Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'crypto.7ossl' command

$ man crypto.7ossl

CRYPTO(7ossl) OpenSSL CRYPTO(7ossl)

NAME

crypto - OpenSSL cryptographic library

SYNOPSIS

See the individual manual pages for details.

DESCRIPTION

The OpenSSL crypto library ("libcrypto") implements 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 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).

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.

Providers

A provider in OpenSSL is a component that collects together algorithm implementations. In order to use an algorithm you must have at least one provider loaded that contains an implementation of it. OpenSSL comes with a number of providers and they may also be obtained from third parties. If you don't load a provider explicitly (either in program code or via config) then the OpenSSL built-in "default" provider will be automatically loaded.

Library contexts

A library context can be thought of as a "scope" within which configuration options take effect. When a provider is loaded, it is only loaded within the scope of a given library context. In this way it is possible for different components of a complex application to each use a different library context and have different providers loaded with different configuration settings.

If an application does not explicitly create a library context then the "default" library context will be used.

Library contexts are represented by the OSSL_LIB_CTX type. Many OpenSSL API functions take a library context as a parameter. Applications can always pass NULL for this parameter to just use the default library context.

The default library context is automatically created the first time it is needed. This will automatically load any available configuration file and will initialise OpenSSL for use. Unlike in earlier versions of OpenSSL (prior to 1.1.0) no explicit initialisation steps need to be taken.

Similarly when the application exits the default library context is automatically destroyed. No explicit de-initialisation steps need to be taken.

See OSSL_LIB_CTX(3) for more information about library contexts. See also "ALGORITHM FETCHING".

Multi-threaded applications

As long as OpenSSL has been built with support for threads (the default case on most platforms) then most OpenSSL functions are thread-safe in the sense that it is safe to call the same function from multiple threads at the same time. However most OpenSSL data structures are not thread-safe. For example the BIO_write(3) and BIO_read(3) functions are thread safe. However it would not be thread safe to call BIO_write() from one thread while calling BIO_read() in another where both functions are passed the same BIO object since both of them may attempt to make changes to the same BIO object.

There are exceptions to these rules. A small number of functions are not thread safe at all. Where this is the case this restriction should be noted in the documentation for the function. Similarly some data structures may be partially or fully thread safe. For example it is safe to use an OSSL_LIB_CTX in multiple threads.

See openssl-threads(7) for a more detailed discussion on OpenSSL threading support.

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.

Property query strings

When fetching an algorithm it is possible to specify a property query string to guide the selection process. For example a property query string of "provider=default" could be used to force the selection to only consider algorithm implementations in the default provider.

Property query strings can be specified explicitly as an argument to a function. It is also possible to specify a default property query string for the whole library context using the EVP_set_default_properties(3) function. Where both default properties and function specific properties are specified then they are combined. Function specific properties will override default properties where there is a conflict.

See property(7) for more information about properties.

Explicit fetching

Users of the OpenSSL libraries never query a provider directly for an algorithm implementation. Instead, the diverse OpenSSL APIs often have explicit fetching functions that do the work, and they return an appropriate algorithm object back to the user. These 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.

A property query string

The property query string used to guide selection of the algorithm implementation.

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 fetch

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.

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 revolve around EVP_PKEY_CTX and EVP_PKEY(3), such as EVP_DigestSignInit(3) and friends, all fetch the implementations implicitly. Because these functions involve both an operation type (such as EVP_SIGNATURE(3)) and an EVP_KEYMGMT(3) for the EVP_PKEY(3), they try the following:

1. Fetch the operation type implementation from any provider given a library context and property string stored in the EVP_PKEY_CTX.

   If the provider of the operation type implementation is different from the provider of the EVP_PKEY(3)'s EVP_KEYMGMT(3)

implementation, try to fetch a EVP_KEYMGMT(3) implementation in the same provider as the operation type implementation and export the EVP_PKEY(3) to it (effectively making a temporary copy of the original key).

If anything in this step fails, the next step is used as a fallback.

2. As a fallback, try to fetch the operation type implementation from the same provider as the original EVP_PKEY(3)'s EVP_KEYMGMT(3), still using the propery string from the EVP_PKEY_CTX.

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 from the default provider in the specified context:

```
EVP_MD *md = EVP_MD_fetch(ctx, "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);
```

OPENSSL PROVIDERS

OpenSSL comes with a set of providers.

The algorithms available in each of these providers may vary due to build time configuration options. The openssl-list(1) command can be used to list the currently available algorithms.

The names of the algorithms shown from openssl-list(1) can be used as an algorithm identifier to the appropriate fetching function. Also see the provider specific manual pages linked below for further details

about using the algorithms available in each of the providers.

As well as the OpenSSL providers third parties can also implement providers. For information on writing a provider see provider(7).

Default provider

The default provider is built in as part of the libcrypto library and contains all of the most commonly used algorithm implementations. Should it be needed (if other providers are loaded and offer implementations of the same algorithms), the property query string "provider=default" can be used as a search criterion for these implementations. The default provider includes all of the functionality in the base provider below.

If you don't load any providers at all then the "default" provider will be automatically loaded. If you explicitly load any provider then the "default" provider would also need to be explicitly loaded if it is required.

See OSSL_PROVIDER-default(7).

Base provider

The base provider is built in as part of the libcrypto library and contains algorithm implementations for encoding and decoding for OpenSSL keys. Should it be needed (if other providers are loaded and offer implementations of the same algorithms), the property query string "provider=base" can be used as a search criterion for these implementations. Some encoding and decoding algorithm implementations are not FIPS algorithm implementations in themselves but support algorithms from the FIPS provider and are allowed for use in "FIPS mode". The property query string "fips=yes" can be used to select such algorithms.

See OSSL_PROVIDER-base(7).

FIPS provider

The FIPS provider is a dynamically loadable module, and must therefore be loaded explicitly, either in code or through OpenSSL configuration (see config(5)). It contains algorithm implementations that have been validated according to the FIPS 140-2 standard. Should it be needed (if

other providers are loaded and offer implementations of the same algorithms), the property query string "provider=fips" can be used as a search criterion for these implementations. All approved algorithm implementations in the FIPS provider can also be selected with the property "fips=yes". The FIPS provider may also contain non-approved algorithm implementations and these can be selected with the property "fips=no".

See OSSL_PROVIDER-FIPS(7) and fips_module(7).

Legacy provider

The legacy provider is a dynamically loadable module, and must therefore be loaded explicitly, either in code or through OpenSSL configuration (see config(5)). It contains algorithm implementations that are considered insecure, or are no longer in common use such as MD2 or RC4. Should it be needed (if other providers are loaded and offer implementations of the same algorithms), the property "provider=legacy" can be used as a search criterion for these implementations.

See OSSL_PROVIDER-legacy(7).

Null provider

The null provider is built in as part of the libcrypto library. It contains no algorithms in it at all. When fetching algorithms the default provider will be automatically loaded if no other provider has been explicitly loaded. To prevent that from happening you can explicitly load the null provider.

See OSSL_PROVIDER-null(7).

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 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.

```
 #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.
     */
    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;
}
```

CONFIGURATION

By default OpenSSL will load a configuration file when it is first used. This will set up various configuration settings within the default library context. Applications that create their own library contexts may optionally configure them with a config file using the OSSL_LIB_CTX_load_config(3) function.

The configuration file can be used to automatically load providers and set up default property query strings.

For information on the OpenSSL configuration file format see config(5).

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).

LIBRARY CONVENTIONS

Many OpenSSL functions that "get" or "set" a value follow a naming convention using the numbers 0 and 1, i.e. "get0", "get1", "set0" and "set1". This can also apply to some functions that "add" a value to an existing set, i.e. "add0" and "add1".

For example the functions:

```
int X509_CRL_add0_revoked(X509_CRL *crl, X509_REVOKED *rev);
int X509_add1_trust_object(X509 *x, const ASN1_OBJECT *obj);
```

In the 0 version the ownership of the object is passed to (for an add or set) or retained by (for a get) the parent object. For example after calling the X509_CRL_add0_revoked() function above, ownership of the rev object is passed to the crl object. Therefore, after calling this function rev should not be freed directly. It will be freed implicitly when crl is freed.

In the 1 version the ownership of the object is not passed to or retained by the parent object. Instead a copy or "up ref" of the object is performed. So after calling the X509_add1_trust_object() function above the application will still be responsible for freeing the obj value where appropriate.

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