

**NAME**

**getrandom** — random number generation from system entropy

**LIBRARY**

Standard C Library (libc, -lc)

**SYNOPSIS**

```
#include <sys/random.h>

ssize_t
getrandom(void *buf, size_t buflen, unsigned int flags);
```

**DESCRIPTION**

The **getrandom** function fills *buf* with up to *buflen* independent uniform random bytes derived from the system's entropy pool.

The function may block until the system has full entropy, meaning that the system has observed enough noise from physical processes that an adversary cannot predict what state it is in:

- When the system has only partial entropy, the output of **getrandom()** may be predictable.
- When the system has full entropy, the output is fit for use as cryptographic key material.

The *flags* argument may be:

0                   Block until the system entropy pool has full entropy; then generate arbitrarily much data. *Recommended*.

If interrupted by a signal, may fail with EINTR or return a short read. If successful, guaranteed to return at least 256 bytes even if interrupted.

GRND\_INSECURE   Do not block; instead fill *buf* with output derived from whatever is in the system entropy pool so far. Equivalent to reading from `/dev/urandom`; see `rnd(4)`.

If interrupted by a signal, may fail with EINTR or return a short read. If successful, guaranteed to return at least 256 bytes even if interrupted.

Despite the name, this is secure as long as you only do it *after* at least one successful call without GRND\_INSECURE, such as `getrandom(..., 0)` or `getrandom(..., GRND_RANDOM)`, or after reading at least one byte from `/dev/random`.

**WARNING:** If you use GRND\_INSECURE *before* the system has full entropy, the output may enable an adversary to search the possible states of the entropy pool by brute force, and thereby reduce its entropy to zero. Thus, incautious use of GRND\_INSECURE can ruin the security of the whole system.

NetBSD attempts to defend against this threat model by resetting the system's entropy estimate to zero in this event, requiring gathering full entropy again before `/dev/random` or **getrandom()** without GRND\_INSECURE will unblock, but other operating systems may not.

GRND\_RANDOM     Block until the system entropy pool has full entropy; then generate a small amount of data. Equivalent to reading from `/dev/random`; see `rnd(4)`. This is provided mainly for source compatibility with Linux; there is essentially no reason to ever use it.

The flag GNRD\_NONBLOCK may also be included with bitwise-OR, in which case if **getrandom()** would have blocked without GRND\_NONBLOCK, it returns EAGAIN instead.

Adding `GRND_NONBLOCK` to `GRND_INSECURE` has no effect; the combination `GRND_INSECURE|GRND_NONBLOCK` is equivalent to `GRND_INSECURE`, since `GRND_INSECURE` never blocks. The combination `GRND_INSECURE|GRND_RANDOM` is nonsensical and fails with `EINVAL`.

## RETURN VALUES

If successful, `getrandom()` returns the number of bytes stored in *buf*. Otherwise, `getrandom()` returns `-1` and sets *errno*.

Since `getrandom(..., 0)` and `getrandom(..., GRND_INSECURE)` are guaranteed to return at least 256 bytes if successful, it is sufficient to use, e.g.,

```
getrandom(buf, 32, 0) == -1
```

or

```
getrandom(buf, 32, GRND_INSECURE) == -1
```

to detect failure. However, with `GRND_RANDOM`, `getrandom()` may return as little as a single byte if successful.

## EXAMPLES

**Recommended usage.** Generate a key for cryptography:

```
uint8_t secretkey[32];

if (getrandom(secretkey, sizeof secretkey, 0) == -1)
    err(EXIT_FAILURE, "getrandom");
crypto_secretbox_xsalsa20poly1305(..., secretkey);
```

Other idioms for illustration:

- Wait for entropy once, and then generate many keys without waiting:

```
struct { uint8_t key[32]; } user[100];

if (getrandom(NULL, 0, 0) == -1)
    err(EXIT_FAILURE, "getrandom");
for (i = 0; i < 100; i++)
    if (getrandom(user[i].key, sizeof user[i].key,
        GRND_INSECURE) == -1)
        err(EXIT_FAILURE, "getrandom");
```

- Twiddle thumbs while waiting for entropy:

```
uint8_t secretkey[32];

while (getrandom(secretkey, sizeof secretkey, GRND_NONBLOCK)
    == -1) {
    if (errno != EAGAIN)
        err(EXIT_FAILURE, "getrandom");
    twiddle_thumbs();
}
crypto_secretbox_xsalsa20poly1305(..., secretkey);
```

(No examples of `GRND_RANDOM` because it is not useful.)

## ERRORS

- [EAGAIN] The GRND\_NONBLOCK flag was specified, and the system entropy pool does not have full entropy.
- [EINTR] The GRND\_NONBLOCK flag was *not* specified, the system entropy pool does not have full entropy, and the process was interrupted by a signal while waiting.
- [EINVAL] *flags* contains an unrecognized flag or a nonsensical combination of flags.
- [EFAULT] *buf* points outside the allocated address space.

**SEE ALSO**

rnd(4)

**HISTORY**

The **getrandom** system call first appeared in Linux 3.17, and was added to NetBSD 10.0.

**AUTHORS**

The NetBSD implementation of **getrandom** and this man page were written by Taylor R Campbell <riastradh@NetBSD.org>.

**BUGS**

There is no way to multiplex waiting for **getrandom()** with other I/O in `select(2)`, `poll(2)`, or `kqueue(2)`. Instead, you can wait for a read from `/dev/random`; see `rnd(4)`.

GRND\_RANDOM is a little silly.