NAME

fips_module − OpenSSL fips module guide

SYNOPSIS

See the individual manual pages for details.

DESCRIPTION

This guide details different ways that OpenSSL can be used in conjunction with the FIPS module. Which is the correct approach to use will depend on your own specific circumstances and what you are attempting to achieve.

Note that the old functions **FIPS_mode()** and **FIPS_mode_set()** are no longer present so you must remove them from your application if you use them.

Applications written to use the OpenSSL 3.0 FIPS module should not use any legacy APIs or features that avoid the FIPS module. Specifically this includes:

- Low level cryptographic APIs (use the high level APIs, such as EVP, instead)
- Engines
- Any functions that create or modify custom ''METHODS'' (for example **EVP_MD_meth_new()**, **EVP_CIPHER_meth_new()**, **EVP_PKEY_meth_new()**, **RSA_meth_new()**, **EC_KEY_METHOD_new()**, etc.)

All of the above APIs are deprecated in OpenSSL 3.0 − so a simple rule is to avoid using all deprecated functions. See **migration_guide** (7) for a list of deprecated functions.

Making all applications use the FIPS module by default

One simple approach is to cause all applications that are using OpenSSL to only use the FIPS module for cryptographic algorithms by default.

This approach can be done purely via configuration. As long as applications are built and linked against OpenSSL 3.0 and do not override the loading of the default config file or its settings then they can automatically start using the FIPS module without the need for any further code changes.

To do this the default OpenSSL config file will have to be modified. The location of this config file will depend on the platform, and any options that were given during the build process. You can check the location of the config file by running this command:

```
$ openssl version -d
OPENSSLDIR: "/usr/local/ssl"
```

Caution: Many Operating Systems install OpenSSL by default. It is a common error to not have the correct version of OpenSSL in your `$PATH`. Check that you are running an OpenSSL 3.0 version like this:

```
$ openssl version -v
OpenSSL 3.0.0-dev xx XXX xxxx (Library: OpenSSL 3.0.0-dev xx XXX xxxx)
```

The **OPENSSLDIR** value above gives the directory name for where the default config file is stored. So in this case the default config file will be called */usr/local/ssl/openssl.cnf*.

Edit the config file to add the following lines near the beginning:

```
config_diagnostics = 1
openssl_conf = openssl_init

.include /usr/local/ssl/fipsmodule.cnf

[openssl_init]
providers = provider_sect

[provider_sect]
fips = fips_sect
base = base_sect
```

```
[base_sect]
activate = 1
```

Obviously the include file location above should match the path and name of the FIPS module config file that you installed earlier. See <https://github.com/openssl/openssl/blob/master/README-FIPS.md>.

For FIPS usage, it is recommened that the **config_diagnostics** option is enabled to prevent accidental use of non-FIPS validated algorithms via broken or mistaken configuration. See **config**(5).

Any applications that use OpenSSL 3.0 and are started after these changes are made will start using only the FIPS module unless those applications take explicit steps to avoid this default behaviour. Note that this configuration also activates the ''base'' provider. The base provider does not include any cryptographic algorithms (and therefore does not impact the validation status of any cryptographic operations), but does include other supporting algorithms that may be required. It is designed to be used in conjunction with the FIPS module.

This approach has the primary advantage that it is simple, and no code changes are required in applications in order to benefit from the FIPS module. There are some disadvantages to this approach:

- You may not want all applications to use the FIPS module.

  It may be the case that some applications should and some should not use the FIPS module.

- If applications take explicit steps to not load the default config file or set different settings.

  This method will not work for these cases.

- The algorithms available in the FIPS module are a subset of the algorithms that are available in the default OpenSSL Provider.

  If any applications attempt to use any algorithms that are not present, then they will fail.

- Usage of certain deprecated APIs avoids the use of the FIPS module.

  If any applications use those APIs then the FIPS module will not be used.

Selectively making applications use the FIPS module by default

A variation on the above approach is to do the same thing on an individual application basis. The default OpenSSL config file depends on the compiled in value for **OPENSSLDIR** as described in the section above. However it is also possible to override the config file to be used via the **OPENSSL_CONF** environment variable. For example the following, on Unix, will cause the application to be executed with a non-standard config file location:

```
$ OPENSSL_CONF=/my/nondefault/openssl.cnf myapplication
```

Using this mechanism you can control which config file is loaded (and hence whether the FIPS module is loaded) on an application by application basis.

This removes the disadvantage listed above that you may not want all applications to use the FIPS module. All the other advantages and disadvantages still apply.

Programmatically loading the FIPS module (default library context)

Applications may choose to load the FIPS provider explicitly rather than relying on config to do this. The config file is still necessary in order to hold the FIPS module config data (such as its self test status and integrity data). But in this case we do not automatically activate the FIPS provider via that config file.

To do things this way configure as per ''Making all applications use the FIPS module by default'' above, but edit the *fipsmodule.cnf* file to remove or comment out the line which says `activate = 1` (note that setting this value to 0 is *not* sufficient). This means all the required config information will be available to load the FIPS module, but it is not automatically loaded when the application starts. The FIPS provider can then be loaded programmatically like this:

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

```
int main(void)
{
    OSSL_PROVIDER *fips;
    OSSL_PROVIDER *base;

    fips = OSSL_PROVIDER_load(NULL, "fips");
    if (fips == NULL) {
        printf("Failed to load FIPS provider\n");
        exit(EXIT_FAILURE);
    }
    base = OSSL_PROVIDER_load(NULL, "base");
    if (base == NULL) {
        OSSL_PROVIDER_unload(fips);
        printf("Failed to load base provider\n");
        exit(EXIT_FAILURE);
    }

    /* Rest of application */

    OSSL_PROVIDER_unload(base);
    OSSL_PROVIDER_unload(fips);
    exit(EXIT_SUCCESS);
}
```

Note that this should be one of the first things that you do in your application. If any OpenSSL functions get called that require the use of cryptographic functions before this occurs then, if no provider has yet been loaded, then the default provider will be automatically loaded. If you then later explicitly load the FIPS provider then you will have both the FIPS and the default provider loaded at the same time. It is undefined which implementation of an algorithm will be used if multiple implementations are available and you have not explicitly specified via a property query (see below) which one should be used.

Also note that in this example we have additionally loaded the ‘‘base’’ provider. This loads a sub-set of algorithms that are also available in the default provider − specifically non cryptographic ones which may be used in conjunction with the FIPS provider. For example this contains algorithms for encoding and decoding keys. If you decide not to load the default provider then you will usually want to load the base provider instead.

In this example we are using the ‘‘default’’ library context. OpenSSL functions operate within the scope of a library context. If no library context is explicitly specified then the default library context is used. For further details about library contexts see the **OSSL_LIB_CTX** (3) man page.

Loading the FIPS module at the same time as other providers

It is possible to have the FIPS provider and other providers (such as the default provider) all loaded at the same time into the same library context. You can use a property query string during algorithm fetches to specify which implementation you would like to use.

For example to fetch an implementation of SHA256 which conforms to FIPS standards you can specify the property query `fips=yes` like this:

```
EVP_MD *sha256;

sha256 = EVP_MD_fetch(NULL, "SHA2-256", "fips=yes");
```

If no property query is specified, or more than one implementation matches the property query then it is undefined which implementation of a particular algorithm will be returned.

This example shows an explicit request for an implementation of SHA256 from the default provider:

```
EVP_MD *sha256;
```

```
sha256 = EVP_MD_fetch(NULL, "SHA2-256", "provider=default");
```

It is also possible to set a default property query string. The following example sets the default property query of `fips=yes` for all fetches within the default library context:

```
EVP_set_default_properties(NULL, "fips=yes");
```

If a fetch function has both an explicit property query specified, and a default property query is defined then the two queries are merged together and both apply. The local property query overrides the default properties if the same property name is specified in both.

There are two important built-in properties that you should be aware of:

The ''provider'' property enables you to specify which provider you want an implementation to be fetched from, e.g. `provider=default` or `provider=fips`. All algorithms implemented in a provider have this property set on them.

There is also the `fips` property. All FIPS algorithms match against the property query `fips=yes`. There are also some non-cryptographic algorithms available in the default and base providers that also have the `fips=yes` property defined for them. These are the encoder and decoder algorithms that can (for example) be used to write out a key generated in the FIPS provider to a file. The encoder and decoder algorithms are not in the FIPS module itself but are allowed to be used in conjunction with the FIPS algorithms.

It is possible to specify default properties within a config file. For example the following config file automatically loads the default and fips providers and sets the default property value to be `fips=yes`. Note that this config file does not load the ''base'' provider. All supporting algorithms that are in ''base'' are also in ''default'', so it is unnecessary in this case:

```
config_diagnostics = 1
openssl_conf = openssl_init

.include /usr/local/ssl/fipsmodule.cnf

[openssl_init]
providers = provider_sect
alg_section = algorithm_sect

[provider_sect]
fips = fips_sect
default = default_sect

[default_sect]
activate = 1

[algorithm_sect]
default_properties = fips=yes
```

Programmatically loading the FIPS module (nondefault library context)

In addition to using properties to separate usage of the FIPS module from other usages this can also be achieved using library contexts. In this example we create two library contexts. In one we assume the existence of a config file called *openssl-fips.cnf* that automatically loads and configures the FIPS and base providers. The other library context will just use the default provider.

```
OSSL_LIB_CTX *fips_libctx, *nonfips_libctx;
OSSL_PROVIDER *defctxnull = NULL;
EVP_MD *fipssha256 = NULL, *nonfipssha256 = NULL;
int ret = 1;

/*
```

```
 * Create two nondefault library contexts. One for fips usage and
 * one for non-fips usage
 */
fips_libctx = OSSL_LIB_CTX_new();
nonfips_libctx = OSSL_LIB_CTX_new();
if (fips_libctx == NULL || nonfips_libctx == NULL)
    goto err;

/* Prevent anything from using the default library context */
defctxnull = OSSL_PROVIDER_load(NULL, "null");

/*
 * Load config file for the FIPS library context. We assume that
 * this config file will automatically activate the FIPS and base
 * providers so we don't need to explicitly load them here.
 */
if (!OSSL_LIB_CTX_load_config(fips_libctx, "openssl-fips.cnf"))
    goto err;

/*
 * We don't need to do anything special to load the default
 * provider into nonfips_libctx. This happens automatically if no
 * other providers are loaded.
 * Because we don't call OSSL_LIB_CTX_load_config() explicitly for
 * nonfips_libctx it will just use the default config file.
 */

/* As an example get some digests */

/* Get a FIPS validated digest */
fipssha256 = EVP_MD_fetch(fips_libctx, "SHA2-256", NULL);
if (fipssha256 == NULL)
    goto err;

/* Get a non-FIPS validated digest */
nonfipssha256 = EVP_MD_fetch(nonfips_libctx, "SHA2-256", NULL);
if (nonfipssha256 == NULL)
    goto err;

/* Use the digests */

printf("Success\n");
ret = 0;

err:
EVP_MD_free(fipssha256);
EVP_MD_free(nonfipssha256);
OSSL_LIB_CTX_free(fips_libctx);
OSSL_LIB_CTX_free(nonfips_libctx);
OSSL_PROVIDER_unload(defctxnull);

return ret;
```

Note that we have made use of the special "null" provider here which we load into the default library context. We could have chosen to use the default library context for FIPS usage, and just create one

additional library context for other usages − or vice versa. However if code has not been converted to use library contexts then the default library context will be automatically used. This could be the case for your own existing applications as well as certain parts of OpenSSL itself. Not all parts of OpenSSL are library context aware. If this happens then you could ‘‘accidentally’’ use the wrong library context for a particular operation. To be sure this doesn’t happen you can load the ‘‘null’’ provider into the default library context. Because a provider has been explicitly loaded, the default provider will not automatically load. This means code using the default context by accident will fail because no algorithms will be available.

See ‘‘Library Context’’ in **migration_guide** (7) for additional information about the Library Context.

Using Encoders and Decoders with the FIPS module

Encoders and decoders are used to read and write keys or parameters from or to some external format (for example a PEM file). If your application generates keys or parameters that then need to be written into PEM or DER format then it is likely that you will need to use an encoder to do this. Similarly you need a decoder to read previously saved keys and parameters. In most cases this will be invisible to you if you are using APIs that existed in OpenSSL 1.1.1 or earlier such as **i2d_PrivateKey** (3). However the appropriate encoder/decoder will need to be available in the library context associated with the key or parameter object. The built-in OpenSSL encoders and decoders are implemented in both the default and base providers and are not in the FIPS module boundary. However since they are not cryptographic algorithms themselves it is still possible to use them in conjunction with the FIPS module, and therefore these encoders/decoders have the `fips=yes` property against them. You should ensure that either the default or base provider is loaded into the library context in this case.

Using the FIPS module in SSL/TLS

Writing an application that uses libssl in conjunction with the FIPS module is much the same as writing a normal libssl application. If you are using global properties and the default library context to specify usage of FIPS validated algorithms then this will happen automatically for all cryptographic algorithms in libssl. If you are using a nondefault library context to load the FIPS provider then you can supply this to libssl using the function **SSL_CTX_new_ex** (3). This works as a drop in replacement for the function **SSL_CTX_new** (3) except it provides you with the capability to specify the library context to be used. You can also use the same function to specify libssl specific properties to use.

In this first example we create two SSL_CTX objects using two different library contexts.

```
/*
 * We assume that a nondefault library context with the FIPS
 * provider loaded has been created called fips_libctx.
 */
SSL_CTX *fips_ssl_ctx = SSL_CTX_new_ex(fips_libctx, NULL, TLS_method());
/*
 * We assume that a nondefault library context with the default
 * provider loaded has been created called non_fips_libctx.
 */
SSL_CTX *non_fips_ssl_ctx = SSL_CTX_new_ex(non_fips_libctx, NULL,
                                           TLS_method());
```

In this second example we create two SSL_CTX objects using different properties to specify FIPS usage:

```
/*
 * The "fips=yes" property includes all FIPS approved algorithms
 * as well as encoders from the default provider that are allowed
 * to be used. The NULL below indicates that we are using the
 * default library context.
 */
SSL_CTX *fips_ssl_ctx = SSL_CTX_new_ex(NULL, "fips=yes", TLS_method());
/*
 * The "provider!=fips" property allows algorithms from any
 * provider except the FIPS provider
 */
```

```
SSL_CTX *non_fips_ssl_ctx = SSL_CTX_new_ex(NULL, "provider!=fips",
                                           TLS_method());
```

Confirming that an algorithm is being provided by the FIPS module

A chain of links needs to be followed to go from an algorithm instance to the provider that implements it. The process is similar for all algorithms. Here the example of a digest is used.

To go from an **EVP_MD_CTX** to an **EVP_MD**, use **EVP_MD_CTX_md** (3) . To go from the **EVP_MD** to its **OSSL_PROVIDER**, use **EVP_MD_get0_provider** (3). To extract the name from the **OSSL_PROVIDER**, use **OSSL_PROVIDER_get0_name** (3).

SEE ALSO

migration_guide (7), **crypto** (7), **fips_config** (5)

COPYRIGHT

Copyright 2021 The OpenSSL Project Authors. All Rights Reserved.

Licensed under the Apache License 2.0 (the ‘‘License’’). You may not use this file except in compliance with the License. You can obtain a copy in the file LICENSE in the source distribution or at <https://www.openssl.org/source/license.html>.