

# CPAN Author's Guide to Random Data for Security

Robert Rothenberg

- [CPAN Author's Guide to Random Data for Security](#)
  - [Good Sources of Random Data](#)
    - [Crypt::URandom](#)
    - [Sys::GetRandom](#)
  - [Using Cryptographic Strength PRNGs](#)
    - [Crypt::PRNG](#)
    - [Math::Random::ISAAC](#)
  - [Generating Tokens and Passwords](#)
    - [Crypt::URandom::Token](#)
    - [Session::Token](#)
    - [Crypt::PRNG](#)
  - [References](#)
  - [License and use of this document](#)
    - [Acknowledgements](#)

## CPAN Author's Guide to Random Data for Security

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 tokens
- nonces

The built-in [rand](#) function is not fit for security purposes: it is seeded by only 32-bits (4 bytes), and the [output can be predicted easily](#).

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)` system call.
- Windows provides a `CryptGenRandom` function in the API.

These sources are easy to access from Perl using several modules. We are listing a few here that are lightweight, and which (generally) have good defaults.

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

## Crypt::URandom

The simplest to use, is `Crypt::URandom`. It is a lightweight module that 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”).

It is important to note that there is a common misconception that `/dev/urandom` is insecure. This is untrue, as `/dev/random` and `/dev/urandom` 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](#) for an in-depth discussion of this.

## 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`, 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](#) that uses the `syscall` function.

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

## 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](#) is part of the [CryptX](#) cryptographic toolkit that uses the [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](#) 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](#).

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
```

## Generating 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](#) and [unpack](#) functions. To convert some data into a string of hex digits, use

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

or a [uuencoded](#) string

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

We can also encode the string using [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](#) 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 “tB6DZ9e4s5HHh9yidQvhwNMG0HnOuPztf95w9hdds5b”.

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](#) 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](#) 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”.

## References

[Far From Random: Three Mistakes From Dart/Flutter’s Weak PRNG](#), December 2024.

[ISAAC](#), a fast cryptographic random number generator.

[Lattice Reduction: a Toolbox for the Cryptanalyst](#), A. Joux and J. Stern, 1994.

[Myths about /dev/urandom](#), March 2014.

[RFC 4086](#), June 2005.

[Predict Random Numbers](#), Perl Monks, March 2002.

## License and use of this document

- Version: 0.1.4
- License: [CC-BY-SA-4.0](#)
- Copyright: © Robert Rothenberg [rrwo@cpan.org](mailto:rrwo@cpan.org), Some rights reserved.

You may use, modify and share this file under the terms of the [CC-BY-SA-4.0](#) license.



# Acknowledgements

Several people have been involved in the development of this document

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

---

CPAN Security Outreach &  
Information Project

 [CPAN-Security](#)  
 [cpansec](#)

CPAN Security Group (CPANSec) 