NAME

EVP_RAND − the random bit generator

SYNOPSIS

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

DESCRIPTION

The default OpenSSL RAND method is based on the EVP_RAND classes to provide non-deterministic inputs to other cryptographic algorithms.

While the RAND API is the 'frontend' which is intended to be used by application developers for obtaining random bytes, the EVP_RAND API serves as the 'backend', connecting the former with the operating systems's entropy sources and providing access to deterministic random bit generators (DRBG) and their configuration parameters. A DRBG is a certain type of cryptographically-secure pseudo-random number generator (CSPRNG), which is described in [NIST SP 800−90A Rev. 1].

Disclaimer

Unless you have very specific requirements for your random generator, it is in general not necessary to utilize the EVP_RAND API directly. The usual way to obtain random bytes is to use **RAND_bytes**(3) or **RAND_priv_bytes**(3), see also **RAND**(7).

Typical Use Cases

Typical examples for such special use cases are the following:

- You want to use your own private DRBG instances. Multiple DRBG instances which are accessed only by a single thread provide additional security (because their internal states are independent) and better scalability in multithreaded applications (because they don't need to be locked).
- You need to integrate a previously unsupported entropy source. Refer to **provider−rand**(7) for the implementation details to support adding randomness sources to EVP_RAND.
- You need to change the default settings of the standard OpenSSL RAND implementation to meet specific requirements.

EVP_RAND CHAINING

An EVP_RAND instance can be used as the entropy source of another EVP_RAND instance, provided it has itself access to a valid entropy source. The EVP_RAND instance which acts as entropy source is called the *parent*, the other instance the *child*. Typically, the child will be a DRBG because it does not make sense for the child to be an entropy source.

This is called chaining. A chained EVP_RAND instance is created by passing a pointer to the parent EVP_RAND_CTX as argument to the **EVP_RAND_CTX_new()** call. It is possible to create chains of more than two DRBG in a row. It is also possible to use any EVP_RAND_CTX class as the parent, however, only a live entropy source may ignore and not use its parent.

THE THREE SHARED DRBG INSTANCES

Currently, there are three shared DRBG instances, the <primary>, <public>, and <private> DRBG. While the <primary> DRBG is a single global instance, the <public> and <private> DRBG are created per thread and accessed through thread-local storage.

By default, the functions **RAND_bytes**(3) and **RAND_priv_bytes**(3) use the thread-local <public> and <private> DRBG instance, respectively.

The <primary> DRBG instance

The <primary> DRBG is not used directly by the application, only for reseeding the two other two DRBG instances. It reseeds itself by obtaining randomness either from os entropy sources or by consuming randomness which was added previously by **RAND_add**(3).

The <public> DRBG instance

This instance is used per default by **RAND_bytes**(3).

The <private> DRBG instance

This instance is used per default by **RAND_priv_bytes** (3)

LOCKING

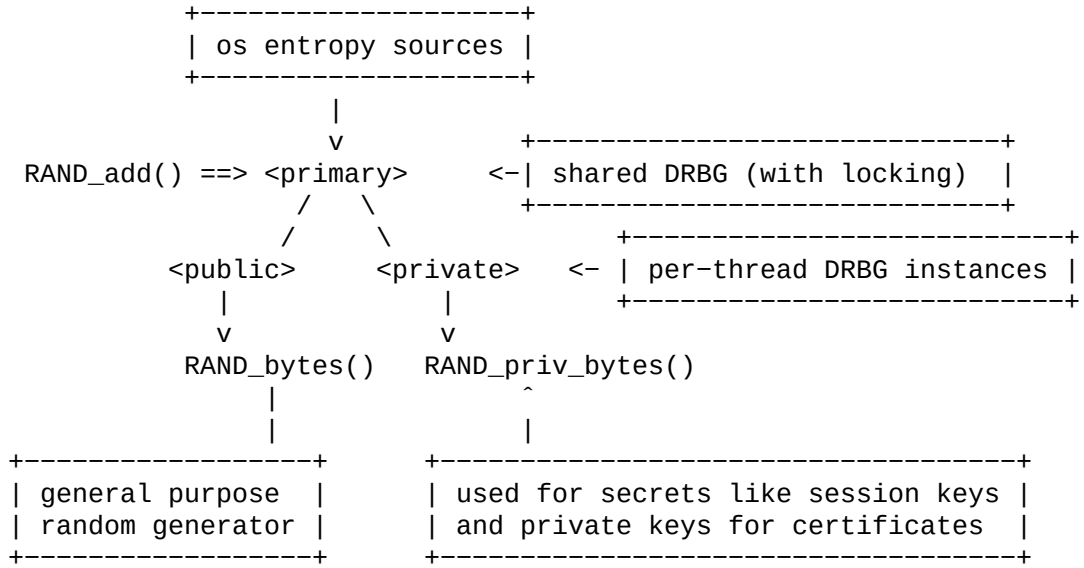
The <primary> DRBG is intended to be accessed concurrently for reseeding by its child DRBG instances. The necessary locking is done internally. It is*not* thread-safe to access the <primary> DRBG directly via the EVP_RAND interface. The <public> and <private> DRBG are thread-local, i.e. there is an instance of each per thread. So they can safely be accessed without locking via the EVP_RAND interface.

Pointers to these DRBG instances can be obtained using **RAND_get0_primary()**, **RAND_get0_public()** and **RAND_get0_private()**, respectively. Note that it is not allowed to store a pointer to one of the thread-local DRBG instances in a variable or other memory location where it will be accessed and used by multiple threads.

All other DRBG instances created by an application don't support locking, because they are intended to be used by a single thread. Instead of accessing a single DRBG instance concurrently from different threads, it is recommended to instantiate a separate DRBG instance per thread. Using the <primary> DRBG as entropy source for multiple DRBG instances on different threads is thread-safe, because the DRBG instance will lock the <primary> DRBG automatically for obtaining random input.

THE OVERALL PICTURE

The following picture gives an overview over how the DRBG instances work together and are being used.



The usual way to obtain random bytes is to call RAND_bytes(...) or RAND_priv_bytes(...). These calls are roughly equivalent to calling EVP_RAND_generate(<public>, ...) and EVP_RAND_generate(<private>, ...), respectively.

RESEEDING

A DRBG instance seeds itself automatically, pulling random input from its entropy source. The entropy source can be either a trusted operating system entropy source, or another DRBG with access to such a source.

Automatic reseeding occurs after a predefined number of generate requests. The selection of the trusted entropy sources is configured at build time using the −−with−rand−seed option. The following sections explain the reseeding process in more detail.

Automatic Reseeding

Before satisfying a generate request (**EVP_RAND_generate** (3)), the DRBG reseeds itself automatically, if one of the following conditions holds:

− the DRBG was not instantiated (=seeded) yet or has been uninstantiated.

− the number of generate requests since the last reseeding exceeds a certain threshold, the so called *reseed_interval*. This behaviour can be disabled by setting the *reseed_interval* to 0.

− the time elapsed since the last reseeding exceeds a certain time interval, the so called *reseed_time_interval*. This can be disabled by setting ther *eseed_time_interval* to 0.

− the DRBG is in an error state.

Note: An error state is entered if the entropy source fails while the DRBG is seeding or reseeding. The last case ensures that the DRBG automatically recovers from the error as soon as the entropy source is available again.

Manual Reseeding

In addition to automatic reseeding, the caller can request an immediate reseeding of the DRBG with fresh entropy by setting the *prediction resistance* parameter to 1 when calling **EVP_RAND_generate** (3).

The document [NIST SP 800−90C] describes prediction resistance requests in detail and imposes strict conditions on the entropy sources that are approved for providing prediction resistance. A request for prediction resistance can only be satisfied by pulling fresh entropy from a live entropy source (section 5.5.2 of [NIST SP 800−90C]). It is up to the user to ensure that a live entropy source is configured and is being used.

For the three shared DRBGs (and only for these) there is another way to reseed them manually: If **RAND_add** (3) is called with a positive *randomness* argument (or **RAND_seed** (3)), then this will immediately reseed the <primary> DRBG. The <public> and <private> DRBG will detect this on their next generate call and reseed, pulling randomness from <primary>.

The last feature has been added to support the common practice used with previous OpenSSL versions to call **RAND_add()** before calling **RAND_bytes()**.

Entropy Input and Additional Data

The DRBG distinguishes two different types of random input: *entropy*, which comes from a trusted source, and *additional input*', which can optionally be added by the user and is considered untrusted. It is possible to add *additional input* not only during reseeding, but also for every generate request.

Configuring the Random Seed Source

In most cases OpenSSL will automatically choose a suitable seed source for automatically seeding and reseeding its <primary> DRBG. In some cases however, it will be necessary to explicitly specify a seed source during configuration, using the −−with−rand−seed option. For more information, see the INSTALL instructions. There are also operating systems where no seed source is available and automatic reseeding is disabled by default.

The following two sections describe the reseeding process of the primary DRBG, depending on whether automatic reseeding is available or not.

Reseeding the primary DRBG with automatic seeding enabled

Calling **RAND_poll()** or **RAND_add()** is not necessary, because the DRBG pulls the necessary entropy from its source automatically. However, both calls are permitted, and do reseed the RNG.

RAND_add() can be used to add both kinds of random input, depending on the value of the *randomness* argument:

randomness == 0:

The random bytes are mixed as additional input into the current state of the DRBG. Mixing in additional input is not considered a full reseeding, hence the reseed counter is not reset.

randomness > 0:

The random bytes are used as entropy input for a full reseeding (resp. reinstantiation) if the DRBG is instantiated (resp. uninstantiated or in an error state). The number of random bits required for reseeding is determined by the security strength of the DRBG. Currently it defaults to 256 bits (32 bytes). It is possible to provide less randomness than required. In this case the missing randomness will be obtained by pulling random input from the trusted entropy sources.

NOTE: Manual reseeding is *not allowed* in FIPS mode, because [NIST SP−800−90Ar1] mandates that entropy *shall not* be provided by the consuming application for instantiation (Section 9.1) or reseeding (Section 9.2). For that reason, the *randomness* argument is ignored and the random bytes provided by the **RAND_add** (3) and **RAND_seed** (3) calls are treated as additional data.

Reseeding the primary DRBG with automatic seeding disabled

Calling **RAND_poll()** will always fail.

RAND_add() needs to be called for initial seeding and periodic reseeding. At least 48 bytes (384 bits) of randomness have to be provided, otherwise the (re−)seeding of the DRBG will fail. This corresponds to one and a half times the security strength of the DRBG. The extra half is used for the nonce during instantiation.

More precisely, the number of bytes needed for seeding depend on the *security strength* of the DRBG, which is set to 256 by default.

SEE ALSO

RAND (7), **EVP_RAND** (3)

HISTORY

This functionality was added in OpenSSL 3.0.

COPYRIGHT