



MasQiTT

Secure MQTT Reference Implementation

User Guide

August 13, 2024



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Intentionally blank

Contents

1	Introduction	1
1.1	Document overview	1
2	MasQiTT reference implementation	2
2.1	MasQiTT API	2
2.2	Crypto API and libraries	3
2.3	KMS Comms	3
2.4	Key Management Server (KMS)	4
2.5	Certification Authority (CA)	4
3	Implementation details	5
3.1	General behavior	5
3.2	Considerations for implementers	5
3.3	Miscellany	8
4	Installing and building MasQiTT	9
4.1	Prerequisites	9
4.2	Building MasQiTT	10
5	Using MasQiTT	12
5.1	Provisioning	12
5.2	Operation	12
6	Porting guide	15
6.1	Set up	15
6.2	Usage	16
6.3	Cleanup	16
6.4	Compilation	17
6.5	Runtime	17
7	Integration guide	18
7.1	Publisher only	18
7.2	Subscriber only	20
7.3	Both	21
A	Acronyms, abbreviations, and references	23
B	Files	24

Figures

2-1	Mosquitto CLI code	2
2-2	MasQiTT code	2



Tables

4-1	Third-party software	9
-----	--------------------------------	---

1 Introduction

MasQiTT is a reference implementation of Secure MQTT as described in [1] and provided by Sandia National Laboratories [2]. Secure MQTT adds security features to MQTT Version 5 [3]. It is intended as a proof of concept and should be considered research-quality code.

MasQiTT is provided as C-language source code and Python 3 scripts. It assumes a Linux environment with reasonably up-to-date versions of common software development tools. It leverages several open source software packages, which are detailed in Section 4.1.

 MasQiTT should not be considered for operational use as-is. There are significant quality of life features which were consciously omitted in favor of implementing the Secure MQTT protocol itself. The most glaring improvements that should be made before adapting MasQiTT for operational use are highlighted with “.

Copyright 2024 National Technology and Engineering Solutions of Sandia LLC.

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.1 Document overview

Use of MasQiTT assumes familiarity with MQTT v5 in general and the Secure MQTT protocol in particular. If you lack those, it would be helpful to skim [3] or an MQTT tutorial, then read [1] before continuing here.

The rest of this document is organized into the following sections.

- Section 2 An overview of the MasQiTT reference implementation.
- Section 3 A discussion of which optional Secure MQTT protocol features are included in MasQiTT and how they are implemented.
- Section 4 Instructions for building and installing MasQiTT.
- Section 5 How to set up and run MasQiTT.
- Section 6 How to modify an existing Mosquitto installation to incorporate MasQiTT.
- Section 7 How to integrate MasQiTT into another MQTT code base.
- Appendix A Abbreviations, acronyms, and references.
- Appendix B A guide to MasQiTT source code.

2 MasQiTT reference implementation

MasQiTT extends the Mosquitto [4] code base to provide Secure MQTT functionality.

In addition to its library for managing MQTT communications, Mosquitto provides a command line interface (CLI) utility that implements Broker, Subscriber, or Publisher functionality, depending on the arguments specified on the command line. The Mosquitto library has a well-defined API [5] for creating, parsing, and handling MQTT packets, and it is used by the CLI utility. Figure 2-1 illustrates the major code components of a Mosquitto CLI Broker or Client.

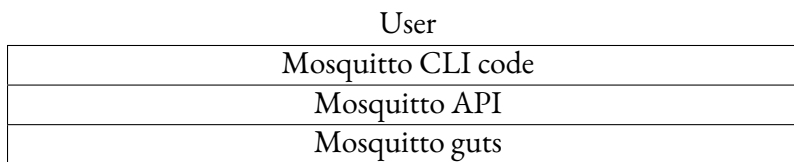


Figure 2-1: Mosquitto CLI code

Secure MQTT specifies User Property fields that must appear in a PUBLISH packet. Secure MQTT Publishers and Subscribers must also communicate with the Key Management Server (KMS) to obtain the cryptographic values needed to encrypt and decrypt Topic Values.

MasQiTT provides a Secure MQTT API layer to hide these gory details from Client code. This layer is illustrated in Figure 2-2 as “MasQiTT API.” New code written for Secure MQTT is identified with a “▷”. Each of these new MasQiTT-specific components is described in the following sections as indicated in the figure.

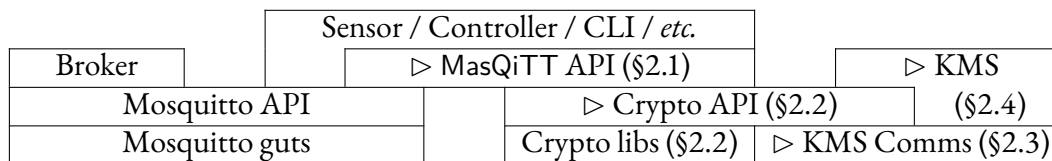


Figure 2-2: MasQiTT code

2.1 MasQiTT API

This MasQiTT API provides a few replacement calls to substitute for those in the stock Mosquitto API with enhancements to support Secure MQTT. Section 6 has detailed information about adapting an existing Mosquitto Client to incorporate MasQiTT’s implementation of the Secure MQTT protocol.

For example, a stock Mosquitto client would call the initialization routine in the Mosquitto API and receive a pointer to an opaque (unparsed by the Client) data structure that must subsequently be passed to every API call so Mosquitto can keep track of the Client’s state. MasQiTT’s Crypto API employs a similar initialization and

pointer to the current state strategy. A MasQiTT Client calls the MasQiTT API initialization routine instead, and that code calls the Mosquitto and Crypto API initialization routines on the Client's behalf, collects the respective state pointers and returns them to the Client in an opaque MasQiTT state pointer.

The MasQiTT API initialization routine requires a few more parameters than its Mosquitto counterpart, including information about how to contact the KMS (host address, port number) and cryptographic keys and certificates needed to establish TLS connections with the KMS.

After initialization, MasQiTT-provided calls are used when a Client sends or receives a PUBLISH message. When a Publisher sends a PUBLISH message, the MasQiTT API code will call the Crypto API code with the Topic Name and Value. Using information provided at initialization time and tracked via the state pointer, the crypto code encrypts the Topic Value and returns the encrypted Value accompanied by User Properties that must be passed to Subscribers so they can decrypt the Topic Value. The MasQiTT API then adds the User Properties to the in-process PUBLISH packet and replaces the Topic Value with its encrypted form before handing the packet to Mosquitto to send.

Similarly, when a Subscriber receives a PUBLISH packet, the MasQiTT API code extracts the User Properties from the packet that Mosquitto received, then provides them with the Topic Name and encrypted Value to the Crypto API to decrypt the Topic Value. The MasQiTT API code passes along the Topic Name and Value to the Subscriber as if the Value had never been encrypted.

2.2 Crypto API and libraries

The cryptographic algorithms used by Secure MQTT are not directly supported by an existing cryptography library, so MasQiTT includes a cryptographic API that employs cryptographic primitives from the MIRACL Core [6] library ("Crypto libs" in Figure 2-2) and provides calls to perform Secure MQTT cryptographic operations.

The Crypto API library provides a small number of high-level functions to support the MasQiTT API code, and these have names starting "MASQ_crypto_api_." A larger number of lower-level functions (MASQ_crypto_) support KMS cryptographic operations and housekeeping functions (MASQ_) including key caching. These lower-level functions are also used by the higher-level code.

2.3 KMS Comms

The KMS Comms component provides functions for creating and parsing each of the types of messages that Clients exchange with a KMS as well as a simplified interface with a third-party open source network communications package (wolfSSL [7]) for sending and receiving those messages over TLS connections.

The Crypto Code component occasionally needs to get cryptographic values from the KMS (*e.g.*, IBE private keys for a Subscriber); when it does, it calls these functions to build a request packet, send the request packet,

receive a response packet, and parse the response packet.

The MasQiTT API does not interact with this component except indirectly as needed by the Crypto Code. The KMS, however, directly calls this code to wait for requests, receive a request, parse the request, form a response, and send the response.

2.4 Key Management Server (KMS)

The MasQiTT Key Management Server provides shared public IBE parameters and public key expiration information to Publishers, as well as IBE private keys to Subscribers. The MasQiTT API interacts with the KMS on behalf of Secure MQTT Clients. As illustrated in Figure 2-2, the KMS employs the same underlying cryptographic support that the MasQiTT API relies upon.

2.5 Certification Authority (CA)

Included here for completeness, MasQiTT's Certification Authority is not shown in Figure 2-2 as it does not directly participate in the Secure MQTT protocol. It serves a support role for protecting Client-KMS communications through Transport Layer Security (TLS) tunnels.

The CA is implemented using a collection of scripts that call OpenSSL utilities to establish a CA signing certificate, generate TLS certificates for Clients, and sign the Client certificates with the CA's certificate.

3 Implementation details

There is latitude within the Secure MQTT protocol to make implementation choices. Here are the choices made for MasQiT T.

3.1 General behavior

MasQiT T Subscribers are long-lived, keeping a connection with a Broker alive until it receives a DISCONNECT packet, the network connection drops, or it is otherwise compelled to quit.

A MasQiT T Publisher can establish a connection with a Broker, send a single PUBLISH packet, and disconnect; or it can keep its connection with a Broker alive, sending PUBLISH packets when it has something to say. If the former, care should be taken to ensure the underlying MasQiT T library is properly initialized before and closed after each Broker connection.

MasQiT T supports Secure MQTT protocol 1.0/1 — 128-bit AES-GCM to encrypt Topic Values, BB₁ Identity-Based Encryption (using BN254 pairing curves and SHA256) to encapsulate AES encryption keys.

The KMS configuration file (`/home/kms/kms.cfg`) is updated by enrollment (Section 5.2.1) activities. The configuration file also includes values that may be customized by an implementer (described below).

3.2 Considerations for implementers

This section parallels Appendix D of the Secure MQTT specification [1] and describes what MasQiT T does for each D.x section.

3.2.1 Public key expiration

MasQiT T code internally handles time in terms of Linux time, the number of seconds after the Linux time epoch. Time values in the `kms.cfg` configuration file can be specified as an integer multiple of seconds, minutes, hours, days, weeks, or years where

- `m` = 1 minute = 60 seconds
- `h` = 1 hour = 60 minutes or 3,600 (60×60) seconds
- `d` = 1 day = 24 hours or 86,400 ($24 \times 3,600$) seconds
- `w` = 1 week = 7 days or 604,800 ($7 \times 86,400$) seconds
- `y` = 1 year = 365 days + 6 hours or 31,557,600 ($365.25 \times 86,400$) seconds

E.g., “8h” specifies 8 hours (28,800 seconds).

The time between public key expirations is specified in the `kms.cfg` configuration file by the “`expiry_interval`” value.

The KMS enforces a minimum expiration interval of one day to avoid the excessive churn of having to recompute IBE private keys each time the key expires. No support is provided for expiring keys on, say, the second Tuesday of each month at noon local time.



A more flexible key expiration schedule may be worth considering for your implementation.

Expiration dates are determined as the next multiple of the expiration interval after the epoch. “epoch” can be specified in `kms.cfg` as a Secure MQTT date/time string. “epoch” defaults “19700101T000000Z,” the Unix/Linux time represented as 0.

For example, if “epoch” is not specified and “`expiry_interval`” is “2d” (two days), the current key expiration date will be set to the next Linux time (seconds) that is a multiple of 172,800 ($2 \times 24 \times 60 \times 60$ seconds), and the next expiration date will be 172,800 seconds after that.

Setting “epoch” can be used to cause key expirations to happen at midday every fourth Tuesday by setting it to “20240102T180000Z” (a random convenient Tuesday in the past) and `key_expiry` to “4w,” assuming a continental US location where 18:00Z is as early as 10:00PST or as late as 14:00EDT. In this case, keys would expire on Jan 2, Jan 30, Feb 27, Mar 26, Apr 23, May 21, Jun 18, Jul 16, Aug 13, Sep 10, Oct 7, Nov 5, Dec 3, and Dec 31 in 2024, and every fourth Tuesday after that.

3.2.2 Clients and Topics

Appendix D.2 applies to how a Secure MQTT installation as a whole is architected. MasQiTT supports all of the options presented.

3.2.3 KMS key caching

The MasQiTT KMS caches IBE private keys that it computes to avoid the overhead of recomputing them later. This cache is held in memory by the KMS and periodically written to disk (the `cache.smqtt` file). The cache is also saved during a KMS orderly shutdown, and initialized from `cache.smqtt` when the KMS is started.

How often the KMS writes its private key cache is determined by the “`cache_save_interval`” value in `kms.cfg` and if not specified defaults to “1h.” This timer is aligned to the time when the KMS is started. In other words, if “`cache_save_interval`” is “1h” and the KMS is started at 09:34:17, the cache will be saved to disk at 10:34:17, 11:34:17, *etc.* (`kms_ctrl -d` does not interrupt this schedule.)

Private keys are deleted from the KMS cache according to the “`expire_cache_after`” configuration value


(default: “4h”). For example, keys with an expiration time of “20240903T180000Z” and the default expiration time will be deleted from the cache at 20240903T220000Z. These deletions will not be reflected in the `cache.smqtt` file until the cache is next written to disk.

If “`precompute_keys`” in `kms.cfg` is true, private keys corresponding to the next expiration date will be computed “`precompute_lead_time`” (default: “2h”) before that expiration date.

MasQiTT keeps track of which keys in its cache are actually requested by a Subscriber. Keys which are not requested will not be precomputed for the next expiration date. The “U” column in the output of `print_cache` indicates whether a key has been used (requested).


3.2.4 Anti-replay

MasQiTT handles sequence numbers as required in the protocol specification, but Subscriber code does not track them for anti-replay purposes. Sequence numbers are randomly initialized on a per-Topic basis and incremented (also on a per-Topic basis) by one for each PUBLISH packet. The sequence number after `ffffffff` is `00000000`.

 Adding anti-replay enforcement is left to the implementer.


3.2.5 KMS-Client authentication

MasQiTT Client certificates include the Client ID as the CN component of the certificate Subject Name. Client certificate Subject Names are set to “`O=MasQiTT CN=ClientID`” by MasQiTT.

The MasQiTT KMS does not validate the Subject Name presented in the Client certificate against the Client ID it receives in the request packet.  This is an opportunity for enhancement.

3.2.6 Topic constraints

MasQiTT does not enforce any restrictions on the Topics a Publisher may publish or a Subscriber may subscribe to.

 Implementers may wish to leverage the “`clients`” entry in `kms.cfg` as a convenient place for specifying whitelists (only these Topics are allowed) and/or blacklists (these Topics are forbidden) for Clients.

3.3 Miscellany

3.3.1 KMS message handling

The MasQiTT KMS is single-threaded, meaning that it waits for an incoming request message, processes it, returns a response message, and then waits for another request. ⚠️ In a production environment a KMS should be multithreaded, *i.e.*, able to handle multiple requests at a time by spinning off a new process or thread for each request.

3.3.2 TLS certificate expiration

MasQiTT does not provide a mechanism for replacing expired TLS certificates. MasQiTT creates certificates with an absurdly long validity period (ten years) for test or evaluation purposes.

⚠️ A full-fledged PKI should be used for a production environment.

4 Installing and building MasQiTT

MasQiTT was developed on Ubuntu 22 Linux systems and has been confirmed to run on Ubuntu 24. Instructions should work on reasonably current releases of Ubuntu and other Debian-based Linux systems, but there are no guarantees.

MasQiTT's focus on the Secure MQTT protocol means most of the development effort was directed there. To the degree practical, third-party open source software, listed in Table 4-1, was leveraged when convenient. Instructions for obtaining and installing those are below in Section 4.1.

Software	License
Mosquitto [4]	Eclipse Public License 2.0 & Eclipse Distribution License 1.0
MIRACL Core [6]	Apache 2.0
wolfSSL [7]	Gnu Public License v2
OpenSSL [8]	Apache 2.0
libconfig [9]	Gnu Lesser Public License

Table 4-1: Third-party software

Section 4.2 explains obtaining and building MasQiTT itself.

For illustration purposes this description assumes that MasQiTT and supporting third-party software packages are all installed in their own directories in \$HOME. Adjust as desired for your preferences.

4.1 Prerequisites

Download and install these third-party open source software packages. The versions used when developing MasQiTT are noted, though newer versions will likely work, too.

- OpenSSL — On Debian systems:

```
$ sudo apt install openssl libssl-dev
```

libssl-dev is required to build Mosquitto, and MasQiTT uses the openssl command to create TLS keys and certificates.

- wolfSSL — On Debian systems:

```
$ sudo apt install libwolfssl-dev
```

This provides TLS support for Client-KMS communication.

- Mosquitto v2.0.18

Download `mosquitto-2.0.18.tar.gz` from <https://mosquitto.org/download>

```
$ cd
$ tar xf mosquitto-2.0.18.tar.gz
$ cd mosquitto-2.0.18
$ make
$ sudo make install
```

- MIRACL Core 4.1

MIRACL Core has a unique interactive command-line interface for building its library as `core.a`. MasQiTT deals with this on your behalf when building `libmasqitt.so`, provided MIRACL's top-level core directory is in the same directory as the top-level `masqitt` directory.

```
$ cd
$ git clone https://github.com/miracl/core.git
```

- libconfig 1.7.3

```
$ cd
$ git clone https://github.com/hyperrealm/libconfig.git
$ cd libconfig
$ ./configure
$ make
$ sudo make install
```

4.2 Building MasQiTT

If you're reading this User Guide, chances are high you already have a copy of the MasQiTT code, but if not,

```
$ cd
$ git clone https://github.com/sandialabs/masqitt.git
```

KMS code will be compiled/installed only if there is a user whose login is "kms" on your system.

```
$ sudo adduser kms
```

Compiling MasQiTT is as simple as

```
$ cd $HOME/masqitt
```

```
$ make
$ sudo make install
```

Depending on your system, running a MasQiTT program may result in an error message complaining that `libmasqitt.so` can not be found. In that case, you will need to set the `LD_LIBRARY_PATH` environment variable to point to MasQiTT's installation directory. It is a good idea to add this initialization to your `$HOME/.bashrc` file to avoid the need to re-do it later.

```
$ LD_LIBRARY_PATH="/usr/local/lib:${LD_LIBRARY_PATH}" export LD_LIBRARY_PATH
```

If you wish to run some sanity tests,

```
$ cd $HOME/masqitt
$ make test
$ sudo make install
$ cd tests
$ ./msgtest
$ ...
```

See `tests/README.md` for instructions on running tests.

5 Using MasQiTT

MasQiTT must first be provisioned as outlined in Section 5.1. After that, Clients must be enrolled, and enrolled Clients can then participate in the Secure MQTT protocol as discussed in Section 5.2.

5.1 Provisioning

The provisioning phase encompasses setting up the CA and KMS and establishing cryptographic values needed for Secure MQTT operation. Provisioning is performed by the `kms` user. ⚠ Re-running the following commands after a system is operational will make the system unstable at best, most likely unusable.

```
$ su - kms
```

Edit `/home/kms/kms.cfg` to your liking. Configuration options are documented by comments in the file and discussed in Section 3.2. These configuration options may be changed later. The KMS will read and parse this file when it starts up and when told to by running `kms_ctrl -c`.

Run the following commands to complete provisioning MasQiTT.

```
kms$ make_params
```

This creates the BB_1 secret values and shared public parameters and saves them in `/home/kms/params.smqtt`.

```
kms$ generate_ca
```

This creates the CA's public/private key, a self-signed CA certificate, and public/private keys for the KMS. These files are found in `/home/kms/ca`.

- `ca-crt.pem` — CA self-signed certificate
- `ca-key.pem` — CA private key
- `kms-crt.pem` — KMS certificate signed by CA
- `kms-key.pem` — KMS private key

5.2 Operation

Once provisioned, a Secure MQTT installation is considered operational. A Client is added to an installation through enrollment (Section 5.2.1), after which it can fully participate in its message-passing responsibilities (Section 5.2.2).

In practical terms, a MasQiTT installation will need an active MQTT Broker and the KMS up and running before Secure MQTT messages can be passed. Refer to Mosquitto [4] (or other applicable) documentation for starting a Broker. The KMS can only be started by the `kms` user.


```
$ su - kms
```

```
kms$ kms
```

This runs the KMS in the foreground and writes status information to its terminal window. Add the `-v` option for more verbose output. Typing control-C in the KMS's terminal window causes the KMS to perform an orderly shutdown, writing its cache to `cache.smqtt` before exiting.

By default, the KMS listens for incoming requests on the loopback interface. This is handy for testing but not useful when Clients and the KMS are not running on the same system. Use the `-a a.b.c.d` option to specify a network-accessible address to accept connections on.

The KMS can be run as a daemon process (in the background) with `kms -d`. In this case, output is redirected to files named `stdout` and `stderr` in `/home/kms`.

This is not an exhaustive description of KMS command line options. `kms -h` for more information.

The `kms_ctrl` command can be used by any user on the same system as the KMS to interrogate and/or command the KMS to do stuff. When the KMS starts up, it writes its process ID to `/home/kms/.kms.pid`. The KMS removes that file when performing an orderly shutdown. `kms_ctrl` reads `.kms.pid` to get the KMS process ID and send it a signal. `kms_ctrl -h` for usage information.

```
$ kms_ctrl -i — Get information on KMS process status (interrogate the process)
```

```
$ kms_ctrl -s — Tell the KMS to shut down (send a SIGHUP)
```

```
$ kms_ctrl -d — Tell the KMS to save its cache to disk (cache.smqtt, send a SIGUSR1)
```

```
$ kms_ctrl -c — Tell the KMS to re-read its configuration file (kms.cfg, send a SIGUSR2)
```

5.2.1 Enrollment

Enrollment refers to the activities needed to add a Client to the installation so that it is known to the KMS and is equipped to communicate with the KMS.

```
$ su - kms
```

```
kms$ generate_client_cert ClientID role
```

ClientID is the 16-character Secure MQTT-prescribed string the Client will use to identify itself. The `rand_id` command can be used to generate a random Client ID if desired, but it is generally more human-friendly to create a descriptive ID that includes a serial number or other data to ensure that it is unique among all Clients, *e.g.*, `DisplayPanel10001`.

role is one of `pub`, `sub`, or `both` to describe the type of Client. The KMS cares about this because certain KMS requests are valid only if received from the correct type of Client.

This creates a public/private key for the Client and as a side-effect adds the Client's information to `/home/kms/kms.cfg`. The public/private keys are placed in `/home/kms/ca` named as *ClientID*-`cert.pem` and *ClientID*-`key.pem`, respectively. These files—and `ca-cert.pem`—must be provided to the Client to enable it to communicate with the KMS.

5.2.2 Message passing

After a Client has been enrolled as described in Section 5.2.1 and has received the necessary cryptographic key files to communicate with the KMS (*ClientID*-crt.pem, *ClientID*-key.pem, and ca-crt.pem), it is ready to pass Secure MQTT messages.

The Client provides those files, plus the IP address and port of the KMS, when initializing its cryptographic state as described in Section 6 or Section 7. Assuming the Broker and KMS are both running, Clients can connect to the Broker and begin sending and receiving Secure MQTT messages.

6 Porting guide

If you have an existing MQTT installation based on Mosquitto, this section describes the steps necessary to adapt it to incorporate MasQiTT.

The code in `examples/` provides working examples of porting Mosquitto Clients to use MasQiTT. The code originally found in `examples/publish/basic-1.c` of the Mosquitto v2.0.15 distribution can be found in its ported update as `examples/masqitt_publisher.c`. Likewise, Mosquitto's `examples/subscribe/basic-1.c` has been ported as `examples/masqitt_subscriber.c`.

6.1 Set up


- Set up a Key Management Server (KMS) (Section 5.1) and know its host address and port.
- Ensure each Client is enrolled with the KMS and has its own TLS key/certificate files and the Certification Authority's certificate (Section 5.2.1).
- Include the MasQiTT library: `#include <masqitt.h>`.
- Replace instances of `struct mosquitto` with `struct masqitt`.
- Remove calls to `mosquitto_lib_init()` and `mosquitto_lib_clean()`. These are no longer necessary, as MasQiTT handles these on your behalf.
- Replace calls to `mosquitto_new()` with `MASQ_new()` to initialize a `struct masqitt`. A pointer to the initialized `struct masqitt` is returned in the same way that `mosquitto_new()` was used. Note that several additional parameters will be needed. See `masqitt.h` for details.
 - MasQiTT (MASQ) Client ID
 - Mosquitto (MQTT) Client ID
 - MQTT role (*i.e.*, publisher, subscriber, both)
 - If a Publisher, the Publisher key management strategy (Persistent or Ephemeral keys); Subscriber strategy is none.
 - If a Publisher, key management strategy value (threshold for generating a new Initial MEK if using Persistent strategy)
 - KMS host address (string in IPv4 dotted decimal notation, or NULL to use 127.0.0.1)
 - KMS host port (0 to use default MasQiTT port of 56788)
 - Name/path of the CA's certificate file (`ca-crt.pem`)
 - Name/path of the Client's certificate file (`ClientID-crt.pem`)
 - Name/path of the Client's private key file (`ClientID-key.pem`)

6.2 Usage

- If MasQiTT does not provide a function in its API but Mosquitto does and you would like to use it, there is a way to do so. The function `MASQ_get_mosquitto()` can be used to retrieve the internal struct `mosquitto *`. This returns a pointer to the internal struct `mosquitto`, NOT a copy, so *DO NOT* free this pointer. With this pointer, the normal Mosquitto API can be used. Do not use the Mosquitto API for functions that MasQiTT provides for publishing, otherwise you will not be using the secure methods for MQTT.

A good example of when to use `MASQ_get_mosquitto()` is for using the callback setters *EXCEPT* for the message callback setter. That is, if the function `mosquitto_connect_callback_set()` is desired, then use `MASQ_get_mosquitto()` to retrieve the struct `mosquitto *` and then call `mosquitto_connect_callback_set()` with the retrieved struct `mosquitto *` as a parameter.

- Replace calls to the message callback setter `mosquitto_message_callback_set()` with `MASQ_message_v5_callback_set()`.

 Since MasQiTT requires using MQTTv5, some functions such as the callback provided to `MASQ_message_v5_callback_set()` will require the additional `mosquitto_property *` parameter, or a compiler warning may occur. Also note that the body of the callback function does not require any changes unless there is interest in utilizing the MQTTv5 functionality of the Mosquitto properties.

- Replace uses of `mosquitto_publish()` with `MASQ_publish_v5()`. Note that the only difference in this API is that the struct `mosquitto *` parameter is replaced by a struct `masqitt *`. The other parameters are the same as a MQTTv5 publish function. If this is replacing `mosquitto_publish()` instead of `mosquitto_publish_v5()`, then an additional parameter to a `mosquitto_property *` is needed (but can be NULL if no properties need to be provided). This change is not because of a MasQiTT requirement, but an MQTTv5 protocol requirement, although MasQiTT does require MQTTv5.

The return value of `MASQ_publish_v5()` is a `MASQ_status_t` status enum instead of the `int` return code that Mosquitto uses. Success is indicated by the value `MASQ_STATUS_SUCCESS`. A status can be converted to a string with the helper function `MASQ_status_to_str()` if the header `api.h` is included.

- No special MasQiTT API function is needed for subscribing. The Mosquitto subscription process can be used unmodified.

6.3 Cleanup

- Be sure to call `MASQ_destroy()` to clean up all instantiations of struct `masqitt`.
- Although not necessary, it is good practice to set local variables retrieved from `MASQ_get_mosquitto()` to NULL.

6.4 Compilation

The compile and link step may vary based on the machine's environment variables and where shared libraries are located.

The first two steps may be necessary only during software development and testing before MasQiTT and Mosquitto have been installed on a system-wide basis.

- Compile with `-I` flags to indicate paths to headers that will be included.
 - `-I<path to masqitt.h>`
 - `-I<path to mosquitto.h>`
- Compile with `-L` flags to indicate paths to libraries that will be linked at runtime (pairs with `-l` flag).
 - `-L<path to libmasqitt.so>`
 - `-L<path to libmosquitto.so>`
- Compile with `-l` flags to indicate libraries that will be dynamically linked (pairs with `-L` flag).
 - `-lmasqitt`
 - `-lmosquitto`
 - `-lwolfssl`

6.5 Runtime

It may be necessary to also update the `LD_LIBRARY_PATH` environment variable as described in Section 4.2 to include paths to `libmasqitt.so` and `libmosquitto.so`, similar to the compile step above with the `-L` flag. Running `sudo make install` on both the Mosquitto and MasQiTT source will copy both libraries to `/usr/local/lib`.

7 Integration guide

MasQiTT is provided as an extension of Mosquitto, but MasQiTT's code can be integrated into other MQTT code bases. KMS code and the CA are unaffected.

Secure MQTT is concerned with the handling of PUBLISH packets, so only the code that creates, sends, receives, or parses PUBLISH packets needs to be updated for Secure MQTT. The files `include/masqitt.h` and `lib/masqitt.c` are needed only for Mosquitto integration; they may be safely replaced for integrating MasQiTT code into other MQTT code bases.

Integration should require calling only routines listed in `api.h`. How these routines are used is different between Publishers and Subscribers, so they are described individually. Processing for Clients who both send and receive PUBLISH packets is described in Section 7.3.

7.1 Publisher only

On startup, a Publisher must initialize the cryptographic library by calling `MASQ_crypto_api_init()` before handling PUBLISH packets. The parameters to this call are as follows.

- `protoid` — Protocol ID as described in [1] §B
- `role` — `MASQ_role_publisher`
- `clientid` — Secure MQTT Client ID
- `strategy` — Publisher key management strategy, one of:
 - `MASQ_key_ephemeral` — Use Ephemeral key management (new key every PUBLISH packet)
 - `MASQ_key_persistent_pkt` — Persistent key with new key after every X PUBLISH packets
 - `MASQ_key_persistent_bytes` — Persistent key with new key after every X bytes of Topic Values
 - `MASQ_key_persistent_time` — Persistent key with new key after every X seconds
 - `MASQ_key_persistent_exp` — Persistent key with new key after every key expiration time
- `strat_val` — If the key management strategy is `MASQ_key_ephemeral` or `MASQ_key_persistent_exp` this parameter is ignored, otherwise the threshold value (X) for generating a new key
- `kms_host` — KMS host address (string in IPv4 dotted decimal notation, or NULL to use 127.0.0.1)
- `kms_port` — KMS host port (0 to use default MasQiTT port of 56788)
- `ca_file` — Name/path of the CA's certificate file (`ca-cert.pem`)
- `cert_file` — Name/path of the Client's certificate file (`ClientID-crt.pem`)
- `key_file` — Name/path of the Client's private key file (`ClientID-key.pem`)

A successful return value (`MASQ_STATUS_SUCCESS`) indicates that the value returned in the `state` parameter is valid. This opaque pointer must be kept by the Client until after closing the cryptographic library.

When ready to send a PUBLISH packet, a Publisher must first call `MASQ_crypto_api_encrypt()` to encrypt the Payload. Parameters to this call are

- `state` — The pointer returned by `MASQ_crypto_api_init()`
- `topic_name` — The MQTT Topic Name ('\\0'-terminated string)
- `topic_value` — The MQTT Topic Value, a pointer to arbitrary data
- `topic_value_len` — Length in bytes of `topic_value`
- `user_properties` — On successful return contains User Property fields
- `outbuf` — Pointer to buffer to receive encrypted Topic Value and/or key management data
- `outbuf_len` — On input, specifies space available in `outbuf`, on successful return contains number of bytes for PUBLISH packet Payload

The number of bytes needed for `outbuf` can be determined by this bit of code.

```
MASQ_mek_strategy_t  strat;
size_t               overhead, mek;

MASQ_crypto_api_get_strategy(state, &strat);
MASQ_crypto_api_overhead(strat, &overhead, &mek);
```

`outbuf` should be at least as large as the larger of `mek` and $(\text{overhead} + \text{topic_value_len})$.

There are two successful return values, `MASQ_STATUS_SUCCESS` and `MASQ_STATUS_ANOTHER`. Any other return value indicates an error.

If the return status is `MASQ_STATUS_SUCCESS`, skip to Section 7.1.2.

7.1.1 Another

A successful return value of `MASQ_STATUS_ANOTHER` indicates that the packet contains key management data. The Publisher should

1. Copy the contents of `user_properties` into the outgoing PUBLISH packet as User Properties in the Variable Header
2. Copy `outbuf_len` bytes of data from `outbuf` into the outgoing PUBLISH packet as the Payload
3. Adjust packet length fields in the PUBLISH packet Fixed Header and Variable Header appropriately
4. Set the Retain bit in the PUBLISH packet Fixed Header to 1
5. Forward the PUBLISH packet to the MQTT Broker

The Publisher must then call `MASQ_crypto_api_encrypt()` again with the same parameters as before, making sure to first reset `outbuf_len` as needed. Processing continues in Section 7.1.2.

7.1.2 Success

A return value of `MASQ_STATUS_SUCCESS` indicates `outbuf` now contains the encrypted Topic Value. Here the Publisher should

1. Copy the contents of `user_properties` into the outgoing PUBLISH packet as User Properties in the Variable Header
2. Copy `outbuf_len` bytes of data from `outbuf` into the outgoing PUBLISH packet as the Payload
3. Adjust packet length fields in the PUBLISH packet Fixed Header and Variable Header appropriately
4. Ensure the Retain bit in the PUBLISH packet Fixed Header is cleared (set to 0)
5. Forward the PUBLISH packet to the MQTT Broker

When the Publisher is shutting down, it is good practice but not absolutely necessary to close the cryptographic library and free the memory used by the state data. This is done by calling `MASQ_crypto_api_close()`; its only argument is the state pointer.

7.2 Subscriber only

Initialization for a Subscriber is the same as for a Publisher with these exceptions when calling `MASQ_crypto_api_init()`

- `role` — `MASQ_role_subscriber`
- `strategy` — `MASQ_key_none`
- `strat_val` — 0 (ignored)

When a PUBLISH packet is received from a Broker, the Subscriber must first parse the incoming packet to obtain information that MasQiTT will need to decrypt the Topic Value, specifically the Topic Name, User Properties (found in the Variable Header), and the Payload contents and size.

A `MASQ_user_properties_t` must be initialized with the User Properties found in the incoming packet. Then the Subscriber can call `MASQ_crypto_api_decrypt()` with parameters that mirror those of `MASQ_crypto_api_encrypt()`.

- `state` — The pointer returned by `MASQ_crypto_api_init()`
- `topic_name` — The MQTT Topic Name ('\\0'-terminated string)
- `user_properties` — As described above

- `inbuf` — Pointer to the received Payload data
- `inbuf_len` — Length of the received Payload data
- `topic_value` — Pointer to buffer to receive decrypted Topic Value
- `topic_value_len` — On input, specifies space available in `topic_value`, on successful return contains number of bytes of decrypted data

`topic_value_len` will always be smaller than `inbuf_len`, so it is convenient (safe) to make sure `topic_value_len` is at least as large as `inbuf_len`.

If `MASQ_crypto_api_decrypt()` returns `MASQ_STATUS_KEY_MGMT`, that is an indication that the received packet was a Persistent MEK packet with a new key. In this case, the contents of `state` have been updated but there is no data to forward. No further processing is needed, and the incoming PUBLISH packet should be discarded.

If `MASQ_crypto_api_decrypt()` returns `MASQ_STATUS_SUCCESS`, then the Subscriber should

1. Remove the User Properties from the PUBLISH packet Variable Header
2. Copy `topic_value_len` bytes of data from `topic_value` into the incoming PUBLISH packet as the Payload
3. Adjust packet length fields in the PUBLISH packet Fixed Header and Variable Header appropriately
4. Forward the PUBLISH packet for Subscriber processing

`MASQ_crypto_api_close(state)` when done.

7.3 Both

Initialization for a Client acting as both a Publisher and a Subscriber is the same as for a Publisher with this exception when calling `MASQ_crypto_api_init()`

- `role` — `MASQ_role_both`

When processing outgoing PUBLISH packets, the Client follows the instructions in Section 7.1.

When processing incoming PUBLISH packets, the Client follows the instructions in Section 7.2.

`MASQ_crypto_api_close(state)` when done.

Intentionally blank

A Acronyms, abbreviations, and references

Abbrev	Meaning	Notes
AES	Advanced Encryption Standard	cryptography
API	Application programming interface	
BB ₁	Boneh-Boyer One	cryptography
CA	Certification Authority	cryptography
CLI	Command line interface	
GCM	Galois/Counter Mode	cryptography
GPIO	General-purpose input/output	
IBE	Identity-Based Encryption	cryptography
KMS	Key Management Server	MasQiTT
MEK	Message Encryption Key	cryptography
MQTT	originally an initialism of “MQ Telemetry Transport,” now used as a name	
PKI	Public Key Infrastructure	cryptography
RPi	Raspberry Pi microcomputer	
SHA256	Secure Hash Algorithm, 256 bits output	cryptography
SSL	Secure Socket Layer	cryptography
TLS	Transport Layer Security	cryptography
VM	Virtual Machine	

References

- [1] Sandia National Laboratories. Securing MQTT with Identity-Based Encryption: Secure MQTT v1.0 Protocol Specification. SAND Report, 2024.
- [2] Sandia National Laboratories. MasQiTT, Secure MQTT Reference Implementation. <https://github.com/sandialabs/masqitt>, 2024.
- [3] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. MQTT Version 5.0. MQTT 5.0, March 2019.
- [4] Eclipse Foundation. Eclipse Mosquitto. <https://mosquitto.org>, 2024.
- [5] Eclipse Foundation. Mosquitto API. <https://mosquitto.org/api>, 2024.
- [6] MIRACL Core Project. The MIRACL Core Cryptographic Library. <https://github.com/miracl/core>, 2024.
- [7] wolfSSL Inc. wolfSSL SSL/TLS Library. <https://wolfssl.com>, 2024.
- [8] OpenSSL Project. OpenSSL Cryptography and SSL/TLS Toolkit. <https://openssl.org>, 2024.
- [9] hyperrealm. libconfig Configuration File Library. <https://github.com/hyperrealm/libconfig>, 2021.

B Files

The files in the MasQiTt code base are listed and described here directory by directory. Not included are miscellaneous administrative files (*e.g.*, `Makefile`, `README.md`, `Doxyfile`) which support MasQiTt code but do not implement Secure MQTT.

- `lib/` — These are the files collected into `libmasqitt.so` (installed in `/usr/local/lib`). They provide the Secure MQTT functionality provided by MasQiTt.

These files contain code protected by “`#ifdef ebug/#endif`” blocks that provide additional reporting of internal processing steps and values to aid in development and debugging. This code can be activated by adding “`-Debug`” to `gcc` on compilation. `make test` (here in the `lib/` directory or the base MasQiTt directory) creates `libmasqittdb.so` with these code blocks included.

- `lib/masqitt.c` — Functions that extend Mosquitto to incorporate Secure MQTT as described in Section 6.
- `lib/api.c` — Functions in this file are called directly by code in `lib/masqitt.c` to encrypt and decrypt the content of MQTT PUBLISH packets. These calls are discussed in Section 7.
- `lib/crypto.c` — These functions provide all the core cryptographic processing to support Secure MQTT. As the glue between the API calls and the supporting cryptographic primitives, they are normally called by `lib/api.c` functions rather than by Clients directly, though these may be useful:
 - * `MASQ_rand_bytes()`
 - * `MASQ_hash()` (SHA256)
 - * `MASQ_hash_init()`
 - * `MASQ_hash_add()`
- `lib/ibe.c` — BB₁ Identity-Based Encryption support routines and interfaces to other useful MIRACL Core primitives, particularly pseudo-random number generation, SHA256, and AES/GCM.
- `lib/keys.c` — Routines used by MasQiTt to cache Client cryptographic keys for re-use. These are used both by Client and KMS code.
- `lib/kms_msg.c` — Functions to create and parse messages exchanged between Clients and the KMS.
- `lib/kms_utils.c` — Low level packet field creation and parsing routines to support the functions in `lib/kms_msg.c`.

- `include/` — These header files are installed in `/usr/include/masqitt` and used in MasQiTt Client and KMS code. Files marked with “*” are not normally directly `#included` by Client code but are indirectly included by other header files.

Except as otherwise noted, `include/foo.h` documents function calls provided by `lib/foo.c`.

- `include/masqitt.h`

- `include/masqlib.h*` — Global enums, sizes, and structs used throughout MasQiTT and #included by most other `.h` files.
 - `include/api.h*` — Useful for non-Mosquitto integration as described in Section 7.
 - `include/crypto.h*`
 - `include/tls.h*` — Glue for connecting `lib/crypto.c` routines to the wolfSSL library for TLS connections between Clients and the KMS.
 - `include/ibe.h*`
 - `include/keys.h*`
 - `include/kms_msg.h*`
 - `include/kms_utils.h*`
- `kms/` — This is the code that implements the KMS itself (first three items below) and supporting utilities. Installation and use of the KMS is described in Section 5.2. Unless otherwise noted, commands are installed in `/home/kms/bin` and available only to the `kms` userid.
 - `kms/kms_main.c` — The main set-up and message processing loop (receive a message, process the message, send a response). The KMS also schedules recurring housekeeping events by setting and receiving alarm events.
 - `kms/cache.c`, `kms/cache.h` — This code manages the KMS cache of IBE private keys previously provided to Subscribers so the keys do not have to be re-computed multiple times.
 - `kms/cfg.c`, `kms/cfg.h` — Glue code for reading information from the KMS configuration file.
 - `kms/kms.cfg` — A starter KMS configuration file with values that can be customized and a list of enrolled Clients. Installed at `/home/kms/kms.cfg`.
 - `kms/make_params.c` — `make_params` is used only for system provisioning (Section 5.1), this creates the IBE private parameters (s_1, s_2, s_3) and shared public parameters (R, T, V), then stores them in `/home/kms/params.smqtt`. Once set, these values should not be changed.
 - `kms/rand_id.c` — `rand_id` simply creates a random 16-character string that can be used for a Client ID.
 - `kms/kms_ctrl.c` — `kms_ctrl` can be used to determine whether or not the KMS is running and to tell the KMS to do things as described in Section 5.2. This command is available to non-`kms` userids on the same system as the KMS and is installed in `/usr/local/bin`.
 - `kms/print_cache.c` — The KMS periodically (and when instructed by `kms_ctrl`) writes its in-memory cache of IBE private keys to disk. `print_cache` prints information about the keys in the cache file (`cache.smqtt`), and optionally the IBE private and shared public parameters (`params.smqtt`).
 - `ca/` — These Python scripts support provisioning (Section 5.1) and enrollment (Section 5.2.1) activities. They are installed in `/home/kms/bin` and are only available to the `kms` userid. The key and certificate files created by these scripts are placed in `/home/kms/ca`.
 - `ca/generate_ca` — Used in provisioning to create the Certification Authority’s signing key and use that key to sign its own certificate. It also creates the KMS’s own key/certificate pair.

- `ca/generate_client_cert` — Used in enrollment to create a key/certificate pair for a Client with the requested Secure MQTT ClientID. This also enrolls the Client by adding an entry for the Client in `/home/kms/kms.cfg` with its role (Publisher, Subscriber, or both). The generated `ClientID-key.pem` and `ClientID-crt.pem` files, along with `ca-crt.pem`, must be given to the Client for its use. A Client's key/certificate files need not be retained by the KMS unless desired for backup purposes.
- `tests/` — This directory includes programs used during the development of MasQiTT to test various aspects of its Secure MQTT implementation. The test programs are compiled with `make test` in this directory or from the base MasQiTT directory. The latter is preferred as these test programs expect the debug version of the MasQiTT library (`libmasqittdb.so`), and `make test` from the base directory ensures it is available.

Each `foo.c` file compiles to a `foo` executable. Except for `msgtest`, these programs require a working and running KMS. They are listed in roughly ascending level of MasQiTT functionality needed for a successful test run.

- `tests/msgtest.c` — This checks that messages exchanged with the KMS can be properly created and parsed.
- `tests/kms-test.cfg` — This file should be copied to `/home/kms/kms.cfg` to support the test programs below as they communicate with the KMS.
- `tests/certs/` — This directory contains a set of TLS keys and associated certificates used by the test programs below as they communicate with the KMS. The files from this directory must be copied to `/home/kms/ca/`, taking care not to overwrite `.pem` files in operational use. After copying, `sudo chown kms:kms /home/kms/ca/*`
- `tests/kmstest.c` — This program tests the communications between a Client and the KMS, ensuring a viable TLS connection and the correct creation and parsing of KMS messages. No cryptographic processing is performed by `kmstest`.
- `tests/cryptotest.c` — This program exercises functions provided by `lib/crypto.c`.
- `tests/apitest.c` — This program exercises functions provided by `lib/api.c`.
- `tests/clienttest.c` — This is a comprehensive end-to-end test of MasQiTT code that exercises `lib/api.c` calls in a manner consistent with a Client's behavior, rather than exercising the individual calls as in `apitest`.
- `tests/masqitt_test.c` — This test exercises the MasQiTT-enhanced interface to Mosquitto and additionally requires an unmodified Mosquitto Broker to run.
- `examples/` — A working example of porting Mosquitto Clients to use MasQiTT.
 - `examples/masqitt_publisher.c` — A MasQiTT port of code originally found in `examples/publish/basic-1.c` of the Mosquitto distribution.
 - `examples/masqitt_subscriber.c` — A MasQiTT port of code originally found in `examples/subscribe/basic-1.c` of the Mosquitto distribution.