libscep Documentation

Release 0.1

Florian Rüchel

November 29, 2015

Contents

Contents:

CHAPTER 1

Narrative Documentation

This part of documentation outlines the usage of `libscep` and describes various topics you might be concerned with such as building, installing and using `libscep`.

1.1 Introduction

Before you start using this library, we need to go over a few basic concepts for generally using `libscep`.

1.1.1 Paramters & Return Values

All `libscep` functions return an error status of type *SCEP_ERROR*. You must always check that this value is `SCEPE_OK`. If it is not, you must not use the return parameters in any way and instead handle the error return by the function and potentially fail gracefully. A typical example would look like this:

```
SCEP *handle;
SCEP_ERROR error = scep_init(&handle);
if(error != SCEPE_OK)
    /* handle error */

/* continue normally */
```

Output parameters are always passed in last. The above example already shows a good example of that though it has no input parameters. Each function documents on how these paramters are used generally they are only every set in case of success and not touched beforehand.

1.1.2 Concept of SCEP

The basic library offers functionality to build and decompose SCEP both for client and server. However, the protocol defines some properties that lie beyond building the messages such as the transport to be used. This is not an integral part of the library itself and is left to the individual implementations on how this is achieved. The bindings in Perl, Python or calls from the command line might have different requirements and the library does not force any kind of behavior on the user here.

1.1.3 Public API

Common Parameters

Many of the functions share similar parameters which we wish to document here instead of separately on each function. The variable names in the signature are the same for all concerned functions.

SCEP_ERROR **scep_message_function**()

Parameters

- **handle** (SCEP *) – SCEP handle, see ?? (init...)
- **sig_cert** (*X509 **) – Sign PKCS#7 request with this. This will often be the old certificate with which to sign the request for renewal. It is also allowed to use a self-signed certificate here (see ??, selfsigned stuff)
- **sig_key** (*EVP_PKEY **) – Key corresponding to `sig_cert`.
- **enc_cert** (*X509 **) – Certificate with which to encrypt the request. Usually this is the CA/RA certificate for the SCEP server.
- **pkiMessage** (*PKCS7 ***) – This is an out-parameter: It will be set to a pointer of a PKCS#7 message if the function completes successfully. Otherwise it will be left in its previous state.

Returns Error status, see *Paramters & Return Values*.

Return type SCEP_ERROR

PKCSReq

SCEP_ERROR **scep_pkcsreq**(*SCEP *handle*, X509_REQ *req*, X509 *sig_cert*, EVP_PKEY *sig_key*, X509 *enc_cert*, PKCS7 ***pkiMessage*)

Create a PKCSReq pkiMessage. See *Common Parameters*. Special parameters:

Parameters

- **req** (*X509_REQ **) – Request for which the PKCSReq message should be created.

CertRep

SCEP_ERROR **scep_certrep**(*SCEP *handle*, char *transactionID*, char *senderNonce*, char *pkiStatus*, *SCEP_FAILINFO failInfo*, X509 *requestedCert*, X509 *sig_cert*, EVP_PKEY *sig_key*, X509 *enc_cert*, STACK_OF(X509) *additionalCerts*, X509_CRL *crl*, PKCS7 ***pkiMessage*)

Parameters

- **transactionID** (*char **) – Transaction ID chosen by the client, needs to be copied over so must stay the same as in the request.
- **senderNonce** (*char **) – Nonce used by sender in original request.
- **pkiStatus** (*char **) – One of `FAILURE`, `SUCCESS` or `PENDING`.
- **failInfo** (SCEP_FAILINFO) – Only makes sense if `pkiStatus` is `FAILURE`. In that case should represent the correct error according to the standard.
- **requestedCert** (*X509 **) – Certificate that was requested. Which certificate that is depends on the request, e.g. may be newly issued cert in case of a PKCSReq.

- **additionalCerts** (*STACK_OF(X509) **) – If you want to add more certificates, to your response, you can use this parameter to add them to the response. The client may ignore them.
- **crl** (*X509_CRL **) – If a CRL was requested instead of a certificate, set this parameter.

GetCertInitial

SCEP_ERROR **scep_get_cert_initial**(*SCEP *handle*, X509_REQ *req*, X509 *sig_cert*, EVP_PKEY *sig_key*, X509 *cacert*, X509 *enc_cert*, PKCS7 ***pkiMessage*)

Parameters

- **req** (*X509_REQ **) – The request for which this message should be created. It basically needs the subject defined here to create the appropriate request to the server.
- **cacert** (*X509 **) – Certificate of the CA from which the request expects a new certificate to be issued. This may be the same as `enc_cert` but can also be different, depending on the PKI setup.

GetCert

SCEP_ERROR **scep_get_cert**(*SCEP *handle*, X509 *sig_cert*, EVP_PKEY *sig_key*, X509_NAME *issuer*, ASN1_INTEGER *serial*, X509 *enc_cert*, PKCS7 ***pkiMessage*)

Parameters

- **issuer** (*X509_NAME **) – Name of the certificate issuer.
- **serial** (*ASN1_INTEGER **) – Serial number of requested certificate.

GetCRL

SCEP_ERROR **scep_get_crl**(*SCEP *handle*, X509 *sig_cert*, EVP_PKEY *sig_key*, X509 *req_cert*, X509 *enc_cert*)

Parameters

- **req_cert** (*X509 **) – Certificate for which CRL should be requested

Unwrapping

Unwrapping of requests is done directly with *`scep_unwrap()`*, responses should be parsed with the wrapper *`scep_unwrap_response()`* as this translates the degenerate PKCS#7 returned by CertRep into their corresponding type, i.e. certificate or CRL.

SCEP_ERROR **scep_unwrap**(*SCEP *handle*, PKCS7 *pkiMessage*, X509 *ca_cert*, X509 *dec_cert*, EVP_PKEY *dec_key*, *SCEP_DATA* ***output*)

Parameters

- **pkiMessage** (*PKCS7 **) – Contrary to the creation cases, this unpacks a PKCS#7 message and so this is an input parameter (the message) received from the client.
- **ca_cert** (*X509 **) – Root CA certificate used for signature verifcation.
- **dec_cert** (*X509 **) – Decryption certificate (either SCEP server or requester certificate).

- **dec_key** (*EVP_PKEY* *) – Private key corresponding to `dec_cert`.
- **output** (SCEP_DATA **) – Data structure in which all information obtained from parsing should be put. See *SCEP_DATA* for information on which fields have which meaning.

SCEP_ERROR **scep_unwrap_response** (*SCEP* *handle*, PKCS7 *pkiMessage*, X509 *ca_cert*, X509 *request_cert*, EVP_PKEY *request_key*, SCEP_OPERATION *request_type*, *SCEP_DATA* **output*)

This is basically the same as *scep_unwrap()* but handles extracting the correct type of response from the degenerate PKCS#7. Thus, parameters are mostly the same as with *scep_unwrap()*. Exception:

Parameters

- **request_type** (*SCEP_OPERATION*) – This indicates the type of request that was made for which this message is a response. This is necessary to interpret the encrypted content.

1.2 Engines

libscep has support for OpenSSL engines. Because the core functionality is completely independent from any engine support due to the generic PKEY interface, we only provide convenience functions and documentation.

OpenSSL offers a high flexibility for using engines, but in 90% of the cases the operations you perform are the same. Thus, the functions offered by libscep take this burden from you in these cases. In the remaining 10% you can use OpenSSL's original support without loss of flexbility or functionality.

1.2.1 Configuration

There are two types of engines with OpenSSL. First, builtin engines exist that OpenSSL already knows about. Second, an engine called `dynamic` is able to load engines not already part of OpenSSL during runtime. To ease usage, both ways are supported through a very similar interface.

To load a builtin engine you configure libscep like this:

```
scep_conf_set(handle, SCEPCFG_ENGINE, "chil");
```

This will load the builtin `chil` engine. On the other hand, a much more common use-case would be to load the engine dynamically:

```
scep_conf_set(handle, SCEPCFG_ENGINE, "dynamic", "pkcs11", "/path/to/engine_pkcs11.so");
```

This will do several things, but the basic gist is this: If you pass `dynamic` as the first configuration parameter, two more will be expected: The first denoting the engine ID (while this is your choice, it is generally clear how it should be named). The second parameter then is the path to the acutal shared object.

In both cases after calling this the engine will be fully operational if no error has been reported. However, some engines might require additional variables to be set up to work. In our example above, the PKCS#11 engine requires a `MODULE_PATH` variable to be set. Thus, it is possible to set any number of variables before loading the engine:

```
scep_conf_set(handle, SCEPCFG_ENGINE_PARAM, "MODULE_PATH", "/path/to/module.so");
```

Before the engine is actually loaded, the `MODULE_PATH` variable is set accordingly. To get a list of possible parameters see *Getting a List of Supported Parameters for SCEPCFG_ENGINE_PARAM*.

Note: Because these parameters have to be set before the engine is loaded it is not allowed to set parameters after an engine has been loaded (this would be useless anyways).

More Flexibility

If you require more flexibility, you can create your own engine object to your liking and then just hand it to the library:

```
scep_conf_set(handle, SCEPCFG_ENGINE_OBJ, engine);
```

In this case, libscep will only keep a reference to it but not take ownership of it: You are responsible for cleaning it up.

Warning: If you create multiple handles and mix `SCEPCFG_ENGINE_OBJ` and `SCEPCFG_ENGINE` you have to take care of the cleanup order: The global cleanup function `ENGINE_cleanup` is called if the last engine libscep knows about is freed. But this only applies if this engine was not passed in through `SCEPCFG_ENGINE_OBJ`. So: Always cleanup in the reverse order you set up and if your explicit engine is the last, you must call `ENGINE_cleanup` yourself, otherwise you **must not**.

1.2.2 Using the Engine

Because of the massive flexibility of the engine API and the diverse usage, we currently do not offer a wrapper around OpenSSL's engine functions. In the most general case, you want to load a private key from your engine:

```
ENGINE *engine = NULL;
scep_engine_get(handle, &engine);
EVP_PKEY *key = ENGINE_load_private_key(engine, "0:01", NULL, NULL);
```

`scep_engine_get` gives you a reference to the configured engine. Even if you configured the engine explicitly with `SCEPCFG_ENGINE_OBJ` you **must** use this interface for the engine. Afterwards, you can freely use the obtained reference on any OpenSSL engine functions.

In the example above, a private key is loaded from our previously configured PKCS#11 engine, loading key with ID `0x01` from slot `0`. We do not provide the optional callback and data parameters.

That's basically it: You now have an `EVP_PKEY` object usable with the library as OpenSSL is completely transparent regarding these anyway. For engine-specific actions and some additional details, refer to the next section.

1.2.3 Special Engines

Unfortunately, it often is not that simple because even though there exists a generic interface, technical differences exist. Thus, special handling is required for most engines. Since libscep does not know about these specialties, it is up to the programmer to take control. This is the main reason why we hand out an engine object instead of offering wrapping functions.

To aid you with this process, we provide documentation for several engines. If you have any suggestions, improvements or similar, please let us know and we will add it here.

pkcs11_engine

With PKCS#11 you are often required to enter a PIN. The engine offers various methods to provide this PIN but the most simple is globally setting it:

```
ENGINE_ctrl_cmd_string(engine, "PIN", "1234", 0);
```

capi

The capi engine for Microsoft's CryptoAPI can also be used, but might sometimes need extra parameters.

First of all, a store name has to be given. The default name for it is `MY` but when a new key with CSR is created, it is stored in the `REQUEST` store:

```
ENGINE_ctrl_cmd_string(engine, "store_name", "REQUEST", 0);
```

Also, if the system store instead of the user's store should be used:

```
ENGINE_ctrl_cmd(engine, "store_flags", 1, NULL, NULL, 0);
```

1.2.4 Tricks

Here are a few tricks that might help you in one case or another.

Getting a List of Supported Parameters for `SCEPCFG_ENGINE_PARAM`

Whenever you call `scep_conf_set` with `SCEPCFG_ENGINE_PARAM`, under the hood, `ENGINE_ctrl_cmd_string` is called. Thus, any parameter supported by an engine can be set here. For builtin engines, getting a list of these is fairly easy. For example, for CHIL:

```
$ openssl engine chil -vvv
(chil) CHIL hardware engine support
 SO_PATH: Specifies the path to the 'hwcrhk' shared library
       (input flags): STRING
 FORK_CHECK: Turns fork() checking on (non-zero) or off (zero)
       (input flags): NUMERIC
 THREAD_LOCKING: Turns thread-safe locking on (zero) or off (non-zero)
       (input flags): NUMERIC
```

Getting this for dynamically loaded engines is a bit more complicated:

```
openssl engine dynamic -pre SO_PATH:path/to/engine_pkcs11.so -pre ID:pkcs11 -pre LIST_ADD:1 -pre LOAD
(dynamic) Dynamic engine loading support
[Success]: SO_PATH:/home/javex/tmp/lib/engines/engine_pkcs11.so
[Success]: ID:pkcs11
[Success]: LIST_ADD:1
[Success]: LOAD
Loaded: (pkcs11) pkcs11 engine
     SO_PATH: Specifies the path to the 'pkcs11-engine' shared library
          (input flags): STRING
     MODULE_PATH: Specifies the path to the pkcs11 module shared library
          (input flags): STRING
     PIN: Specifies the pin code
          (input flags): STRING
     VERBOSE: Print additional details
          (input flags): NO_INPUT
     QUIET: Remove additional details
          (input flags): NO_INPUT
     INIT_ARGS: Specifies additional initialization arguments to the pkcs11 module
          (input flags): STRING
```

Note: Currently, we only support string values here. `NO_INPUT` might also work if you pass `NULL` as a value but this is untested.

CHAPTER 2

API Documentation

This document describes the API of `libscep` in detail. If you are looking for specific functions and implementation details you are correct here. If you are looking for just using this library, the *narrative documentation* might be more for you.

2.1 Functions

2.1.1 General functions

SCEP_ERROR **scep_init** (*SCEP* ***handle*)

Initializes the *SCEP* data structure and returns a success status. The memory for the contained structs is pre-allocated and can later be filled with some data, e.g. configuration values.

Make sure to call *scep_cleanup()* when you are done.

void **scep_cleanup** (*SCEP** *handle*)

Deallocate all memory that was reserved by the client during the process. Afterwards the data that was allocated is no longer accessible. Should be called at the end of the process, in conjuction with calling *scep_init()* at the beginning.

Note that there is some data that is not cleaned up. This is data which is documented to not be copied. Take a look at the specific configuration options you are using to avoid memory leaks.

SCEP_ERROR **scep_conf_set** (*SCEP** *handle*, SCEPCFG_TYPE *type*, ...)

Set the option for `handle` of type `type` to the value passed as the last argument. The documentation for `SCEPCFG_TYPE` describes which options are available and which parameters the function expects.

All values passed to this function are copied (except if explicitly stated otherwise), so any memory allocated can be freed after the option has been set. Freeing of the internal memory will be done by *scep_cleanup()*.

2.1.2 Utility functions

char* **scep_strerror** (*SCEP_ERROR err*)

Turns an internal error code into a human-readable string explaining the error code.

Example usage:

```
printf("Error message: %s\n", strerror(SCEPE_MEMORY));
```

2.2 Data Types

This section lists the data types used within `libscep`.

SCEP
A handle to a single instance for `libscep`. This needs to be passed to all functions that execute operations. It includes the configuration and some additional information.

SCEP_ERROR
An error code indicating a problem. Can be converted into human readable string using *scep_strerror()*. `SCEPE_OK` indicates that no error has happened and should be checked after calling any function that returns this type.

SCEP_PKISTATUS
Prefixed by `SCEP_` with possible suffixes `SUCCESS`, `PENDING` or `FAILURE` according to SCEP standard.

SCEP_FAILINFO
Enum that represents the `failInfo` field in a native way. All values are prefixed by `SCEP_BAD_`. The suffix decides which type of error it is. Available suffixes: `ALG`, `MESSAGE_CHECK`, `REQUEST`, `TIME`, `CERT_ID`, each corresponding to the failInfo field of an SCEP message. Only relevant if *SCEP_PKISTATUS* is `SCEP_FAILURE`.

SCEP_MESSAGE_TYPE
Enum that represents all possible messageType fields for SCEP. Prefixed by `SCEP_MSG_` and suffixed by one of `PKCSREQ`, `CERTREP`, `GETCERTINITIAL`, `GETCERT`, `GETCRL`. The integers in the enum correspond to their defined value in the standard, e.g. `SCEP_MSG_PKCSREQ` has the value `19`.

SCEP_DATA
Structure with all information contained in an SCEP pkiMessage in a more accessible way. Produced by *scep_unwrap()* and *scep_unwrap_response()*. The following field are defined:

Parameters

- **pkiStatus** (SCEP_PKISTATUS) – The status of a CertRep message, irrelevant for others
- **failInfo** (SCEP_FAILINFO) – If `pkiStatus` is FAILURE, this contains additional information.
- **transactionID** (*char **) – Transaction ID contained in request. This is always present. Stored hex encoded
- **senderNonce** (*unsigned char **) – Always present, exactly 16 byte long. Stored unencoded
- **recipientNonce** (*unsigned char **) – Only present in CertRep, format like `snederNonce`
- **challenge_passowrd** – Challenge password extracted from a PKCSReq, otherwise unset. Left at generic `ASN1_TYPE` to make no assumptions about its content, encoding, etc.
- **signer_certificate** (*X509 **) – The certificate used to sign the message. Currently unused.
- **messageType_str** (*char **) – Representation of message type as a stringified integer, e.g. `"19"` for PKCSReq. Provided for convenience.
- **messageType** (SCEP_MESSAGE_TYPE) – Message type represented by an enum, can assume any valid SCEP messageType.
- **request** (*X509_REQ **) – Only set when messageType is PKCSReq, contains the CSR.

- **initialEnrollment** (*int*) – Only PKCSReq. Whether this is an initial enrollment message, determined by whether the request was self-signed. 1 if it is initial enrollment, 0 otherwise.
- **issuer_and_serial** (*PKCS7_ISSUER_AND_SERIAL*) – Only GetCert and GetCRL.
- **issuer_and_subject** (*PKCS7_ISSUER_AND_SUBJECT*) – Only GetCertInitial.
- **certs** (*STACK_OF(X509) **) – Only CertRep if not response to GetCRL. Contains one or more certificate where the first one is the requested certificate (e.g. the newly issued in case of PKCSReq).
- **crl** (*X509_CRL **) – Only CertRep if response to GetCRL. Contains requested CRL.

CHAPTER 3

Developer Documentation

3.1 Running Tests

Running tests is designed to be as easy as possible. However, due to our engine support and the corresponding tests, various dependencies are introduced. Now, the easiest way to get things running is not to care at all. Just go ahead and run this:

```
mkdir build
cd build
cmake ..
make build_test
ctest --output-on-failure
```

This should create everything as it is needed without no need for intervention. However, this is by far not the quickest way because a lot of libraries have to be built (and if you delete the build directory, they will be built again).

3.1.1 Manually Installing Dependencies

If you want to have quicker builds, you can manually install the dependencies, possibly from your package manager. Here is a list of all the required packages:

- libbotan
- SoftHSM
- libp11
- engine_pkcs11

If you installed everything and their are fairly sane locations, running the code from the previous section should find these. If not, it will probably just build them anyway. It should also find all the correct paths to modules and libraries it requires or will complain if it doesn't. If you have suggestions on how to improve this process, please let us know.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

S