NAME

provider-decoder - The OSSL_DECODER library <-> provider functions

SYNOPSIS

```
#include <openssl/core_dispatch.h>

/*
 * None of these are actual functions, but are displayed like this for
 * the function signatures for functions that are offered as function
 * pointers in OSSL_DISPATCH arrays.
 */

/* Decoder parameter accessor and descriptor */
const OSSL_PARAM *OSSL_FUNC_decoder_gettable_params(void *provctx);
int OSSL_FUNC_decoder_get_params(OSSL_PARAM params[]);

/* Functions to construct / destruct / manipulate the decoder context */
void *OSSL_FUNC_decoder_newctx(void *provctx);
void OSSL_FUNC_decoder_freectx(void *ctx);
const OSSL_PARAM *OSSL_FUNC_decoder_settable_ctx_params(void *provctx);
int OSSL_FUNC_decoder_set_ctx_params(void *ctx, const OSSL_PARAM params[]);

/* Functions to check selection support */
int OSSL_FUNC_decoder_does_selection(void *provctx, int selection);

/* Functions to decode object data */
int OSSL_FUNC_decoder_decode(void *ctx, OSSL_CORE_BIO *in,
                             int selection,
                             OSSL_CALLBACK *data_cb, void *data_cbarg,
                             OSSL_PASSPHRASE_CALLBACK *cb, void *cbarg);

/* Functions to export a decoded object */
int OSSL_FUNC_decoder_export_object(void *ctx,
                                    const void *objref, size_t objref_sz,
                                    OSSL_CALLBACK *export_cb,
                                    void *export_cbarg);
```

DESCRIPTION

The term "decode" is used throughout this manual. This includes but is not limited to deserialization as individual decoders can also do decoding into intermediate data formats.

The DECODER operation is a generic method to create a provider-native object reference or intermediate decoded data from an encoded form read from the given **OSSL_CORE_BIO**. If the caller wants to decode data from memory, it should provide a **BIO_s_mem**(3) **BIO**. The decoded data or object reference is passed along with eventual metadata to the *metadata_cb* as **OSSL_PARAM**(3) parameters.

The decoder doesn't need to know more about the **OSSL_CORE_BIO** pointer than being able to pass it to the appropriate BIO upcalls (see "Core functions" in **provider-base**(7)).

The DECODER implementation may be part of a chain, where data is passed from one to the next. For example, there may be an implementation to decode an object from PEM to DER, and another one that decodes DER to a provider-native object.

The last decoding step in the decoding chain is usually supposed to create a provider-native object referenced by an object reference. To import that object into a different provider the **OSSL_FUNC_decoder_export_object()** can be called as the final step of the decoding process.

All "functions" mentioned here are passed as function pointers between *libcrypto* and the provider in **OSSL_DISPATCH**(3) arrays via **OSSL_ALGORITHM**(3) arrays that are returned by the provider's **provider_query_operation()** function (see "Provider Functions" in **provider-base**(7)).

All these "functions" have a corresponding function type definition named **OSSL_FUNC_{name}_fn**, and a helper function to retrieve the function pointer from an **OSSL_DISPATCH**(3) element named **OSSL_FUNC_{name}**. For example, the "function" **OSSL_FUNC_decoder_decode()** has these:

```
typedef int
    (OSSL_FUNC_decoder_decode_fn)(void *ctx, OSSL_CORE_BIO *in,
                                  int selection,
                                  OSSL_CALLBACK *data_cb, void *data_cbarg,
                                  OSSL_PASSPHRASE_CALLBACK *cb, void *cbarg);
static ossl_inline OSSL_FUNC_decoder_decode_fn*
    OSSL_FUNC_decoder_decode(const OSSL_DISPATCH *opf);
```

OSSL_DISPATCH(3) arrays are indexed by numbers that are provided as macros in **openssl-core_dispatch.h**(7), as follows:

```
OSSL_FUNC_decoder_get_params        OSSL_FUNC_DECODER_GET_PARAMS
OSSL_FUNC_decoder_gettable_params   OSSL_FUNC_DECODER_GETTABLE_PARAMS

OSSL_FUNC_decoder_newctx            OSSL_FUNC_DECODER_NEWCTX
```

OSSL_FUNC_decoder_freectx	OSSL_FUNC_DECODER_FREECTX
OSSL_FUNC_decoder_set_ctx_params	OSSL_FUNC_DECODER_SET_CTX_PARAMS
OSSL_FUNC_decoder_settable_ctx_params	OSSL_FUNC_DECODER_SETTABLE_CTX_PARAMS
OSSL_FUNC_decoder_does_selection	OSSL_FUNC_DECODER_DOES_SELECTION
OSSL_FUNC_decoder_decode	OSSL_FUNC_DECODER_DECODE
OSSL_FUNC_decoder_export_object	OSSL_FUNC_DECODER_EXPORT_OBJECT

Names and properties

The name of an implementation should match the target type of object it decodes. For example, an implementation that decodes an RSA key should be named "RSA". Likewise, an implementation that decodes DER data from PEM input should be named "DER".

Properties can be used to further specify details about an implementation:

input

This property is used to specify what format of input the implementation can decode.

This property is *mandatory*.

OpenSSL providers recognize the following input types:

pem An implementation with that input type decodes PEM formatted data.

der An implementation with that input type decodes DER formatted data.

msblob

An implementation with that input type decodes MSBLOB formatted data.

pvk An implementation with that input type decodes PVK formatted data.

structure

This property is used to specify the structure that the decoded data is expected to have.

This property is *optional*.

Structures currently recognised by built-in decoders:

"type-specific"
: Type specific structure.

"pkcs8"
: Structure according to the PKCS#8 specification.

"SubjectPublicKeyInfo"
: Encoding of public keys according to the Subject Public Key Info of RFC 5280.

The possible values of both these properties is open ended. A provider may very well specify input types and structures that libcrypto doesn't know anything about.

Subset selections

Sometimes, an object has more than one subset of data that is interesting to treat separately or together. It's possible to specify what subsets are to be decoded, with a set of bits *selection* that are passed in an **int**.

This set of bits depend entirely on what kind of provider-side object is to be decoded. For example, those bits are assumed to be the same as those used with **provider-keymgmt**(7) (see "Key Objects" in **provider-keymgmt**(7)) when the object is an asymmetric keypair - e.g., **OSSL_KEYMGMT_SELECT_PRIVATE_KEY** if the object to be decoded is supposed to contain private key components.

OSSL_FUNC_decoder_does_selection() should tell if a particular implementation supports any of the combinations given by *selection*.

Context functions

OSSL_FUNC_decoder_newctx() returns a context to be used with the rest of the functions.

OSSL_FUNC_decoder_freectx() frees the given *ctx* as created by **OSSL_FUNC_decoder_newctx()**.

OSSL_FUNC_decoder_set_ctx_params() sets context data according to parameters from *params* that it recognises. Unrecognised parameters should be ignored. Passing NULL for *params* should return true.

OSSL_FUNC_decoder_settable_ctx_params() returns a constant **OSSL_PARAM**(3) array describing the parameters that **OSSL_FUNC_decoder_set_ctx_params()** can handle.

See **OSSL_PARAM**(3) for further details on the parameters structure used by **OSSL_FUNC_decoder_set_ctx_params()** and **OSSL_FUNC_decoder_settable_ctx_params()**.

Export function

When a provider-native object is created by a decoder it would be unsuitable for direct use with a foreign provider. The export function allows for exporting the object into that foreign provider if the foreign provider supports the type of the object and provides an import function.

OSSL_FUNC_decoder_export_object() should export the object of size *objref_sz* referenced by *objref* as an **OSSL_PARAM**(3) array and pass that into the *export_cb* as well as the given *export_cbarg*.

Decoding functions

OSSL_FUNC_decoder_decode() should decode the data as read from the **OSSL_CORE_BIO** *in* to produce decoded data or an object to be passed as reference in an **OSSL_PARAM**(3) array along with possible other metadata that was decoded from the input. This **OSSL_PARAM**(3) array is then passed to the *data_cb* callback. The *selection* bits, if relevant, should determine what the input data should contain. The decoding functions also take an **OSSL_PASSPHRASE_CALLBACK**(3) function pointer along with a pointer to application data *cbarg*, which should be used when a pass phrase prompt is needed.

It's important to understand that the return value from this function is interpreted as follows:

True (1)
: This means "carry on the decoding process", and is meaningful even though this function couldn't decode the input into anything, because there may be another decoder implementation that can decode it into something.

  The *data_cb* callback should never be called when this function can't decode the input into anything.

False (0)
: This means "stop the decoding process", and is meaningful when the input could be decoded into some sort of object that this function understands, but further treatment of that object results into errors that won't be possible for some other decoder implementation to get a different result.

The conditions to stop the decoding process are at the discretion of the implementation.

Decoder operation parameters

There are currently no operation parameters currently recognised by the built-in decoders.

Parameters currently recognised by the built-in pass phrase callback:

"info" (**OSSL_PASSPHRASE_PARAM_INFO**) <UTF8 string>

A string of information that will become part of the pass phrase prompt. This could be used to give the user information on what kind of object it's being prompted for.

RETURN VALUES

OSSL_FUNC_decoder_newctx() returns a pointer to a context, or NULL on failure.

OSSL_FUNC_decoder_set_ctx_params() returns 1, unless a recognised parameter was invalid or caused an error, for which 0 is returned.

OSSL_FUNC_decoder_settable_ctx_params() returns a pointer to an array of constant **OSSL_PARAM**(3) elements.

OSSL_FUNC_decoder_does_selection() returns 1 if the decoder implementation supports any of the *selection* bits, otherwise 0.

OSSL_FUNC_decoder_decode() returns 1 to signal that the decoding process should continue, or 0 to signal that it should stop.

SEE ALSO

provider(7)

HISTORY

The DECODER interface was introduced in OpenSSL 3.0.

COPYRIGHT