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 'provider-base.7ossl' command

$ man provider-base.7ossl

PROVIDER-BASE(7ossl) OpenSSL PROVIDER-BASE(7ossl)

NAME

provider-base - The basic OpenSSL 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.
 */
/* Functions offered by libcrypto to the providers */
const OSSL_ITEM *core_gettable_params(const OSSL_CORE_HANDLE *handle);
int core_get_params(const OSSL_CORE_HANDLE *handle, OSSL_PARAM params[]);
typedef void (*OSSL_thread_stop_handler_fn)(void *arg);
int core_thread_start(const OSSL_CORE_HANDLE *handle,
                      OSSL_thread_stop_handler_fn handfn,
                      void *arg);
OPENSSL_CORE_CTX *core_get_libctx(const OSSL_CORE_HANDLE *handle);
void core_new_error(const OSSL_CORE_HANDLE *handle);
void core_set_error_debug(const OSSL_CORE_HANDLE *handle,
                          const char *file, int line, const char *func);
void core_vset_error(const OSSL_CORE_HANDLE *handle,
                     uint32_t reason, const char *fmt, va_list args);
```

```
int core_obj_add_sigid(const OSSL_CORE_HANDLE *prov, const char  *sign_name,
                       const char *digest_name, const char *pkey_name);
int core_obj_create(const OSSL_CORE_HANDLE *handle, const char *oid,
                    const char *sn, const char *ln);
/*
 * Some OpenSSL functionality is directly offered to providers via
 * dispatch
 */
void *CRYPTO_malloc(size_t num, const char *file, int line);
void *CRYPTO_zalloc(size_t num, const char *file, int line);
void CRYPTO_free(void *ptr, const char *file, int line);
void CRYPTO_clear_free(void *ptr, size_t num,
                       const char *file, int line);
void *CRYPTO_realloc(void *addr, size_t num,
                     const char *file, int line);
void *CRYPTO_clear_realloc(void *addr, size_t old_num, size_t num,
                           const char *file, int line);
void *CRYPTO_secure_malloc(size_t num, const char *file, int line);
void *CRYPTO_secure_zalloc(size_t num, const char *file, int line);
void CRYPTO_secure_free(void *ptr, const char *file, int line);
void CRYPTO_secure_clear_free(void *ptr, size_t num,
                              const char *file, int line);
int CRYPTO_secure_allocated(const void *ptr);
void OPENSSL_cleanse(void *ptr, size_t len);
unsigned char *OPENSSL_hexstr2buf(const char *str, long *buflen);
OSSL_CORE_BIO *BIO_new_file(const char *filename, const char *mode);
OSSL_CORE_BIO *BIO_new_membuf(const void *buf, int len);
int BIO_read_ex(OSSL_CORE_BIO *bio, void *data, size_t data_len,
                size_t *bytes_read);
int BIO_write_ex(OSSL_CORE_BIO *bio, const void *data, size_t data_len,
                 size_t *written);
int BIO_up_ref(OSSL_CORE_BIO *bio);
int BIO_free(OSSL_CORE_BIO *bio);
```

```
int BIO_vprintf(OSSL_CORE_BIO *bio, const char *format, va_list args);
int BIO_vsnprintf(char *buf, size_t n, const char *fmt, va_list args);
void OSSL_SELF_TEST_set_callback(OSSL_LIB_CTX *libctx, OSSL_CALLBACK *cb,
                                 void *cbarg);
size_t get_entropy(const OSSL_CORE_HANDLE *handle,
                   unsigned char **pout, int entropy,
                   size_t min_len, size_t max_len);
void cleanup_entropy(const OSSL_CORE_HANDLE *handle,
                     unsigned char *buf, size_t len);
size_t get_nonce(const OSSL_CORE_HANDLE *handle,
                 unsigned char **pout, size_t min_len, size_t max_len,
                 const void *salt, size_t salt_len);
void cleanup_nonce(const OSSL_CORE_HANDLE *handle,
                   unsigned char *buf, size_t len);
/* Functions for querying the providers in the application library context */
int provider_register_child_cb(const OSSL_CORE_HANDLE *handle,
                   int (*create_cb)(const OSSL_CORE_HANDLE *provider,
                                    void *cbdata),
                   int (*remove_cb)(const OSSL_CORE_HANDLE *provider,
                                    void *cbdata),
                   int (*global_props_cb)(const char *props, void *cbdata),
                   void *cbdata);
void provider_deregister_child_cb(const OSSL_CORE_HANDLE *handle);
const char *provider_name(const OSSL_CORE_HANDLE *prov);
void *provider_get0_provider_ctx(const OSSL_CORE_HANDLE *prov);
const OSSL_DISPATCH *provider_get0_dispatch(const OSSL_CORE_HANDLE *prov);
int provider_up_ref(const OSSL_CORE_HANDLE *prov, int activate);
int provider_free(const OSSL_CORE_HANDLE *prov, int deactivate);
/* Functions offered by the provider to libcrypto */
void provider_teardown(void *provctx);
const OSSL_ITEM *provider_gettable_params(void *provctx);
int provider_get_params(void *provctx, OSSL_PARAM params[]);
const OSSL_ALGORITHM *provider_query_operation(void *provctx,
```

```
                                    int operation_id,
                                    const int *no_store);
    void provider_unquery_operation(void *provctx, int operation_id,
                                    const OSSL_ALGORITHM *algs);
    const OSSL_ITEM *provider_get_reason_strings(void *provctx);
    int provider_get_capabilities(void *provctx, const char *capability,
                                  OSSL_CALLBACK *cb, void *arg);
    int provider_self_test(void *provctx);
```

DESCRIPTION

All "functions" mentioned here are passed as function pointers between libcrypto and the provider in OSSL_DISPATCH arrays, in the call of the provider initialization function. See "Provider" in provider(7) for a description of the initialization function. They are known as "upcalls".

All these "functions" have a corresponding function type definition named OSSL_FUNC_{name}_fn, and a helper function to retrieve the function pointer from a OSSL_DISPATCH element named OSSL_FUNC_{name}.

For example, the "function" core_gettable_params() has these:

```
typedef OSSL_PARAM *
    (OSSL_FUNC_core_gettable_params_fn)(const OSSL_CORE_HANDLE *handle);
static ossl_inline OSSL_NAME_core_gettable_params_fn
    OSSL_FUNC_core_gettable_params(const OSSL_DISPATCH *opf);
```

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

For in (the OSSL_DISPATCH array passed from libcrypto to the provider):

```
core_gettable_params        OSSL_FUNC_CORE_GETTABLE_PARAMS
core_get_params             OSSL_FUNC_CORE_GET_PARAMS
core_thread_start           OSSL_FUNC_CORE_THREAD_START
core_get_libctx             OSSL_FUNC_CORE_GET_LIBCTX
core_new_error              OSSL_FUNC_CORE_NEW_ERROR
core_set_error_debug        OSSL_FUNC_CORE_SET_ERROR_DEBUG
core_vset_error             OSSL_FUNC_CORE_VSET_ERROR
core_obj_add_sigid          OSSL_FUNC_CORE_OBJ_ADD_SIGID
```

core_obj_create OSSL_FUNC_CORE_OBJ_CREATE

CRYPTO_malloc OSSL_FUNC_CRYPTO_MALLOC

CRYPTO_zalloc OSSL_FUNC_CRYPTO_ZALLOC

CRYPTO_free OSSL_FUNC_CRYPTO_FREE

CRYPTO_clear_free OSSL_FUNC_CRYPTO_CLEAR_FREE

CRYPTO_realloc OSSL_FUNC_CRYPTO_REALLOC

CRYPTO_clear_realloc OSSL_FUNC_CRYPTO_CLEAR_REALLOC

CRYPTO_secure_malloc OSSL_FUNC_CRYPTO_SECURE_MALLOC

CRYPTO_secure_zalloc OSSL_FUNC_CRYPTO_SECURE_ZALLOC

CRYPTO_secure_free OSSL_FUNC_CRYPTO_SECURE_FREE

CRYPTO_secure_clear_free OSSL_FUNC_CRYPTO_SECURE_CLEAR_FREE

CRYPTO_secure_allocated OSSL_FUNC_CRYPTO_SECURE_ALLOCATED

BIO_new_file OSSL_FUNC_BIO_NEW_FILE

BIO_new_mem_buf OSSL_FUNC_BIO_NEW_MEMBUF

BIO_read_ex OSSL_FUNC_BIO_READ_EX

BIO_write_ex OSSL_FUNC_BIO_WRITE_EX

BIO_up_ref OSSL_FUNC_BIO_UP_REF

BIO_free OSSL_FUNC_BIO_FREE

BIO_vprintf OSSL_FUNC_BIO_VPRINTF

BIO_vsnprintf OSSL_FUNC_BIO_VSNPRINTF

BIO_puts OSSL_FUNC_BIO_PUTS

BIO_gets OSSL_FUNC_BIO_GETS

BIO_ctrl OSSL_FUNC_BIO_CTRL

OPENSSL_cleanse OSSL_FUNC_OPENSSL_CLEANSE

OSSL_SELF_TEST_set_callback OSSL_FUNC_SELF_TEST_CB

ossl_rand_get_entropy OSSL_FUNC_GET_ENTROPY

ossl_rand_cleanup_entropy OSSL_FUNC_CLEANUP_ENTROPY

ossl_rand_get_nonce OSSL_FUNC_GET_NONCE

ossl_rand_cleanup_nonce OSSL_FUNC_CLEANUP_NONCE

provider_register_child_cb OSSL_FUNC_PROVIDER_REGISTER_CHILD_CB

provider_deregister_child_cb OSSL_FUNC_PROVIDER_DEREGISTER_CHILD_CB

provider_name OSSL_FUNC_PROVIDER_NAME

provider_get0_provider_ctx OSSL_FUNC_PROVIDER_GET0_PROVIDER_CTX

provider_get0_dispatch OSSL_FUNC_PROVIDER_GET0_DISPATCH

provider_up_ref OSSL_FUNC_PROVIDER_UP_REF

provider_free OSSL_FUNC_PROVIDER_FREE

For *out (the OSSL_DISPATCH array passed from the provider to libcrypto):

provider_teardown OSSL_FUNC_PROVIDER_TEARDOWN

provider_gettable_params OSSL_FUNC_PROVIDER_GETTABLE_PARAMS

provider_get_params OSSL_FUNC_PROVIDER_GET_PARAMS

provider_query_operation OSSL_FUNC_PROVIDER_QUERY_OPERATION

provider_unquery_operation OSSL_FUNC_PROVIDER_UNQUERY_OPERATION

provider_get_reason_strings OSSL_FUNC_PROVIDER_GET_REASON_STRINGS

provider_get_capabilities OSSL_FUNC_PROVIDER_GET_CAPABILITIES

provider_self_test OSSL_FUNC_PROVIDER_SELF_TEST

Core functions

core_gettable_params() returns a constant array of descriptor OSSL_PARAM, for parameters that core_get_params() can handle.

core_get_params() retrieves parameters from the core for the given handle. See "Core parameters" below for a description of currently known parameters.

The core_thread_start() function informs the core that the provider has stated an interest in the current thread. The core will inform the provider when the thread eventually stops. It must be passed the handle for this provider, as well as a callback handfn which will be called when the thread stops. The callback will subsequently be called, with the supplied argument arg, from the thread that is stopping and gets passed the provider context as an argument. This may be useful to perform thread specific clean up such as freeing thread local variables.

core_get_libctx() retrieves the core context in which the library object for the current provider is stored, accessible through the handle. This function is useful only for built-in providers such as the default provider. Never cast this to OSSL_LIB_CTX in a provider that is not built-in as the OSSL_LIB_CTX of the library loading the

provider might be a completely different structure than the OSSL_LIB_CTX of the library the provider is linked to. Use OSSL_LIB_CTX_new_child(3) instead to obtain a proper library context that is linked to the application library context.

core_new_error(), core_set_error_debug() and core_vset_error() are building blocks for reporting an error back to the core, with reference to the handle.

core_new_error()

allocates a new thread specific error record.

This corresponds to the OpenSSL function ERR_new(3).

core_set_error_debug()

sets debugging information in the current thread specific error record. The debugging information includes the name of the file file, the line line and the function name func where the error occurred.

This corresponds to the OpenSSL function ERR_set_debug(3).

core_vset_error()

sets the reason for the error, along with any addition data. The reason is a number defined by the provider and used to index the reason strings table that's returned by provider_get_reason_strings(). The additional data is given as a format string fmt and a set of arguments args, which are treated in the same manner as with BIO_vsnprintf(). file and line may also be passed to indicate exactly where the error occurred or was reported.

This corresponds to the OpenSSL function ERR_vset_error(3).

The core_obj_create() function registers a new OID and associated short name sn and long name ln for the given handle. It is similar to the OpenSSL function OBJ_create(3) except that it returns 1 on success or 0 on failure. It will treat as success the case where the OID already exists (even if the short name sn or long name ln provided as arguments differ from those associated with the existing OID, in which case the new names are not associated). This function is not thread safe.

The core_obj_add_sigid() function registers a new composite signature algorithm (sign_name) consisting of an underlying signature algorithm (pkey_name) and digest algorithm (digest_name) for the given handle. It assumes that the OIDs for the composite signature algorithm as well as for the underlying signature and digest algorithms are either already known to OpenSSL or have been registered via a call to core_obj_create(). It corresponds to the OpenSSL function OBJ_add_sigid(3), except that the objects are identified by name rather than a numeric NID. Any name (OID, short name or long name) can be used to identify the object. It will treat as success the case where the composite signature algorithm already exists (even if registered against a different underlying signature or digest algorithm). For digest_name, NULL or an empty string is permissible for signature algorithms that do not need a digest to operate correctly. The function returns 1 on success or 0 on failure. This function is not thread safe.

CRYPTO_malloc(), CRYPTO_zalloc(), CRYPTO_free(), CRYPTO_clear_free(), CRYPTO_realloc(), CRYPTO_clear_realloc(), CRYPTO_secure_malloc(), CRYPTO_secure_zalloc(), CRYPTO_secure_free(), CRYPTO_secure_clear_free(), CRYPTO_secure_allocated(), BIO_new_file(), BIO_new_mem_buf(), BIO_read_ex(), BIO_write_ex(), BIO_up_ref(), BIO_free(), BIO_vprintf(), BIO_vsnprintf(), BIO_gets(), BIO_puts(), BIO_ctrl(), OPENSSL_cleanse() and OPENSSL_hexstr2buf() correspond exactly to the public functions with the same name. As a matter of fact, the pointers in the OSSL_DISPATCH array are typically direct pointers to those public functions. Note that the BIO functions take an OSSL_CORE_BIO type rather than the standard BIO type. This is to ensure that a provider does not mix BIOs from the core with BIOs used on the provider side (the two are not compatible).

OSSL_SELF_TEST_set_callback() is used to set an optional callback that can be passed into a provider. This may be ignored by a provider.

get_entropy() retrieves seeding material from the operating system. The seeding material will have at least entropy bytes of randomness and

the output will have at least min_len and at most max_len bytes. The buffer address is stored in *pout and the buffer length is returned to the caller. On error, zero is returned.

cleanup_entropy() is used to clean up and free the buffer returned by get_entropy(). The entropy pointer returned by get_entropy() is passed in buf and its length in len.

get_nonce() retrieves a nonce using the passed salt parameter of length salt_len and operating system specific information. The salt should contain uniquely identifying information and this is included, in an unspecified manner, as part of the output. The output is stored in a buffer which contrains at least min_len and at most max_len bytes. The buffer address is stored in *pout and the buffer length returned to the caller. On error, zero is returned.

cleanup_nonce() is used to clean up and free the buffer returned by get_nonce(). The nonce pointer returned by get_nonce() is passed in buf and its length in len.

provider_register_child_cb() registers callbacks for being informed about the loading and unloading of providers in the application's library context. handle is this provider's handle and cbdata is this provider's data that will be passed back to the callbacks. It returns 1 on success or 0 otherwise. These callbacks may be called while holding locks in libcrypto. In order to avoid deadlocks the callback implementation must not be long running and must not call other OpenSSL API functions or upcalls.

create_cb is a callback that will be called when a new provider is loaded into the application's library context. It is also called for any providers that are already loaded at the point that this callback is registered. The callback is passed the handle being used for the new provider being loadded and this provider's data in cbdata. It should return 1 on success or 0 on failure.

remove_cb is a callback that will be called when a new provider is unloaded from the application's library context. It is passed the handle being used for the provider being unloaded and this provider's

data in cbdata. It should return 1 on success or 0 on failure.

global_props_cb is a callback that will be called when the global properties from the parent library context are changed. It should return 1 on success or 0 on failure.

provider_deregister_child_cb() unregisters callbacks previously registered via provider_register_child_cb(). If provider_register_child_cb() has been called then provider_deregister_child_cb() should be called at or before the point that this provider's teardown function is called.

provider_name() returns a string giving the name of the provider identified by handle.

provider_get0_provider_ctx() returns the provider context that is associated with the provider identified by prov.

provider_get0_dispatch() gets the dispatch table registered by the provider identified by prov when it initialised.

provider_up_ref() increments the reference count on the provider prov. If activate is nonzero then the provider is also loaded if it is not already loaded. It returns 1 on success or 0 on failure.

provider_free() decrements the reference count on the provider prov. If deactivate is nonzero then the provider is also unloaded if it is not already loaded. It returns 1 on success or 0 on failure.

Provider functions

provider_teardown() is called when a provider is shut down and removed from the core's provider store. It must free the passed provctx.

provider_gettable_params() should return a constant array of descriptor OSSL_PARAM, for parameters that provider_get_params() can handle.

provider_get_params() should process the OSSL_PARAM array params, setting the values of the parameters it understands.

provider_query_operation() should return a constant OSSL_ALGORITHM that corresponds to the given operation_id. It should indicate if the core may store a reference to this array by setting *no_store to 0 (core may store a reference) or 1 (core may not store a reference).

provider_unquery_operation() informs the provider that the result of a

provider_query_operation() is no longer directly required and that the function pointers have been copied. The operation_id should match that passed to provider_query_operation() and algs should be its return value.

provider_get_reason_strings() should return a constant OSSL_ITEM array that provides reason strings for reason codes the provider may use when reporting errors using core_put_error().

The provider_get_capabilities() function should call the callback cb passing it a set of OSSL_PARAMs and the caller supplied argument arg. The OSSL_PARAMs should provide details about the capability with the name given in the capability argument relevant for the provider context provctx. If a provider supports multiple capabilities with the given name then it may call the callback multiple times (one for each capability). Capabilities can be useful for describing the services that a provider can offer. For further details see the "CAPABILITIES" section below. It should return 1 on success or 0 on error.

The provider_self_test() function should perform known answer tests on a subset of the algorithms that it uses, and may also verify the integrity of the provider module. It should return 1 on success or 0 on error. It will return 1 if this function is not used.

None of these functions are mandatory, but a provider is fairly useless without at least provider_query_operation(), and provider_gettable_params() is fairly useless if not accompanied by provider_get_params().

Provider parameters

provider_get_params() can return the following provider parameters to the core:

"name" (OSSL_PROV_PARAM_NAME) <UTF8 ptr>

This points to a string that should give a unique name for the provider.

"version" (OSSL_PROV_PARAM_VERSION) <UTF8 ptr>

This points to a string that is a version number associated with this provider. OpenSSL in-built providers use OPENSSL_VERSION_STR,

but this may be different for any third party provider. This string is for informational purposes only.

"buildinfo" (OSSL_PROV_PARAM_BUILDINFO) <UTF8 ptr>

This points to a string that is a build information associated with this provider. OpenSSL in-built providers use OPENSSL_FULL_VERSION_STR, but this may be different for any third party provider.

"status" (OSSL_PROV_PARAM_STATUS) <unsigned integer>

This returns 0 if the provider has entered an error state, otherwise it returns 1.

provider_gettable_params() should return the above parameters.

Core parameters

core_get_params() can retrieve the following core parameters for each provider:

"openssl-version" (OSSL_PROV_PARAM_CORE_VERSION) <UTF8 string ptr>

This points to the OpenSSL libraries' full version string, i.e. the string expanded from the macro OPENSSL_VERSION_STR.

"provider-name" (OSSL_PROV_PARAM_CORE_PROV_NAME) <UTF8 string ptr>

This points to the OpenSSL libraries' idea of what the calling provider is named.

"module-filename" (OSSL_PROV_PARAM_CORE_MODULE_FILENAME) <UTF8 string ptr>

This points to a string containing the full filename of the providers module file.

Additionally, provider specific configuration parameters from the config file are available, in dotted name form. The dotted name form is a concatenation of section names and final config command name separated by periods.

For example, let's say we have the following config example:

```
config_diagnostics = 1
openssl_conf = openssl_init
[openssl_init]
providers = providers_sect
```

```
[providers_sect]
foo = foo_sect
[foo_sect]
activate = 1
data1 = 2
data2 = str
more = foo_more
[foo_more]
data3 = foo,bar
```

The provider will have these additional parameters available:

"activate"

pointing at the string "1"

"data1"

pointing at the string "2"

"data2"

pointing at the string "str"

"more.data3"

pointing at the string "foo,bar"

For more information on handling parameters, see OSSL_PARAM(3) as OSSL_PARAM_int(3).

CAPABILITIES

Capabilities describe some of the services that a provider can offer. Applications can query the capabilities to discover those services.

"TLS-GROUP" Capability

The "TLS-GROUP" capability can be queried by libssl to discover the list of TLS groups that a provider can support. Each group supported can be used for key exchange (KEX) or key encapsulation method (KEM) during a TLS handshake. TLS clients can advertise the list of TLS groups they support in the supported_groups extension, and TLS servers can select a group from the offered list that they also support. In this way a provider can add to the list of groups that libssl already supports with additional ones.

Each TLS group that a provider supports should be described via the

callback passed in through the provider_get_capabilities function. Each
group should have the following details supplied (all are mandatory,
except OSSL_CAPABILITY_TLS_GROUP_IS_KEM):
"tls-group-name" (OSSL_CAPABILITY_TLS_GROUP_NAME) <UTF8 string>
    The name of the group as given in the IANA TLS Supported Groups
    registry
    <https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-8>.
"tls-group-name-internal" (OSSL_CAPABILITY_TLS_GROUP_NAME_INTERNAL)
<UTF8 string>
    The name of the group as known by the provider. This could be the
    same as the "tls-group-name", but does not have to be.
"tls-group-id" (OSSL_CAPABILITY_TLS_GROUP_ID) <unsigned integer>
    The TLS group id value as given in the IANA TLS Supported Groups
    registry.
"tls-group-alg" (OSSL_CAPABILITY_TLS_GROUP_ALG) <UTF8 string>
    The name of a Key Management algorithm that the provider offers and
    that should be used with this group. Keys created should be able to
    support key exchange or key encapsulation method (KEM), as implied
    by the optional OSSL_CAPABILITY_TLS_GROUP_IS_KEM flag. The
    algorithm must support key and parameter generation as well as the
    key/parameter generation parameter, OSSL_PKEY_PARAM_GROUP_NAME. The
    group name given via "tls-group-name-internal" above will be passed
    via OSSL_PKEY_PARAM_GROUP_NAME when libssl wishes to generate
    keys/parameters.
"tls-group-sec-bits" (OSSL_CAPABILITY_TLS_GROUP_SECURITY_BITS)
<unsigned integer>
    The number of bits of security offered by keys in this group. The
    number of bits should be comparable with the ones given in table 2
    and 3 of the NIST SP800-57 document.
"tls-group-is-kem" (OSSL_CAPABILITY_TLS_GROUP_IS_KEM) <unsigned
integer>
    Boolean flag to describe if the group should be used in key
    exchange (KEX) mode (0, default) or in key encapsulation method

(KEM) mode (1).

This parameter is optional: if not specified, KEX mode is assumed as the default mode for the group.

In KEX mode, in a typical Diffie-Hellman fashion, both sides execute keygen then derive against the peer public key. To operate in KEX mode, the group implementation must support the provider functions as described in provider-keyexch(7).

In KEM mode, the client executes keygen and sends its public key, the server executes encapsulate using the client's public key and sends back the resulting ciphertext, finally the client executes decapsulate to retrieve the same shared secret generated by the server's encapsulate. To operate in KEM mode, the group implementation must support the provider functions as described in provider-kem(7).

Both in KEX and KEM mode, the resulting shared secret is then used according to the protocol specification.

"tls-min-tls" (OSSL_CAPABILITY_TLS_GROUP_MIN_TLS) <integer>

"tls-max-tls" (OSSL_CAPABILITY_TLS_GROUP_MAX_TLS) <integer>

"tls-min-dtls" (OSSL_CAPABILITY_TLS_GROUP_MIN_DTLS) <integer>

"tls-max-dtls" (OSSL_CAPABILITY_TLS_GROUP_MAX_DTLS) <integer>

These parameters can be used to describe the minimum and maximum TLS and DTLS versions supported by the group. The values equate to the on-the-wire encoding of the various TLS versions. For example TLSv1.3 is 0x0304 (772 decimal), and TLSv1.2 is 0x0303 (771 decimal). A 0 indicates that there is no defined minimum or maximum. A -1 indicates that the group should not be used in that protocol.

EXAMPLES

This is an example of a simple provider made available as a dynamically loadable module. It implements the fictitious algorithm "FOO" for the fictitious operation "BAR".

```
#include <malloc.h>
#include <openssl/core.h>
```

```
#include <openssl/core_dispatch.h>
/* Errors used in this provider */
#define E_MALLOC       1
static const OSSL_ITEM reasons[] = {
    { E_MALLOC, "memory allocation failure" }.
    { 0, NULL } /* Termination */
};
/*
 * To ensure we get the function signature right, forward declare
 * them using function types provided by openssl/core_dispatch.h
 */
OSSL_FUNC_bar_newctx_fn foo_newctx;
OSSL_FUNC_bar_freectx_fn foo_freectx;
OSSL_FUNC_bar_init_fn foo_init;
OSSL_FUNC_bar_update_fn foo_update;
OSSL_FUNC_bar_final_fn foo_final;
OSSL_FUNC_provider_query_operation_fn p_query;
OSSL_FUNC_provider_get_reason_strings_fn p_reasons;
OSSL_FUNC_provider_teardown_fn p_teardown;
OSSL_provider_init_fn OSSL_provider_init;
OSSL_FUNC_core_put_error *c_put_error = NULL;
/* Provider context */
struct prov_ctx_st {
    OSSL_CORE_HANDLE *handle;
}
/* operation context for the algorithm FOO */
struct foo_ctx_st {
    struct prov_ctx_st *provctx;
    int b;
};
static void *foo_newctx(void *provctx)
{
    struct foo_ctx_st *fooctx = malloc(sizeof(*fooctx));
```

```
    if (fooctx != NULL)
        fooctx->provctx = provctx;
    else
        c_put_error(provctx->handle, E_MALLOC, __FILE__, __LINE__);
    return fooctx;
}
static void foo_freectx(void *fooctx)
{
    free(fooctx);
}
static int foo_init(void *vfooctx)
{
    struct foo_ctx_st *fooctx = vfooctx;
    fooctx->b = 0x33;
}
static int foo_update(void *vfooctx, unsigned char *in, size_t inl)
{
    struct foo_ctx_st *fooctx = vfooctx;
    /* did you expect something serious? */
    if (inl == 0)
        return 1;
    for (; inl-- > 0; in++)
        *in ^= fooctx->b;
    return 1;
}
static int foo_final(void *vfooctx)
{
    struct foo_ctx_st *fooctx = vfooctx;
    fooctx->b = 0x66;
}
static const OSSL_DISPATCH foo_fns[] = {
    { OSSL_FUNC_BAR_NEWCTX, (void (*)(void))foo_newctx },
    { OSSL_FUNC_BAR_FREECTX, (void (*)(void))foo_freectx },
```

```
    { OSSL_FUNC_BAR_INIT, (void (*)(void))foo_init },
    { OSSL_FUNC_BAR_UPDATE, (void (*)(void))foo_update },
    { OSSL_FUNC_BAR_FINAL, (void (*)(void))foo_final },
    { 0, NULL }
};
static const OSSL_ALGORITHM bars[] = {
    { "FOO", "provider=chumbawamba", foo_fns },
    { NULL, NULL, NULL }
};
static const OSSL_ALGORITHM *p_query(void *provctx, int operation_id,
                                     int *no_store)
{
    switch (operation_id) {
    case OSSL_OP_BAR:
        return bars;
    }
    return NULL;
}
static const OSSL_ITEM *p_reasons(void *provctx)
{
    return reasons;
}
static void p_teardown(void *provctx)
{
    free(provctx);
}
static const OSSL_DISPATCH prov_fns[] = {
    { OSSL_FUNC_PROVIDER_TEARDOWN, (void (*)(void))p_teardown },
    { OSSL_FUNC_PROVIDER_QUERY_OPERATION, (void (*)(void))p_query },
    { OSSL_FUNC_PROVIDER_GET_REASON_STRINGS, (void (*)(void))p_reasons },
    { 0, NULL }
};
int OSSL_provider_init(const OSSL_CORE_HANDLE *handle,
```

```
                    const OSSL_DISPATCH *in,
                    const OSSL_DISPATCH **out,
                    void **provctx)
{
    struct prov_ctx_st *pctx = NULL;
    for (; in->function_id != 0; in++)
        switch (in->function_id) {
        case OSSL_FUNC_CORE_PUT_ERROR:
            c_put_error = OSSL_FUNC_core_put_error(in);
            break;
        }
    *out = prov_fns;
    if ((pctx = malloc(sizeof(*pctx))) == NULL) {
        /*
         * ALEA IACTA EST, if the core retrieves the reason table
         * regardless, that string will be displayed, otherwise not.
         */
        c_put_error(handle, E_MALLOC, __FILE__, __LINE__);
        return 0;
    }
    pctx->handle = handle;
    return 1;
}
```

This relies on a few things existing in openssl/core_dispatch.h:

```
#define OSSL_OP_BAR            4711
#define OSSL_FUNC_BAR_NEWCTX      1
typedef void *(OSSL_FUNC_bar_newctx_fn)(void *provctx);
static ossl_inline OSSL_FUNC_bar_newctx(const OSSL_DISPATCH *opf)
{ return (OSSL_FUNC_bar_newctx_fn *)opf->function; }
#define OSSL_FUNC_BAR_FREECTX     2
typedef void (OSSL_FUNC_bar_freectx_fn)(void *ctx);
static ossl_inline OSSL_FUNC_bar_newctx(const OSSL_DISPATCH *opf)
{ return (OSSL_FUNC_bar_freectx_fn *)opf->function; }
```

```
#define OSSL_FUNC_BAR_INIT        3
typedef void *(OSSL_FUNC_bar_init_fn)(void *ctx);
static ossl_inline OSSL_FUNC_bar_init(const OSSL_DISPATCH *opf)
{ return (OSSL_FUNC_bar_init_fn *)opf->function; }
#define OSSL_FUNC_BAR_UPDATE      4
typedef void *(OSSL_FUNC_bar_update_fn)(void *ctx,
                                        unsigned char *in, size_t inl);
static ossl_inline OSSL_FUNC_bar_update(const OSSL_DISPATCH *opf)
{ return (OSSL_FUNC_bar_update_fn *)opf->function; }
#define OSSL_FUNC_BAR_FINAL       5
typedef void *(OSSL_FUNC_bar_final_fn)(void *ctx);
static ossl_inline OSSL_FUNC_bar_final(const OSSL_DISPATCH *opf)
{ return (OSSL_FUNC_bar_final_fn *)opf->function; }
```

SEE ALSO

provider(7)

HISTORY

The concept of providers and everything surrounding them was introduced in OpenSSL 3.0.

COPYRIGHT