

NAME

EVP_RAND, EVP_RAND_fetch, EVP_RAND_free, EVP_RAND_up_ref, EVP_RAND_CTX, EVP_RAND_CTX_new, EVP_RAND_CTX_free, EVP_RAND_instantiate, EVP_RAND_uninstantiate, EVP_RAND_generate, EVP_RAND_reseed, EVP_RAND_nonce, EVP_RAND_enable_locking, EVP_RAND_verify_zeroization, EVP_RAND_get_strength, EVP_RAND_get_state, EVP_RAND_get0_provider, EVP_RAND_CTX_get0_rand, EVP_RAND_is_a, EVP_RAND_get0_name, EVP_RAND_names_do_all, EVP_RAND_get0_description, EVP_RAND_CTX_get_params, EVP_RAND_CTX_set_params, EVP_RAND_do_all_provided, EVP_RAND_get_params, EVP_RAND_gettable_ctx_params, EVP_RAND_settable_ctx_params, EVP_RAND_CTX_gettable_params, EVP_RAND_CTX_settable_params, EVP_RAND_gettable_params, EVP_RAND_STATE_UNINITIALISED, EVP_RAND_STATE_READY, EVP_RAND_STATE_ERROR - EVP_RAND routines

SYNOPSIS

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

```
typedef struct evp_rand_st EVP_RAND;
```

```
typedef struct evp_rand_ctx_st EVP_RAND_CTX;
```

```
EVP_RAND *EVP_RAND_fetch(OSSL_LIB_CTX *libctx, const char *algorithm,
                        const char *properties);
int EVP_RAND_up_ref(EVP_RAND *rand);
void EVP_RAND_free(EVP_RAND *rand);
EVP_RAND_CTX *EVP_RAND_CTX_new(EVP_RAND *rand, EVP_RAND_CTX *parent);
void EVP_RAND_CTX_free(EVP_RAND_CTX *ctx);
EVP_RAND *EVP_RAND_CTX_get0_rand(EVP_RAND_CTX *ctx);
int EVP_RAND_get_params(EVP_RAND *rand, OSSL_PARAM params[]);
int EVP_RAND_CTX_get_params(EVP_RAND_CTX *ctx, OSSL_PARAM params[]);
int EVP_RAND_CTX_set_params(EVP_RAND_CTX *ctx, const OSSL_PARAM params[]);
const OSSL_PARAM *EVP_RAND_gettable_params(const EVP_RAND *rand);
const OSSL_PARAM *EVP_RAND_gettable_ctx_params(const EVP_RAND *rand);
const OSSL_PARAM *EVP_RAND_settable_ctx_params(const EVP_RAND *rand);
const OSSL_PARAM *EVP_RAND_CTX_gettable_params(EVP_RAND_CTX *ctx);
const OSSL_PARAM *EVP_RAND_CTX_settable_params(EVP_RAND_CTX *ctx);
const char *EVP_RAND_get0_name(const EVP_RAND *rand);
const char *EVP_RAND_get0_description(const EVP_RAND *rand);
int EVP_RAND_is_a(const EVP_RAND *rand, const char *name);
const OSSL_PROVIDER *EVP_RAND_get0_provider(const EVP_RAND *rand);
void EVP_RAND_do_all_provided(OSSL_LIB_CTX *libctx,
```

```

        void (*fn)(EVP RAND *rand, void *arg),
        void *arg);
int EVP RAND_names_do_all(const EVP RAND *rand,
        void (*fn)(const char *name, void *data),
        void *data);

int EVP RAND_instantiate(EVP RAND_CTX *ctx, unsigned int strength,
        int prediction_resistance,
        const unsigned char *pstr, size_t pstr_len,
        const OSSL_PARAM params[]);
int EVP RAND_uninstantiate(EVP RAND_CTX *ctx);
int EVP RAND_generate(EVP RAND_CTX *ctx, unsigned char *out, size_t outlen,
        unsigned int strength, int prediction_resistance,
        const unsigned char *addin, size_t addin_len);
int EVP RAND_reseed(EVP RAND_CTX *ctx, int prediction_resistance,
        const unsigned char *ent, size_t ent_len,
        const unsigned char *addin, size_t addin_len);
int EVP RAND_nonce(EVP RAND_CTX *ctx, unsigned char *out, size_t outlen);
int EVP RAND_enable_locking(EVP RAND_CTX *ctx);
int EVP RAND_verify_zeroization(EVP RAND_CTX *ctx);
unsigned int EVP RAND_get_strength(EVP RAND_CTX *ctx);
int EVP RAND_get_state(EVP RAND_CTX *ctx);

#define EVP RAND_STATE_UNINITIALISED  0
#define EVP RAND_STATE_READY          1
#define EVP RAND_STATE_ERROR          2

```

DESCRIPTION

The EVP RAND routines are a high-level interface to random number generators both deterministic and not. If you just want to generate random bytes then you don't need to use these functions: just call **RAND_bytes()** or **RAND_priv_bytes()**. If you want to do more, these calls should be used instead of the older **RAND** and **RAND_DRBG** functions.

After creating a **EVP RAND_CTX** for the required algorithm using **EVP RAND_CTX_new()**, inputs to the algorithm are supplied either by passing them as part of the **EVP RAND_instantiate()** call or using calls to **EVP RAND_CTX_set_params()** before calling **EVP RAND_instantiate()**. Finally, call **EVP RAND_generate()** to produce cryptographically secure random bytes.

Types

EVP RAND is a type that holds the implementation of a **RAND**.

EVP RAND_CTX is a context type that holds the algorithm inputs. **EVP RAND_CTX** structures are reference counted.

Algorithm implementation fetching

EVP RAND_fetch() fetches an implementation of a RAND *algorithm*, given a library context *libctx* and a set of *properties*. See "ALGORITHM FETCHING" in **crypto(7)** for further information.

The returned value must eventually be freed with **EVP RAND_free(3)**.

EVP RAND_up_ref() increments the reference count of an already fetched RAND.

EVP RAND_free() frees a fetched algorithm. NULL is a valid parameter, for which this function is a no-op.

Context manipulation functions

EVP RAND_CTX_new() creates a new context for the RAND implementation *rand*. If not NULL, *parent* specifies the seed source for this implementation. Not all random number generators need to have a seed source specified. If a parent is required, a NULL *parent* will utilise the operating system entropy sources. It is recommended to minimise the number of random number generators that rely on the operating system for their randomness because this is often scarce.

EVP RAND_CTX_free() frees up the context *ctx*. If *ctx* is NULL, nothing is done.

EVP RAND_CTX_get0_rand() returns the **EVP RAND** associated with the context *ctx*.

Random Number Generator Functions

EVP RAND_instantiate() processes any parameters in *params* and then instantiates the RAND *ctx* with a minimum security strength of <strength> and personalisation string *pstr* of length <pstr_len>. If *prediction_resistance* is specified, fresh entropy from a live source will be sought. This call operates as per NIST SP 800-90A and SP 800-90C.

EVP RAND_uninstantiate() uninstantiates the RAND *ctx* as per NIST SP 800-90A and SP 800-90C. Subsequent to this call, the RAND cannot be used to generate bytes. It can only be freed or instantiated again.

EVP RAND_generate() produces random bytes from the RAND *ctx* with the additional input *addin* of length *addin_len*. The bytes produced will meet the security *strength*. If *prediction_resistance* is specified, fresh entropy from a live source will be sought. This call operates as per NIST SP 800-90A and SP 800-90C.

EVP RAND_reseed() reseeds the RAND with new entropy. Entropy *ent* of length *ent_len* bytes can be supplied as can additional input *addin* of length *addin_len* bytes. In the FIPS provider, both are treated as additional input as per NIST SP-800-90Ar1, Sections 9.1 and 9.2. Additional seed material is also drawn from the RAND's parent or the operating system. If *prediction_resistance* is specified, fresh entropy from a live source will be sought. This call operates as per NIST SP 800-90A and SP 800-90C.

EVP RAND_nonce() creates a nonce in *out* of maximum length *outlen* bytes from the RAND *ctx*. The function returns the length of the generated nonce. If *out* is NULL, the length is still returned but no generation takes place. This allows a caller to dynamically allocate a buffer of the appropriate size.

EVP RAND_enable_locking() enables locking for the RAND *ctx* and all of its parents. After this *ctx* will operate in a thread safe manner, albeit more slowly. This function is not itself thread safe if called with the same *ctx* from multiple threads. Typically locking should be enabled before a *ctx* is shared across multiple threads.

EVP RAND_get_params() retrieves details about the implementation *rand*. The set of parameters given with *params* determine exactly what parameters should be retrieved. Note that a parameter that is unknown in the underlying context is simply ignored.

EVP RAND_CTX_get_params() retrieves chosen parameters, given the context *ctx* and its underlying context. The set of parameters given with *params* determine exactly what parameters should be retrieved. Note that a parameter that is unknown in the underlying context is simply ignored.

EVP RAND_CTX_set_params() passes chosen parameters to the underlying context, given a context *ctx*. The set of parameters given with *params* determine exactly what parameters are passed down. Note that a parameter that is unknown in the underlying context is simply ignored. Also, what happens when a needed parameter isn't passed down is defined by the implementation.

EVP RAND_gettable_params() returns an **OSSL_PARAM(3)** array that describes the retrievable and settable parameters. **EVP RAND_gettable_params()** returns parameters that can be used with **EVP RAND_get_params()**.

EVP RAND_gettable_ctx_params() and **EVP RAND_CTX_gettable_params()** return constant **OSSL_PARAM(3)** arrays that describe the retrievable parameters that can be used with **EVP RAND_CTX_get_params()**. **EVP RAND_gettable_ctx_params()** returns the parameters that can be retrieved from the algorithm, whereas **EVP RAND_CTX_gettable_params()** returns the parameters that can be retrieved in the context's current state.

EVP RAND_settable_ctx_params() and **EVP RAND_CTX_settable_params()** return constant

OSSL_PARAM(3) arrays that describe the settable parameters that can be used with **EVP_RAND_CTX_set_params()**. **EVP_RAND_settable_ctx_params()** returns the parameters that can be retrieved from the algorithm, whereas **EVP_RAND_CTX_settable_params()** returns the parameters that can be retrieved in the context's current state.

Information functions

EVP_RAND_get_strength() returns the security strength of the RAND *ctx*.

EVP_RAND_get_state() returns the current state of the RAND *ctx*. States defined by the OpenSSL RNGs are:

- ⊕ **EVP_RAND_STATE_UNINITIALISED**: this RNG is currently uninitialised. The instantiate call will change this to the ready state.
- ⊕ **EVP_RAND_STATE_READY**: this RNG is currently ready to generate output.
- ⊕ **EVP_RAND_STATE_ERROR**: this RNG is in an error state.

EVP_RAND_is_a() returns 1 if *rand* is an implementation of an algorithm that's identifiable with *name*, otherwise 0.

EVP_RAND_get0_provider() returns the provider that holds the implementation of the given *rand*.

EVP_RAND_do_all_provided() traverses all RAND implemented by all activated providers in the given library context *libctx*, and for each of the implementations, calls the given function *fn* with the implementation method and the given *arg* as argument.

EVP_RAND_get0_name() returns the canonical name of *rand*.

EVP_RAND_names_do_all() traverses all names for *rand*, and calls *fn* with each name and *data*.

EVP_RAND_get0_description() returns a description of the rand, meant for display and human consumption. The description is at the discretion of the rand implementation.

EVP_RAND_verify_zeroization() confirms if the internal DRBG state is currently zeroed. This is used by the FIPS provider to support the mandatory self tests.

PARAMETERS

The standard parameter names are:

"state" (**OSSL_RAND_PARAM_STATE**) <integer>

Returns the state of the random number generator.

"strength" (**OSSL_RAND_PARAM_STRENGTH**) <unsigned integer>

Returns the bit strength of the random number generator.

For rands that are also deterministic random bit generators (DRBGs), these additional parameters are recognised. Not all parameters are relevant to, or are understood by all DRBG rands:

"reseed_requests" (**OSSL_DRBG_PARAM_RESEED_REQUESTS**) <unsigned integer>

Reads or set the number of generate requests before reseeding the associated RAND ctx.

"reseed_time_interval" (**OSSL_DRBG_PARAM_RESEED_TIME_INTERVAL**) <integer>

Reads or set the number of elapsed seconds before reseeding the associated RAND ctx.

"max_request" (**OSSL_DRBG_PARAM_RESEED_REQUESTS**) <unsigned integer>

Specifies the maximum number of bytes that can be generated in a single call to `OSSL_FUNC_rand_generate`.

"min_entropylen" (**OSSL_DRBG_PARAM_MIN_ENTROPYLEN**) <unsigned integer>

"max_entropylen" (**OSSL_DRBG_PARAM_MAX_ENTROPYLEN**) <unsigned integer>

Specify the minimum and maximum number of bytes of random material that can be used to seed the DRBG.

"min_noncelen" (**OSSL_DRBG_PARAM_MIN_NONCELEN**) <unsigned integer>

"max_noncelen" (**OSSL_DRBG_PARAM_MAX_NONCELEN**) <unsigned integer>

Specify the minimum and maximum number of bytes of nonce that can be used to seed the DRBG.

"max_perslen" (**OSSL_DRBG_PARAM_MAX_PERSLEN**) <unsigned integer>

"max_adinlen" (**OSSL_DRBG_PARAM_MAX_ADINLEN**) <unsigned integer>

Specify the minimum and maximum number of bytes of personalisation string that can be used with the DRBG.

"reseed_counter" (**OSSL_DRBG_PARAM_RESEED_COUNTER**) <unsigned integer>

Specifies the number of times the DRBG has been seeded or reseeded.

"properties" (**OSSL_RAND_PARAM_PROPERTIES**) <UTF8 string>

"mac" (**OSSL_RAND_PARAM_MAC**) <UTF8 string>

"digest" (**OSSL_RAND_PARAM_DIGEST**) <UTF8 string>

"cipher" (**OSSL RAND PARAM CIPHER**) <UTF8 string>

For RAND implementations that use an underlying computation MAC, digest or cipher, these parameters set what the algorithm should be.

The value is always the name of the intended algorithm, or the properties in the case of **OSSL RAND PARAM PROPERTIES**.

NOTES

The use of a nonzero value for the *prediction_resistance* argument to **EVP RAND instantiate()**, **EVP RAND generate()** or **EVP RAND reseed()** should be used sparingly. In the default setup, this will cause all public and private DRBGs to be reseeded on next use. Since, by default, public and private DRBGs are allocated on a per thread basis, this can result in significant overhead for highly multi-threaded applications. For normal use-cases, the default "reseed_requests" and "reseed_time_interval" thresholds ensure sufficient prediction resistance over time and you can reduce those values if you think they are too high. Explicitly requesting prediction resistance is intended for more special use-cases like generating long-term secrets.

An **EVP RAND_CTX** needs to have locking enabled if it acts as the parent of more than one child and the children can be accessed concurrently. This must be done by explicitly calling **EVP RAND enable_locking()**.

The RAND life-cycle is described in **life_cycle-rand(7)**. In the future, the transitions described there will be enforced. When this is done, it will not be considered a breaking change to the API.

RETURN VALUES

EVP RAND fetch() returns a pointer to a newly fetched **EVP RAND**, or NULL if allocation failed.

EVP RAND get0_provider() returns a pointer to the provider for the RAND, or NULL on error.

EVP RAND_CTX get0_rand() returns a pointer to the **EVP RAND** associated with the context.

EVP RAND get0_name() returns the name of the random number generation algorithm.

EVP RAND up_ref() returns 1 on success, 0 on error.

EVP RAND names_do_all() returns 1 if the callback was called for all names. A return value of 0 means that the callback was not called for any names.

EVP RAND_CTX new() returns either the newly allocated **EVP RAND_CTX** structure or NULL if an error occurred.

EVP RAND_CTX_free() does not return a value.

EVP RAND_nonce() returns the length of the nonce.

EVP RAND_get_strength() returns the strength of the random number generator in bits.

EVP RAND_gettable_params(), **EVP RAND_gettable_ctx_params()** and **EVP RAND_settable_ctx_params()** return an array of **OSSL_PARAMS**.

EVP RAND_verify_zeroization() returns 1 if the internal DRBG state is currently zeroed, and 0 if not.

The remaining functions return 1 for success and 0 or a negative value for failure.

SEE ALSO

RAND_bytes(3), **EVP RAND-CTR-DRBG(7)**, **EVP RAND-HASH-DRBG(7)**,
EVP RAND-HMAC-DRBG(7), **EVP RAND-TEST-RAND(7)**, **provider-rand(7)**, **life_cycle-rand(7)**

HISTORY

This functionality was added to OpenSSL 3.0.

COPYRIGHT

Copyright 2020-2023 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>.