Rocky Enterprise Linux 9.2 Manual Pages on command 'random.7'

C:\>man random.7

RANDOM(7) Linux Programmer's Manual RANDOM(7)

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 pseudo? random 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 ge? trandom(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 initial? ized.

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 urandom 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 par? tially 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 simula? tions or other programs/algorithms which are doing probabilistic sampling will be slow. Furthermore, it is unnecessary, because such applications do not need cryp? tographically 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 gen? erator 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 initial? ized.

```
????????????????????????????????????????????????????????????????????
?Interface     ? Pool         ? Blocking       ? Behavior when pool ?
?              ?              ? behavior       ? is not yet ready   ?
????????????????????????????????????????????????????????????????????
?/dev/random   ? Blocking     ? If entropy too ? Blocks until       ?
?              ? pool         ? low, blocks    ? enough entropy     ?
?              ?              ? until there is ? gathered           ?
?              ?              ? enough entropy ?                    ?
?              ?              ? again          ?                    ?
```

??

?/dev/urandom ? CSPRNG out? ? Never blocks ? Returns output ?

? ? put ? ? from uninitialized ?

? ? ? ? CSPRNG (may be low ?

? ? ? ? entropy and un? ?

? ? ? ? suitable for cryp? ?

? ? ? ? tography) ?

??

?getrandom() ? Same as ? Does not block ? Blocks until pool ?

? ? /dev/urandom ? once is pool ? ready ?

? ? ? ready ? ?

??

?getrandom() ? Same as ? If entropy too ? Blocks until pool ?

?GRND_RANDOM ? /dev/random ? low, blocks ? ready ?

? ? ? until there is ? ?

? ? ? enough entropy ? ?

? ? ? again ? ?

??

?getrandom() ? Same as ? Does not block ? EAGAIN ?

?GRND_NONBLOCK ? /dev/urandom ? once is pool ? ?

? ? ? ready ? ?

??

?getrandom() ? Same as ? EAGAIN if not ? EAGAIN ?

?GRND_RANDOM + ? /dev/random ? enough entropy ? ?

?GRND_NONBLOCK ? ? available ? ?

??

Generating cryptographic keys

The amount of seed material required to generate a cryptographic key equals the ef? fective 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 in? terval (not less than one minute), that should be taken as a sign that its cryptog? raphy is not skillfully implemented.

SEE ALSO

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

COLOPHON

This page is part of release 5.05 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.