NAME

random − overview of interfaces for obtaining randomness

DESCRIPTION

The kernel random-number generator relies on entropy gathered from device drivers and other sources of environmental noise to seed a cryptographically secure pseudorandom number generator (CSPRNG). It is designed for security, rather than speed.

The following interfaces provide access to output from the kernel CSPRNG:

- The */dev/urandom* and */dev/random* devices, both described in **random**(4). These devices have been present on Linux since early times, and are also available on many other systems.
- The Linux-specific **getrandom**(2) system call, available since Linux 3.17. This system call provides access either to the same source as */dev/urandom* (called the *urandom* source in this page) or to the same source as */dev/random* (called the *random* source in this page). The default is the *urandom* source; the *random* source is selected by specifying the **GRND_RANDOM** flag to the system call. (The **getentropy**(3) function provides a slightly more portable interface on top of **getrandom**(2).)

Initialization of the entropy pool

The kernel collects bits of entropy from the environment. When a sufficient number of random bits has been collected, the entropy pool is considered to be initialized.

Choice of random source

Unless you are doing long-term key generation (and most likely not even then), you probably shouldn't be reading from the */dev/random* device or employing **getrandom**(2) with the **GRND_RANDOM** flag. Instead, either read from the */dev/urandom* device or employ **getrandom**(2) without the **GRND_RANDOM** flag. The cryptographic algorithms used for the*ur andom* source are quite conservative, and so should be sufficient for all purposes.

The disadvantage of **GRND_RANDOM** and reads from */dev/random* is that the operation can block for an indefinite period of time. Furthermore, dealing with the partially fulfilled requests that can occur when using **GRND_RANDOM** or when reading from */dev/random* increases code complexity.

Monte Carlo and other probabilistic sampling applications

Using these interfaces to provide large quantities of data for Monte Carlo simulations or other programs/algorithms which are doing probabilistic sampling will be slow. Furthermore, it is unnecessary, because such applications do not need cryptographically secure random numbers. Instead, use the interfaces described in this page to obtain a small amount of data to seed a user-space pseudorandom number generator for use by such applications.

Comparison between getrandom, /dev/urandom, and /dev/random

The following table summarizes the behavior of the various interfaces that can be used to obtain randomness. **GRND_NONBLOCK** is a flag that can be used to control the blocking behavior of **getrandom**(2). The final column of the table considers the case that can occur in early boot time when the entropy pool is not yet initialized.

Interface	Pool	Blocking behavior	Behavior when pool is not yet ready
/dev/random	Blocking pool	If entropy too low, blocks until there is enough entropy again	Blocks until enough entropy gathered
/dev/urandom	CSPRNG output	Never blocks	Returns output from uninitialized CSPRNG (may be low entropy and unsuitable for cryptography)
getrandom()	Same as */dev/urandom*	Does not block once is pool ready	Blocks until pool ready
getrandom() **GRND_RANDOM**	Same as */dev/random*	If entropy too low, blocks until there is enough entropy again	Blocks until pool ready
getrandom() **GRND_NONBLOCK**	Same as */dev/urandom*	Does not block once is pool ready	**EAGAIN**
getrandom() **GRND_RANDOM** + **GRND_NONBLOCK**	Same as */dev/random*	**EAGAIN** if not enough entropy available	**EAGAIN**

Generating cryptographic keys

The amount of seed material required to generate a cryptographic key equals the effective key size of the key. For example, a 3072-bit RSA or Diffie-Hellman private key has an effective key size of 128 bits (it requires about 2^128 operations to break) so a key generator needs only 128 bits (16 bytes) of seed material from */dev/random*.

While some safety margin above that minimum is reasonable, as a guard against flaws in the CSPRNG algorithm, no cryptographic primitive available today can hope to promise more than 256 bits of security, so if any program reads more than 256 bits (32 bytes) from the kernel random pool per invocation, or per reasonable reseed interval (not less than one minute), that should be taken as a sign that its cryptography is *not* skillfully implemented.

SEE ALSO

getrandom(2), **getauxval**(3), **getentropy**(3), **random**(4), **urandom**(4), **signal**(7)