CPAN Author’s Guide to Random Data for Security (https://security.metacpan.org/docs/guides/random-data-for-security.html)

8 minute read • Robert Rothenberg

Random in general

Any secret token that allows someone to access a resource or perform an action should be generated with a secure random number generator. That includes:

- encryption keys
- message padding and initialisation vectors
- password generation
- password salts and peppers
- authentication tokens
- session ids
- nonces
- captcha strings
- multifactor authentication codes

For any implementation of a protocol that specifies a random value, it is safer to use a secure random number generator, even when the protocol does not explicitly call for that.

The built-in rand (https://perldoc.perl.org/functions/rand) function is not fit for security purposes: it is seeded by only 32-bits (4 bytes), and the output can be predicted easily (https://www.perlmonks.org/?node_id=151595).

A cryptographic-strength pseudo-random number generator (PRNG) won’t improve security if it was seeded with data from `rand`: ultimately the output is of that algorithm still comes from a 32-bit seed.

Good Sources of Random Data

Modern operating systems provide access to random data:

- Linux and BSD variants have special devices like `/dev/random` and `/dev/urandom`.
- Newer Linux and BSD variants have the getrandom(2) (http://man.he.net/man2/getrandom) system call.
- Windows provides a `CryptGenRandom` function in the API.

These sources are easy to access from Perl using several modules, which we discuss below.

Recommended Perl Modules

It's better to use existing and up-to-date modules than to roll your own method for reading or generating random data. The benefits of reducing non-core dependencies are outweighed by potential bugs introduced by duplicating code that needs to be maintained separately.

We are listing a few here that are portable, lightweight and which have good defaults.

Crypt::URandom

One of the simplest to use is Crypt::URandom (https://metacpan.org/pod/Crypt::URandom). It reads from a random data source on a variety of systems, using the `/dev/urandom` device or equivalents on other operating systems, including Windows. Newer versions will also use the `getrandom` or `getentropy` calls on systems that support those calls.

To obtain 256-bits (32 bytes) of data:

```
use Crypt::URandom qw( urandom );

my $bytes = urandom(32);
```

Since this is a wrapper around the operating system's random data source, there is no worry about child processes with the same parent returning the same data (i.e., it is "fork safe").

Crypt::SysRandom

Crypt::SysRandom (https://metacpan.org/pod/Crypt::SysRandom) is a pure-perl module for reading from `/dev/urandom` or the Win32::API (https://metacpan.org/pod/Win32::API) random function call.

To obtain 256-bits (32 bytes) of data:

```
use Crypt::SysRandom 0.006 qw( random_bytes );

my $bytes = random_bytes(32);
```

Likewise, since this is also a wrapper around the operating system's random data source, there is no worry about child processes with the same parent returning the same data (i.e., it is "fork safe").

If Crypt::SysRandom::XS (https://metacpan.org/pod/Crypt::SysRandom::XS) is installed, it will use that to retrieve random bytes from system calls.

Sys::GetRandom

If you are writing code that only runs on Linux or BSD systems with the `getrandom` system call, you can use Sys::GetRandom (https://metacpan.org/pod/Sys::GetRandom), an XS module that calls the function directly. To obtain 256-bits (32 bytes) of data using it:

```
use Sys::GetRandom qw( random_bytes );

my $bytes = random_bytes(32);
```

There is also a pure-Perl version Sys::GetRandom::PP (https://metacpan.org/pod/Sys::GetRandom::PP) that uses the `syscall` function.

Note there are some caveats when using `getrandom` to retrieve more than 256 bytes at a time, as the amount of data returned may be less due to interrupts.

Other modules

There are several older modules on CPAN that have been intentionally omitted from this document:

- They require users to select a source of randomness.

  Many consumers of these modules do not select the source, and in some cases these modules have had insecure defaults.

  For most cases there is no reason to choose a source when there is a standard source of random data provided by modern operating systems.

- They add an unnecessary layer of complexity by wrappers around different sources.

  Some will use a PRNG seeded by random data, which provides no improvement over `/dev/urandom`. These implementations have had less reviewers than the operating systems, and may be less secure.

- They incorrectly label `/dev/urandom` as weaker than `/dev/random`.

  It is important to note that there is a common misconception, as both devices use the same entropy pool and PRNG internally. In newer Linux kernels, `/dev/random` no longer blocks and is an alias for `/dev/urandom`. See Myths about /dev/urandom (https://www.thomas-huehn.com/myths-about-urandom/) for an in-depth discussion of this.

Using Cryptographic Strength PRNGs

It's often faster to use random data to seed a cryptographic strength PRNG (CSPRNG) and pull data from that, than to repeatedly request random data from the operating system.

It's important to note that CSPRNGs are not "less secure". If the algorithm is considered secure, then a PRNG seeded with 256 or more bits is difficult to predict the output of. If the use case is to generate short-lived tokens then it is more than adequate.

Crypt::PRNG

Crypt::PRNG (https://metacpan.org/pod/Crypt::PRNG) is part of the CryptX (https://metacpan.org/dist/CryptX) cryptographic toolkit that uses the libtomcrypt (https://github.com/libtom/libtomcrypt). It will initialise a cryptographic-strength PRNG from `/dev/urandom`, and claims to be thread and fork safe.

It also supports several utility methods for returning base-64 strings, URL-safe base-64 strings, random strings or strings with custom alphabets, random integers and random floating point numbers.

For example,

```
use Crypt::PRNG;

my $bytes = Crypt::PRNG->new->bytes(32);
```

will return 256 bits of random data.

Math::Random::ISAAC

Math::Random::ISAAC (https://metacpan.org/pod/Math::Random::ISAAC) is a small and very fast CSPRNG to generate 32-bit integers and floating point numbers. There is also an XS-version Math::Random::ISAAC::XS (https://metacpan.org/pod/Math::Random::ISAAC::XS).

One caveat of this module is that it needs to be manually seeded by 256 long integers. This is not difficult:

```
use Crypt::URandom qw( urandom );
use Math::Random::ISAAC;

my $rng = Math::Random::ISAAC->new( unpack( "N*", urandom(1024) ) ); # 8192 bits
```

Crypt::OpenSSL::Random

Crypt::OpenSSL::Random (https://metacpan.org/pod/Crypt::OpenSSL::Random) will return bytes from OpenSSL or LibreSSL libraries' pseudo-random number generators.

```
use Crypt::OpenSSL::Random qw( random_bytes );

my $bytes = random_bytes(32)
  or die "not enough randomness";
```

This module requires the OpenSSL or LibreSSL libraries to be installed, which may make it non-portable.

Note that on systems without `/dev/random` device, the random seed may need to be initialised. (There is a `random_status` function that indicates whether there is sufficient seeding.)

Generating IDs, Tokens and Passwords

When generating raw random data for encryption keys or initialisation vectors, a common need is to generate a printable string, for example as

- part of a secret URL or file path,
- a session id, key or token in a cookie
- a random password
- a nonce for a web site's Content Security Policy

The simplest way to convert a string of random bytes into something readable is to use the built-in pack (https://perldoc.perl.org/functions/pack) and unpack (https://perldoc.perl.org/functions/unpack) functions. To convert some data into a string of hex digits, use

```
my $str = unpack("H*", $bytes);
```

or a uuencoded (https://en.wikipedia.org/wiki/Uuencoding) string

```
my $str = pack("u*", $bytes);
```

We can also encode the string using MIME::Base64 (https://metacpan.org/pod/MIME::Base64):

```
use MIME::Base64 qw( encode_base64 );

my $str = encode_base64($bytes);
```

There are times where you may want a more restricted alphabet, such as base-62. There are modules that let you generate random strings with custom alphabets or URL-safe encodings.

Note that returning a message digest of random bytes adds no security. Likewise, mixing random data with other information such as a timestamp or PID is unnecessary and does not improve the security.

Crypt::URandom::Token

Crypt::URandom::Token (https://metacpan.org/pod/Crypt::URandom::Token) will generate strings for a specific alphabet using data from Crypt::URandom directly. For example,

```
use Crypt::URandom::Token qw(urandom_token);

my $token = urandom_token();
```

will return a random printable string such as "tB6DZ9e4s5HHh9yidQvhwNMG0HnOuPztfd95w9hdds5b".

A potential downside of this module is that it reads from the system's random source directly, and may not be as fast as using a PRNG.

Session::Token

Session::Token (https://metacpan.org/pod/Session::Token) is an XS module that seeds the ISAAC PRNG with `/dev/urandom` (or the Windows equivalent) and uses it to generate tokens to a desired length and custom alphabet. For example,

```
use Session::Token;

my $token = Session::Token->new->get;
```

will return a string of mixed-case letters and digits, such as "z5DfxjKRu5dCkA3RRjj3F5".

Caution: There are some caveats about initialising this properly after forking. There are also unreleased bug fixes in the git repository that may affect how it is used.

Crypt::PRNG

As we noted above, Crypt::PRNG (https://metacpan.org/pod/Crypt::PRNG) supports several utility methods for returning base-64 strings, URL-safe base-64 strings, random strings or strings with custom alphabets, random integers and random floating point numbers. For example,

```
use Crypt::PRNG;

my $token = Crypt::PRNG->new->string(22);
```

will return a string of mixed-case letters and digits, such as "y1FpfRQszS72GH4h4zTXov".

UUID::URandom

UUIDs should not be used for security tokens, including session ids. RFC 9562 Security Considerations (https://www.rfc-editor.org/rfc/rfc9562.html#name-security-considerations) says:

> *Implementations SHOULD NOT assume that UUIDs are hard to guess. For example, they MUST NOT be used as security capabilities (identifiers whose mere possession grants access). Discovery of predictability in a random number source will result in a vulnerability.*

The UUID Version 4 is the only type that contains random bits, besides the specific fields required by the UUID format. However, unless you need the data to be a valid UUID, then you should probably be returning a string of hex digits using `unpack` or one of the other modules described above to return random tokens.

UUID::URandom (https://metacpan.org/pod/UUID::URandom) is a wrapper around Crypt::Random to generate UUIDs:

```
use UUID::URandom qw( create_uuid_string );

my $id = create_uuid_string();
```

This module also has a function `create_uuid_hex` that returns the UUID as a hexidecimal string.

References

Far From Random: Three Mistakes From Dart/Flutter's Weak PRNG (https://www.zellic.io/blog/proton-dart-flutter-csprng-prng/), December 2024.

ISAAC, a fast cryptographic random number generator (https://burtleburtle.net/bob/rand/isaacafa.html).

Lattice Reduction: a Toolbox for the Cryptanalyst (https://www.di.ens.fr/~stern/data/St54.pdf), A. Joux and J. Stern, 1994.

Myths about /dev/urandom (https://www.thomas-huehn.com/myths-about-urandom/), March 2014.

Predict Random Numbers (https://www.perlmonks.org/?node_id=151595), Perl Monks, March 2002.

RFC 4086 (https://www.rfc-editor.org/info/rfc4086), June 2005.

RFC 9562 (https://www.rfc-editor.org/rfc/rfc9562.html), May 2024.

License and use of this document

Acknowledgements

Several people have been involved in the development of this document

- Robert Rothenberg (main author)
- Alexander Hartmaier
- H. Merijn Brand
- Leon Timmermans
- Salve J. Nilsen
- Stig Palmquist
- Thibault Duponchelle
- Timothy Legge