NAME

ossl−guide−libcrypto−introduction, crypto − OpenSSL Guide: An introduction to libcrypto

INTRODUCTION

The OpenSSL cryptography library (`libcrypto`) enables access to a wide range of cryptographic algorithms used in various Internet standards. The services provided by this library are used by the OpenSSL implementations of TLS and CMS, and they have also been used to implement many other third party products and protocols.

The functionality includes symmetric encryption, public key cryptography, key agreement, certificate handling, cryptographic hash functions, cryptographic pseudo−random number generators, message authentication codes (MACs), key derivation functions (KDFs), and various utilities.

Algorithms

Cryptographic primitives such as the SHA256 digest, or AES encryption are referred to in OpenSSL as "algorithms". Each algorithm may have multiple implementations available for use. For example the RSA algorithm is available as a "default" implementation suitable for general use, and a "fips" implementation which has been validated to FIPS 140 standards for situations where that is important. It is also possible that a third party could add additional implementations such as in a hardware security module (HSM).

Algorithms are implemented in providers. See **ossl−guide−libraries−introduction**(7) for information about providers.

Operations

Different algorithms can be grouped together by their purpose. For example there are algorithms for encryption, and different algorithms for digesting data. These different groups are known as "operations" in OpenSSL. Each operation has a different set of functions associated with it. For example to perform an encryption operation using AES (or any other encryption algorithm) you would use the encryption functions detailed on the **EVP_EncryptInit**(3) page. Or to perform a digest operation using SHA256 then you would use the digesting functions on the **EVP_DigestInit**(3) page.

ALGORITHM FETCHING

In order to use an algorithm an implementation for it must first be "fetched". Fetching is the process of looking through the available implementations, applying selection criteria (via a property query string), and finally choosing the implementation that will be used.

Two types of fetching are supported by OpenSSL − "Explicit fetching" and "Implicit fetching".

Explicit fetching

Explicit fetching involves directly calling a specific API to fetch an algorithm implementation from a provider. This fetched object can then be passed to other APIs. These explicit fetching functions usually have the name `APINAME_fetch`, where `APINAME` is the name of the operation. For example **EVP_MD_fetch**(3) can be used to explicitly fetch a digest algorithm implementation. The user is responsible for freeing the object returned from the `APINAME_fetch` function using `APINAME_free` when it is no longer needed.

These fetching functions follow a fairly common pattern, where three arguments are passed:

The library context

See **OSSL_LIB_CTX**(3) for a more detailed description. This may be NULL to signify the default (global) library context, or a context created by the user. Only providers loaded in this library context (see **OSSL_PROVIDER_load**(3)) will be considered by the fetching function. In case no provider has been loaded in this library context then the default provider will be loaded as a fallback (see **OSSL_PROVIDER−default**(7)).

An identifier

For all currently implemented fetching functions this is the algorithm name. Each provider supports a list of algorithm implementations. See the provider specific documentation for information on the algorithm implementations available in each provider: "OPERATIONS AND ALGORITHMS" in **OSSL_PROVIDER−default**(7), "OPERATIONS AND ALGORITHMS" in **OSSL_PROVIDER−FIPS**(7), "OPERATIONS AND ALGORITHMS" in

OSSL_PROVIDER−legacy(7) and "OPERATIONS AND ALGORITHMS" in **OSSL_PROVIDER−base**(7).

Note, while providers may register algorithms against a list of names using a string with a colon separated list of names, fetching algorithms using that format is currently unsupported.

A property query string
The property query string used to guide selection of the algorithm implementation. See "PROPERTY QUERY STRINGS" in **ossl−guide−libraries−introduction**(7).

The algorithm implementation that is fetched can then be used with other diverse functions that use them. For example the **EVP_DigestInit_ex**(3) function takes as a parameter an **EVP_MD** object which may have been returned from an earlier call to **EVP_MD_fetch**(3).

Implicit fetching

OpenSSL has a number of functions that return an algorithm object with no associated implementation, such as **EVP_sha256**(3), **EVP_aes_128_cbc**(3), **EVP_get_cipherbyname**(3) or **EVP_get_digestbyname**(3). These are present for compatibility with OpenSSL before version 3.0 where explicit fetching was not available.

When they are used with functions like **EVP_DigestInit_ex**(3) or **EVP_CipherInit_ex**(3), the actual implementation to be used is fetched implicitly using default search criteria (which uses NULL for the library context and property query string).

In some cases implicit fetching can also occur when a NULL algorithm parameter is supplied. In this case an algorithm implementation is implicitly fetched using default search criteria and an algorithm name that is consistent with the context in which it is being used.

Functions that use an **EVP_PKEY_CTX** or an **EVP_PKEY**(3), such as **EVP_DigestSignInit**(3), all fetch the implementations implicitly. Usually the algorithm to fetch is determined based on the type of key that is being used and the function that has been called.

Performance

If you perform the same operation many times with the same algorithm then it is recommended to use a single explicit fetch of the algorithm and then reuse the explicitly fetched algorithm each subsequent time. This will typically be faster than implicitly fetching the algorithm every time you use it. See an example of Explicit fetching in "USING ALGORITHMS IN APPLICATIONS".

Prior to OpenSSL 3.0, functions such as **EVP_sha256()** which return a "const" object were used directly to indicate the algorithm to use in various function calls. If you pass the return value of one of these convenience functions to an operation then you are using implicit fetching. If you are converting an application that worked with an OpenSSL version prior to OpenSSL 3.0 then consider changing instances of implicit fetching to explicit fetching instead.

If an explicitly fetched object is not passed to an operation, then any implicit fetch will use an internally cached prefetched object, but it will still be slower than passing the explicitly fetched object directly.

The following functions can be used for explicit fetching:

EVP_MD_fetch(3)
Fetch a message digest/hashing algorithm implementation.

EVP_CIPHER_fetch(3)
Fetch a symmetric cipher algorithm implementation.

EVP_KDF_fetch(3)
Fetch a Key Derivation Function (KDF) algorithm implementation.

EVP_MAC_fetch(3)
Fetch a Message Authentication Code (MAC) algorithm implementation.

EVP_KEM_fetch(3)
Fetch a Key Encapsulation Mechanism (KEM) algorithm implementation

OSSL_ENCODER_fetch(3)
Fetch an encoder algorithm implementation (e.g. to encode keys to a specified format).

OSSL_DECODER_fetch(3)
Fetch a decoder algorithm implementation (e.g. to decode keys from a specified format).

EVP_RAND_fetch(3)
Fetch a Pseudo Random Number Generator (PRNG) algorithm implementation.

See "OPERATIONS AND ALGORITHMS" in **OSSL_PROVIDER-default**(7), "OPERATIONS AND ALGORITHMS" in **OSSL_PROVIDER-FIPS**(7), "OPERATIONS AND ALGORITHMS" in **OSSL_PROVIDER-legacy**(7) and "OPERATIONS AND ALGORITHMS" in **OSSL_PROVIDER-base**(7) for a list of algorithm names that can be fetched.

FETCHING EXAMPLES

The following section provides a series of examples of fetching algorithm implementations.

Fetch any available implementation of SHA2-256 in the default context. Note that some algorithms have aliases. So "SHA256" and "SHA2-256" are synonymous:

```
EVP_MD *md = EVP_MD_fetch(NULL, "SHA2-256", NULL);
...
EVP_MD_free(md);
```

Fetch any available implementation of AES-128-CBC in the default context:

```
EVP_CIPHER *cipher = EVP_CIPHER_fetch(NULL, "AES-128-CBC", NULL);
...
EVP_CIPHER_free(cipher);
```

Fetch an implementation of SHA2-256 from the default provider in the default context:

```
EVP_MD *md = EVP_MD_fetch(NULL, "SHA2-256", "provider=default");
...
EVP_MD_free(md);
```

Fetch an implementation of SHA2-256 that is not from the default provider in the default context:

```
EVP_MD *md = EVP_MD_fetch(NULL, "SHA2-256", "provider!=default");
...
EVP_MD_free(md);
```

Fetch an implementation of SHA2-256 that is preferably from the FIPS provider in the default context:

```
EVP_MD *md = EVP_MD_fetch(NULL, "SHA2-256", "provider=?fips");
...
EVP_MD_free(md);
```

Fetch an implementation of SHA2-256 from the default provider in the specified library context:

```
EVP_MD *md = EVP_MD_fetch(libctx, "SHA2-256", "provider=default");
...
EVP_MD_free(md);
```

Load the legacy provider into the default context and then fetch an implementation of WHIRLPOOL from it:

```
/* This only needs to be done once - usually at application start up */
OSSL_PROVIDER *legacy = OSSL_PROVIDER_load(NULL, "legacy");

EVP_MD *md = EVP_MD_fetch(NULL, "WHIRLPOOL", "provider=legacy");
...
EVP_MD_free(md);
```

Note that in the above example the property string "provider=legacy" is optional since, assuming no other providers have been loaded, the only implementation of the "whirlpool" algorithm is in the "legacy"

provider. Also note that the default provider should be explicitly loaded if it is required in addition to other providers:

```
/* This only needs to be done once - usually at application start up */
OSSL_PROVIDER *legacy = OSSL_PROVIDER_load(NULL, "legacy");
OSSL_PROVIDER *default = OSSL_PROVIDER_load(NULL, "default");

EVP_MD *md_whirlpool = EVP_MD_fetch(NULL, "whirlpool", NULL);
EVP_MD *md_sha256 = EVP_MD_fetch(NULL, "SHA2-256", NULL);
...
EVP_MD_free(md_whirlpool);
EVP_MD_free(md_sha256);
```

USING ALGORITHMS IN APPLICATIONS

Cryptographic algorithms are made available to applications through use of the "EVP" APIs. Each of the various operations such as encryption, digesting, message authentication codes, etc., have a set of EVP function calls that can be invoked to use them. See the **evp**(7) page for further details.

Most of these follow a common pattern. A "context" object is first created. For example for a digest operation you would use an **EVP_MD_CTX**, and for an encryption/decryption operation you would use an **EVP_CIPHER_CTX**. The operation is then initialised ready for use via an "init" function - optionally passing in a set of parameters (using the **OSSL_PARAM**(3) type) to configure how the operation should behave. Next data is fed into the operation in a series of "update" calls. The operation is finalised using a "final" call which will typically provide some kind of output. Finally the context is cleaned up and freed.

The following shows a complete example for doing this process for digesting data using SHA256. The process is similar for other operations such as encryption/decryption, signatures, message authentication codes, etc. Additional examples can be found in the OpenSSL demos (see "DEMO APPLICATIONS" in **ossl-guide-libraries-introduction**(7)).

```
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/bio.h>
#include <openssl/err.h>

int main(void)
{
    EVP_MD_CTX *ctx = NULL;
    EVP_MD *sha256 = NULL;
    const unsigned char msg[] = {
        0x00, 0x01, 0x02, 0x03
    };
    unsigned int len = 0;
    unsigned char *outdigest = NULL;
    int ret = 1;

    /* Create a context for the digest operation */
    ctx = EVP_MD_CTX_new();
    if (ctx == NULL)
        goto err;

    /*
     * Fetch the SHA256 algorithm implementation for doing the digest. We're
     * using the "default" library context here (first NULL parameter), and
     * we're not supplying any particular search criteria for our SHA256
     * implementation (second NULL parameter). Any SHA256 implementation will
     * do.
```

```
         * In a larger application this fetch would just be done once, and could
         * be used for multiple calls to other operations such as EVP_DigestInit_ex().
         */
        sha256 = EVP_MD_fetch(NULL, "SHA256", NULL);
        if (sha256 == NULL)
            goto err;

       /* Initialise the digest operation */
       if (!EVP_DigestInit_ex(ctx, sha256, NULL))
           goto err;

        /*
         * Pass the message to be digested. This can be passed in over multiple
         * EVP_DigestUpdate calls if necessary
         */
        if (!EVP_DigestUpdate(ctx, msg, sizeof(msg)))
            goto err;

        /* Allocate the output buffer */
        outdigest = OPENSSL_malloc(EVP_MD_get_size(sha256));
        if (outdigest == NULL)
            goto err;

        /* Now calculate the digest itself */
        if (!EVP_DigestFinal_ex(ctx, outdigest, &len))
            goto err;

        /* Print out the digest result */
        BIO_dump_fp(stdout, outdigest, len);

        ret = 0;

     err:
        /* Clean up all the resources we allocated */
        OPENSSL_free(outdigest);
        EVP_MD_free(sha256);
        EVP_MD_CTX_free(ctx);
        if (ret != 0)
           ERR_print_errors_fp(stderr);
        return ret;
    }
```

ENCODING AND DECODING KEYS

Many algorithms require the use of a key. Keys can be generated dynamically using the EVP APIs (for example see **EVP_PKEY_Q_keygen**(3)). However it is often necessary to save or load keys (or their associated parameters) to or from some external format such as PEM or DER (see **openssl-glossary**(7)). OpenSSL uses encoders and decoders to perform this task.

Encoders and decoders are just algorithm implementations in the same way as any other algorithm implementation in OpenSSL. They are implemented by providers. The OpenSSL encoders and decoders are available in the default provider. They are also duplicated in the base provider.

For information about encoders see **OSSL_ENCODER_CTX_new_for_pkey**(3). For information about decoders see **OSSL_DECODER_CTX_new_for_pkey**(3).

As well as using encoders/decoders directly there are also some helper functions that can be used for certain well known and commonly used formats. For example see **PEM_read_PrivateKey**(3) and

PEM_write_PrivateKey(3) for information about reading and writing key data from PEM encoded files.

FURTHER READING

See **ossl-guide-libssl-introduction**(7) for an introduction to using `libssl`.

SEE ALSO

openssl(1), **ssl**(7), **evp**(7), **OSSL_LIB_CTX**(3), **openssl-threads**(7), **property**(7), **OSSL_PROVIDER-default**(7), **OSSL_PROVIDER-base**(7), **OSSL_PROVIDER-FIPS**(7), **OSSL_PROVIDER-legacy**(7), **OSSL_PROVIDER-null**(7), **openssl-glossary**(7), **provider**(7)

COPYRIGHT