Keylime Documentation Documentation

Release 7.10.0

Keylime Developers

Apr 16, 2024

CONTENTS:

Warning: This documentation is still under development and not complete. It will be so until this warning is removed.

Welcome to the Keylime Documentation site!

Keylime is a TPM-based highly scalable remote boot attestation and runtime integrity measurement solution. Keylime enables cloud users to monitor remote nodes using a hardware based cryptographic root of trust.

Keylime was originally born out of the security research team in MIT's Lincoln Laboratory and is now developed and maintained by the Keylime community.

This Documentation site contains guides to install, use and administer keylime as well as guides to enable developers to make contributions to keylime or develop services against Keylime's Rest API(s).

We recommend newcomers to read the *design section* to get an understanding what the goals of Keylime are and how they are implemented.

CHAPTER

ONE

INSTALLATION

There are three current methods for installing Keylime: the Ansible role, the Keylime installer or a manual installation.

1.1 Ansible Keylime Roles

An Ansible role to deploy Keylime , alongside the Keylime Rust agent

This role deploys Keylime for use with a Hardware TPM.

Should you wish to deploy Keylime with a software TPM emulator for development or getting your feet wet, use the Ansible Keylime Soft TPM role instead.

1.1.1 Usage

Download or clone Ansible Keylime from its repository and follow the usage section.

Run the example playbook against your target remote host(s):

```
ansible-playbook -i your_hosts playbook.yml
```

1.1.2 TPM Version Control (Software TPM)

Ansible Keylime Soft TPM provides a role type for 2.0 TPM versions.

TPM 2.0 support can be configured by simply adding the role in the `playbook.yml` file here

For TPM 2.0 use:

```
- ansible-keylime-tpm20
```

This rule uses the TPM 2.0 Emulator (IBM software TPM).

1.1.3 Rust agent

Note: The Rust agent is the official agent for Keylime and replaces the Python implementation. For the rust agent a different configuration file is used (by default `/etc/keylime/agent.conf`) which is **not** interchangeable with the old Python configuration.

Installation instructions can be found in the README.md for the Rust agent.

1.2 Keylime Bash installer

Keylime requires Python 3.6 or greater.

Installation can be performed via an automated shell script, `installer.sh`. The following command line options are available:

```
Usage: ./installer.sh [option...]
Options:
-k              Download Keylime (stub installer mode)
-m              Use modern TPM 2.0 libraries; this is the default
-s              Install & use a Software TPM emulator (development only)
-p PATH         Use PATH as Keylime path
-h              This help info
```

1.3 Docker - Deployment

The verifier, registrar and tenant can also be deployed using Docker images. Keylime's official images can be found here. Those are automatically generated for every commit and release.

For building those images locally see here.

1.4 Manual

Keylime requires Python 3.6 or greater.

1.4.1 Python-based prerequisites

The following Python packages are required:

- cryptography>=3.3.2
- tornado>=5.0.2
- pyzmq>=14.4
- pyyaml>=3.11
- requests>=2.6
- sqlalchemy>=1.3.12
- alembic>=1.1.0

- packaging>=20.0
- psutil>=5.4.2
- lark>=1.0.0
- pyasn1>=0.4.2
- pyasn1-modules>=0.2.1
- jinja2>=3.0.0
- gpg (Note: the GPG bindings must match the local GPG version and therefore this package should not be installed via PyPI)
- typing-extensions>=3.7.4 (only for Python versions < 3.8)

The current list of required packages can be found here.

All of them should be available as distro packages. See installer.sh for more information if you want to install them this way. You can also let Keylime's `setup.py` install them via PyPI.

1.4.2 TPM 2.0 Support

Keylime uses the Intel TPM2 software set to provide TPM 2.0 support. You will need to install the tpm2-tss software stack (available here) and tpm2-tools utilities available here. See README.md in these projects for detailed instructions on how to build and install.

The brief synopsis of a quick build/install (after installing dependencies) is:

```
# tpm2-tss
git clone https://github.com/tpm2-software/tpm2-tss.git tpm2-tss
pushd tpm2-tss
./bootstrap
./configure --prefix=/usr
make
sudo make install
popd
# tpm2-tools
git clone https://github.com/tpm2-software/tpm2-tools.git tpm2-tools
pushd tpm2-tools
./bootstrap
./configure --prefix=/usr/local
make
sudo make install
popd
```

To ensure that you have the recent version installed ensure that you have the `tpm2_checkquote` utility in your path.

Note: Keylime by default (all versions after 6.2.0) uses the kernel TPM resource manager. For kernel versions older than 4.12 we recommend to use the tpm2-abrmd resource manager (available here).

How the TPM is accessed by tpm2-tools can be set using the `TPM2TOOLS_TCTI` environment variable. More information about that can be found here.

Talk to the swtpm emulator directly:

```
export TPM2TOOLS_TCTI="mssim:port=2321"
```

To talk to the TPM directly (not recommended):

```
export TPM2TOOLS_TCTI="device:/dev/tpm0"
```

1.4.3 Install Keylime

You're finally ready to install Keylime:

```
sudo python setup.py install
```

1.5 Configuring basic (m)TLS setup

Keylime uses mTLS authentication between the different components. By default the verifier creates a CA for this under `/var/lib/keylime/cv_ca/` on first startup. The directory contains files for three different components:

- *Root CA*: `cacert.crt` contains the root CA certificate. **Important:** this certificate needs to be also be deployed on the agent, otherwise the tenant and verifier cannot connect to the agent!
- *Server certificate and key*: `server-cert.crt` and `server-{private,public}.pem` are used by the registrar and verifier for their HTTPS interface.
- *Client certificate and key*: `client-cert.crt` and `client-{private,public}.pem` are used by the tenant to authenticate against the verifier, registrar and agent. The verifier uses this key and certificate to authenticate against the agent.

Keylime allows each component to use their own server and client keys and also a list of trusted certificates for mTLS connections. Please refer to options the the respective configuration files for more details.

1.6 Database support

Keylime supports the following databases:

- SQLite
- PostgreSQL
- MySQL
- MariaDB

SQLite is configured as default (`database_url = sqlite`) where the databases are stored under `/var/lib/keylime`.

Starting with Keylime version 6.4.0 only supports SQLAlchemy's URL format to allow a more flexible configuration. The format for the supported databases can be found in the SQLAlchemy engine configuration documentation.

CHAPTER

TWO

USER GUIDE

2.1 User Selected PCR Monitoring

Warning: This page is still under development and not complete. It will be so until this warning is removed.

Using use the *tpm_policy* feature in Keylime, it is possible to monitor a remote machine for any given PCR.

This can be used for Trusted Boot checks for both the *rhboot* shim loader and Trusted Grub 2.

Note: On larger deployments the PCR values might be insufficient. In this case use the UEFI event log for measured boot: *Use Measured Boot*.

2.1.1 How to use

Select which PCRs you would like Keylime to measure, by using the *tpm2_pcrread* from the tpm2-tools tool.

You can add a node to using *keylime_tenant*:

```
# First create a payload to send to the agent (in our case this is empty)
touch payload

# Now actually add the node
keylime_tenant -c add \
--uuid d432fbb3-d2f1-4a97-9ef7-75bd81c00000 \
-f payload \
--tpm_policy '{"22":["0000000000000000000000000000000000000001",
↪"0000000000000000000000000000000000000000000000000000000000000001",
↪"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
↪","ffffffffffffffffffffffffffffffffffffffff",
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff",
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
↪"],"15":["0000000000000000000000000000000000000000",
↪"0000000000000000000000000000000000000000000000000000000000000000",
↪"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"]}'
```

2.1.2 rhboot shim-loader

The following is sourced from the rhboot shim repository please visit the upstream README to ensure information is still accurate

The following PCRs are extended by shim:

PCR4:

- the Authenticode hash of the binary being loaded will be extended into PCR4 before SB verification.
- the hash of any binary for which Verify is called through the shim_lock protocol

PCR7:

- Any certificate in one of our certificate databases that matches a binary we try to load will be extended into PCR7. That includes:
  - DBX - the system denylist, logged as "dbx"
  - MokListX - the Mok denylist, logged as "MokListX"
  - vendor_dbx - shim's built-in vendor denylist, logged as "dbx"
  - DB - the system allowlist, logged as "db"
  - MokList the Mok allowlist, logged as "MokList"
  - vendor_cert - shim's built-in vendor allowlist, logged as "Shim"
  - shim_cert - shim's build-time generated allowlist, logged as "Shim"
- MokSBState will be extended into PCR7 if it is set, logged as "MokSBState".

PCR8:

- If you're using the grub2 TPM patchset we cary in Fedora, the kernel command line and all grub commands (including all of grub.cfg that gets run) are measured into PCR8.

PCR9:

- If you're using the grub2 TPM patchset we cary in Fedora, the kernel, initramfs, and any multiboot modules loaded are measured into PCR9.

PCR14:

- MokList, MokListX, and MokSBState will be extended into PCR14 if they are set.

2.2 Use Measured Boot

> **Warning:** This page is still under development and not complete. It will be so until this warning is removed.

2.2.1 Introduction

In any real-world large-scale production environment, a large number of different types of nodes will typically be found. The TPM 2.0 defines a specific meaning - measurement of UEFI bios, measurement of boot device firmware - for each of the lower-numbered PCRs (e.g., PCRs 0-9), as these are extended during the multiple events of a measured boot log. However, simply comparing the contents of these PCRs against a well-known "golden value" becomes unfeasible. The reason for this is, in addition to the potentially hundreds of variations due to node type, it can be experimentally

demonstrated that some PCRs (e.g, PCR 1) vary for each physical machine, if such machine is netbooted (as it encodes the MAC address of the NIC used during boot.)

Fortunately, the UEFI firmware is now exposing the event log through an ACPI table and a "recent enough" Linux kernel (e.g., 5.4 or later) is now consuming this table and exposing this boot event log through the securityfs, typically at the path */sys/kernel/security/tpm0/binary_bios_measurements*. When combined with *secure boot* and a "recent enough" version of grub (2.06 or later), the aforementioned PCR set can be fully populated, including measurements of all components, up to the *kernel* and *initrd*.

In addition to these sources of (boot log) data, a "recent enough" version of *tpm2-tools* (5.0 or later) can be used to consume the contents of such logs and thus rebuild the contents of PCRs [0-9] (and potentially PCRs [11-14]).

2.2.2 Implementation

Keylime can make use of this new capability in a very flexible manner. A "measured boot reference state" or *mb_refstate* for short can be specified by the *keylime* operator (i.e. the *tenant*). This operator-provided piece of information is used, in a fashion similar to the "IMA policy" (previously known as "allowlist"), by the *keylime_verifier*, to compare the contents of the information shipped from the *keylime_agent* (boot log in one case, IMA log on the other), against such reference state.

Due to the fact that physical node-specific information can be encoded on the "measured boot log", it became necessary to specify (optionally) a second piece of information, a "measured boot policy" or *mb_policy* . This information is used to instruct the *keylime_verifier* on how to do the comparison (e.g., using a regular expression, rather than a simple equality match). The policy name is specified in *keylime.conf*, under the *[cloud_verifier]* section of the file, with parameter named *measured_boot_policy_name*. The default value for it is *accept-all*, meaning "just don't try to match the contents, just replay the log and make sure the values of PCRs [0-9] and [11-14] match".

Whenever a "measured boot reference state" is defined - via a new command-line option in *keylime_tenant* - *–mb_refstate*, the following actions will be taken.

1) PCRs [0-9] and [11-14] will be included in the quote sent by *keylime_agent*

2) The *keylime_agent* will also send the contents of */sys/kernel/security/tpm0/binary_bios_measurements*

3) The *keylime_verifier* will replay the boot log from step 2, ensuring the correct values for PCRs collected in step 1. Again, this is very similar to what it is done with "IMA logs" and PCR 10.

4) The very same *keylime_verifier* will take the boot log, now deemed "attested" and compare it against the "measured boot reference state", according to the "measured boot policy", causing the attestation to fail if it does not conform.

2.2.3 How to use

The simplest way to use this new functionality is by providing an empty "measured boot reference state" and an *accept-all* "measured boot policy", which will cause the *keylime_verifier* to simply skip the aforementioned step 4.

An example follows:

```
echo "{}" > measured_boot_reference_state.txt

keylime_tenant -c add -t <AGENT IP> -v <VERIFIER IP> -u <AGENT UUID> --mb_refstate ./
↪measured_boot_reference_state.txt
```

Note: please keep in mind that the IMA-specific options can be combined with the above options in the example, resulting in a configuration where a *keylime_agent* sent a quote with PCRs [0-15] and both logs (boot and IMA)

Evidently, to be fully used in a meaningful manner, keylime operators need to provide its own custom *mb_refstate* and *mb_policy*. While an user can write a policy that performs an "exact match" on a carefully constructed refstate, the key

idea here is to create a pair of specification files which are at once meaningful (for the purposes of trusted computing attestation) and generic (enough to be applied to a set of nodes).

The most convenient way to crate an *mb_refstate* is starting from the contents of an UEFI boot log from a given node, and then tweak and customize it to make more generic. Keylime includes a tool (under *scripts* directory) - *generate_mb_refstate* - which will consume a boot log and output a JSON file containing an *mb_refstate*. An example follows:

```
keylime/scripts/create_mb_refstate /sys/kernel/security/tpm0/binary_bios_measurements␣
↪measured_boot_reference_state.json

keylime_tenant -c add -t <AGENT IP> -v <VERIFIER IP> -u <AGENT UUID> --mb_refstate ./
↪measured_boot_reference_state.json
```

This reference state can be (as in the example above) consumed "as is", or it can be tweaked to be made more generic (or even more strict, if the keylime operator chooses so).

The *mb_policy* is defined within a framework specified in *policies.py*, where some "trivial" policies such as *accept-all* and *reject-all* are pre-defined. The Domain-Specific Language (DSL) used by the framework are defined in *tests.py* and an illustrative use of it can be seen in the policy *example.py*, all under the *elchecking* directory. This example policy was crafted to be meaningful (i.e., with a relevant number of parameters tests) and yet applicable to a large set of nodes. It consumes a *mb_refstate* such as the one generated by the aforementioned tool or the *example_reference_state.json*, located under the same directory.

Just to quickly exemplify what this policy does, it for instance tests if a node has *SecureBoot* enabled (*tests.FieldTest("Enabled", tests.StringEqual("Yes"))*) and if a node has a well-formed kernel command line boot parameters (e.g., *tests.FieldTest("String", tests.RegExp(r".*/grub.*"))*). The policy is well documented, and operators are encouraged to just read through the comments in order to understand how the tests are implemented.

While an operator can attempt to write its own policy from scratch, it is recommended that one just copies *example.py* into *mypolicy.py*, change it as required and then just points to this new policy on *keylime.conf* (*measured_boot_policy_name*) for its own use.

2.3 Runtime Integrity Monitoring

Keylime's runtime integrity monitoring requires the set up of Linux IMA. More information about IMA in general can be found in the openSUSE Wiki.

You should refer to your Linux Distributions documentation to enable IMA, but as a general guide most recent versions already have `CONFIG_IMA` toggled to `Y` as a value during Kernel compile.

It is then just a case of deploying an `ima-policy` file. On a Fedora or Debian system, the file is located in `/etc/ima/ima-policy`.

For configuration of your IMA policy, please refer to the IMA Documentation.

Within Keylime we use the following for demonstration (found in `demo/ima-policies/ima-policy-keylime`):

```
# PROC_SUPER_MAGIC
dont_measure fsmagic=0x9fa0
# SYSFS_MAGIC
dont_measure fsmagic=0x62656572
# DEBUGFS_MAGIC
dont_measure fsmagic=0x64626720
# TMPFS_MAGIC
dont_measure fsmagic=0x01021994
```

(continues on next page)

(continued from previous page)

```
# RAMFS_MAGIC
dont_measure fsmagic=0x858458f6
# SECURITYFS_MAGIC
dont_measure fsmagic=0x73636673
# SELINUX_MAGIC
dont_measure fsmagic=0xf97cff8c
# CGROUP_SUPER_MAGIC
dont_measure fsmagic=0x27e0eb
# OVERLAYFS_MAGIC
# when containers are used we almost always want to ignore them
dont_measure fsmagic=0x794c7630
# Don't measure log, audit or tmp files
dont_measure obj_type=var_log_t
dont_measure obj_type=auditd_log_t
dont_measure obj_type=tmp_t
# MEASUREMENTS
measure func=BPRM_CHECK
measure func=FILE_MMAP mask=MAY_EXEC
measure func=MODULE_CHECK uid=0
```

This default policy measures all executables in `bprm_check` and all files `mmapped` executable in `file_mmap` and module checks and skips several irrelevant files (logs, audit, tmp, etc).

Once your `ima-policy` is in place, reboot your machine (or even better have it present in your image for first boot).

You can then verify IMA is measuring your system:

```
# head -5 /sys/kernel/security/ima/ascii_runtime_measurements
PCR                                template-hash filedata-hash                               ␣
↪         filename-hint
10 3c93cea361cd6892bc8b9e3458e22ce60ef2e632 ima-ng␣
↪sha1:ac7dd11bf0e3bec9a7eb2c01e495072962fb9dfa boot_aggregate
10 3d1452eb1fcbe51ad137f3fc21d3cf4a7c2e625b ima-ng␣
↪sha1:a212d835ca43d7deedd4ee806898e77eab53dafa /usr/lib/systemd/systemd
10 e213099a2bf6d88333446c5da617e327696f9eb4 ima-ng␣
↪sha1:6da34b1b7d2ca0d5ca19e68119c262556a15171d /usr/lib64/ld-2.28.so
10 7efd8e2a3da367f2de74b26b84f20b37c692b9f9 ima-ng␣
↪sha1:af78ea0b455f654e9237e2086971f367b6bebc5f /usr/lib/systemd/libsystemd-shared-239.so
10 784fbf69b54c99d4ae82c0be5fca365a8272414e ima-ng␣
↪sha1:b0c601bf82d32ff9afa34bccbb7e8f052c48d64e /etc/ld.so.cache
```

2.3.1 Keylime Runtime Policies

A runtime policy in its most basic form is a set of "golden" cryptographic hashes of files' un-tampered state or of keys that may be loaded onto keyrings for IMA verification.

Keylime will load the runtime policy into the Keylime Verifier. Keylime will then poll tpm quotes to *PCR 10* on the agents TPM and validate the agents file(s) state against the policy. If the object has been tampered with or an unexpected key was loaded onto a keyring, the hashes will not match and Keylime will place the agent into a failed state. Likewise, if any files invoke the actions stated in `ima-policy` that are not matched in the allowlist, keylime will place the agent into a failed state.

Allowlists are contained in Keylime runtime policies - see below for more details.

Generate a Runtime Policy

Runtime policies heavily depend on the IMA configuration and used files by the operating system. Keylime provides two helper scripts for getting started.

Note: Those scripts only provide a reference point to get started and **not** a complete solution. We encourage developers / users of Keylime to be creative and come up with their own process for securely creating and maintaining runtime policies.

Create Runtime Policy from a Running System

The first script generates a runtime policy from the `initramfs`, IMA log (just for the `boot aggregate`) and files located on the root filesystem of a running system.

The `create_runtime_policy.sh` script is available here

Run the script as follows:

```
# create_runtime_policy.sh -o runtime_policy_keylime.json
```

For more options see the help page `create_runtime_policy.sh`:

```
Usage: $0 -o/--output_file FILENAME [-a/--algo ALGO] [-x/--ramdisk-location PATH] [-y/--
↪boot_aggregate-location PATH] [-z/--rootfs-location PATH] [-e/--exclude_list FILENAME]␣
↪[-s/--skip-path PATH]"

optional arguments:
-a/--algo                    (checksum algorithmi to be used, default: sha1sum)
-x/--ramdisk-location        (path to initramdisk, default: /boot/, set to "none" to␣
↪skip)
-y/--boot_aggregate-location (path for IMA log, used for boot aggregate extraction,␣
↪default: /sys/kernel/security/ima/ascii_runtime_measurements, set to "none" to skip)
-z/--rootfs-location         (path to root filesystem, default: /, cannot be skipped)
-e/--exclude_list            (filename containing a list of paths to be excluded (i.e.,␣
↪verifier will not try to match checksums), default: none)
-s/--skip-path               (comma-separated path list, files found there will not have␣
↪checksums calculated, default: none)
-h/--help                    show this message and exit
```

Note: note, you need the OpenSSL installed to have the sha*sum CLI executables available

The resulting *runtime_policy_keylime.json* file can be directly used by `keylime_tenant` (option `--runtime-policy`)

Warning: It's best practice to create the runtime policy in a secure environment. Ideally, this should be on a fully encrypted, air gapped computer that is permanently isolated from the Internet. Disable all network cards and sign the runtime policy hash to ensure no tampering occurs when transferring to other machines.

Creating more Complex Policies

The second script allows the user to build more complex policies by providing options to include: keyring verification, IMA verification keys, generating allowlist from IMA measurement log and extending existing policies.

A basic policy can be easily created by using a IMA measurement log from system:

```
keylime_create_policy -m /path/to/ascii_runtime_measurements -o runtime_policy.json
```

For more options see the help page `keylime_create_policy -h`:

```
usage: keylime_create_policy [-h] [-B BASE_POLICY] [-k] [-b] [-a ALLOWLIST] [-m IMA_
↪MEASUREMENT_LIST] [-i IGNORED_KEYRINGS] [-o OUTPUT] [--no-hashes] [-A IMA_SIGNATURE_
↪KEYS]

This is an experimental tool for adding items to a Keylime's IMA runtime policy

options:
  -h, --help            show this help message and exit
  -B BASE_POLICY, --base-policy BASE_POLICY
                        Merge new data into the given JSON runtime policy
  -k, --keyrings        Create keyrings policy entries
  -b, --ima-buf         Process ima-buf entries other than those related to keyrings
  -a ALLOWLIST, --allowlist ALLOWLIST
                        Use given plain-text allowlist
  -m IMA_MEASUREMENT_LIST, --ima-measurement-list IMA_MEASUREMENT_LIST
                        Use given IMA measurement list for keyrings and critical data␣
↪extraction rather than /sys/kernel/security/ima/ascii_runtime_measurements
  -i IGNORED_KEYRINGS, --ignored-keyrings IGNORED_KEYRINGS
                        Ignored the given keyring; this option may be passed multiple␣
↪times
  -o OUTPUT, --output OUTPUT
                        File to write JSON policy into; default is to print to stdout
  --no-hashes           Do not add any hashes to the policy
  -A IMA_SIGNATURE_KEYS, --add-ima-signature-verification-key IMA_SIGNATURE_KEYS
                        Add the given IMA signature verification key to the Keylime-
↪internal 'tenant_keyring'; the key should be an x509 certificate in DER or PEM format␣
↪but may also be a public or private key
                        file; this option may be passed multiple times
```

Runtime Policy Entries for Keys

IMA can measure which keys are loaded onto different keyrings. Keylime has the option to verify those keys and automatically use them for signature verification.

The hash of the an key can be generated for example with:

```
sha256sum /etc/keys/ima/rsakey-rsa.crt.der
```

As seen the the JSON schema below, the hash (sha1 or sha256) depending on the IMA configuration can be added as the following where in `.ima` is the keyring the key gets loaded onto and <SHA256_HASH> is the hash of that key:

```
jq '.keyrings += {".ima" : ["<SHA256_HASH>"]}'  runtime_policy.json  > runtime_policy_
↪with_keyring.json
```

The following rule should be added to the IMA policy so that IMA reports keys loaded onto keyrings .ima and .evm (since Linux 5.6):

```
measure func=KEY_CHECK keyrings=.ima|.evm
```

If the key should only be verified and not be used for IMA signature verification, then it can be added to the ignore list:

```
jq '.ima.ignored_keyrings += [".ima"]' runtime_policy.json > runtime_policy_ignore_ima.
↪json
```

If * is added no verified keyring is used for IMA signature verification.

Runtime Policy JSON Schema

The tenant parses the allow and exclude list into a JSON object that is then sent to the verifier. Depending of the use case the object can also be constructed manually instead of using the tenant.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "title": "Keylime IMA policy",
    "type": "object",
    "properties": {
        "meta": {
            "type": "object",
            "properties": {
                "version": {
                    "type": "integer",
                    "description": "Version number of the IMA policy schema"
                }
            },
            "required": ["version"],
            "additionalProperties": false
        },
        "release": {
            "type": "number",
            "title": "Release version",
            "description": "Version of the IMA policy (arbitrarily chosen by the user)"
        },
        "digests": {
            "type": "object",
            "title": "File paths and their digests",
            "patternProperties": {
                ".*": {
                    "type": "array",
                    "title": "Path of a valid file",
                    "items": {
                        "type": "string",
                        "title": "Hash of an valid file"
                    }
                }
            }
        },
        "excludes": {
```

(continues on next page)

(continued from previous page)

```
            "type": "array",
            "title": "Excluded file paths",
            "items": {
                "type": "string",
                "format": "regex"
            }
        },
        "keyrings": {
            "type": "object",
            "patternProperties": {
                ".*": {
                    "type": "string",
                    "title": "Hash of the content in the keyring"
                }
            }
        },
        "ima-buf": {
            "type": "object",
            "title": "Validation of ima-buf entries",
            "patternProperties": {
                ".*": {
                    "type": "string",
                    "title": "Hash of the ima-buf entry"
                }
            }
        },
        "verification-keys": {
            "type": "array",
            "title": "Public keys to verify IMA attached signatures",
            "items": {
                "type": "string"
            }
        },
        "ima": {
            "type": "object",
            "title": "IMA validation configuration",
            "properties": {
                "ignored_keyrings": {
                    "type": "array",
                    "title": "Ignored keyrings for key learning",
                    "description": "The IMA validation can learn the used keyrings␣
↪embedded in the kernel. Use '*' to never learn any key from the IMA keyring␣
↪measurements",
                    "items": {
                        "type": "string",
                        "title": "Keyring name"
                    }
                },
                "log_hash_alg": {
                    "type": "string",
                    "title": "IMA entry running hash algorithm",
                    "description": "The hash algorithm used for the running hash in IMA␣
```

(continues on next page)

(continued from previous page)

```
↪entries (second value). The kernel currently hardcodes it to sha1.",
                    "const": "sha1"
                }
            },
            "required": ["ignored_keyrings", "log_hash_alg"],
            "additionalProperties": false
        }
    },
    "required": ["meta", "release", "digests", "excludes", "keyrings", "ima", "ima-buf",
↪"verification-keys"],
    "additionalProperties": false
}
```

2.3.2 Remotely Provision Agents

Now that we have our runtime policy available, we can send it to the verifier.

Note: If you're using a TPM Emulator (for example with the ansible-keylime-tpm-emulator, you will also need to run the keylime ima emulator. To do this, open a terminal and run `keylime_ima_emulator`

Using the `keylime_tenant` we can send the runtime policy as follows:

```
touch payload  # create empty payload for example purposes
keylime_tenant -c add --uuid <agent-uuid> -f payload --runtime-policy /path/to/policy.
↪json
```

Note: If your agent is already registered, you can use `-c update`

2.3.3 How can I test this?

Create a script that does anything (for example `echo "hello world"`) that is not present in your runtime policy. Run the script as root on the agent machine. You will then see the following output on the verifier showing the agent status change to failed:

```
keylime.tpm - INFO - Checking IMA measurement list...
keylime.ima - WARNING - File not found in allowlist: /root/evil_script.sh
keylime.ima - ERROR - IMA ERRORS: template-hash 0 fnf 1 hash 0 good 781
keylime.cloudverifier - WARNING - agent D432FBB3-D2F1-4A97-9EF7-75BD81C00000 failed,␣
↪stopping polling
```

2.3.4 IMA File Signature Verification

Keylime supports the verification of IMA file signatures, which also helps to detect modifications on immutable files and can be used to complement or even replace the allowlist of hashes in the runtime policy if all relevant executables and libraries are signed. However, the set up of a system that has *all* files signed is beyond the scope of this documentation.

In the following we will show how files can be signed and how a system with signed files must be registered. We assume that the system has already been set up for runtime-integrity monitoring following the above steps and that the system would not show any errors on the Keylime Verifier side. It should not be registered with the keylime verifier at this point. If it is, we now deregister it:

```
keylime_tenant -c delete -u <agent-uuid>
```

Our first step is to enable IMA Appraisal in Linux. Recent Fedora kernels for example have IMA Appraisal support built-in but not activated. To enable it, we need to add the following Linux kernel parameters to the Linux boot command line:

```
ima_appraise=fix ima_template=ima-sig ima_policy=tcb
```

For this we edit */etc/default/grub* and append the above parameters to the *GRUB_CMDLINE_LINUX* line and then recreate the system's grub configuration file with the following command:

```
sudo grub2-mkconfig -o /boot/grub2/grub.cfg
```

IMA will be in IMA Appraisal fix-mode when the system is started up the next time. Fix-mode, unlike enforcement mode, does not require that all files be signed but will give us the benefit that the verifier receives all file signatures of signed executables.

For IMA Appraisal to append the file signatures to the IMA log, we need to append the following line to the above IMA policy:

```
appraise func=BPRM_CHECK fowner=0 appraise_type=imasig
```

We now create our IMA file signing key using the following commands:

```
openssl genrsa -out ima-filesigning.pem 2048
openssl rsa -in ima-filesigning.pem -pubout -out ima-pub.pem
```

Next, we determine the hash (sha1 or sha256) that IMA is using for file measurements by looking at the IMA measurement log and then use evmctl to sign a demo executable that we derive from the echo tool:

```
sudo dnf -y install ima-evm-utils
cp /bin/echo ./myecho
sudo evmctl ima_sign --key ima-filesigning.pem -a <hash> myecho
```

Note: It is important that we use the same hash for signing the file that IMA also uses for file measurements. In the case we use 'sha1' since the IMA measurement log further above shows sha1 filedata-hashes in the 4th column. On more recent systems we would likely use 'sha256'.

Note: If the IMA measurement log contains invalid signatures, the system will have to be rebooted to start over with a clean log that the Keylime Verifier can successfully verify.

Invalid signatures may for example be in the log if executables were accidentally signed with the wrong hash, such as sha1 instead of sha256. In this case they all need to be re-signed to match the hash that IMA is using for file signatures.

Another reason for an invalid signature may be that a file was modified after it was signed. Any file modification will invalidate the signature. Similarly, a malformatted or altered *security.ima* extended attribute will lead to a signature verification failure.

Yet another reason may be that an unknown key was used for signing files. In this case the system should be re-registered with that additional key using the Keylime tenant tool.

To verify that the file has been properly signed, we can use the following command, which will show the security.ima extended attribute's value:

```
getfattr -m ^security.ima --dump myecho
```

We now reboot the machine:

```
reboot
```

After the reboot the IMA measurement log should not have any measurement of the *myecho* tool. The following command should not return anything:

```
grep myecho /sys/kernel/security/ima/ascii_runtime_measurements
```

We now create a new policy that includes the signing key using the `keylime_create_policy` tool:

```
keylime_create_policy -B /path/to/runtime_policy.json -A /path/to/ima-pub.pem  -o /
↪output/path/runtime_policy_with_key.json
```

After that we register the agent with the new policy:

```
keylime_tenant -c add --uuid <agent-uuid> -f payload --runtime-policy /path/to/runtime_
↪policy_with_key.json
```

We can now execute the `myecho` tool as root:

```
sudo ./myecho
```

At this point we should not see any errors on the verifier side and there should be one entry of 'myecho' in the IMA measurement log that contains a column after the file path containing the file signature:

```
grep myecho /sys/kernel/security/ima/ascii_runtime_measurements
```

To test that signature verification works, we can now invalidate the signature by *appending* a byte to the file and executing it again:

```
echo >> ./myecho
sudo ./myecho
```

We should now see two entries in the IMA measurement log. Each one should have a different measurement:

```
grep myecho /sys/kernel/security/ima/ascii_runtime_measurements
```

The verifier log should now indicating a bad file signature:

```
keylime.tpm - INFO - Checking IMA measurement list on agent: D432FBB3-D2F1-4A97-9EF7-
↪75BD81C00000
keylime.ima - WARNING - signature for file /home/test/myecho is not valid
```

(continues on next page)

(continued from previous page)

```
keylime.ima - ERROR - IMA ERRORS: template-hash 0 fnf 0 hash 0 bad-sig 1 good 3042
keylime.cloudverifier - WARNING - agent D432FBB3-D2F1-4A97-9EF7-75BD81C00000 failed,␣
↪stopping polling
```

2.3.5 Using Key Learning to Verify Files

Note: The following has been tested with RHEL 9.3 and keylime 7.3. It is work-in-progress on CentOS and Fedora.

Using key learning to verify files requires that files logged by IMA are appropriately signed. If files are not signed or have a bad signature then they must be either in the exclude list of the runtime policy or their hashes must be part of the runtime policy. It should also be noted that IMA signature verification provides lock-down of a system and ensures the provenance of files from a trusted source but, unlike file hashes, does not provide protection for file renaming or replacing files and signatures with other versions (downgrading).

For the following setup we use RHEL 9.3 since this distribution carries file signatures in its rpm packages and the Dracut scripts have been added to load the IMA signature verification keys onto the `.ima` keyring.

All below steps are run as *root*.

To ensure that file signatures are installed when packages are installed, run the following command:

```
dnf -y install rpm-plugin-ima
```

Since some packages did not carry file signatures until recently, update all packages to ensure that the signatures are installed:

```
dnf -y update
```

In case the system was previously not installed with file signatures, run the following command to reinstall all packages with file signatures:

```
dnf -y reinstall \*
```

To verify whether a particular file has its file signature installed use the following command to display the contents of `security.ima`. If nothing is displayed then this file misses its file signature:

```
getfattr -m ^security.ima -e hex --dump /usr/bin/bash
```

We must setup the system with the kernel command line option `ima_template=ima-sig` so that IMA signatures become part of the measurement log. It is not necessary to enable signature enforcement on the system, measuring executed applications is sufficient for the purpose of 'key learning'. For this we edit `/etc/default/grub` and adjust the following line:

```
GRUB_CMDLINE_LINUX="rhgb quiet ima_template=ima-sig"
```

Then run the following command to update the kernel command line options:

```
grub2-mkconfig -o /boot/grub2/grub.conf   # grub.cfg on CentOS/RHEL
```

Set the following IMA policy in `/etc/ima/ima-policy` when systemd will load the policy:

```
# PROC_SUPER_MAGIC
dont_measure fsmagic=0x9fa0
# SYSFS_MAGIC
dont_measure fsmagic=0x62656572
# DEBUGFS_MAGIC
dont_measure fsmagic=0x64626720
# TMPFS_MAGIC
dont_measure fsmagic=0x01021994
# RAMFS_MAGIC
dont_measure fsmagic=0x858458f6
# SECURITYFS_MAGIC
dont_measure fsmagic=0x73636673
# SELINUX_MAGIC
dont_measure fsmagic=0xf97cff8c
# CGROUP_SUPER_MAGIC
dont_measure fsmagic=0x27e0eb
# OVERLAYFS_MAGIC
# when containers are used we almost always want to ignore them
dont_measure fsmagic=0x794c7630

# Measure and log keys loaded onto the .ima keyring
measure func=KEY_CHECK keyrings=.ima
# Measure and log executables
measure func=BPRM_CHECK
# Measure and log shared libraries
measure func=FILE_MMAP mask=MAY_EXEC
```

Copy IMA signature verification key(s) so that Dracut scripts can load the keys onto the `.ima` keyring early during system startup:

```
mkdir -p /etc/keys/ima
cp /usr/share/doc/kernel-keys/$(uname -r)/ima.cer /etc/keys/ima # RHEL/CentOS
```

Enable the IMA Dracut scripts in the initramfs:

```
dracut --kver $(uname -r) --force --add integrity
```

Then reboot the system:

```
reboot
```

Once the system has been rebooted it must show at least two entries in the IMA log where keys were loaded onto the .ima keyring:

grep -E " .ima " /sys/kernel/security/ima/ascii_runtime_measurements

The first entry represents the Linux kernel signing key and the second entry is the IMA file signing key.

We now create the policy:

```
grep \
  -E "(boot_aggregate| ima-buf )" \
  /sys/kernel/security/ima/ascii_runtime_measurements > trimmed_ima_log

keylime_create_policy -k -m ./trimmed_ima_log -o mypolicy.json
```

The 1st command creates a trimmed-down IMA measurement log that only contains the boot_aggregate and ima-buf entries. The latter show the key(s) that were loaded onto the `.ima` keyring.

The 2nd command creates the runtime policy that holds the boot_aggregate entry and a hash over keys that were loaded onto the .ima keyring. This hash is used to verify that only trusted keys are learned.

We can now start to monitor this system:

```
touch payload  # create empty payload for example purposes
keylime_tenant -c update --uuid <agent-uuid> -f payload --runtime-policy ./mypolicy.json
```

In case the verification of the system fails we need to inspect the verifier log and add those files to the `trimmed_ima_log` that failed verification. Assuming files with the filename pattern `livesys` failed verification we repeat the steps above as follows by adding files with the file pattern `livesys` to the trimmed log. These files will then be verified using their hashes rather than signatures. Another possibility would be to add these files to the list of excluded files. We may need to repeat the following steps until the system passes verification:

```
grep \
  -E "(boot_aggregate| ima-buf |livesys)" \
  /sys/kernel/security/ima/ascii_runtime_measurements > trimmed_ima_log

keylime_create_policy -k -m ./trimmed_ima_log -o mypolicy.json

keylime_tenant -c update --uuid <agent-uuid> -f payload --runtime-policy ./mypolicy.json
```

To trigger a verification failure an unsigned application can be started:

```
cat <<EOF > test.sh
#!/usr/bin/env bash
echo Test
EOF

chmod 0755 test.sh

./test.sh
```

To re-enable the verification of the system the policy needs to be updated to contain `test.sh` and possibly all other applications that are not signed:

grep
: -E "(boot_aggregate| ima-buf |test.sh)" /sys/kernel/security/ima/ascii_runtime_measurements > trimmed_ima_log

keylime_create_policy -k -m ./trimmed_ima_log -o mypolicy.json

keylime_tenant -c update –uuid <agent-uuid> -f payload –runtime-policy ./mypolicy.json

2.3.6 Legacy allowlist and excludelist Format

Since Keylime 6.6.0 the old JSON and flat file formats for runtime policies are deprecated. Keylime provides with `keylime_convert_runtime_policy` a utility to convert those into the new format.

2.4 Secure Payloads

Warning: This page is still under development and not complete. It will be so until this warning is removed.

Secure payloads offer the ability to provision encrypted data to an enrolled node. This encrypted data can be used to deliver secrets needed by the node such as keys, passwords, certificate roots of trust, etc.

Secure payloads are for anything which requires strong confidentiality and integrity to bootstrap your system.

The payload itself is encrypted and sent via the Keylime Tenant CLI (or rest API) to the Keylime Agent. The Agent also sends part of the key needed to decrypt the payload, a key share, called the *u_key* or user key. Only when the Agent has passed its enrolment criteria (including any *tpm_policy* or IMA allowlist), will the other key share of the decryption key, called the *v_key* or verification key, be passed to the Agent by the Keylime Verifier to decrypt the payload.

Note: An alternative to secure payloads is to deliver the encrypted data to the node through some other mechanism like *cloud-init* or pre-embedded in a disk image. The Keylime protocol described above will still run to derive the decryption key for this data, but the data itself will never been seen or transported by Keylime. This guide does not discuss this method.

Keylime offers two modes for sending secure payloads: single file encryption and certificate package mode. In the following sections we describe each. If you're interested in using the more advanced certificate package mode, we recommend you also read the Single File Encryption section as it contains configuration options and other information that both modes share.

2.4.1 Single File Encryption

In this mode, a file you specify to the *keylime_tenant* application with the *-f* option will be encrypted by the Tenant using the bootstrap key and securely delivered to the Agent. Once the Keylime protocol with the Tenant and Verifier has completed, the Keylime Agent will decrypt this file and place it in */var/lib/keylime/secure/decrypted_payload* This is the default file name, but you can adjust the name of this file using the *dec_payload_file* option in *keylime.conf*. You can also optionally specify a zip file as the file to be securely delivered. If the *extract_payload_zip* option in *keylime.conf* is set (which it is by default), then Keylime will automatically extract the zip file to */var/lib/keylime/secure/unzipped*. Finally, Keylime can also execute a script contained in the zip file once it has been unzipped. You can think of this as a very simple form of cloud-init. By default this script is called *autorun.sh*. You can override this default with a different script name by adjusting the *payload_script* option in *keylime.conf*. Note also that this script must be contained in the encrypted zip file, from which it will be extraced and then placed in */var/lib/keylime/secure/unzipped*.

Because the keys that Keylime uses to decrypt the data and the decrypted data itself are very sensitive, Keylime will only write those files to the memory-backed (and therefore non-persistent) */var/lib/keylime/secure* directory. This is a bind-mounted tmpfs partition. As such, depending on how large your payload is, you may need to increase the size of this mounted partition by adjusting the *secure_size* option in *keylime.conf*.

This simple mode of operation is suitable for many situations where the secrets and other bootstrapping information are basic. However, there are several features that Keylime supports like revocation and certificate management that do not work in this mode. For those, you'll need the next mode: Certificate Package Mode.

2.4.2 Certificate Package Mode

This mode of Keylime automates many common actions that tenants will want to take when provisioning their Agents. First, Keylime can create an X509 certificate authority (CA) using *keylime_ca -d listen* and then issue certificates and the corresponding private keys to each provisioned node. This CA lives on the same host where the tenant issues the *keylime_ca* command and can be used to bootstrap many other security solutions like mutual TLS or SSH. To use this mode, pass the *–cert* option and a directory where the CA is located as the parameter to this option. Keylime will then create a certificate for the node (with the common name set to the Agent's UUID) and then create a zip file containing the newly generated X509 certificates, trust roots, and private keys. It then uses the same process for single file encryption as described above to securely deliver all the keys to the Agent. Optionally, the user can specify with the *–include* option a directory of additional files to be put into the certification package zip and securely delivered to the Agent.

This mode of operation also natively supports certificate revocation. If the Keylime Verifier detects an Agent that no longer satisfies its integrity policy (e.g., it booted an unauthorized kernel or ran an unauthorized binary not on the IMA allowlist), it will create a signed revocation notification. These revocation notifications are signed by a special certificate/private key called the RevocationNotifier. Keylime will automatically create this certificate and pass it to the verifier when you add a new Agent to the verifier. Keylime will also include the public certificate for this key in the zip it sends to the Agent. This way Agents can validate the revocation notifications they receive from the verifier.

By default all Keylime Agents listen for these revocation notifications (see the *listen_notifications* option in *keylime.conf*). Using the keys in the unzipped certificate package, Agents can check that the revocations are valid. Keylime Agents can also take actions in response to a valid revocation. You can configure these actions by putting additional files into the delivered zip file using *–include*.

Revocation actions are small Python scripts that will run on an Agent when a valid revocation is received. They should contain an *execute* function that takes one argument. This argument is a Python dictionary of metadata that can be used to tailor what the revocation action does. In the cert package mode, Keylime will specify the certificate serial number and common name (aka UUID) of the node that has failed its integrity check inside this metadata passed to the revocation action. For example, you can use this info to revoke the the offending X509 certificate.

One subtlety to revocation actions is that they are not intended for the Agent that has been revoked. If an Agent has failed its integrity check, then we really can't trust that it won't ignore the revocations and do arbitrarily malicious things. So, revocation actions are for other well-behaving Agents in the system to take action against the revoked Agent. For example, by revoking its certificate as described above or firewalling it from the network, etc.

There are some conventions to specifying revocation actions. As described above, their names must start with *local_action* to be executed. They also must be listed (without *.py* extensions) in a comma separated list in a file called *action_list* in the zip file. For example to run *local_action_a.py* and *local_action_b.py* the *action_list* file should contain *local_action_a,local_action_b*.

So far we've described all the details of this in fine detail, but much of this automation will happen by default.

2.4.3 Certificate Package Example

Let's put all of the above together with an example.

For the following example, we will provision some SSH keys onto the Agent.

1. Create a directory to host the files and *autorun.sh* script. For this example, we will use the directory *payload*
2. Create an *autorun.sh* script in the *payload* directory:

```
#!/bin/bash

# this will make it easier for us to find our own cert
ln -s `ls *-cert.crt | grep -v Revocation` mycert.crt
```

(continues on next page)

(continued from previous page)

```
mkdir -p /root/.ssh/
cp payload_id_rsa* /root/.ssh/
chmod 600 /root/.ssh/payload_id_rsa*
```

3. Copy the files you wish to provision into the *payload* directory.

```
$ ls payload/
autorun.sh
payload_id_rsa.pub
payload_id_rsa
```

Send the files using the Keylime Tenant tool:

```
keylime_tenant -t <agent-ip> --cert myca --include payload
```

Recall that the *–cert* option tells Keylime to create a certificate authority at the default location */var/lib/keylime/ca* and give this machine an X509 identity with its UUID. Keylime will also create a revocation notifier certificate for this CA and make it available to the verifier. Finally, the *–include* option tells Keylime to securely deliver the files in the specified directory (*payload* in our case) along with the X509 certs to the targeted Agent machine.

If the enrolment was been successful, you will be able to see the contents of the *payload* directory in */var/lib/keylime/secure/unzipped* along with the certs and included files. You should also see the SSH keys we included made in */root/.ssh* directory from where the autorun.sh script was ran.

Now, let's extend this example with revocation. In this example, we're going to execute a simple revocation action on the node that was revoked.

It is also possible to configure scripts for execution should a node fail any given criteria (IMA measurements, for example).

To configure this, we will use our *payload* directory again.

First create a Python script with the preface of *local_action*

For example *local_action_rm_ssh.py*

Within this script create an *execute* function:

```
import os
import json
import keylime.ca_util as ca_util
import keylime.secure_mount as secure_mount

async def execute(event):
    if event['type'] != 'revocation':
        return

    json_meta = json.loads(event['meta_data'])
    serial = json_meta['cert_serial']

    # load up my own cert
    secdir = secure_mount.mount()
    mycert = ca_util.load_cert_by_path(f'{secdir}/unzipped/mycert.crt')

    # is this revocation meant for me?
```

(continues on next page)

(continued from previous page)

```
    if serial == mycert.serial_number:
        os.remove("/root/.ssh/payload_id_rsa")
        os.remove("/root/.ssh/payload_id_rsa.pub")
```

Next, in the *payload* directory create the *action_list* file containing *local_action_rm_ssh* (remember not to put the *.py* extension).

Warning: In the above example, the node that fails its integrity check is the same one that we're expecting to run the revocation action to delete the key. Since the node is potentially compromised, we really can't expect that it will actually do this and not just ignore the revocation. A more realistic scenario for SSH keys is to provision one node with the SSH key generated as above, then provision a second server and add *payload_id_rsa.pub* to *.ssh/authorized_keys* using an autorun script. At this point, you can SSH from the first server to the second one. Should the first machine fail its integrity, then an revocation action on the second server can remove the compromised first machine from its list of Secure machines in *.ssh/authorized_keys*

Many actions can be executed based on CA revocation. For more details and examples, please refer to the *Agent Revocation* page.

2.5 Agent Revocation

Warning: This page is still under development and not complete. It will be so until this warning is removed.

2.6 Configuration

Keylime is configured by files installed by default in `/etc/keylime`. The files are loaded following a hierarchical order in which the values set for the options can be overridden by the next level.

For each component, a base configuration file (e.g. `/etc/keylime/verifier.conf`) is loaded, setting the base values. Then, the configuration snippets placed in the respective directory (e.g. `/etc/keylime/verifier.conf.d`) are loaded, overriding the previously set values. Finally, options can be overridden via environment variables.

The following components can be configured:

Table 1: Components and configuration

Component	Default base config file	Default snippets directory
agent	`/etc/keylime/agent.conf`	`/etc/keylime/agent.conf.d`
verifier	`/etc/keylime/verifier.conf`	`/etc/keylime/verifier.conf.d`
registrar	`/etc/keylime/registrar.conf`	`/etc/keylime/registrar.conf.d`
tenant	`/etc/keylime/tenant.conf`	`/etc/keylime/tenant.conf.d`
ca	`/etc/keylime/ca.conf`	`/etc/keylime/ca.conf.d`
logging	`/etc/keylime/logging.conf`	`/etc/keylime/logging.conf.d`

The next sections contain details of the configuration files

2.6.1 Configuration file processing order

The configurations are loaded in the following order:

1. Default (hardcoded) option values
2. Base configuration options, overriding previously set values
   - By default, located in `/etc/keylime/<component>.conf` for each component
3. Configuration snippets, overriding previously set values
   - By default, located in `/etc/keylime/<component>.conf.d` for each component
   - The configuration snippets are loaded in lexicographic order
4. Environment variables, overriding previously set values
   - The environment variables are in the form `KEYLIME_{COMPONENT}_[SECTION_]{OPTION}`, for example `KEYLIME_VERIFIER_SERVER_KEY`.

2.6.2 Configuration file format

The configuration files for all components are written in INI format, except for the agent which is written in TOML format.

Each component contains a main section named after the component name (for historical reasons). For example, the main section of `verifier.conf` is `[verifier]`.

2.6.3 Override configurations via configuration snippets

To override a configuration option using a configuration snippet, simply create a file in the component's configuration snippets directory (e.g. `/etc/keylime/verifier.conf.d` to override options from `/etc/keylime/verifier.conf`).

Note that the configuration snippets are loaded and processed in lexicographic order, keeping the last value set for each option. It is recommended to use a numeric prefix for the files to control the order in which they should be processed (e.g. `001-custom.conf`).

The configuration snippets need the section to be explicitly provided, meaning that the snippets are required to be valid INI (or TOML in the case of the agent) files.

For example, the following snippet placed in `/etc/keylime/verifier.conf.d/0001-ip.conf` can override the default verifier ip address:

```
[verifier]
ip = 172.30.1.10
```

2.6.4 Override configurations via environment variables

It is possible to override configuration options by setting the desired value through environment variables. The environment variables are defined as `KEYLIME_{COMPONENT}_[SECTION_]{OPTION}`

The section can be omitted if the option to set is located in the main section (the section named after the component). Otherwise the section is required.

For example, to set the `webhook_url` option from the `[revocations]` section in the `verifier.conf` file, the environment variable to set is `KEYLIME_VERIFIER_REVOCATIONS_WEBHOOK_URL`.

To set an option located in the main section, for example the `server_key` option from the `[verifier]` section in the `verifier.conf`, the environment variable to set is `KEYLIME_VERIFIER_SERVER_KEY` (note that the section can be omitted).

2.6.5 Configuraton upgrades

When updating keylime, it is also recommended to upgrade the configuration to make sure that the used configuration is compatible with the keylime version.

The `keylime_upgrade_config` script is installed by default with the keylime components, and is the script that performs the configuration upgrade following the upgrade mappings and templates.

By default, the configuration templates are installed in `/usr/share/keylime/templates`. For each configuration version a respective directory is installed in the templates directory.

The `keylime_upgrade_config` will take the current configuration files as input (by default from `/etc/keylime/`). For each component, the version of the configuration file is checked against the available upgrade template versions to decide if an upgrade is necessary or not. In case an upgrade is not necessary, meaning all the components configuration files are up-to-date, the script does not modify the configuration files.

For each component that needs upgrade, the script will process all the transformations needed by each configuration version. For example, suppose the installed configuration file is in version `1.0` and there are upgrade templates available for the versions `2.0` and `3.0`. The upgrade script will process the transformations defined from the version `1.0` to the version `2.0` and then from the version `2.0` to the version `3.0`.

For each configuration upgrade template version, there are the following files:

- `mapping.json`: This file defines the transformations that should be performed, mapping the configuration option from the older version to the configuration options in the new version. If the mapping has the `update` type, then it describes operations to transform the previous version to the next (adding, removing, or replacing options)
- `<component>.j2`: These are templates for the configuration files. It defines the output format for the configuration file for each component.
- `adjust.py`: This is an optional script that defines special adjustments that cannot be specified through the `mapping.json` file. It is executed after the mapping transformations are applied.

The main goal of the upgrade script is to keep the configuration changes made by the user and keep the configuration files up-to-date. For new options, the default values are used.

2.6.6 The configuration upgrade script `keylime_upgrade_script`

Run `keylime_upgrade_config --help` for the description of the supported options.

When executed by the `root` user, the default output directory for the `keylime_upgrade_config` script is the `/etc/keylime` directory. The existing configuration files are kept intact as backup and renamed with the `.bkp` extension appended to the file names.

In case the `--output` option is provided to the `keylime_upgrade_config` script, the configuration files are written even when they were alredy up-to-date using the available templates. It can be seen as a way to force the creation of the configuration fiels, fitting the options read into the new templates.

Passing the `--debug` option to the `keylime_upgrade_config`, the logging level is set to `DEBUG`, making the script more verbose.

The templates directory to be processed can be passed via the `--templates` option. If provided, the script will try to find the configuration upgrade templates in the provided path instead of the default location (`/usr/share/keylime/templates`)

To output files only for a subset of the components, the `--component` can be provided multiple times.

To override input files (by default the `/etc/keylime/<component>.conf` for each component), the `--input` option can be passed multiple times. Unknown components are ignored.

To stop the processing in a target version, set the target version with the `--version` option.

To ignore the input files and use the default value for all options, the `--defaults` option can be provided

Finally, to process a single mapping file, the mapping file path can be passed via the `--mapping` option

CHAPTER

THREE

DESIGN OF KEYLIME

3.1 Overview of Keylime

Keylime mainly consists of an agent, two server components (verifier and registrar) and a commandline tool the tenant.

3.1.1 Agent

The agent is a service that runs on the operating system that should be attested. It communicates with the TPM to enroll the AK and to generate quotes and collects the necessary data like the UEFI and IMA event logs to make state attestation possible.

The agent provides an interface to provision the device further once it was attested successfully for the first time using the secure payload mechanism. For more details see: *Secure Payloads*.

It is possible for the agent to listen to revocation events that are sent by the verifier if an agent attestation failed. This is useful for environments where attested systems directly communicate with each other and require that the other systems are trusted. In this case a revocation message might change local policies so that the compromised system cannot access any resources from other systems.

3.1.2 Registrar

The agent registers itself in the registrar. The registrar manages the agent enrollment process which includes getting an UUID for the agent, collecting the EK_{pub}, EK certificate and AK_{pub} from an agent and verifying that the *AK* belongs to the *EK* (using *MakeCredential* and *ActivateCredential*).

Once an agent has been registered in the registrar, it is ready to be enrolled for attestation. The tenant can use the EK certificate to verify the trustworthiness of the TPM.

Note: If *EK* or *AK* are mentioned outside of internal TPM signing operations, it usually references the EK_{pub} or AK_{pub} because it should not be possible extract the private keys out of the TPM.

Note: The Keylime agent currently generates a *AK* on every startup and sends the EK and EK certificate. This is done to keep then design simple by not requiring a third party to verify the EK. The EK (and EK certificate) is required to verify the authenticity of the AK once and Keylime does not require a new AK but currently registration only with an AK is not enabled because the agent does not implement persisting the AK.

3.1.3 Verifier

The verifier implements the actual attestation of an agent and sends revocation messages if an agent leaves the trusted state.

Once an agent is registered for attestation (using the tenant or the API directly) the verifier continuously pulls the required attestation data from the agent. This can include: a quote over the PCRs, the PCR values, NK public key, IMA log and UEFI event log. After that the quote is validated additional validation of the data can be configured.

Static PCR values

The *tpm_policy* allows for simple checking of PCR values against a known good allowlist. In most cases this is only useful when the boot chain does not change often, there is a way to retrieve the values beforehand and the UEFI event log is unavailable. More information can be found in *User Selected PCR Monitoring*.

Measured Boot using the UEFI Event Log

On larger deployments it is not feasible to collect golden values for the PCR values used for measured boot. To make attestation still possible Keylime includes a policy engine for validating the UEFI event log. This is the preferred method because static PCR values are fragile because they change if something in the boot chain is updated (firmware, Shim, GRUB, Linux kernel, initrd, ...). More information can be found in *Use Measured Boot*.

IMA validation

Keylime allows to verify files measured by IMA against either a provided allowlist or a signature. This makes it for example easy to attest all files that were executed by root. More information can be found in *Runtime Integrity Monitoring*.

3.1.4 Tenant

The tenant is a commandline management tool shipped by Keylime to manage agents. This includes adding or removing the agent from attestation, validation of the EK certificate against a cert store and getting the status of an agent. It also provides the necessary tools for the payload mechanism and revocation actions.

For all the features of the tenant see the output of `keylime_tenant --help`.

3.2 Threat Model

Keylime was originally developed with the intention of using it in combination with hypervisors to protect the VMs against by using the vTPM support in Xen. vTPM support for TPM2.0 was never implemented into Keylime and swtpm+libvirt never supported it, so this model no longer fits. Keylime is commonly used either on bare metal hardware or in VMs where the TPM is emulated but from VM side treated the same as a hardware TPM. Therefore the common threat model is defined on the latter use case.

Note: The term vTPM can be confusing because it originally described the deep quote feature in Xen which Keylime used for TPM 1.2. Now it commonly refers to a software implementation of a TPM (e.g. swtpm) or the Virtual TPM Proxy Driver in the Linux kernel.

From Keylime's perspective the core hardware like CPU, memory, motherboard is trusted, because it does not provide mechanisms to detect tampering with the hardware itself. Keylime chains its root of trust into the TPM therefore the TPM is deemed in general trustworthy. This trust is verified using the EK or EK certificate.

The goal of Keylime is to attest the state of a running system. For this to work the entire boot chain has to be verified. The UEFI with Secure Boot enabled firmware and CRTM are generally trusted because it provides the UEFI event log and the API for other EFI applications to use the TPM. All the other applications in the boot chain are either measured by the firmware or the application that loads them (e.g. GRUB2 loads the kernel). The threat model does not require to trust arbitrary EFI applications during the boot process because it can be verified after boot what was executed.

The threat model includes that an adversary has full control over the network and can either sent rogue messages, drop or modify them. Also the Keylime agent and running operating system itself is not deemed trustworthy by default. Only after the successful initial attestation the system is deemed trustworthy, but still can leave the trusted state at any moment and is therefore continuously attested.

3.2.1 Types of Attacks to detect

Keylime tries to detect the following attacks.

TPM Quote Manipulation

Because the TPM is the root-of-trust for Keylime, it ensures that the quote is valid. This is vital for all the other attestation steps because the quote is used to validate the data.

Keylime ensures this through three steps:

- EK validation: The tenant allows Keylime to verify the EK certificate against the CAs of hardware manufacturers or add custom validation steps. This is done to ensure that the EK belongs to an actual hardware TPM or a trusted software TPM.
- AK enrollment: Using the TPM commands *MakeCredential*, *ActivateCredential* and enforcing certain key properties (restricted, user with auth, sign encrypt, fixed TPM, fixed parent and sensitive data origin) Keylime ensures that the used AK belongs to the provided EK and has the right properties for signing quotes.
- Quote validation: Each quote generated by the TPM is verified with the AK provided during agent registration. The verifier provides a fresh nonce that is included in the quote to prohibit replay attacks.

Modification of the boot process

Checking the security of the running system does only make sense if it can be ensured that the system was correctly booted. Therefore Keylime provides two ways to allow users to verify the entire boot chain up to the running system: static PCR value checks (*User Selected PCR Monitoring*) and the measured boot policy engine (*Use Measured Boot*).

Runtime file and system integrity

Keylime can attest the state of a Linux system and the files using the Linux Integrity Measurement Architecture (IMA). Therefore Keylime can be used to remotely check for attacks that IMA detects.

CHAPTER

FOUR

ADDITIONAL READING

This section contains a list of additional blog posts, research and talks about or related to Keylime.

4.1 Blogs entries

- Daniele Buono, Marcio A. Silva, Maurizio Drocco, Gheorghe Almasi, and James Bottomley. Extending server integrity with Durable Attestation. URL: https://research.ibm.com/blog/durable-attestation-cloud-security (visited on 2023-12-02).
- Kylie Foy. Keylime security software is deployed to IBM cloud. URL: https://news.mit.edu/2021/keylime-security-software-deployed-ibm-cloud-0727 (visited on 2023-12-02).
- Vikram Narayanan, Claudio Carvalho, Angelo Ruocco, Gheorghe Almási, James Bottomley, Mengmei Ye, Tobin Feldman-Fitzthum, Daniele Buono, Hubertus Franke, and Anton Burtsev. Remote attestation of SEV-SNP confidential VMs using e-vTPMs. URL: http://arxiv.org/abs/2303.16463 (visited on 2023-12-02), arXiv:2303.16463, doi:10.48550/arXiv.2303.16463.
- Michael Peters and Gheorghe Almasi. IBM implements remote attestation on Linux with a hardware root-of-trust using Keylime. URL: https://www.cncf.io/blog/2021/07/06/ibm-implements-remote-attestation-on-linux-with-a-hardware-root-of-trust-using-keylime/ (visited on 2023-12-02).
- Michael Peters, Marcio A. Silva, George Almási, James Bottomley, and Lily Sturmann. Keylime's durable attestation makes security auditable. URL: https://next.redhat.com/2023/04/25/keylimes-durable-attestation-makes-security-auditable/ (visited on 2023-12-02).
- Alberto Planas. MicroOS Remote Attestation with TPM and Keylime. URL: https://microos.opensuse.org/blog/2021-11-08-MicroOS-Keylime-TPM/ (visited on 2023-12-02).
- Patrick Uiterwijk. TPM2 Key Trust: where did Keylime go wrong. URL: https://puiterwijk.org/posts/tpm2-attestation-keylime-vulnerability/ (visited on 2023-12-02).
- Kimberly Underwood. Keylime Provides Root-of-Trust at Scale. URL: https://www.afcea.org/signal-media/keylime-provides-root-trust-scale (visited on 2023-12-02).

4.2 Academic Research

4.2.1 Original papers

- Amin Mosayyebzadeh, Gerardo Ravago, Apoorve Mohan, Ali Raza, Sahil Tikale, Nabil Schear, Trammell Hudson, Jason Hennessey, Naved Ansari, Kyle Hogan, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. A Secure Cloud with Minimal Provider Trust. *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, July 2018.
- Nabil Schear, Patrick T. Cable, Thomas M. Moyer, Bryan Richard, and Robert Rudd. Bootstrapping and maintaining trust in the cloud. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 65–77. ACM, December 2016. URL: https://dl.acm.org/doi/10.1145/2991079.2991104 (visited on 2023-12-02), doi:10.1145/2991079.2991104.

4.2.2 Research using Keylime

- Diana Gratiela Berbecaru and Silvia Sisinni. Counteracting software integrity attacks on IoT devices with remote attestation: a prototype. In *2022 26th International Conference on System Theory, Control and Computing (ICSTCC)*, 380–385. October 2022. URL: https://ieeexplore.ieee.org/document/9931765 (visited on 2023-12-02), doi:10.1109/ICSTCC55426.2022.9931765.
- Antonio Lioy, Dr Ignazio Pedone, and Dr Silvia Sisinni. TPM 2.0-based Attestation of a Kubernetes Cluster. *Politecnico di Torino*, 2023.
- Vikram Narayanan, Claudio Carvalho, Angelo Ruocco, Gheorghe Almási, James Bottomley, Mengmei Ye, Tobin Feldman-Fitzthum, Daniele Buono, Hubertus Franke, and Anton Burtsev. Remote attestation of SEV-SNP confidential VMs using e-vTPMs. URL: http://arxiv.org/abs/2303.16463 (visited on 2023-12-02), arXiv:2303.16463, doi:10.48550/arXiv.2303.16463.
- Silvia Sisinni, Davide Margaria, Ignazio Pedone, Antonio Lioy, and Andrea Vesco. Integrity Verification of Distributed Nodes in Critical Infrastructures. *Sensors 2022*, 22(18):6950, September 2022. URL: https://www.mdpi.com/1424-8220/22/18/6950 (visited on 2023-12-02), doi:10.3390/s22186950.

4.3 Talks and Live Demos

- Luke Hinds. Keylime - An Open Source TPM Project for Remote Trust of IoT. URL: https://www.youtube.com/watch?v=jtbWnod5hoY (visited on 2023-12-02).
- Luke Hinds. Keylime - An Open Source TPM Project for Remote Trust. URL: https://www.youtube.com/watch?v=YtPsruEqGeY (visited on 2023-12-02).
- Luke Hinds. Keylime Demo: Remote Trust for IoT, edge, and cloud. URL: https://www.youtube.com/watch?v=Qhr_aVBCZPw (visited on 2023-12-02).
- Luke Hinds. Keylime: Bootstrapping and Maintaining Trust. URL: https://www.youtube.com/watch?v=xPmv-G5V4I8 (visited on 2023-12-02).
- Alberto Planas. MicroOS Remote Attestation with TPM and Keylime. URL: https://www.youtube.com/watch?v=6F2mxG4YRKg (visited on 2023-12-02).
- Alberto Planas. Remote Attestation in MicroOS. 00:00:00 +0200. URL: https://media.ccc.de/v/3710-remote-attestation-in-microos (visited on 2023-12-02).
- Anderson Sasaki and Thore Sommer. Remote Attestation with Keylime. URL: https://archive.fosdem.org/2023/schedule/event/security_keylime/ (visited on 2023-12-02).

- Thore Sommer. Remote Attestation of the UEFI Event log. URL: https://vimeo.com/770419457 (visited on 2023-12-02).
- Thore Sommer. Writing Digital Exams secured by Remote Attestation and Cloud Computing. URL: https://www.youtube.com/watch?v=EXaPg2Yji4s (visited on 2023-12-02).
- Lily Sturmann and Michael Peters. Keylime: Bootstrap and Maintain Trust on the Edge, Cloud, and IoT. URL: https://www.youtube.com/watch?v=e_g32LxvOCk (visited on 2023-12-02).
- Andrew Toth. Keylime, Securing your Slice of the Cloud. URL: https://www.youtube.com/watch?v=O2x9gwq3BQQ (visited on 2023-12-02).

CHAPTER

FIVE

REST API'S

All Keylime APIs use *REST (Representational State Transfer)*.

5.1 Authentication

Most API interactions are secured using mTLS connections. By default there are two CAs involved, but the components can be configured to accommodate more complex setups.

(The revocation process also uses a CA, but this is different to those CAs)

5.1.1 Server Components CA

This CA is created by verifier on startup. It contains the server certificates and keys used by the verifier and registrar for their respective HTTPS interfaces. Then it also contains the client certificates and keys that are used by the tenant to connect to the registrar, verifier and agent. Also the verifier uses that certificate to authenticate itself against the agent.

5.1.2 Agent Keylime CA

The agent runs an HTTPS server and provides its certificate to the registrar (`mtls_cert`).

The server component CA certificate is also required on the agent to authenticate connections from the tenant and verifier. By default `/var/lib/keylime/cv_ca/cacert.crt` is used.

5.2 RESTful API for Keylime (v2.1)

Keylime API is versioned. More information can be found here: https://github.com/keylime/enhancements/blob/master/45_api_versioning.md

> **Warning:** API version 1.0 will no longer be officially supported starting with Keylime 6.4.0.

5.2.1 General responses

ANY /

Generic fields in responses

Response JSON Object

- **code** (*int*) – HTTP status code
- **status** (*string*) – textual context of that status
- **results** (*object*) – Holds the actual data.

5.2.2 Cloud verifier (CV)

GET /v2.1/agents/{agent_id:UUID}

Get status of agent *agent_id* from CV

Example response:


```
{
  "code": 200,
  "status": "Success",
  "results": {
    "operational_state": 7,
    "v": "yyNnlWwFRz1ZUzSe2YEpz9A5urtv6oywgttTF7VbBP4=",
    "ip": "127.0.0.1",
    "port": 9002,
    "tpm_policy": "{\"22\": [\"0000000000000000000000000000000000000000\", \
↪"0000000000000000000000000000000000000000000000000000000000000001\", \
↪"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪", \"ffffffffffffffffffffffffffffffffffffffff\", \
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff\", \
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
↪"], \"15\": [\"0000000000000000000000000000000000000000\", \
↪"0000000000000000000000000000000000000000000000000000000000000000\", \
↪"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"], \"mask\": \"0x408000\"}",
    "vtpm_policy": "{\"23\": [\"ffffffffffffffffffffffffffffffffffffffff\", \
↪"0000000000000000000000000000000000000000\"], \"15\": [\
↪"0000000000000000000000000000000000000000\"], \"mask\": \"0x808000\"}",
    "meta_data": "{}",
    "has_mb_refstate": 0,
    "has_runtime_policy": 0,
    "accept_tpm_hash_algs": [
      "sha512",
      "sha384",
      "sha256",
      "sha1"
    ],
    "accept_tpm_encryption_algs": [
      "ecc",
      "rsa"
    ],
```


(continues on next page)

(continued from previous page)


```
    "accept_tpm_signing_algs": [
      "ecschnorr",
      "rsassa"
    ],
    "hash_alg": "sha256",
    "enc_alg": "rsa",
    "sign_alg": "rsassa",
    "verifier_id": "default",
    "verifier_ip": "127.0.0.1",
    "verifier_port": 8881,
    "severity_level": 6,
    "last_event_id": "qoute_validation.quote_validation",
    "attestation_count": 240,
    "last_received_quote": 1676644582,
    "last_successful_attestation": 1676644462
  }
}
```


Response JSON Object

- **code** (*int*) – HTTP status code
- **status** (*string*) – Status as string
- **operational_state** (*int*) – Current state of the agent in the CV. Defined in https://github.com/keylime/keylime/blob/master/keylime/common/states.py
- **v** (*string*) – V key for payload base64 encoded. Decoded length is 32 bytes
- **ip** (*string*) – Agents contact ip address for the CV
- **port** (*string*) – Agents contact port for the CV
- **tpm_policy** (*string*) – Static PCR policy and mask for TPM
- **vtpm_policy** (*string*) – Static PCR policy and mask for vTPM
- **meta_data** (*string*) – Metadata about the agent. Normally contains certificate information if a CA is used.
- **has_mb_refstate** (*int*) – 1 if a measured boot refstate was provided via tenant, 0 otherwise.
- **has_runtime_policy** (*int*) – 1 if a runtime policy (allowlist and excludelist) was provided via tenant, 0 otherwise.
- **accept_tpm_hash_algs** (*list[string]*) – Accepted TPM hashing algorithms. sha1 must be enabled for IMA validation to work.
- **accept_tpm_encryption_algs** (*list[string]*) – Accepted TPM encryption algorithms.
- **accept_tpm_signing_algs** (*list[string]*) – Accepted TPM signing algorithms.
- **hash_alg** (*string*) – Used hashing algorithm.
- **enc_alg** (*string*) – Used encryption algorithm.
- **sign_alg** (*string*) – Used signing algorithm.

- **verifier_id** (*string*) – Name of the verifier that is used. (Only important if multiple verifiers are used)
- **verifier_ip** (*string*) – IP of the verifier that is used.
- **verifier_port** (*int*) – Port of the verifier that is used.
- **severity_level** (*int*) – Severity level of the agent. Might be *null*. Levels are the numeric representation of the severity labels.
- **last_event_id** (*string*) – ID of the last revocation event. Might be *null*.
- **attestation_count** (*int*) – Number of quotes received from the agent which have verified successfully.
- **last_received_quote** (*int*) – Timestamp of the last quote received from the agent irrespective of validity. A value of 0 indicates no quotes have been received. May be *null* after upgrading from a previous Keylime version.
- **last_successful_attestation** (*int*) – Timestamp of the last quote received from the agent which verified successfully. A value of 0 indicates no valid quotes have been received. May be *null* after upgrading from a previous Keylime version.

`POST /v2.1/agents/{agent_id:UUID}`

Add new agent *instance_id* to CV.

Example request:


```
{
  "v": "3HZMmIEc6yyjfoxdCwcOgPk/6X1GuNG+tlCmNgqBM/I=",
  "cloudagent_ip": "127.0.0.1",
  "cloudagent_port": 9002,
  "tpm_policy": "{\"22\": [\"0000000000000000000000000000000000000001\", \
↪"0000000000000000000000000000000000000000000000000000000000000001\", \
↪"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
↪", \"ffffffffffffffffffffffffffffffffffffffff\", \
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff\", \
↪"fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
↪"], \"15\": [\"0000000000000000000000000000000000000000\", \
↪"0000000000000000000000000000000000000000000000000000000000000000\", \
↪"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"], \"mask\": \"0x408000\"}",
  "ak_tpm": "ARgAAQALAAUAcgAAABAAFAALCAAAAAAAAQDjZ4J2HO7ekIONAX/eYIzt7ziiVAqE/
↪1D7I9oEwIE88dIfqH0FQLJAg8u3+ZOgsJDQr9HiMhZRPhv8hRuia8ULdAomyOFA1cVzlBF+xcPUEemOIofbvcBNAoTY/
↪x49r8LpqAEUBBiUeOniQbjfRaV2S5cEAA92wHLQAPLF9Sbf3zNxCnbhtRkEi6C3NYl8/
↪FJqyu5Z9vvwEBBOFFTPasAxMtPm6a+Z5KJ4rDflipfaVcUvTKLIBRI7wkuXqhTR8BeIByK9upQ3iBo+FbYjWSf+BaN+wodMNg
↪",
  "mtls_cert": "-----BEGIN CERTIFICATE----- (...) -----END CERTIFICATE-----",
  "runtime_policy_name": null,
  "runtime_policy": "",
  "runtime_policy_sig": "",
  "runtime_policy_key": "",
  "mb_refstate": "null",
  "ima_sign_verification_keys": "[]",
  "metadata": "{\"cert_serial\": 719066720466992686663564415155145407427243959000, \
↪"subject\": \"/C=US/ST=MA/L=Lexington/O=MITLL/OU=53/CN=D432FBB3-D2F1-4A97-9EF7-
↪75BD81C00000\"}",
```


(continues on next page)

(continued from previous page)

```
  "revocation_key": "-----BEGIN PRIVATE KEY----- (...) -----END PRIVATE KEY-----\n",
  "accept_tpm_hash_algs": [
    "sha512",
    "sha384",
    "sha256",
    "sha1"
  ],
  "accept_tpm_encryption_algs": [
    "ecc",
    "rsa"
  ],
  "accept_tpm_signing_algs": [
    "ecschnorr",
    "rsassa"
  ],
  "supported_version": "2.0"
}
```

Request JSON Object

- **v** (*string*) – V key for payload base64 encoded. Decoded length is 32 bytes.
- **cloudagent_ip** (*string*) – Agents contact ip address for the CV.
- **cloudagent_port** (*string*) – Agents contact port for the CV.
- **tpm_policy** (*string*) – Static PCR policy and mask for TPM. Is a string encoded dictionary that also includes a *mask* for which PCRs should be included in a quote.
- **ak_tpm** (*string*) – AK of the agent, base64-encoded, same as *aik_tpm* in the registrar.
- **mtls_cert** (*string*) – MTLS certificate of the agent, PEM encoded, same as in the registrar.
- **runtime_policy_name** (*string*) – Optional. If specified with a *runtime_policy* it is saved under that name, if specified without, then the policy with that name is loaded.
- **runtime_policy** (*string*) – Runtime policy JSON object, base64 encoded.
- **runtime_policy_sig** (*string*) – Optional runtime policy detached signature, base64-encoded. Must also provide *runtime_policy_key*.
- **runtime_policy_key** (*string*) – Optional runtime policy detached signature key, base64-encoded. Must also provide *runtime_policy_sig*.
- **mb_refstate** (*string*) – Measured boot reference state policy.
- **ima_sign_verification_keys** (*string*) – IMA signature verification public keyring JSON object string encoded.
- **metadata** (*string*) – Metadata about the agent. Contains *cert_serial* and *subject* if a CA is used with the tenant.
- **revocation_key** (*string*) – Key which is used to sign the revocation message of the agent.
- **accept_tpm_hash_algs** (*list[string]*) – Accepted TPM hashing algorithms. sha1 must be enabled for IMA validation to work.
- **accept_tpm_encryption_algs** (*list[string]*) – Accepted TPM encryption algorithms.

- **accept_tpm_signing_algs** (*list[string]*) – Accepted TPM signing algorithms.
- **supported_version** (*string*) – supported API version of the agent. *v* prefix must not be included.

DELETE /v2.1/agents/{agent_id:UUID}

Terminate instance *agent_id*.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {}
}
```

PUT /v2.1/agents/{agent_id:UUID}/reactivate

Start agent *agent_id* (for an already bootstrapped *agent_id* node)

PUT /v2.1/agents/{agent_id:UUID}/stop

Stop cv polling on *agent_id*, but don't delete (for an already started *agent_id*). This will make the agent verification fail.

POST /v2.1/allowlists/{runtime_policy_name:string}

Add new named IMA policy *runtime_policy_name* to CV.

Example request:

```
{
  "tpm_policy": "{\"22\": [\"0000000000000000000000000000000000000001\", \
↪"0000000000000000000000000000000000000000000000000000000000000001\", \
↪"000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
↪", \"ffffffffffffffffffffffffffffffffffffffff\", \
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff\", \
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
↪"], \"15\": [\"0000000000000000000000000000000000000000\", \
↪"0000000000000000000000000000000000000000000000000000000000000000\", \
↪"000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"], \"mask\": \"0x408000\"}",
  "runtime_policy": "",
  "runtime_policy_sig": "",
  "runtime_policy_key": ""
}
```

Request JSON Object

- **tpm_policy** (*string*) – Static PCR policy and mask for TPM. Is a string encoded dictionary that also includes a *mask* for which PCRs should be included in a quote.
- **runtime_policy** (*string*) – Runtime policy JSON object, base64 encoded.
- **runtime_policy_sig** (*string*) – Optional runtime policy detached signature, base64-encoded. Must also provide *runtime_policy_key*.
- **runtime_policy_key** (*string*) – Optional runtime policy detached signature key, base64-encoded. Must also provide *runtime_policy_sig*.

GET /v2.1/allowlists/{runtime_policy_name:string}

Retrieve named runtime policy *runtime_policy_name* from CV.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {
    "name": "",
    "tpm_policy": "{\"22\": [\"0000000000000000000000000000000000000001\", \
↪"0000000000000000000000000000000000000000000000000000000000000001\", \
↪"000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
↪", \"ffffffffffffffffffffffffffffffffffffffff\", \
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff\", \
↪"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
↪"], \"15\": [\"0000000000000000000000000000000000000000\", \
↪"0000000000000000000000000000000000000000000000000000000000000000\", \
↪"000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"], \"mask\": \"0x408000\"}",
    "runtime_policy": ""
  }
}
```

Request JSON Object

- **name** (*string*) – Name of the requested IMA policy.
- **tpm_policy** (*string*) – Static PCR policy and mask for TPM. Is a string encoded dictionary that also includes a *mask* for which PCRs should be included in a quote.
- **runtime_policy** (*string*) – Runtime policy JSON object, base64 encoded.

DELETE /v2.1/allowlist/{runtime_policy_name:string}

Delete IMA policy *runtime_policy_name*.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {}
}
```

5.2.3 Cloud Agent

GET /v2.1/keys/pubkey

Retrieves agents public key.

Example response:

```
{
  "code": 200,
  "status": "Success",
```

(continues on next page)

(continued from previous page)

```
  "results": {
    "pubkey": "-----BEGIN PUBLIC KEY----- (...) -----END PUBLIC KEY-----\n"
  }
}
```

Response JSON Object

- **pubkey** (*string*) – Public rsa key of the agent used for encrypting V and U key.

`GET /version`

Returns what API version the agent supports. This endpoint might not be implemented by all agents.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {
    "supported_version": "2.0"
  }
}
```

Response JSON Object

- **supported_version** (*string*) – The latest version the agent supports.

`POST /v2.1/keys/vkey`

Send *v_key* to node.

Example request:

```
{
  "encrypted_key": "MN/F33jjuLiIuRH8fF7pMtw6Hoe2KG10zg+/xuuZLa5d1WB2aR6feVCwknZDe/
↪dhG51yB0tKau8fCNUz8KMxyWoFkalIY4vVG6DNpLouDjb+vMvI6RmVmCBwO5zx6R802wK2z2yUbcn11TU/
↪k2zHq34CNFIgI5pQu7cnLMzCLW6NLEp8N0IOQL6D+uV9emkheJH1g40xYwUaKoABWjZeaJN5dvKwbkpIf2m+CROmCNPCidh87
↪tZErh1zk+nUamtrgl25pEImw+Cn9RIVTd6fBkmzlGzch5foAqZCyZ0AhQ0ONuWw=="
}
```

Request JSON Object

- **encrypted_key** (*string*) – V key encrypted with agents public key base64 encoded.

`POST /v2.1/keys/ukey`

Send *u_key* to node (with optional payload)

Example request:

```
{
  "auth_tag" :
↪"3876c08b30c16c4140ee04300bb4262bbcc9034d8a2ed8c90784f13b484a570bf9da3d5c372141bd16d85de05c4c7cce
↪",
  "encrypted_key":
↪"iAckMZgZc8r43pF0iW8iwwAorD+rvnvF7AShhlz6+am+ryqW+907UynOrWrIrAseyVRE7ouHpr547gnwfF7oKeBFlEdWnE6E
```

(continues on next page)

(continued from previous page)


```
↪y/
↪MmSuNR5pGQwZCueKI0ji3Nqq6heOgSvnMRC0PHgyumOsYiAnbDNyryvfwO4HsqdqMcEsBu1IVzU3EtJWhfQ8i/
↪UpvHy6Jq4bBh+mw5HZwmK93bmsLXNKgjPWAicsCZINUAPVMCUL7dcDd4zijsBxMxiZF7Js7V25wKKFer2zqKsE5omLy9sKotF
↪",
  "payload": "WcXpUr4G9yfvVaojNx6K2XZuDYRkFoZQhHrvZB+TKZqsq41g"
}
```


Request JSON Object

- **auth_tag** (*string*) – HMAC calculated with K key as key and UUID as data, using SHA-384 as the underlying hash algorithm
- **encrypted_key** (*string*) – U key encrypted with agents public key base64 encoded
- **payload** (*string*) – (optional) payload encrypted with K key base64 encoded.

GET /v2.1/keys/verify

Get confirmation of bootstrap key derivation

Example request:

```
/v2.1/keys/verify?challenge=1234567890ABCDEFHIJ
```

Parameters

- **challenge** (*string*) – 20 character random string with [a-Z,0-9] as symbols.

Example response:


```
{
  "code": 200,
  "status": "Success",
  "results": {
    "hmac":
↪"719d992fb7d2a0761785fd023fe1cf8a584b835e465e71e2ef2632ff4e9938c080bdefba26194d8ea69dd7f9adee6c18
↪"
  }
}
```


Response JSON Object

- **hmac** (*string*) – hmac with K key as key and the challenge

GET /v2.1/quotes/integrity

Get integrity quote from node

Example request:

```
/v2.1/quotes/integrity?nonce=1234567890ABCDEFHIJ&mask=0x10401&partial=0
```

Parameters

- **nonce** (*string*) – 20 character random string with [a-Z,0-9] as symbols.
- **mask** (*string*) – Mask for what PCRs from the TPM are included in the quote.

- **partial** (*string*) – Is either "0" or "1". If set to "1" the public key is excluded in the response.
- **ima_ml_entry** (*string*) – (optional) Line offset of the IMA entry list. If not present, 0 is assumed.

Example Response:

```
{
  "code": 200,
  "status": "Success",
  "results": {
    "quote": "r/
↪1RDR4AYABYABPihP2yz+HcGF0vD0c4qiKt4nvSOAARURVNUAAAAAAAyQ9AAAAAAAAAAAAEgGRAjABY2NgAAAAEABAMAAAEAF
↪yx6OVUze9jTDvML9xkkK1ghXObCJ5gH+QX0udKfrLacm/
↪iMds28SBtVO0rjqDIoYqGgXhH2ZhwGNDwjRCp6HquvtBe7pGEgtZlxf7Hr3wQRLO3FtliBPBR6gjOo7NC/
↪uGsuPjdPU7c9ls29NgYSqdwShuNdRzwmZrF57umuUgF6GREFlxqLkGcbDIT1itV4zJZtI1caLVxqiH0Qv3sNqlNLsSHggkgc5
↪TsEZOq/
↪leCoLtyVGYghPeGwg0RJfbe8cdyBWCQ6nOA==:AQAAAAQAAwAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪ntmsqy2aDi6NhKnLKz4k4uEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪",
    "hash_alg": "sha256",
    "enc_alg": "rsa",
    "sign_alg": "rsassa",
    "pubkey": "-----BEGIN PUBLIC KEY----- (...) -----END PUBLIC KEY-----\n"
    "boottime": 123456,
    "ima_measurement_list": "10 367a111b682553da5340f977001689db8366056a ima-ng␣
↪sha256:94c0ac6d0ff747d8f1ca7fac89101a141f3e8f6a2c710717b477a026422766d6 boot_
↪aggregate\n",
    "ima_measurement_list_entry": 0,
    "mb_measurement_list":
↪"AAAAAAMAAAAAAAAAAAAAAAAAAAAAAAAAAAAACEAAABTcGVjIElEIEV2ZW50MDMAAAAAAAACAAIBAAAACwAgAAAAAAACAAA
↪[....]"
  }
}
```

Response JSON Object

- **quote** (*string*) – TPM integrity quote
- **hash_alg** (*string*) – Used hash algorithm used in the quote (e.g. sha1, sha256, sha512).
- **enc_alg** (*string*) – Encryption algorithm used in the quote (ecc, rsa).
- **sign_alg** (*string*) – Signing algorthm used in the quote (rsassa, rsapss, ecdsa, ecdaa or ecschnorr).
- **pubkey** (*string*) – PEM encoded public portion of the NK (digest is measured into PCR 16).
- **boottime** (*int*) – Seconds since the system booted
- **ima_measurement_list** (*string*) – (optional) IMA entry list. Is included if *IMA_PCR* (10) is included in the mask
- **ima_measurement_list_entry** (*int*) – (optional) Starting line offset of the IMA entry list returned

- **mb_measurement_list** (*string*) – (optional) UEFI Eventlog list base64 encoded. Is included if PCR 0 is included in the mask

Quote format: The quote field contains the quote, the signature and the PCR values that make up the quote.

```
QUOTE_DATA := rTPM_QUOTE:TPM_SIG:TPM_PCRS
TPM_QUOTE  := base64(TPMS_ATTEST)
TPM_SIG    := base64(TPMT_SIGNATURE)
TPM_PCRS   := base64(tpm2_pcrs) // Can hold more that 8 PCR entries. This is a data␣
↪structure generated by tpm2_quote
```

GET /v2.1/quotes/identity

Get identity quote from node

Example request:

```
/v2.1/quotes/identity?nonce=1234567890ABCDEFHIJ
```

Parameters

- **nonce** (*string*) – 20 character random string with [a-Z,0-9] as symbols.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {
    "quote": "r/
↪1RDR4AYABYABPihP2yz+HcGF0vD0c4qiKt4nvSOAARURVNUAAAAAAAyQ9AAAAAAAAAAAAEgGRAjABY2NgAAAAEABAMAAAEAF
↪yx6OVUze9jTDvML9xkkK1ghXObCJ5gH+QX0udKfrLacm/
↪iMds28SBtVO0rjqDIoYqGgXhH2ZhwGNDwjRCp6HquvtBe7pGEgtZlxf7Hr3wQRLO3FtliBPBR6gjOo7NC/
↪uGsuPjdPU7c9ls29NgYSqdwShuNdRzwmZrF57umuUgF6GREFlxqLkGcbDIT1itV4zJZtI1caLVxqiH0Qv3sNqlNLsSHggkgc5
↪TsEZOq/
↪leCoLtyVGYghPeGwg0RJfbe8cdyBWCQ6nOA==:AQAAAAQAAwAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪ntmsqy2aDi6NhKnLKz4k4uEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪",
    "hash_alg": "sha256",
    "enc_alg": "rsa",
    "sign_alg": "rsassa",
    "pubkey": "-----BEGIN PUBLIC KEY----- (...) -----END PUBLIC KEY-----\n"
    "boottime": 123456
  }
}
```

Response JSON Object

- **quote** (*string*) – See *quotes/integrity*
- **hash_alg** (*string*) – See *quotes/integrity*
- **enc_alg** (*string*) – See *quotes/integrity*
- **sign_alg** (*string*) – See *quotes/integrity*
- **pubkey** (*string*) – See *quotes/integrity*

- **boottime** (*int*) – See *quotes/integrity*

5.2.4 Cloud Registrar

GET /v2.1/agents/

Get ordered list of registered agents

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {
    "uuids": [
      "5e600bce-a5cb-4f5a-bf08-46d0b45081c5",
      "6dab10e4-6619-4ff9-9062-ee6ad23ec24d",
      "d432fbb3-d2f1-4a97-9ef7-75bd81c00000"
    ]
  }
}
```

GET /v2.1/agents/{agent_id:UUID}

Get EK certificate, AIK and optinal contact ip and port of agent *agent_id*.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {
    "aik_tpm": "ARgAAQALAAUAcgAAABAAFAALCAAAAAAAAQDjZ4J2HO7ekIONAX/eYIzt7ziiVAqE/
↪1D7I9oEwIE88dIfqH0FQLJAg8u3+ZOgsJDQr9HiMhZRPhv8hRuia8ULdAomyOFA1cVzlBF+xcPUEemOIofbvcBNAoTY/
↪x49r8LpqAEUBBiUeOniQbjfRaV2S5cEAA92wHLQAPLF9Sbf3zNxCnbhtRkEi6C3NYl8/
↪FJqyu5Z9vvwEBBOFFTPasAxMtPm6a+Z5KJ4rDflipfaVcUvTKLIBRI7wkuXqhTR8BeIByK9upQ3iBo+FbYjWSf+BaN+wodMNg
↪",
    "ek_tpm": "AToAAQALAAMAsgAgg3GXZ0SEs/
↪gakMyNRqXXJP1S124GUgtk8qHaGzMUaaoABgCAAEMAEAgAAAAAAAEA0YwlPPIoXryMvbD5cIokN9OkljL2mV1oDxy7ETBXBe1
↪gDAqXryb+F192IJLKShHYSN32LJjCYOKrvNX1lrmr377juICFSRClE4q+pCfzhNj0Izw/
↪eplaAI7gq41vrlnymWYGIEi4McErWG7qwr7LR9CXwiM7nhBYGtvobqoaOm4+f6zo3jQuks/
↪KYjk0BR3mgAec/Qkfefw2lgSSYaPNl/8ytg6Dhla1LK8f7wWy/
↪bv+3z7L11KLr8DZiFAzKBMiIDfaqNGYPhiFLKAMJ0MmJx63obCqx9z5BltV5YQ==",
    "ekcert":
↪"MIIEGTCCAoGgAwIBAgIBBTANBgkqhkiG9w0BAQsFADAYMRYwFAYDVQQDEw1zd3RwbS1sb2NhbGNhMB4XDTIxMDQwOTEyNDAy
↪gDAqXryb+F192IJLKShHYSN32LJjCYOKrvNX1lrmr377juICFSRClE4q+pCfzhNj0Izw/
↪eplaAI7gq41vrlnymWYGIEi4McErWG7qwr7LR9CXwiM7nhBYGtvobqoaOm4+f6zo3jQuks/
↪KYjk0BR3mgAec/Qkfefw2lgSSYaPNl/8ytg6Dhla1LK8f7wWy/
↪bv+3z7L11KLr8DZiFAzKBMiIDfaqNGYPhiFLKAMJ0MmJx63obCqx9z5BltV5YQIDAQABo4HNMIHKMBAGA1UdJQQJMAcGBWeBB
↪wRIMEakRDBCMRYwFAYFZ4EFAgEMC2lkOjAwMDAxMDE0MRAwDgYFZ4EFAgIMBXN3dHBtMRYwFAYFZ4EFAgMMC2lkOjIwMTkxMD
↪wQCMAAwIgYDVR0JBBswGTAXBgVngQUCEDEOMAwMAzIuMAIBAAICAKIwHwYDVR0jBBgwFoAUaO+9FEi5yX/
↪GEnU+Vc6b3Si6JeAwDwYDVR0PAQH/BAUDAwcgADANBgkqhkiG9w0BAQsFAAOCAYEAaP/jI2i/
↪hXDrthtaZypQ8VUG5AWFnMDtgiMhDSaKwOBfyxiUiYMTggGYXLOXGIu1SJGBtRJsh3QSYgs2tJCnntWF9Jcpmk6kIW/
↪MC8shE+hdu/
↪gQZKjAPZS4QCLIldv+GVZdNYEIv2FYDsKl6Bq1qUsYhAb7z29Nu1itpdvja2qy7ODJ0u+ThccBuH60VGFclFdJg19dvVQMnf
```

(continues on next page)

(continued from previous page)


```
↪ZPTLNutJHmF0/Vk9W2pRym8SrUe8G6mwxVW8lP9M7fhovKTzoXVFW3gQWQeUxhvWOncXxtARFLp/
↪+f2mzGBRWxIslW17vpZ3QLlCdJ2C7P3U8x2tvkuyyDfz3/
↪pq+8ECupZhdSvpHlBnWvqs1tAWKW0qI9d0xNYjj3Kfl3Lfy7kqqe6FIkvbDlVhw3vnJlclW+M6D86jBulL9ze+3zyMxy2z8m7
↪",
    "mtls_cert": "-----BEGIN CERTIFICATE----- (...) -----END CERTIFICATE-----",
    "ip": "127.0.0.1",
    "port": 9002,
    "regcount": 1
  }
}
```


Response JSON Object

- **aik_tpm** (*string*) – base64 encoded AIK. The AIK format is TPM2B_PUBLIC from tpm2-tss.
- **ek_tpm** (*string*) – base64 encoded EK. When a *ekcert* is submitted it will be the public key of that certificate.
- **ekcert** (*string*) – base64 encoded EK certificate. Should be in *DER* format. Gets extracted from NV *0x1c00002*.
- **mtls_cert** (*string*) – Agent HTTPS server certificate. PEM encoded.
- **ip** (*string*) – IPv4 address for contacting the agent. Might be *null*.
- **port** (*integer*) – Port for contacting the agent. Might be *null*.

POST /v2.1/agents/{agent_id:UUID}

Add agent *agent_id* to registrar.

Example request:


```
{
  "ekcert":
↪"MIIEGTCCAoGgAwIBAgIBBTANBgkqhkiG9w0BAQsFADAYMRYwFAYDVQQDEw1zd3RwbS1sb2NhbGNhMB4XDTIxMDQwOTEyNDAy
↪gDAqXryb+F192IJLKShHYSN32LJjCYOKrvNX1lrmr377juICFSRClE4q+pCfzhNj0Izw/
↪eplaAI7gq41vrlnymWYGIEi4McErWG7qwr7LR9CXwiM7nhBYGtvobqoaOm4+f6zo3jQuks/
↪KYjk0BR3mgAec/Qkfefw2lgSSYaPNl/8ytg6Dhla1LK8f7wWy/
↪bv+3z7L11KLr8DZiFAzKBMiIDfaqNGYPhiFLKAMJ0MmJx63obCqx9z5BltV5YQIDAQABo4HNMIHKMBAGA1UdJQQJMAcGBWeBB
↪wRIMEakRDBCMRYwFAYFZ4EFAgEMC2lkOjAwMDAxMDE0MRAwDgYFZ4EFAgIMBXN3dHBtMRYwFAYFZ4EFAgMMC2lkOjIwMTkxMD
↪wQCMAAwIgYDVR0JBBswGTAXBgVngQUCEDEOMAwMAzIuMAIBAAICAKIwHwYDVR0jBBgwFoAUaO+9FEi5yX/
↪GEnU+Vc6b3Si6JeAwDwYDVR0PAQH/BAUDAwcgADANBgkqhkiG9w0BAQsFAAOCAYEAaP/jI2i/
↪hXDrthtaZypQ8VUG5AWFnMDtgiMhDSaKwOBfyxiUiYMTggGYXLOXGIu1SJGBtRJsh3QSYgs2tJCnntWF9Jcpmk6kIW/
↪MC8shE+hdu/
↪gQZKjAPZS4QCLIldv+GVZdNYEIv2FYDsKl6Bq1qUsYhAb7z29Nu1itpdvja2qy7ODJ0u+ThccBuH60VGFclFdJg19dvVQMnf
↪ZPTLNutJHmF0/Vk9W2pRym8SrUe8G6mwxVW8lP9M7fhovKTzoXVFW3gQWQeUxhvWOncXxtARFLp/
↪+f2mzGBRWxIslW17vpZ3QLlCdJ2C7P3U8x2tvkuyyDfz3/
↪pq+8ECupZhdSvpHlBnWvqs1tAWKW0qI9d0xNYjj3Kfl3Lfy7kqqe6FIkvbDlVhw3vnJlclW+M6D86jBulL9ze+3zyMxy2z8m7
↪",
  "aik_tpm": "ARgAAQALAAUAcgAAABAAFAALCAAAAAAAAQCg5mMzNFqdlUbW8uI/
↪GuMcIIvOXXTohHFTas59JlwrJQVed+5klWP+j7tI7492YPmCnoZvP4T4YdT1PN7tHHGfF81AeMnuw5GV5RkW/
↪QeSD+ssB4f6AfuzYJgBkc28zKmpRRHUbwN4rb/
↪HnJgRXdXsuIcnOqGcC39pD0kiu5TrN6hekjxTQtfAbIlQwwDwHCxKWdtH5x7avd15hqc6cBc2gjTQksXrk+OiMwOFTJ68n0qY
↪mVmd8XhPeYUoMlweXBOwc3e9zM9lZmMvregrFHKYc7CXChz",
```


(continues on next page)

(continued from previous page)

```
  "mtls_cert": "-----BEGIN CERTIFICATE----- (...) -----END CERTIFICATE-----",
  "ip": "127.0.0.1",
  "port": "9002"
}
```

Request JSON Object

- **ekcert** (*string*) – base64 encoded EK certificate. Should be in *DER* format. Gets extracted from NV *0x1c00002*.
- **aik_tpm** (*string*) – base64 encoded AIK. The AIK format is TPM2B_PUBLIC from tpm2-tss.
- **mtls_cert** (*string*) – Agent HTTPS server certificate. PEM encoded.
- **ip** (*string*) – (Optional) contact IPv4 address for the verifier and tenant to use.
- **port** (*string*) – (Optional) contact port for the verifier and tenant to use.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {
    "blob": "utzA3gAAAAEARAAgC/
↪w9LP1PKZ9thEk+GkMg4m+tkc9TkavcvFiFL6xbXM2q2fTRyKmQnxuCJc0tQdgsRXMftGiKJyA/
↪SUo8kGNVmcNfAQCs79kl9Ir49JJ8rfyMfDIqOuSVlu9PhxGUOeVzAdxyUmPxq5Qp0s431n/KeL/
↪5nUaVXC+qpOftF4bmVtXwLGTTUbKtyT3GG+9ujkjiwHCQhSKTQ8HiuARgXXh13ntFsJ75PBD5dWauLTuciYZI/
↪WQDVXAcgMnQNxodJUi9ir1GxJWz8zufjVQTVjrlgsgeBdOKbB6+H81K1d9prWhZaVLP+wIwO3YuWgtNHNi90E1z/
↪dah2pzfUpLvJo3lNZ4bJgrJUR507AokGKIFm7EfOf+5WWWAvGxGtgqTJB27vgE0CVBLEuDUHoRcLVBi1Np4GGNTByalxbulg8
↪"
  }
}
```

Response JSON Object

- **blob** (*string*) – base64 encoded blob containing the *aik_tpm* name and a challenge. Is encrypted with *ek_tpm*.

`DELETE /v2.1/agents/{agent_id:UUID}`

Remove agent *agent_id* from registrar.

Example response:

```
{
  "code": 200,
  "status": "Success",
  "results": {}
}
```

`PUT /v2.1/agents/{agent_id:UUID}/activate`

Activate physical agent *agent_id*

Example request:

```
{
  "auth_tag":
↪"7087ba88746886262de743587ed97aea6b6e3f32755de5d85415c40feef3169bc58d38855ddb96e32efdd8745d0bdfe
↪"
}
```


Request JSON Object

- **auth_tag** (*string*) – hmac containing the challenge from *blob* and the *agent_id*.

5.3 Changelog

Changes between the different API versions.

5.3.1 Changes from v2.0 to v2.1

API version 2.1 was first implemented in Keylime 6.4.0.

- Added *ak_tpm* field to *POST /v2.1/agents/{agent_id:UUID}* in cloud verifier.
- Added *mtls_cert* field to *POST /v2.1/agents/{agent_id:UUID}* in cloud verifier.
- Removed *vmask* parameter from

This removed the requirement for the verifier to connect to the registrar.

5.3.2 Changes from v1.0 to v2.0

API version 2.0 was first implemented in Keylime 6.3.0.

- Added mTLS authentication to agent endpoints.
- Added *supported_version* field to *POST /v2.0/agents/{agent_id:UUID}* in cloud verifier.
- Added *mtls_cert* field to *POST/GET /v2.0/agents/{agent_id:UUID}* in registrar.
- Added */version* endpoint to agent. Note that this endpoint is not implemented by all agents.
- Dropped zlib encryption for *quote* field data in *GET /v2.0/quotes/integrity*/*GET /v2.0/quotes/identity*.

CHAPTER

SIX

KEYLIME DEVELOPMENT

6.1 Contributing

When contributing any keylime repository, please first discuss the change you wish to make via an issue in the relevant repository for your change or ask the community in the `#keylime` channel on the CNCF Slack workspace

6.1.1 Pull Request Process

1. Create an issue outlining the fix or feature.
2. Fork the keylime repository to your own github account and clone it locally.
3. Hack on your changes.
4. Update the README.md or documentation with details of changes to any interface, this includes new environment variables, exposed ports, useful file locations, CLI parameters and configuration values.
5. Add and commit your changes with some descriptive text on the nature of the change / feature in your commit message. Also reference the issue raised at [1] as follows: *Fixes #45*. See the following link for more message types
6. Ensure that CI passes, if it fails, fix the failures.
7. Every pull request requires a review from the core keylime team
8. If your pull request consists of more than one commit, please squash your commits as described in see *Squash Commits*.

Commit Message Guidelines

We follow the commit formatting recommendations found on Chris Beams' How to Write a Git Commit Message article.

Well formed commit messages not only help reviewers understand the nature of the Pull Request, but also assists the release process where commit messages are used to generate release notes.

A good example of a commit message would be as follows:

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
```

(continues on next page)

(continued from previous page)

```
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded
  by a single space, with blank lines in between, but conventions
  vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

Note the *Resolves #123* tag, this references the issue raised and allows us to ensure issues are associated and closed when a pull request is merged.

Please refer to the github help page on message types for a complete list of issue references.

Squash Commits

Should your pull request consist of more than one commit (perhaps due to a change being requested during the review cycle), please perform a git squash once a reviewer has approved your pull request.

A squash can be performed as follows. Let's say you have the following commits:

```
initial commit
second commit
final commit
```

Run the command below with the number set to the total commits you wish to squash (in our case 3 commits):

```
git rebase -i HEAD~3
```

You default text editor will then open up and you will see the following:

```
pick eb36612 initial commit
pick 9ac8968 second commit
pick a760569 final commit

# Rebase eb1429f..a760569 onto eb1429f (3 commands)
```

We want to rebase on top of our first commit, so we change the other two commits to *squash*:

```
pick eb36612 initial commit
squash 9ac8968 second commit
squash a760569 final commit
```

After this, should you wish to update your commit message to better summarise all of your pull request, run:

```
git commit --amend
```

You will then need to force push (assuming your initial commit(s) were posted to github):

```
git push origin your-branch --force
```

Docker Test Environment

Python Keylime with a TPM emulator can be deployed using Docker. Since this docker configuration uses a TPM emulator, it should only be used for development or testing and NOT in production.

Please see either the Dockerfiles or our local CI script which will automate the build and pull of Keylime.

6.2 Updating Configurations

When configuration changes are introduced, a new mapping and configuration templates should be created (at least once per release where configuration changes were introduced).

When only adding new configuration options, the configuration file is still compatible with the previous version since all the options that existed before are still present. In this case, the configuration minor version number should be bumped.

When an option is removed or renamed, the configuration file is no longer compatible with the previous version. In this case, the major version number should be bumped.

6.2.1 Adding upgrade template files

For each configuration version, it should exist a corresponding directory under the `templates`. For example, for the configuration version `2.0`, the directory `templates/2.0` was added.

For each new version, the following files should be created:

- A directory for the new version number should be created in the `templates` directory
- The `mapping.json` file that specify the transformations from the previous configuration version to the new version should be added. See the format in *Mapping file format*
- For each existing component, the corresponding configuration file jinja2 template should be added. For example, for the `verifier`, the `verifier.j2` should be created in the directory for the new version
- If special transformations that cannot be expressed through the simple operations in the `mapping.json` file, the `adjust.py` script can be added. See below the requirements for the `adjust.py` in *Adjust script requirements*

For example, when adding templates for a new version `X.Y`, the following directory tree should be added:

```
templates
└── X.Y
    ├── adjust.py (optional)
```

(continues on next page)

(continued from previous page)

```
    ├── agent.j2
    ├── ca.j2
    ├── logging.j2
    ├── mapping.json
    ├── registrar.j2
    ├── tenant.j2
    ├── test.md
    └── verifier.j2
```

6.2.2 Mapping file format

For each configuration version, a mapping from the previous version to the new version is required. The mapping is provided as a JSON file, with the following fields:

- `version`: The mapping version, in the `MAJOR.MINOR` format. The value assigned to this field should match the template directory name. For example, for the mapping in the `templates/2.0` directory, the `version` field should be set as `2.0`.
- `type`: The mapping type. The supported values are `update` and `full`. See the requirements for each type below in *Update mapping* and *Full mapping*, respectively.
- `components`: The components to be modified by the mapping. Depending on the mapping type, the contents of the `components` field are different. See the requirements for each type below in *Update mapping* and *Full mapping*.
- `subcomponents`: Only present in the `full` mapping type, this field is required to associate sections to their parent components.
- Full mapping: The `type` field in the `mapping.json` file should be set as `full` for the mapping to be processed as a full mapping. In this type of mapping, all the options for all the components are treated as replacements of options. If an option is omitted, it means the option is removed in the new configuration version.

6.2.3 Update mapping

For the update mapping, the `type` field in the `mapping.json` file should be set as `update`. The update mapping is used to create a new configuration version by applying changes to the existing options through operations and preserve all the other options. This mapping type is the recommended way to introduce small changes to the configuration files.

In the update mapping, the `components` field list the components that are modified from the previous version, and the operations applied to them. See the format of the components dictionary below in the *Update mapping components format*

The update mapping does not use or need the `subcomponents` field.

Update mapping components format

For the update mapping, the `components` field should be set as a dictionary which maps the sections to be modified to the operations to be applied to them. Only the sections that are modified need to be present in the dictionary, all the omitted components are preserved as they were in the previous version.

The supported operations to modify the sections are:

- `add`: This operation adds a new option to the section. It should be assigned as a dictionary mapping the new option name to its default value.

  For example, the following mapping file adds 2 new options to the `[comp_a]` section:

```
{
    "version": "3.1",
    "type": "update",
    "components": {
        "comp_a": {
            "add": {
                "new_option": "value",
                "new_option2": "value2"
            }
        }
    }
}
```

- `remove`: This operation removes options that exists in the previous configuration version. An array of options to be removed should be assigned to the `remove` field for the section/component to be modified.

  For example, the following mapping file removes 2 options from the `[comp_a]` section:

```
{
    "version": "3.1",
    "type": "update",
    "components": {
        "comp_a": {
            "remove": ["unused_option", "another_unused_option"]
        }
    }
}
```

- `replace`: This operation replaces an option that exists in the previous configuration version with another in the new version. During the processing of the mapping file, the value found in the replaced option will be preserved, and assigned to the new option in the output. If the option is not found in the input configuration file, the default value is used instead.

  The `replace` field should be set as a dictionary mapping the option to be replaced to the parameters for the new option. The dictionary should have the following fields:

  - `section`: The section to which the new option should be added
  - `option`: The new option name
  - `default`: The default value to be used in case the old option is not found in the input configuration.

  For example, the following mapping file replaces two options from the `[comp_a]` section:

```
{
    "version": "3.1",
    "type": "update",
    "components": {
        "comp_a": {
            "replace": {
                "old_option_to_replace": {
                    "section": "new_section",
                    "option": "new_option",
                    "default": "value"
                },
                "old_value": {
                    "section": "other_section",
                    "option": "other_new_option",
                    "default": "value"
                }
            }
        }
    }
}
```

6.2.4 Full mapping

In the full mapping, al the options of the new configuration version should be declared. If an option is omitted, it means the option is removed.

The format of the fields in the full mapping file are:

- `version`: Should be in the `MAJOR.MINOR` format. The version should match the directory name
- `type`: Should be set as `full`. If omitted, the mapping will be treated as as full mapping
- `components`: Should be set as a dictionary which maps each component to a dictionary of options. See the option dictionary format below in *Full mapping components format* section.
- `subcomponents`: Should be set as a dictionary mapping subcomponents to its main component. This is necessary to create a relationship between the sections of the files that are not components (e.g. The `[revocations]` section in the `verifier.conf` file should be declared in the `subcomponents` dictionary as `"revocations": "verifier"`). See the format below in *Subcomponents format*

Full mapping components format

For each component, the options transformations should be declared through dictionaries that map the **new option** name to the **option it is replacing**.

The upgrade script will search the options to be replaced in the old configuration, in the section provided in the `section` field. If the option is found, the value is preserved in the news configuration, otherwise the value provided in the `default` field is used instead.

As an example, follows an excerpt of the `templates/2.0/mapping.json` file:

```
"components": {
    "agent": {
        "version": {
```

(continues on next page)

(continued from previous page)

```
            "section": "agent",
            "option": "version",
            "default": "2.0"
        },
        "revocation_notification_ip": {
            "section": "general",
            "option": "receive_revocation_ip",
            "default": "127.0.0.1"
        },
}
```

In the excerpt above, in the `agent` component two options are declared, `version` and `revocation_notification_ip`.

The new option `revocation_notification_ip` will receive the value from the `receive_revocation_ip` from the `general` section in the old configuration file. If the option is not found, the value `127.0.0.1` provided in the `default` field is used instead.

Subcomponents format

The configuration file for some of the components (e.g. the `verifier.conf`) have more than one section. The main section is named after the component (e.g. `[verifier]` section of the `verifier.conf` file). The other sections are considered subsections, or subcomponents, by the configuration upgrade script. The subsections are associated with their main section in the `Subcomponents` dictionary, which maps a subsection to the associated main section.

For example, the following excerpt from the `templates/2.0/mapping.json` file:

```
"subcomponents": {
    "revocations": "verifier",
    "loggers": "logging",
    "handlers": "logging",
    "formatters": "logging",
    "formatter_formatter": "logging",
    "logger_root": "logging",
    "handler_consoleHandler": "logging",
    "logger_keylime": "logging"
}
```

In the excerpt, the `[revocations]` section is declared as a subsection (or subcomponent) of the `[verifier]` section.

6.2.5 Adjust script requirements

The optional `adjust.py` script can perform complex operations that cannot be expressed using the `mapping.json` file. For example, deciding the value of an option depending on the presence of another option in the configuration file.

The `adjust.py` script is processed after the `mapping.json` is applied. For this reason, when writing the `adjust.py` script, the author should consider the input to be the output of the processing of the associated `mapping.json` file.

The only requirement for the `adjust.py` script is to implement the `adjust()` function, defined as the following:

```
def adjust(
    config: RawConfigParser, mapping: Dict, logger: Logger = logging.getLogger(__name__)
) -> None:
```

The `config` parameter is the result after applying the transformations defined by the `mapping.json` file.

The `mapping` parameter is the dictionary read from the `mapping.json` JSON file.

The optional `logger` parameter will receive the logger from the `keylime_upgrade_config` script, so that all the log messages are in a single place. It is recommended to keep the default assigned as `logging.getLogger(__name__)` so that the `keylime_upgrade_config` can set the log level accordingly.

The `adjust()` function should make the changes to the parser received through the `config` parameter directly.

CHAPTER

SEVEN

SECURING KEYLIME

Warning: This page is still under development and not complete. It will be so until this warning is removed.

7.1 System Hardening

7.2 TLS configuration

7.3 Reporting an issue

Please contact us directly at security@keylime.groups.io for any bug that might impact the security of this project. Do not use a github issue to report any potential security bugs.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- search

HTTP ROUTING TABLE

/

/v2.1

/version