)
with SCPClient(client.get_transport()) as scp:
    scp.get(remote_path=master_key_id_file, local_path=master_key_id_file,
preserve_times=True)
```

- Connects to the **master device over SSH** using `createSSHClient`.
- Uses `SCPClient` to **copy the master Key ID file** from the master device to the local system.
- `preserve_times=True` keeps the original file timestamps.

4: Error handling**

```
except SCPException as e:
    print(f'SCP get exception error: {e}')
    return None
```

- If the SCP transfer fails (network issues, missing file, permission errors), logs the exception and returns `None`.

5: Return local path**

```
return master_key_id_file
```

- Returns the **path to the retrieved Key ID file** on the local system.
- This file can then be read to get the master's current key IDs.

Summary

1. Determine the **master device** and its Key ID filename.

2. If running off-device (`not onbox`):

    - Check if the file exists locally → return path or `None`.

3. If running on-device (`onbox`):

    - Connect to the master over SSH.
    - Copy the Key ID file via SCP.
    - Return the local path.

4. Logs any errors encountered during SCP transfer.

💡 **Notes / Suggestions**

- Currently uses `print()` for messages; consider using `log.info()` or `log.error()` for consistency.
- Could add retries or timeout handling for SCP in case of transient network issues.
- Ensures that **all devices in the QKD network can access the master's Key ID** to stay synchronized.

`process` Function Explained

The `process` function is the **core logic for applying MACSEC keys to a Junos device**. It handles fetching keys from the KME (Key Management Entity), updating the device configuration, and storing the key IDs for future runs.

Function Signature

```python
def process(dev, targets_dict, log):
```

Parameters:

- `dev`: Juniper device object from `jnpr.junos.Device`
- `targets_dict`: Dictionary containing all device info, roles, credentials, interfaces, and KME info
- `log`: Logger object for logging events

1: Get MACSEC configuration**

```python
conf_filter = E.configuration(E.security(E.macsec()))
config_xml = dev.rpc.get_config(filter_xml=conf_filter)
macsec_xml = config_xml.find('security/macsec')
```

- Fetches the current MACSEC configuration from the device using Junos XML RPC.
- Prepares the configuration object for further modifications.

2: Prepare key fetching session**

```python
session = requests.Session()
new_key_dict = {}
qkd_macsec_xml = E.macsec()
commit = False
```

- Creates a `requests.Session` for communicating with the KME API.
- `new_key_dict` will store the Key IDs fetched for this device.
- `qkd_macsec_xml` is a placeholder for new MACSEC configuration.
- `commit` flag indicates whether to apply changes.

3: Determine CA name and local hostname**

```python
ca_name = macsec_xml.findtext('connectivity-association/name')
local_name = dev.facts['hostname'].split('-re')[0]
kme_host = targets_dict[local_name]["kme"]["kme_name"]
```

- Extracts the connectivity association (CA) name from the MACSEC config.
- Determines the local hostname (removes `-re` suffix).
- Finds the KME host associated with this device.

4: Fetch Key from KME**

- **Master Device Logic:**

```
if targets_dict["qkd_roles"]['master'] == local_name:
    remote_mnmgt_add = targets_dict["qkd_roles"]['slave']
    r = fetch_kme_key(session, local_name, log, remote_mnmgt_add,
kme_host, key_id=None)
    key = r['keys'][0]
    new_key_dict[local_name] = key['key_ID'].strip()
```

- If this device is the **Master**, fetch a new encryption key from the KME for the Slave device.

- Updates `new_key_dict` with the received Key ID.

- **Slave Device Logic:**

```
else:
    remote_mnmgt_add = targets_dict["qkd_roles"]['master']
    # Retry loop for master key
    master_key_id_file = get_key_id_from_master(dev, log, targets_dict)
    master_key_dict = get_previous_key_ids(log, remote_mnmgt_add)
    new_key_dict[local_name] = master_key_dict[remote_mnmgt_add]
```

- If the device is a **Slave**, fetch the **Key ID from the Master device**.
- Retries every 5 seconds until a new Key ID is available.
- Updates `new_key_dict` with the Master's Key ID.

5: Update MACSEC CAK and CKM**

```
qkd_ca_xml.find('pre-shared-key/ckn').text = CKN_PREFIX +
uuid.UUID(key['key_ID']).hex
qkd_ca_xml.find('pre-shared-key/cak').text =
str(base64.b64decode(key['key']).hex())[:64]
qkd_macsec_xml.append(qkd_ca_xml)
commit = True
```

- Updates the **connectivity association key name (CKN)** and **connectivity association key (CAK)** in the configuration XML.
- Appends the new MACSEC config to `qkd_macsec_xml`.
- Sets `commit = True` so it will be applied.

6: Root Authentication**

```
root_auth_xml = E.system(
    E("root-authentication",
        E("encrypted-password", targets_dict[local_name]["root_enc_pass"])
    )
)
```

- Prepares root authentication XML (encrypted password) to be included in the configuration if needed.

7: Commit Configuration**

```
qkd_config_xml = E.configuration(E.security(qkd_macsec_xml))
with Config(dev) as cu:
    cu.lock()
    cu.load(qkd_config_xml, format='xml', merge=True)
    cu.commit()
    cu.unlock()
```

- Converts XML to a Junos configuration object.
- Locks the configuration database.
- Loads the new configuration and **merges it with existing settings**.
- Commits the configuration changes to the device.
- Unlocks the configuration database.
- Logs success or failure of the commit.

8: Save Key IDs**

```
save_key_ids(new_key_dict, local_name)
```

- Saves the updated Key IDs to a local JSON file for future reference.

9: Optional Verification**

```
mka_session_info_xml =
dev.rpc.get_mka_session_information({'format':'text'}, summary=True)
```

- Fetches **MKA (MACsec Key Agreement) session information** for debugging or verification.

Summary

1. Fetch current MACSEC configuration from the device.
2. Determine if the device is **Master or Slave** in the QKD network.
3. **Master:** request a new key from the KME for Slave.
4. **Slave:** fetch the key ID from the Master device. Retry until a new key is available.
5. Update MACSEC keys (`CKN` and `CAK`) in the device configuration.
6. Commit the configuration changes to the device.
7. Save the new key IDs locally for future runs.
8. Optionally, fetch MACSEC session information for verification.

💡 Notes / Suggestions

- Currently prints a lot of debug information — you could replace most `print` statements with `log.debug` for cleaner logs.
- The retry logic for Slave devices ensures **synchronization between Master and Slave**, but it might hang if the Master never updates the key file.
- `commit = True` is controlled dynamically based on whether new keys are available.

`get_args()` Function Explained

The `get_args` function is responsible for **defining and parsing command-line arguments** for the script. It uses Python's `argparse` module.

Function Signature

```
def get_args():
```

- No input parameters.
- Returns a **Namespace object** containing the parsed arguments.

1: Create Argument Parser**

```
parser = argparse.ArgumentParser()
```

- Creates an `ArgumentParser` object to handle command-line arguments.
- This object will automatically generate help messages.

2: Define Arguments**

- Threads (`-t` / `--threads`)**

```
parser.add_argument("-t", "--threads", type=int, help="Number of threads to use")
```

- Optional argument to specify how many threads the script should run.
- Example usage:

```
python script.py -t 5
```

- Verbose (`-v` / `--verbose`)**

```
parser.add_argument('-v','--verbose', default=0, action='count', help="increase verbosity level")
```

- Optional flag to increase the verbosity of logs.
- Can be specified multiple times (`-vv` for more verbose).
- Default is `0` (no extra verbosity).

Example:

```
python script.py -vv
```

- `action='count'` automatically counts how many times `-v` is used.

- Trace (`-tr` / `--trace`)**

```
parser.add_argument('-tr','--trace', action='store_true', help="dump debug level logs to trace.log file")
```

- Optional flag to enable **trace logging**.
- Stores a boolean `True` if present on the command line, otherwise `False`.

Example:

```
python script.py --trace
```

3: Parse Arguments**

```
return parser.parse_args()
```

- Reads the command-line arguments provided when the script is run.
- Returns them as a `Namespace` object, which can be accessed like:

```
args = get_args()
print(args.threads)  # Number of threads
print(args.verbose)  # Verbosity level
print(args.trace)    # True/False
```

Summary

1. Prepares the script to accept command-line options.
2. Supports **threads**, **verbosity**, and **trace logging** flags.
3. Returns a convenient object (`args`) containing all user-specified options.

`main()` Function Explanation

The `main()` function orchestrates the full workflow for **configuring devices with QKD MACSEC and fetching keys from KME**.

---

1. **Parse Command-Line Arguments**

```
args = get_args()
```

- Reads arguments like:
  - Number of threads to use (`--threads`)
  - Verbosity level (`-v`)
  - Trace mode (`--trace`)

---

2. **Initialize Logging**

```
log = initialize_logging(args)
```

- Sets up logging based on verbosity or trace mode.
- Logs to console, file, or rotating log file depending on environment (`onbox` or offbox).

---

3. **Define Target Devices and System Info**

```
targets_dict = {...}
```

- Contains:
  - **System-wide settings** (e.g., max threads, start time for events)
  - **Secrets** (SSH username/password)
  - **CA server info** (certificate fetch/generate options, paths)
  - **QKD roles** (master, slave, additional devices)
  - **Device-specific data** (IP, root password, interfaces, KME details)

Essentially, this is your **entire network and KME configuration** in a single dictionary.

---

4. **Offbox vs Onbox Workflow**

```
if not onbox:
    # Offbox processing
elif onbox:
    # Onbox processing
```

- **Offbox**: Script runs from a separate host (not on the devices themselves).
  - It calculates **threads** and distributes devices across them.
  - Launches `req_thread` for each set of devices.
  - Waits for all threads to finish (`t.join()`).
- **Onbox**: Script runs directly on a device.
  - Connects to the device using `Device()`.
  - Runs the configuration steps **synchronously**:
    1. `check_and_apply_initial_config()` — applies initial MACSEC configuration if not already present.
    2. `process()` — fetches keys from KME and updates MACSEC configuration.
  - Uses profiling (`prof.start` / `prof.stop`) to measure execution time for each step.

---

5. **Threading Logic (Offbox)**

- Creates a list of devices to process in threads:

```
dlist = [master, slave, additional slaves]
```

- Splits the list according to the number of threads.

- Calls `req_thread()` for each batch, which is **decorated with `@background`** to run in a separate thread.
- Waits for all threads to finish before continuing.

---

6. **Onbox Logic**

- Runs everything sequentially on the device.
- Uses profiling to measure time taken by:
  - Certificate renewal (if needed)
  - Initial MACSEC configuration
  - Fetching keys and updating MACSEC configuration

---

7. **Error Handling**

- Logs and prints exceptions if SSH or device operations fail.

Summary

- **Purpose:** Automate QKD MACSEC configuration on multiple devices.
- **Supports two modes:** Onbox (direct device) or Offbox (external host with threading).
- **Workflow:**
  1. Parse CLI args
  2. Initialize logging
  3. Load device and KME configuration
  4. Apply initial MACSEC config
  5. Fetch keys from KME
  6. Update device configuration
  7. Save new Key IDs for tracking

Below is the **"How to Run"** section. It includes clear steps, examples for both **onbox** and **offbox**, and environment expectations.

How to Run the Script

This section explains how to run the QKD–MACsec automation script in both **offbox** (management host) and **onbox** (directly on a router) modes, with real and practical examples.

Requirements

Offbox Mode (recommended for managing many devices)

You need a Linux/BSD/Unix host with:

- Python 3.7+
- Juniper PyEZ (`junos-eznc`)
- `paramiko`, `scp`, `pyOpenSSL`
- SSH reachability to each device
- The `targets.yaml` or `targets.json` configuration file

Onbox Mode (script runs directly on JunOS)

You need:

- JunOS with `event-options` and Python enabled
- Script copied to `/var/db/scripts/event/`
- Proper permissions for `python-script-user`

Script Usage

The script supports the following command-line options:

Option	Description
`-t, --threads`	Number of parallel threads (offbox only)
`-v`	Increase verbosity (can be used multiple times)
`-tr, --trace`	Log debug-level output to `trace.log`

Running in Offbox Mode

Basic Example

Run the script with default settings:

```
python3 qkd_macsec.py
```

If no thread count is provided, the script uses **1 thread**.

Using Multiple Threads

To run on 10 devices in parallel:

```
python3 qkd_macsec.py --threads 10
```

This will:

- Spawn 10 worker threads
- Each thread executes `req_thread()`
- Each device goes through the full workflow:
    - Initial MACsec config (if not applied)
    - Certificate validation/renewal
    - Key ID retrieval (master → slave)
    - KME key fetch
    - MACsec key/CA update
    - Key ID persistence

Verbose Output

More `-v` flags increase log detail.

```
python3 qkd_macsec.py -vv
```

- `-v`: INFO
- `-vv`: DEBUG
- `-vvv`: VERY verbose

Write Debug Information to trace.log

```
python3 qkd_macsec.py --trace
```

This enables debug-level trace logging in:

```
./trace.log
```

Full Example with All Options

```
python3 qkd_macsec.py --threads 5 -vv --trace
```

Meaning:

- Use 5 threads
- Verbose debug messages to console

- Full debug tracing logged to `trace.log`

Running in Onbox Mode (JunOS Event Script)

The script can also run as an **event-triggered script** using `event-options`.

Copy the Script to the Device

Place the file in:

```
/var/db/scripts/event/qkd_macsec.py
```

Make sure permissions allow execution:

```
chmod 755 /var/db/scripts/event/qkd_macsec.py
```

Configure Event-Options (example)

Example JunOS config:

```
set event-options generate-event every10mins time-interval 600 start-time
00:00:00
set event-options policy qkd events every10mins
set event-options policy qkd then event-script qkd_macsec.py
set event-options event-script file qkd_macsec.py python-script-user admin
set event-options traceoptions file script.log
set event-options traceoptions file size 10m
```

This will:

- Trigger the script every 10 minutes
- Run it as user `admin`
- Log to `/var/log/script.log`

Manual Onbox Execution

You can manually run the script on the router:

```
python3 /var/db/scripts/event/qkd_macsec.py
```

This follows the **same device workflow**:

- Certificate validation/renewal
- Key ID fetch (from master or local file)
- KME communication
- MACsec updates

File Outputs

During execution, the script generates:

Key ID file

```
<hostname>_keyID.json
```

Certificates

```
ca_cert.pem
ca_key.pem
client_cert.pem
client_key.pem
```

Logs

- `script.log` (onbox trace)
- `trace.log` (offbox debug trace if enabled)
- Console output based on `-v` flags

Typical Workflow Example

Scenario

You have 3 devices:

- `PE1` (master)
- `PE2` (slave)
- `PE3` (slave)

And you want to configure all of them at once with QKD–MACsec.

Run:

```
python3 qkd_macsec.py --threads 3 -vv --trace
```

The script will:

1. Connect to PE1, PE2, PE3
2. Apply initial MACsec config (if needed)
3. Validate certificates; renew if needed
4. Fetch CA and client certificates
5. PE1 fetches key from KME → saves key ID
6. PE2/PE3 retrieve master key ID
7. Each device fetches its KME key
8. Final MACsec config is applied
9. All key IDs saved for next iteration

Confirming Device Configuration

On a JunOS device, check MACsec settings:

```
show security macsec
show security certificates local
show log script.log
```

Check event triggers:

```
show event-options policy
show event-options event-script
```

Appendix

- A complete `targets.yaml` example**
- A full Troubleshooting section**

---

targets.yaml (Example Configuration File)

This file defines all devices, roles (master/slave), KME endpoints, credentials, CA behavior, and system parameters.

```
system:
  maxthreads: 4
  event_options:
    start_time: "2025-03-23.13:00:00"

secrets:
  username: "admin"
  password: "admin123!"

CA_server:
  CA_cert:
    fetch: false
    generate: false
  c_a: "CA_basic"
  ca_server_ip: "192.168.10.10"
  ca_path: "/opt/ca/"
  ca_cert_name: "root_ca_cert.pem"
  ca_key_name: "root_ca_key.pem"
  ca_user: "caadmin"
  ca_pass: "capassword12!"

qkd_roles:
  master: "acx-1"
  slave: "acx-2"
  additional_slave_SAE_IDs: ["acx-3", "acx-4"]

acx-1:
  root_enc_pass:
"$6$3eHulK1c$Yq.kaamV8hcuviwRebQI4gUMRSOGVIiBN8o/QTw7sfZ4GCfExd3TjyuUrsyrg
foBW3xNQVT5/gtGg6.S09okg0"
  ip: "9.173.8.201"
  interfaces:
    - "et-0/0/20:2"
  kme:
    kme_name: "https://idq-1"
    kme_ip: "9.173.9.102"

acx-2:
  root_enc_pass:
"$6$bOeXyUQ7$oefu0aDycBhyLGDE.TCExBrdVkYOhg2IOesMVwRQvid9iDpMzwm5yZPvYhKlB
u3sZ0YbHBAH0ro5SQWTnscWf."
  ip: "9.173.8.202"
  interfaces:
    - "et-0/0/20:2"
  kme:
    kme_name: "https://idq-2"
    kme_ip: "9.173.9.103"

acx-3:
  root_enc_pass: "$6$someotherpass$morehashdata"
  ip: "9.173.8.203"
  interfaces:
    - "et-0/0/20:2"
  kme:
    kme_name: "https://idq-3"
    kme_ip: "9.173.9.104"

acx-4:
  root_enc_pass: "$6$randompass$hashxxxyyy"
  ip: "9.173.8.204"
```

```
    interfaces:
      - "et-0/0/20:2"
    kme:
      kme_name: "https://idq-4"
      kme_ip: "9.173.9.105"
```

Troubleshooting Guide

This section helps diagnose common issues when running the QKD–MACsec automation script.

Script Fails to Connect to Device (SSH Error)

Symptoms:

- `TimeoutError`
- `Authentication failed`
- `SSHException: Error reading SSH protocol banner`

Causes:

- Wrong username/password
- Device not reachable
- SSH disabled or filtered by firewall

Fix:

```
ssh admin@<device-ip>
```

If SSH fails manually → fix network or credentials.

Ensure device allows Python script execution:

```
set system scripts language python3
```

`KeyID JSON file not found`

Message:

```
File to read previous keyId(s) not found
```

This is **not an error** during the first run.

Fix:

None required — the script will create `<device>_keyID.json` automatically.

MACsec Not Coming Up After Script Execution

Checks:

1. Verify MACsec status:

```
show security macsec
```

2. Verify key exchange status:

```
show security macsec connectivity-association
```

3. Check KME communication:

```
show log messages | match kme
```

Common Causes:

- Key mismatch between master and slaves
- Wrong interface in `targets.yaml`
- KME IP unreachable
- CA certificate mismatch

Certificate Creation or Upload Fails

Symptoms:

- Missing certificate fields
- “invalid certificate format”
- Upload fails on device

Fix Checklist:

- Ensure OpenSSL is installed (offbox)
- Ensure CA paths are correct
- Ensure CA and client keys are readable by script
- Check certificate validity:

```
openssl x509 -in client_cert.pem -text
```

`SCPException` When Fetching Master KeyID File

Causes:

- Wrong master hostname
- Wrong file path
- SCP blocked

Fix:

Check remote file existence:

```
file list /var/home/admin/acx-1_keyID.json
```

Verify scp manually:

```
scp admin@acx-1:/var/home/admin/acx-1_keyID.json .
```

Threads Start but Nothing Happens (Offbox Mode)

Causes:

- `targets.yaml` has no devices
- Threads created but `req_thread()` exits early due to local error
- Logging verbosity too low to see details

Fix:

Run with high verbosity:

```
python3 qkd_macsec.py --threads 5 -vvv
```

Check `trace.log` if enabled.

Onbox Mode: Script Not Triggering

Causes:

- Event-options misconfigured

- Script not in correct directory
- Wrong owner/permissions

Fix Steps:

1. Verify script placement:

```
/var/db/scripts/event/<your-script>.py
```

2. Verify JunOS event options:

```
show event-options policy
show event-options event-script
```

3. Check logs:

```
show log script.log
```

KME Not Returning Key

Symptoms:

- “key_id not found”
- “failed to fetch key”
- “400/403 from KME”

Fix:

- Verify KME is reachable:

```
curl https://<kme-ip>/status
```

- Validate TLS certificates
- Ensure device hostname matches KME policy

Device Fails at `check_and_apply_initial_config()`

Symptoms:

- Interface not found
- MACsec configuration rejected

Fix:

Verify interfaces in `targets.yaml` match device:

```
show interfaces terse | match et-0/0/20
```

Script Crashes with Python Exceptions

Common Causes:

- Missing dependencies
- Invalid path
- Typo in hostname
- Corrupted JSON keyID file

Fix:

Delete corrupted keyID file:

```
rm <device>_keyID.json
```

Re-run script.

Python Profiler Class

This document describes a **Python 3 profiler** script that allows you to measure and log the execution time of tasks (or code sections) in your Python code, in milliseconds. It's designed to help you see **how long different parts of a script take to run**, including counts, cumulative time, min/max, and averages. The profiler includes a **context manager interface** for clean and hierarchical task timing.

Overview

The profiler allows you to:

- Measure **execution time** of named tasks (allows timing **named tasks** with start/stop calls).
- Collects stats: track **count**, **cumulative time**, **average**, **minimum**, and **maximum** durations.
- Writes a detailed **profiling log** to a Log file, in a human-readable format.
- Profile and support **nested tasks** (Outer → Inner → Loop) with automatic hierarchical timing.
- Useful for **benchmarking parts of a Python program** without external libraries.

The script is fully compatible with **Python 3** and can be used as a module in other scripts.

Features

- Task timing using `start(task)` and `stop(task)` methods.
- Verbose logging of task start/stop events.
- Automatic statistics: cumulative time, min/max time, average, and iteration counts.
- Annotate the profiling log with custom text.
- Context manager interface for clean nested profiling:

```python
with profiler.task("MyTask"):
    # code to profile
```

- Clean, modern Python 3 implementation using f-strings and `with open(...)`.
- Output summary sorted by **cumulative execution time**.

Installation

Simply copy the `profiler.py` file into your project. No external dependencies are required. The script uses only Python standard library modules (`time`, `datetime`, `contextlib`).

Usage

Basic start/stop profiling

```python
from profiler import Profile
import time

profiler = Profile(file_path="profile.log", verbose=True)

profiler.start("TaskA")
time.sleep(1)  # Simulate work
profiler.stop("TaskA")

profiler.close()
```

Context manager usage (recommended)

```python
from profiler import Profile
import time

profiler = Profile(file_path="profile.log", verbose=True)

with profiler.task("Outer"):
    time.sleep(1)  # Some code to profile

profiler.close()
```

- Using the **context manager** is cleaner and automatically handles `start()` and `stop()`.
- Supports nested tasks without manually tracking start/stop.

Nested loops example

This example demonstrates **nested profiling** for loops, similar to the original test case:

```python
from profiler import Profile
import time

profiler = Profile(file_path="prof.test", verbose=True)
profiler.annotate("Unit test of profiler class with context manager")

with profiler.task("Outer"):
    for i in range(10):
        with profiler.task("Inner"):
            for j in range(5):
                with profiler.task("Loop"):
                    print("Processing inner loop work...")
                    time.sleep(0.1)  # Simulate work

profiler.close()
```

- Tasks `"Outer"`, `"Inner"`, and `"Loop"` are tracked separately.
- Automatically calculates **count, cumulative, min/max, and average times** for each task.

Profiler Output

A typical profiling log looks like this:

```
Profiler started: 2025-11-27 12:00:00
Unit test of profiler class with context manager
Starting task "Outer"
Starting task "Inner"
Starting task "Loop"
Finished task "Loop" iter 1 elapsed 100ms (0:00:00.100000)
...

=== Summary ===
Task: Loop | Count: 50 | Cumulative: 5000ms (0:00:05) | Avg: 100.00ms |
Min: 100ms | Max: 100ms
Task: Inner | Count: 10 | Cumulative: 5000ms (0:00:05) | Avg: 500.00ms |
Min: 500ms | Max: 500ms
Task: Outer | Count: 1 | Cumulative: 5000ms (0:00:05) | Avg: 5000.00ms |
Min: 5000ms | Max: 5000ms
```

```
GLOBAL execution elapsed: 6000ms (0:00:06)
##################################################
```

- Shows **task counts**, **cumulative execution time**, **average**, **min/max**, and **global execution time**.
- Nested tasks are automatically tracked.

Advanced Notes

- **Enable/Disable profiling**: Pass `enabled=False` when creating the profiler to temporarily disable profiling.
- **Verbose mode**: Set `verbose=True` to log start/stop events to the profiling file.
- **Custom annotations**: Use `profiler.annotate("My note")` to add custom messages in the log.
- **File management**: The profiler appends to the specified log file; previous logs are preserved unless you manually delete the file.

Appendix

Glossary of MACsec Concepts

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.

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.

CAK - Connectivity Association Key

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

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.

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.

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.

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.

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.

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).

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.

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.

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.

Imported Packages and Modules Synthesis

`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.

`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).

`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.

`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.

`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.

`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.

`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.

`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.

`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.

`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.

`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.

`uuid`

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

`re`

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

`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.

`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.

`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.

`datetime`

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

`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.

`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.

`time`

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

Logs

Sample logs

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':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xOp0
qOiDLH9ldIb4Q=='}]}
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':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xOp0
qOiDLH9ldIb4Q=='}]}
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</ckn
>

<cak>9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f</cak
>
        </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':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xOp0
qOiDLH9ldIb4Q=='}]}
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</ckn
>

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

local_name: mx10008-24: {'keys': [{'key_ID': 'd9b6fa63-dd88-47d2-8afe-
94fcecf40d19', 'key':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xOp0
qOiDLH9ldIb4Q=='}]}
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</ckn
>
```

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

local_name: mx10008-23: {'keys': [{'key_ID': 'd9b6fa63-dd88-47d2-8afe-
94fcecf40d19', 'key':
'nK45tzFFjDA42/1Ko5B6Hz2wi35gr72VsiHyVbMD9X9qxRbHOKe8D4NpHUy1nX96JeVF4xOp0
qOiDLH9ldIb4Q=='}]}
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</ckn
>

<cak>9cae39b731458c3038dbfd4aa3907a1f3db08b7e60afbd95b221f255b303f57f</cak
>
        </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
```

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+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8Ixk
DnDP/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+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8Ixk
DnDP/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</ckn
>

<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</cak
>
                </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
=============================
```

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+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8Ixk
DnDP/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</ckn
>
```

```
<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</cak
>
                </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>
```

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+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8Ixk
DnDP/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</ckn
>

<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</cak
>
                </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>
```

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+zl1B6Wh71jkcBql9VayN1rFigJqw0R8IWiej8Ixk
DnDP/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</ckn
>

<cak>13f9f4741e876383dfa6862f24eb9a1076daf4f841d7f873fb397507a5a1ef58</cak
>
                </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>
```