libkmip Documentation

Release 0.1.rel

Peter Hamilton

Jan 26, 2022

Contents

libkmip is an ISO C11 implementation of the Key Management Interoperability Protocol (KMIP), an OASIS communication standard for the management of objects stored and maintained by key management systems. KMIP defines how key management operations and operation data should be encoded and communicated between client and server applications. Supported operations include creating, retrieving, and destroying keys. Supported object types include:

- symmetric/asymmetric encryption keys

For more information on KMIP, check out the OASIS KMIP Technical Committee and the OASIS KMIP Documentation.

CHAPTER 1

Installation

You can install libkmip from source using `make`:

```
$ cd libkmip
$ make
$ make install
```

See *Installation* for more information.

CHAPTER 2

Layout

libkmip provides client functionality, allowing developers to integrate the key management lifecycle into their projects. For more information, check out the various articles below:

2.1 Installation

2.1.1 Dependencies

Building libkmip requires the following dependencies:

- OpenSSL 1.1.0

These may come installed by default on your target system or they may require separate installation procedures. See each individual dependency's documentation for more details.

2.1.2 Building libkmip on Linux

You can install libkmip from source via `git`:

```
$ git clone https://github.com/openkmip/libkmip.git
$ cd libkmip
$ make
$ make install
```

The default build settings will direct `make` to install libkmip under `/usr/local`, which may require `sudo` access. There are several different libkmip components that will be installed, including the documentation, the source code and header files, the shared library, and the example demo applications. The following list defines the default install directories and the files that can be found in them:

- **`/usr/local/bin/kmip`** Contains demo libkmip applications showing how to use the supported KMIP operations.
- **`/usr/local/include/kmip`** Contains the libkmip header files for use in third-party applications.

- **/usr/local/lib/kmip** Contains the libkmip shared library, `libkmip.so`.
- **/usr/local/src/kmip** Contains the libkmip source files.
- **/usr/local/share/doc/kmip/src** Contains the libkmip documentation source files.
- **/usr/local/share/doc/kmip/html** Contains the libkmip documentation HTML files *if they have have already been built*.

You can override the build defaults when invoking `make install`. The following list defines the build variables used by `make` and what their default values are:

- **PREFIX** Defines where libkmip will be installed. Defaults to `/usr/local`.
- **KMIP** Defines the common name of the libkmip subdirectories that will be created under `PREFIX`. Defaults to `kmip`.
- **DESTDIR** Defines an alternative root of the file system where libkmip will be installed. Used primarily to test the installation process without needing to modify the default values of `PREFIX` or `KMIP`. Defaults to the empty string.

For example, to install libkmip under your home directory, you could use the following command:

```
$ make PREFIX=$HOME/.local install
```

This would create all of the normal installation directories (e.g., `bin`, `include`, `lib`) under `$HOME/.local` instead of `/usr/local`.

To ensure that your system is up-to-date after you install libkmip, make sure to run `ldconfig` to update the dynamic linker's run-time bindings.

```
$ ldconfig
```

For more information see the project Makefile (insert link here).

2.1.3 Uninstalling libkmip

You can uninstall libkmip using the provided `make uninstall` target:

```
$ cd libkmip
$ make uninstall
```

This will simply remove all of the installation directories and files created during the above installation process. Like with `make install`, the default build settings will direct `make` to remove libkmip from under `/usr/local`, which may require `sudo` access. If you customize the installation settings, be sure to use those same settings when uninstalling.

Like the installation process, run `ldconfig` again after uninstall to make the dynamic linker is up-to-date.

2.2 Changelog

2.2.1 0.2 - July 12, 2019

- Add the BSD 3-clause license to the library
- Add KMIP 2.0 attributes
- Add deep copy utilities for all attribute types

- Upgrade Create support to enable KMIP 2.0 encodings
- Upgrade the unit test suite to use intelligent test tracking
- Upgrade the linked list to support enqueue and double linkage
- Fix an implicit infinite loop in the test suite application
- Fix a usage issue when passing no args to the demo applications
- Fix Travis CI config to redirect OpenSSL install logs to a file

2.2.2 0.1 - November 15, 2018

- Initial release
- Add encoding/decoding support for Symmetric/Public/Private Keys
- Add encoding/decoding support for Create/Get/Destroy operations
- Add KMIP 1.0 - 1.4 support for all supported KMIP structures
- Add an OpenSSL BIO client to securely connect to KMIP servers
- Add demo applications that show how to use the client API
- Add a unit test suite that covers the encoding/decoding library
- Add library documentation built and managed by Sphinx
- Add a Makefile that can build/install static/shared libraries

2.3 Frequently Asked Questions

Table of Contents

2.3.1 There are no results when I run `man libkmip`. Why not?

The current build and install process does not generate `man` compatible documentation output.

The libkmip documentation is written in RST and is built as HTML by `sphinx`. It is typically installed under `/usr/local/share/doc/kmip`. It is available online on ReadTheDocs.

For more information, see *Writing Documentation* and *Building libkmip on Linux*.

2.4 Development

Development for libkmip is open to all contributors. Use the information provided here to inform your contributions and help the project maintainers review and accept your work.

2.4.1 Getting Started

File a new issue on the project issue tracker on GitHub describing the work you intend on doing. This is especially recommended for any sizable contributions, like adding support for a new KMIP operation or object type. Provide as much information on your feature request as possible, using information from the KMIP specifications or existing feature support in libkmip where applicable.

The issue number for your new issue should be included at the end of the commit message of each patch related to that issue.

If you simply want to request a new feature but do not intend on working on it, file your issue as normal and the project maintainers will triage it for future work.

2.4.2 Writing Code

New code should be written in its own `git` branch, ideally branched from `HEAD` on `master`. If other commits are merged into `master` after your branch was created, be sure to rebase your work on the current state of `master` before submitting a pull request to GitHub.

New code should generally follow the style used in the surrounding libkmip codebase.

2.4.3 Writing Documentation

Like new code, new documentation should be written in its own `git` branch. All libkmip documentation is written in RST format and managed using `sphinx`. It can be found under `docs/source`.

If you are interested in contributing to the project documentation, install the project documentation requirements:

```
$ pip install -r doc-requirements.txt
```

To build the documentation, navigate into the `docs` directory and run:

```
$ make html
```

This will build the libkmip documentation as HTML and place it under the new `docs/build/html` directory. View it using your preferred web browser.

2.4.4 Commit Messages

Commit messages should include a single line title (75 characters max) followed by a blank line and a description of the change, including feature details, testing and documentation updates, feature limitations, known issues, etc.

The issue number for the issue associated with the commit should be included at the end of the commit message, if it exists. If the commit is the final one for a specific issue, use `Closes #XXX` or `Fixes #XXX` to link the issue and close it simultaneously.

2.4.5 Bug Fixes

If you have found a bug in libkmip, file a new issue and use the title format `Bug: <brief description here>`. In the body of the issue please provide as much information as you can, including platform, compiler version, dependency version, and any stacktraces or error information produced by libkmip related to the bug. See What to put in your bug report for a breakdown of bug reporting best practices.

If you are working on a bug fix for a bug in `master`, follow the general guidelines above for branching and code development (see *Writing Code*).

If you are working on a bug fix for an older version of libkmip, your branch should be based on the latest commit of the repository branch for the version of libkmip the bug applies to (e.g., branch `release-0.1.0` for libkmip 0.1). The pull request for your bug fix should also target the version branch in question. If appliable, it will be pulled forward to newer versions of libkmip, up to and including `master`.

2.4.6 Running Tests

libkmip comes with its own testing application that primarily covers the encoding/decoding functionality of the library. It is built with the default `make` target and can be run locally by invoking the `tests` binary:

```
$ cd libkmip
$ make
$ ./tests
```

2.5 Security

The security of libkmip is the top priority for the project. Use the information provided below to inform your security posture.

2.5.1 Handling Sensitive Data

Given that libkmip is an ISO C11 implementation of a key management protocol, the most sensitive aspect of the library is its handling of memory containing cryptographic material. All memory allocation and deallocation routines explicitly zero memory to prevent inadvertent leaks of sensitive data. This approach relies on the use of the standard `memset_s` function (see memset_s) included in C11 Annex K. If `memset_s` is unavailable at build time, memory clearing is done through a volatile function pointer to prevent the optimizer from optimizing away the clearing operation.

Warning: Despite the precautions taken here, it is possible that your build system will still optimize away the memory clearing operation. If this occurs, sensitive cryptographic material will be left behind in memory during and after application execution. Examine your application binary directly to determine if this is true for your setup.

Other security concerns, such as locking memory pages, are left up to the parent application and are not the domain of libkmip.

2.5.2 Reporting a Security Issue

Please do not report security issues to the normal GitHub project issue tracker. Contact the project maintainers directly via email to report and discuss security issues.

When reporting a security issue, please include as much detail as possible. This includes a high-level description of the issue, information on how to cause or reproduce the issue, and any details on specific portions of the project code base related to the issue.

Once you have submitted an issue, you should receive an acknowledgement. Depending upon the severity of the issue, the project maintainers will respond to collect additional information and work with you to address the security issue.

If applicable, a new library subrelease will be produced across all actively supported releases to address and fix the issue.

2.6 API

libkmip is composed of several components:

- an encoding/decoding library
- a client library
- a utilities library

The encoding library transforms KMIP message structures to and from the KMIP binary TTLV encoding format. The client library uses the OpenSSL BIO library to create secure connections with a KMIP server, sending and receiving TTLV-encoded messages. Finally, the utilities library is used to create and manage the library context and its associated structures which are used by the client library. Together, these components can be used to conduct secure key management operations.

2.6.1 Client API

The libkmip Client API supports varying levels of granularity, allowing parent applications access to everything from the low-level encoded message buffer up to high-level KMIP operation functions that handle all of the message building and encoding details automatically.

The following function signatures define the client API and can be found in `kmip_bio.h`:

```
/* High-level API */
int kmip_bio_create_symmetric_key(BIO *, TemplateAttribute *, char **, int *);
int kmip_bio_get_symmetric_key(BIO *, char *, int, char **, int *);
int kmip_bio_destroy_symmetric_key(BIO *, char *, int);

/* Mid-level API */
int kmip_bio_create_symmetric_key_with_context(KMIP *, BIO *, TemplateAttribute *,␣
↪char **, int *);
int kmip_bio_get_symmetric_key_with_context(KMIP *, BIO *, char *, int, char **, int␣
↪*);
int kmip_bio_destroy_symmetric_key_with_context(KMIP *, BIO *, char *, size_t);

/* Low-level API */
int kmip_bio_send_request_encoding(KMIP *, BIO *, char *, int, char **, int *);
```

High-level API

The high-level client API contains KMIP operation functions that simply require the inputs for a specific KMIP operation. Using these functions, the library will automatically:

- create the libkmip library context (see *The libkmip Context*)
- create the request message structure
- encode the request message structure into a request encoding
- send the request encoding to the BIO-connected KMIP server
- receive the response encoding back from the BIO-connected KMIP server

- decode the response encoding into the response message structure
- extract the relevant output from the response message structure
- clean up the library context and the encoding buffers
- handle any errors that occur throughout the request/response process

Because the library context and encoding processes are handled internally, the parent application has no access to additional debugging or error information when the KMIP operation fails. There is also no way to control or manage the dynamic memory allocation process required for the encoding buffers and the decoding process. If this information and/or capability is needed by the parent application, consider switching to use the *Mid-level API* or *Low-level API* which provide these capabilities.

The function header details for each of the high-level API functions are provided below.

int **kmip_bio_create_symmetric_key**(BIO *, TemplateAttribute *, char **, int *)

Create a symmetric key with the attributes provided in the `TemplateAttribute` structure.

Parameters

- **BIO*** – An OpenSSL `BIO` structure containing a connection to the KMIP server that will create the symmetric key.
- **TemplateAttribute*** – A libkmip `TemplateAttribute` structure containing the attributes for the symmetric key (e.g., cryptographic algorithm, cryptographic length).
- **char**** – A double pointer that can be used to access the UUID of the newly created symmetric key.

  Note: This pointer will point to a newly allocated block of memory. The parent application is responsible for clearing and freeing this memory once it is done using the UUID.

- **int*** – A pointer that can be used to access the length of the UUID string pointed to by the above double pointer.

Returns

A status code indicating success or failure of the operation. A negative status code indicates a libkmip error occurred while processing the request. A positive status code indicates a KMIP error occurred while the KMIP server processed the request. A status code of 0 indicates the operation succeeded.

The following codes are returned explicitly by this function. If the code returned is negative and is not listed here, it is the result of the request encoding or response decoding process. See *Status Codes* for all possible status code values.

- **KMIP_ARG_INVALID** One or more of the function arguments are invalid or unset and no work can be done. This failure can occur if any of the following are true:
  - the OpenSSL `BIO` pointer is set to `NULL`
  - the `TemplateAttribute` pointer is set to `NULL`
  - the `char **` UUID double pointer is set to `NULL`
  - the `int *` UUID size pointer is set to `NULL`
- **KMIP_MEMORY_ALLOC_FAILED** Memory allocation failed during the key creation call. This failure can occur during any of the following steps:
  - creation/resizing of the encoding buffer

  - creation of the decoding buffer

- **KMIP_IO_FAILURE** A `BIO` error occurred during the key creation call. This failure can occur during any of the following steps:
  - sending the encoded request message to the KMIP server
  - receiving the encoded response message from the KMIP server
- **KMIP_EXCEED_MAX_MESSAGE_SIZE** The received response message from the KMIP server exceeds the maximum allowed message size defined in the default libkmip library context. Switching to the *Mid-level API* will allow the parent application to set the max message size in the library context directly.
- **KMIP_MALFORMED_RESPONSE** The received response message from the KMIP server is malformed and does not contain valid operation result information.

int **kmip_bio_get_symmetric_key**(BIO *, char *, int, char **, int *)

Retrieve a symmetric key identified by a specific UUID.

Parameters

- **BIO*** – An OpenSSL `BIO` structure containing a connection to the KMIP server that stores the symmetric key.
- **char*** – A string containing the UUID of the symmetric key to retrieve.
- **int** – The length of the above UUID string.
- **char**** – A double pointer that can be used to access the bytes of the retrieved symmetric key.

  Note: This pointer will point to a newly allocated block of memory. The parent application is responsible for clearing and freeing this memory once it is done using the symmetric key.

- **int*** – A pointer that can be used to access the length of the symmetric key pointed to by the above double pointer.

Returns

A status code indicating success or failure of the operation. A negative status code indicates a libkmip error occurred while processing the request. A positive status code indicates a KMIP error occurred while the KMIP server processed the request. A status code of 0 indicates the operation succeeded.

The following codes are returned explicitly by this function. If the code returned is negative and is not listed here, it is the result of the request encoding or response decoding process. See *Status Codes* for all possible status code values.

- **KMIP_ARG_INVALID** One or more of the function arguments are invalid or unset and no work can be done. This failure can occur if any of the following are true:
  - the OpenSSL `BIO` pointer is set to `NULL`
  - the `char *` UUID pointer is set to `NULL`
  - the `int` UUID size argument is set to a non-positive integer
  - the `char **` bytes double pointer is set to `NULL`
  - the `int *` bytes size pointer is set to `NULL`

- **KMIP_MEMORY_ALLOC_FAILED** Memory allocation failed during the key retrieval call. This failure can occur during any of the following steps:
  - creation/resizing of the encoding buffer
  - creation of the decoding buffer
- **KMIP_IO_FAILURE** A `BIO` error occurred during the key retrieval call. This failure can occur during any of the following steps:
  - sending the encoded request message to the KMIP server
  - receiving the encoded response message from the KMIP server
- **KMIP_EXCEED_MAX_MESSAGE_SIZE** The received response message from the KMIP server exceeds the maximum allowed message size defined in the default libkmip library context. Switching to the *Mid-level API* will allow the parent application to set the max message size in the library context directly.
- **KMIP_MALFORMED_RESPONSE** The received response message from the KMIP server is malformed and does not contain valid operation result information.

int **kmip_bio_destroy_symmetric_key**(BIO *, char *, int)

Destroy a symmetric key identified by a specific UUID.

Parameters

- **BIO*** – An OpenSSL `BIO` structure containing a connection to the KMIP server that stores the symmetric key.
- **char*** – A string containing the UUID of the symmetric key to destroy.
- **int** – The length of the above UUID string.

Returns

A status code indicating success or failure of the operation. A negative status code indicates a libkmip error occurred while processing the request. A positive status code indicates a KMIP error occurred while the KMIP server processed the request. A status code of 0 indicates the operation succeeded.

The following codes are returned explicitly by this function. If the code returned is negative and is not listed here, it is the result of the request encoding or response decoding process. See *Status Codes* for all possible status code values.

- **KMIP_ARG_INVALID** One or more of the function arguments are invalid or unset and no work can be done. This failure can occur if any of the following are true:
  - the OpenSSL `BIO` pointer is set to `NULL`
  - the `char *` UUID pointer is set to `NULL`
  - the `int` UUID size argument is set to a non-positive integer
- **KMIP_MEMORY_ALLOC_FAILED** Memory allocation failed during the key destruction call. This failure can occur during any of the following steps:
  - creation/resizing of the encoding buffer
  - creation of the decoding buffer
- **KMIP_IO_FAILURE** A `BIO` error occurred during the key destruction call. This failure can occur during any of the following steps:
  - sending the encoded request message to the KMIP server

  - receiving the encoded response message from the KMIP server

- **KMIP_EXCEED_MAX_MESSAGE_SIZE** The received response message from the KMIP server exceeds the maximum allowed message size defined in the default libkmip library context. Switching to the *Mid-level API* will allow the parent application to set the max message size in the library context directly.
- **KMIP_MALFORMED_RESPONSE** The received response message from the KMIP server is malformed and does not contain valid operation result information.

Mid-level API

The mid-level client API is similar to the high-level API except that it allows the parent application to create and supply the library context to each KMIP operation function. This allows the parent application to set the KMIP message settings relevant to its own use case, including the KMIP version to use for message encoding, the maximum message size to accept from the KMIP server, and the list of credentials to use when sending a KMIP request message. The application can also substitute its own memory management system using the standard memory function hooks provided in the context.

Should an error occur during the request encoding or response decoding process, error information, including an error message and a stack trace detailing the function call path triggering the error, can be obtained from the library context. For more information on the context, see *The libkmip Context*.

Using these functions, the library will automatically:

- create the request message structure
- encode the request message structure into a request encoding
- send the request encoding to the BIO-connected KMIP server
- receive the response encoding back from the BIO-connected KMIP server
- decode the response encoding into the response message structure
- extract the relevant output from the response message structure
- clean up the encoding buffers
- handle any errors that occur throughout the request/response process

The function header details for each of the mid-level API functions are provided below.

int **kmip_bio_create_symmetric_key_with_context**(KMIP *, BIO *, TemplateAttribute *, char **, int *)

Create a symmetric key with the attributes provided in the `TemplateAttribute` structure.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures.

  Note: This structure should be properly destroyed by the parent application once it is done conducting KMIP operations. See *The libkmip Context* and *Utility Functions* for more information.

- **BIO*** – An OpenSSL `BIO` structure containing a connection to the KMIP server that will create the symmetric key.
- **TemplateAttribute*** – A libkmip `TemplateAttribute` structure containing the attributes for the symmetric key (e.g., cryptographic algorithm, cryptographic length).

- **char**** – A double pointer that can be used to access the UUID of the newly created symmetric key.

  Note: This pointer will point to a newly allocated block of memory. The parent application is responsible for clearing and freeing this memory once it is done using the UUID.

- **int*** – A pointer that can be used to access the length of the UUID string pointed to by the above double pointer.

Returns

A status code indicating success or failure of the operation. A negative status code indicates a libkmip error occurred while processing the request. A positive status code indicates a KMIP error occurred while the KMIP server processed the request. A status code of 0 indicates the operation succeeded.

The following codes are returned explicitly by this function. If the code returned is negative and is not listed here, it is the result of the request encoding or response decoding process. See *Status Codes* for all possible status code values.

- **KMIP_ARG_INVALID** One or more of the function arguments are invalid or unset and no work can be done. This failure can occur if any of the following are true:
  - the libkmip `KMIP` pointer is set to `NULL`
  - the OpenSSL `BIO` pointer is set to `NULL`
  - the `TemplateAttribute` pointer is set to `NULL`
  - the `char **` UUID double pointer is set to `NULL`
  - the `int *` UUID size pointer is set to `NULL`
- **KMIP_MEMORY_ALLOC_FAILED** Memory allocation failed during the key creation call. This failure can occur during any of the following steps:
  - creation/resizing of the encoding buffer
  - creation of the decoding buffer
- **KMIP_IO_FAILURE** A `BIO` error occurred during the key creation call. This failure can occur during any of the following steps:
  - sending the encoded request message to the KMIP server
  - receiving the encoded response message from the KMIP server
- **KMIP_EXCEED_MAX_MESSAGE_SIZE** The received response message from the KMIP server exceeds the maximum allowed message size defined in the provided libkmip library context.
- **KMIP_MALFORMED_RESPONSE** The received response message from the KMIP server is malformed and does not contain valid operation result information.

int **kmip_bio_get_symmetric_key_with_context**(KMIP *, BIO *, char *, int, char **, int *)

Retrieve a symmetric key identified by a specific UUID.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures.

Note: This structure should be properly destroyed by the parent application once it is done conducting KMIP operations. See *The libkmip Context* and *Utility Functions* for more information.

- **BIO*** – An OpenSSL `BIO` structure containing a connection to the KMIP server that stores the symmetric key.
- **char*** – A string containing the UUID of the symmetric key to retrieve.
- **int** – The length of the above UUID string.
- **char**** – A double pointer that can be used to access the bytes of the retrieved symmetric key.

Note: This pointer will point to a newly allocated block of memory. The parent application is responsible for clearing and freeing this memory once it is done using the symmetric key.

- **int*** – A pointer that can be used to access the length of the symmetric key pointed to by the above double pointer.

Returns

A status code indicating success or failure of the operation. A negative status code indicates a libkmip error occurred while processing the request. A positive status code indicates a KMIP error occurred while the KMIP server processed the request. A status code of 0 indicates the operation succeeded.

The following codes are returned explicitly by this function. If the code returned is negative and is not listed here, it is the result of the request encoding or response decoding process. See *Status Codes* for all possible status code values.

- **KMIP_ARG_INVALID** One or more of the function arguments are invalid or unset and no work can be done. This failure can occur if any of the following are true:
  - the libkmip `KMIP` pointer is set to `NULL`
  - the OpenSSL `BIO` pointer is set to `NULL`
  - the `char *` UUID pointer is set to `NULL`
  - the `int` UUID size argument is set to a non-positive integer
  - the `char **` bytes double pointer is set to `NULL`
  - the `int *` bytes size pointer is set to `NULL`
- **KMIP_MEMORY_ALLOC_FAILED** Memory allocation failed during the key retrieval call. This failure can occur during any of the following steps:
  - creation/resizing of the encoding buffer
  - creation of the decoding buffer
- **KMIP_IO_FAILURE** A `BIO` error occurred during the key retrieval call. This failure can occur during any of the following steps:
  - sending the encoded request message to the KMIP server
  - receiving the encoded response message from the KMIP server

- **KMIP_EXCEED_MAX_MESSAGE_SIZE** The received response message from the KMIP server exceeds the maximum allowed message size defined in the provided libkmip library context.
- **KMIP_MALFORMED_RESPONSE** The received response message from the KMIP server is malformed and does not contain valid operation result information.

int **kmip_bio_destroy_symmetric_key_with_context**(KMIP *, BIO *, char *, int)

Destroy a symmetric key identified by a specific UUID.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures.

  Note: This structure should be properly destroyed by the parent application once it is done conducting KMIP operations. See *The libkmip Context* and *Utility Functions* for more information.

- **BIO*** – An OpenSSL `BIO` structure containing a connection to the KMIP server that stores the KMIP managed object.
- **char*** – A string containing the UUID of the KMIP managed object to destroy.
- **int** – The length of the above UUID string.

Returns

A status code indicating success or failure of the operation. A negative status code indicates a libkmip error occurred while processing the request. A positive status code indicates a KMIP error occurred while the KMIP server processed the request. A status code of 0 indicates the operation succeeded.

The following codes are returned explicitly by this function. If the code returned is negative and is not listed here, it is the result of the request encoding or response decoding process. See *Status Codes* for all possible status code values.

- **KMIP_ARG_INVALID** One or more of the function arguments are invalid or unset and no work can be done. This failure can occur if any of the following are true:
  - the libkmip `KMIP` pointer is set to `NULL`
  - the OpenSSL `BIO` pointer is set to `NULL`
  - the `char *` UUID pointer is set to `NULL`
  - the `int` UUID size argument is set to a non-positive integer
- **KMIP_MEMORY_ALLOC_FAILED** Memory allocation failed during the key destruction call. This failure can occur during any of the following steps:
  - creation/resizing of the encoding buffer
  - creation of the decoding buffer
- **KMIP_IO_FAILURE** A `BIO` error occurred during the key destruction call. This failure can occur during any of the following steps:
  - sending the encoded request message to the KMIP server
  - receiving the encoded response message from the KMIP server

- **KMIP_EXCEED_MAX_MESSAGE_SIZE** The received response message from the KMIP server exceeds the maximum allowed message size defined in the provided libkmip library context.
- **KMIP_MALFORMED_RESPONSE** The received response message from the KMIP server is malformed and does not contain valid operation result information.

Low-level API

The low-level client API differs from the mid and high-level APIs. It provides a single function that is used to send and receive encoded KMIP messages. The request message structure construction and encoding, along with the response message structure decoding, is left up to the parent application. This provides the parent application complete control over KMIP message processing.

Using this function, the library will automatically:

- send the request encoding to the BIO-connected KMIP server
- receive the response encoding back from the BIO-connected KMIP server
- handle any errors that occur throughout the send/receive process

The function header details for the low-level API function is provided below.

int **kmip_bio_send_request_encoding**(KMIP *, BIO *, char *, int, char **, int *)
Send a KMIP encoded request message to the KMIP server.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures. Primarily used here to control the maximum response message size.

  Note: This structure should be properly destroyed by the parent application once it is done conducting KMIP operations. See *The libkmip Context* and *Utility Functions* for more information.

- **BIO*** – An OpenSSL `BIO` structure containing a connection to the KMIP server.
- **char*** – A string containing the KMIP encoded request message bytes.
- **int** – The length of the above encoded request message.
- **char**** – A double pointer that can be used to access the bytes of the received KMIP encoded response message.

  Note: This pointer will point to a newly allocated block of memory. The parent application is responsible for clearing and freeing this memory once it is done processing the encoded response message.

- **int*** – A pointer that can be used to access the length of the encoded response message pointed to by the above double pointer.

Returns

A status code indicating success or failure of the operation. A negative status code indicates a libkmip error occurred while processing the request. A positive status code indicates a KMIP

error occurred while the KMIP server processed the operation. A status code of 0 indicates the operation succeeded. The following codes are returned explicitly by this function.

- **KMIP_ARG_INVALID** One or more of the function arguments are invalid or unset and no work can be done. This failure can occur if any of the following are true:
  - the libkmip `KMIP` pointer is set to `NULL`
  - the OpenSSL `BIO` pointer is set to `NULL`
  - the `char *` encoded request message bytes pointer is set to `NULL`
  - the `int` encoded request message bytes size argument is set to a non-positive integer
  - the `char **` encoded response message bytes double pointer is set to `NULL`
  - the `int *` encoded response message bytes size pointer is set to `NULL`
- **KMIP_MEMORY_ALLOC_FAILED** Memory allocation failed during message handling. This failure can occur during the following step:
  - creation of the decoding buffer
- **KMIP_IO_FAILURE** A `BIO` error occurred during message handling. This failure can occur during any of the following steps:
  - sending the encoded request message to the KMIP server
  - receiving the encoded response message from the KMIP server
- **KMIP_EXCEED_MAX_MESSAGE_SIZE** The received response message from the KMIP server exceeds the maximum allowed message size defined in the provided libkmip library context.

Status Codes

The following tables list the status codes that can be returned by the client API functions. The first table lists the status codes related to the functioning of libkmip.

Status Code	Value
KMIP_OK	0
KMIP_NOT_IMPLEMENTED	-1
KMIP_ERROR_BUFFER_FULL	-2
KMIP_ERROR_ATTR_UNSUPPORTED	-3
KMIP_TAG_MISMATCH	-4
KMIP_TYPE_MISMATCH	-5
KMIP_LENGTH_MISMATCH	-6
KMIP_PADDING_MISMATCH	-7
KMIP_BOOLEAN_MISMATCH	-8
KMIP_ENUM_MISMATCH	-9
KMIP_ENUM_UNSUPPORTED	-10
KMIP_INVALID_FOR_VERSION	-11
KMIP_MEMORY_ALLOC_FAILED	-12
KMIP_IO_FAILURE	-13
KMIP_EXCEED_MAX_MESSAGE_SIZE	-14
KMIP_MALFORMED_RESPONSE	-15
KMIP_OBJECT_MISMATCH	-16

The second table lists the operation result status codes that can be returned by a KMIP server as the result of a successful or unsuccessful operation.

Status Code	Value
KMIP_STATUS_SUCCESS	0
KMIP_STATUS_OPERATION_FAILED	1
KMIP_STATUS_OPERATION_PENDING	2
KMIP_STATUS_OPERATION_UNDONE	3

2.6.2 Encoding API

The libkmip Encoding API supports encoding and decoding a variety of message structures and substructures to and from the KMIP TTLV encoding format. The *Client API* functions use the resulting encoded messages to communicate KMIP operation instructions to the KMIP server. While each substructure contained in a request or response message structure has its own corresponding set of encoding and decoding functions, parent applications using libkmip should only need to use the encoding and decoding functions for request and response messages respectively.

The following function signatures define the encoding API and can be found in `kmip.h`:

```
int kmip_encode_request_message(KMIP *, const RequestMessage *);
int kmip_decode_response_message(KMIP *, ResponseMessage *);
```

The function header details for each of the encoding API functions are provided below.

int **kmip_encode_request_message**(KMIP *, const RequestMessage *)
Encode the request message and store the encoding in the library context.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures.
- **RequestMessage*** – A libkmip `RequestMessage` structure containing the request message information that will be encoded. The structure will not be modified during the encoding process.

Returns A status code indicating success or failure of the encoding process. See *Status Codes* for all possible status code values. If `KMIP_OK` is returned, the encoding succeeded.

int **kmip_decode_response_message**(KMIP *, ResponseMessage *)
Decode the encoding in the library context into the response message.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures.
- **ResponseMessage*** – A libkmip `ResponseMessage` structure that will be filled out by the decoding process.

Note: This structure will contain pointers to newly allocated substructures created during the decoding process. The calling function is responsible for clearing and freeing these substructures once it is done processing the response message. See (ref here) for more information.

> **Warning:** Any attributes set in the structure before it is passed in to this decoding function will be overwritten and lost during the decoding process. Best practice is to pass in a pointer to a freshly initialized, empty structure to ensure this does not cause application errors.

Returns A status code indicating success or failure of the decoding process. See *Status Codes* for all possible status code values. If `KMIP_OK` is returned, the decoding succeeded.

2.6.3 Utilities API

The libkmip Utilities API supports a wide variety of helper functions and structures that are used throughout libkmip, ranging from the core library context structure that is used for all encoding and decoding operations to structure initializers, deallocators, and debugging aides.

> **Warning:** Additional capabilities are included in libkmip that may not be discussed here. These capabilities are generally for internal library use only and are subject to change in any release. Parent applications that use these undocumented features should not expect API stability.

The libkmip Context

The libkmip library context is a structure that contains all of the settings and controls needed to create KMIP message encodings. It is defined in `kmip.h`:

```
typedef struct kmip
{
    /* Encoding buffer */
    uint8 *buffer;
    uint8 *index;
    size_t size;

    /* KMIP message settings */
    enum kmip_version version;
    int max_message_size;
    LinkedList *credentials;

    /* Error handling information */
    char *error_message;
    size_t error_message_size;
    LinkedList *error_frames;

    /* Memory management function pointers */
    void *(*calloc_func)(void *state, size_t num, size_t size);
    void *(*realloc_func)(void *state, void *ptr, size_t size);
    void  (*free_func)(void *state, void *ptr);
    void *(*memset_func)(void *ptr, int value, size_t size);
    void *state;
} KMIP;
```

The structure includes the encoding/decoding buffer, KMIP message settings, error information, and memory management hooks.

The Encoding/Decoding Buffer

The library context contains a pointer to the main target buffer, `buffer`, used for both encoding and decoding KMIP messages. This buffer should only be set and accessed using the defined context utility functions defined below. It should never be accessed or manipulated directly.

KMIP Message Settings

The library context contains several attributes that are used throughout the encoding and decoding process.

The `version` enum attribute is used to control what KMIP structures are included in operation request and response messages. It should be set by the parent application to the desired KMIP version:

```
enum kmip_version
{
    KMIP_1_0 = 0,
    KMIP_1_1 = 1,
    KMIP_1_2 = 2,
    KMIP_1_3 = 3,
    KMIP_1_4 = 4
};
```

The `max_message_size` attribute defines the maximum size allowed for incoming response messages. Since KMIP message encodings define the total size of the message at the beginning of the encoding, it is important for the parent application to set this attribute to a reasonable default suitable for its operation.

The `credentials` list is intended to store a set of authentication credentials that should be included in any request message created with the library context. This is primarily intended for use with the *Mid-level API*.

Each of these attributes will be set to reasonable defaults by the `kmip_init` context utility and can be overridden as needed.

Error Information

The library context contains several attributes that are used to track and store error information. These are only used when errors occur during the encoding or decoding process. Once an error is detected, a libkmip stack trace will be constructed, with each frame in the stack containing the function name and source line number where the error occurred to facilitate debugging.

```
typedef struct error_frame
{
    char *function;
    int line;
} ErrorFrame;
```

The original error message will be captured in the `error_message` attribute for use in logging or user-facing status messages.

See the context functions below for using and accessing this error information.

Memory Management

The library context contains several function pointers that can be used to wrap or substitute common memory management utilities. All memory management done by libkmip is done through these function pointers, allowing the calling

application to easily substitute its own memory management system. Note specifically the `void *state` attribute in the library context; it is intended to contain a reference to the parent application's custom memory management system, if one exists. This attribute is passed to every call made through the context's memory management hooks, allowing the parent application complete control of the memory allocation process. By default, the `state` attribute is ignored in the default memory management hooks. The `kmip_init` utility function will automatically set these hooks to the default memory management functions if any of them are unset.

Utility Functions

The following function signatures define the Utilities API and can be found in `kmip.h`:

```
/* Library context utilities */
void kmip_clear_errors(KMIP *);
void kmip_init(KMIP *, void *, size_t, enum kmip_version);
void kmip_init_error_message(KMIP *);
int  kmip_add_credential(KMIP *, Credential *);
void kmip_remove_credentials(KMIP *);
void kmip_reset(KMIP *);
void kmip_rewind(KMIP *);
void kmip_set_buffer(KMIP *, void *, size_t);
void kmip_destroy(KMIP *);
void kmip_push_error_frame(KMIP *, const char *, const int);

/* Message structure initializers */
void kmip_init_protocol_version(ProtocolVersion *, enum kmip_version);
void kmip_init_attribute(Attribute *);
void kmip_init_request_header(RequestHeader *);
void kmip_init_response_header(ResponseHeader *);

/* Message structure deallocators */
void kmip_free_request_message(KMIP *, RequestMessage *);
void kmip_free_response_message(KMIP *, ResponseMessage *);

/* Message structure debugging utilities */
void kmip_print_request_message(RequestMessage *);
void kmip_print_response_message(ResponseMessage *);
```

Library Context Utilities

The libkmip context contains various fields and attributes used in various ways throughout the encoding and decoding process. In general, the context fields should not be modified directly. All modifications should be done using one of the context utility functions described below.

The function header details for each of the relevant context utility functions are provided below.

void **kmip_init**(KMIP *, void *, size_t, enum *kmip_version*)

Initialize the `KMIP` context.

This function initializes the different fields and attributes used by the context to encode and decode KMIP messages. Reasonable defaults are chosen for certain fields, like the maximum message size and the error message size. If any of the memory allocation function hooks are `NULL`, they will be set to system defaults.

Parameters

- **KMIP*** – The libkmip `KMIP` context to be initialized. If `NULL`, the function does nothing and returns.

- **void*** – A `void` pointer to a buffer to be used for encoding and decoding KMIP messages. If setting up the context for use with the *Mid-level API* it is fine to use `NULL` here.
- **size_t** – The size of the above buffer. If setting up the context for use with the *Mid-level API* it is fine to use 0 here.
- **kmip_version** (*enum*) – A KMIP version enumeration that will be used by the context to decide how to encode and decode messages.

Returns None

void **kmip_clear_errors**(KMIP *)

Clean up any error-related information stored in the `KMIP` context.

This function clears and frees any error-related information or structures contained in the context, should any exist. It is intended to be used between encoding or decoding operations so that repeated use of the context is possible without causing errors. It is often used by other context handling utilities. See the utility source code for more details.

Parameters

- **KMIP*** – The libkmip `KMIP` context containing error-related information to be cleared.

Returns None

void **kmip_init_error_message**(KMIP *)

Initialize the error message field of the `KMIP` context.

This function allocates memory required to store the error message string in the library context. If an error message string already exists, nothing is done. Primarily used internally by other utility functions.

Parameters

- **KMIP*** – The libkmip `KMIP` context whose error message memory should be allocated.

Returns None

int **kmip_add_credential**(KMIP *, Credential *)

Add a `Credential` structure to the list of credentials used by the `KMIP` context.

This function dynamically adds a node to the `LinkedList` of `Credential` structures stored by the context. These credentials are used automatically by the *Mid-level API* when creating KMIP operation requests.

Parameters

- **KMIP*** – The libkmip `KMIP` context to add a credential to.
- **Credential*** – The libkmip `Credential` structure to add to the list of credentials stored by the context.

Returns

A status code indicating if the credential was added to the context. The code will be one of the following:

- **KMIP_OK** The credential was added successfully.
- **KMIP_UNSET** The credential was not added successfully.

void **kmip_remove_credentials**(KMIP *)

Remove all `Credential` structures stored by the `KMIP` context.

This function clears and frees all of the `LinkedList` nodes used to store the `Credential` structures associated with the context.

Note: If the underlying `Credential` structures were themselves dynamically allocatted, they must be freed separately by the parent application.

Parameters

- **KMIP*** – The libkmip `KMIP` context containing credentials to be removed.

Returns None

void **kmip_reset**(KMIP *)

Reset the `KMIP` context buffer so that encoding can be reattempted.

This function resets the context buffer to its initial empty starting state, allowing the context to be used for another encoding attempt if the prior attempt failed. The buffer will be overwritten with zeros to ensure that no information leaks across encoding attempts. This function also calls `kmip_clear_errors` to clear out any error information that was generated by the encoding failure.

Parameters

- **KMIP*** – The libkmip `KMIP` context that contains the buffer needing to be reset.

Returns None

void **kmip_rewind**(KMIP *)

Rewind the `KMIP` context buffer so that decoding can be reattempted.

This function rewinds the context buffer to its initial starting state, allowing the context to be used for another decoding attempt if the prior attempt failed. This function also calls `kmip_clear_errors` to clear out any error information that was generated by the decoding failure.

Parameters

- **KMIP*** – The libkmip `KMIP` context that contains the buffer needing to be rewound.

Returns None

void **kmip_set_buffer**(KMIP *, void *, size_t)

Set the encoding buffer used by the `KMIP` context.

Parameters

- **KMIP*** – The libkmip `KMIP` context that will contain the buffer.
- **void*** – A `void` pointer to a buffer to be used for encoding and decoding KMIP messages.
- **size_t** – The size of the above buffer.

Returns None

void **kmip_destroy**(KMIP *)

Deallocate the content of the `KMIP` context.

This function resets and deallocates all of the fields contained in the context. It calls `kmip_reset` and `kmip_set_buffer` to clear the buffer and overwrite any leftover pointers to it. It calls `kmip_clear_credentials` to clear out any referenced credential information. It also unsets all of the memory allocation function hooks.

Note: The buffer memory itself will not be deallocated by this function, nor will any of the `Credential` structures if they are dynamically allocatted. The parent application is responsible for clearing and deallocating those segments of memory.

void **kmip_push_error_frame**(KMIP *, const char *, const *int*)
Add an error frame to the stack trace contained in the KMIP context.

This function dynamically adds a new error frame to the context stack trace, using the information provided to record where an error occurred.

Parameters

- **KMIP*** – The libkmip KMIP context containing the stack trace.
- **char*** – The string containing the function name for the new stack trace error frame.
- **int** – The line number for the new stack trace error frame.

Returns None

Message Structure Initializers

There are many different KMIP message structures and substructures that are defined and supported by libkmip. In general, the parent application should zero initialize any libkmip structures before using them, like so:

```
RequestMessage message = {0};
```

In most cases, optional fields in KMIP substructures are excluded from the encoding process when set to 0. However, in some cases 0 is a valid value for a specific optional field. In these cases, we set these values to KMIP_UNSET. The parent application should never need to worry about manually initialize these types of fields. Instead, the following initializer functions should be used for the associated structures to handle properly setting default field values.

The function header details for each of the relevant initializer functions are provided below.

void **kmip_init_protocol_version**(ProtocolVersion *, enum *kmip_version*)
Initialize a ProtocolVersion structure with a KMIP version enumeration.

Parameters

- **ProtocolVersion*** – A libkmip ProtocolVersion structure to be initialized.
- **kmip_version** (*enum*) – A KMIP version enumeration whose value will be used to initialize the ProtocolVersion structure.

Returns None

void **kmip_init_attribute**(Attribute *)
Initialize an Attribute structure.

Parameters

- **Attribute*** – A libkmip Attribute structure to be initialized.

Returns None

void **kmip_init_request_header**(RequestHeader *)
Initialize a RequestHeader structure.

Parameters

- **RequestHeader*** – A libkmip RequestHeader structure to be initialized.

Returns None

void **kmip_init_response_header**(ResponseHeader *)
Initialize a ResponseHeader structure.

Parameters

- **ResponseHeader*** – A libkmip `ResponseHeader` structure to be initialized.

Returns None

Message Structure Deallocators

Along with structure initializers, there are corresponding structure deallocators for every supported KMIP structure. The deallocator behaves like the initializer; it takes in a pointer to a specific libkmip structure and will set all structure fields to safe, initial defaults. If a structure field is a non `NULL` pointer, the deallocator will attempt to clear and free the associated memory.

Note: A deallocator will not free the actual structure passed to it. It will only attempt to free memory referenced by the structure fields. The parent application is responsible for freeing the structure memory if it was dynamically allocated and should set any pointers to the structure to `NULL` once it is done with the structure.

Given how deallocators handle memory, they should only ever be used on structures that are created from the decoding process (i.e., structures created on the heap). The decoding process dynamically allocates memory to build out the message structure in the target encoding and it is beyond the capabilities of the client API or the parent application to manually free all of this memory directly.

Warning: If you use a deallocator on a structure allocated fully or in part on the stack, the deallocator will attempt to free stack memory and will trigger undefined behavior. This can lead to program instability and may cause the application to crash.

While there are deallocators for every supported structure, parent applications should only need to use the deallocators for request and response messages. Given these are the root KMIP structures, using these will free all associated substructures used to represent the message.

The function header details for each of the deallocator functions are provided below.

void **kmip_free_request_message**(KMIP *, RequestMessage *)
Deallocate the content of a `RequestMessage` structure.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures. Primarily used here for memory handlers.
- **RequestMessage*** – A libkmip `RequestMessage` structure whose content should be reset and/or freed.

Returns None

void **kmip_free_response_message**(KMIP *, ResponseMessage *)
Deallocate the content of a `ResponseMessage` structure.

Parameters

- **KMIP*** – A libkmip `KMIP` structure containing the context information needed to encode and decode message structures. Primarily used here for memory handlers.
- **ResponseMessage*** – A libkmip `ResponseMessage` structure whose content should be reset and/or freed.

Returns None

Message Structure Debugging Utilities

If the parent application is using the *Low-level API*, it will have access to the `RequestMessage` and `ResponseMessage` structures used to generate the KMIP operation encodings. These structures can be used with basic printing utilities to display the content of these structures in an easy to view and debug format.

The function header details for each of the printing utilities are provided below.

void **kmip_print_request_message**(RequestMessage *)
Print the contents of a `RequestMessage` structure.

Parameters

- **RequestMessage*** – A libkmip `RequestMessage` structure to be displayed.

Returns None

void **kmip_print_response_message**(ResponseMessage *)
Print the contents of a `ResponseMessage` structure.

Parameters

- **ResponseMessage*** – A libkmip `ResponseMessage` structure to be displayed.

Returns None

2.7 Examples

To demonstrate how to use libkmip, several example applications are built and deployed with the library to get developers started.

2.7.1 Demos

Three demo applications are included with libkmip, one for each of the following KMIP operations:

- `Create`
- `Get`
- `Destroy`

If libkmip is built, the demo applications can be found in the local build directory. If libkmip is installed, the demo applications can also be found in the bin directory, by default located at `/usr/local/bin/kmip`.

Run any of the demo applications with the `-h` flag to see usage information.

Create Demo

The `Create` demo, `demo_create.c`, uses the *Low-level API* to issue a KMIP request to the KMIP server to create a symmetric key. The application manually creates the library context and initalizes it. It then manually builds the request message structure, creating the following attributes for the symmetric key:

- cryptographic algorithm (AES)
- cryptographic length (256 bits)
- cryptographic usage mask (Encrypt and Decrypt usage)

The demo application encodes the request and then sends it through the low-level API to retrieve the response encoding. It decodes the response encoding into the response message structure and then extracts the UUID of the newly created symmetric key.

Get Demo

The `Get` demo, `demo_get.c`, uses the *Mid-level API* to issue a KMIP request to the KMIP server to retrieve a symmetric key. The application manually creates the library context and initializes it. It sets its own custom memory handlers to override the default ones supplied by libkmip and then invokes the mid-level API with the UUID of the symmetric key it wants to retrieve.

The client API internally builds the corresponding request message, encodes it, sends it via BIO to the KMIP server, retrieves the response encoding, and then decodes the response into the corresponding response message structure. Finally, it extracts the symmetric key bytes and copies them to a separate block of memory that will be handed back to the demo application. Finally, it cleans up the buffers used for the encoding and decoding process and cleans up the response message structure.

Destroy Demo

The `Destroy` demo, `demo_destroy.c`, use the *High-level API* to issue a KMIP request to the KMIP server to destroy a symmetric key. The application invokes the high-level API with the UUID of the symmetric key it wants to destroy.

The client API internally builds the library context along with the corresponding request message. It encodes the request, sends it via BIO to the KMIP server, retrieves the response encoding, and then decodes the response into the corresponding response message structure. Finally, it extracts the result of the KMIP operation from the response message structure and returns it.

2.7.2 Tests

A test application is also included with libkmip to exercise the encoding and decoding capabilities for all support KMIP features. The source code for this application, `tests.c`, contains numerous examples of how to build and use different libkmip structures.

Index

K