Secure random values

> This article was originally published at https://gist.github.com/joepie91/7105003c3b26e65efcea63f3db82dfba.

Not all random values are created equal - for security-related code, you need a *specific kind* of random value.

A summary of this article, if you don't want to read the entire thing:

- **Don't use `Math.random()`.** There are *extremely* few cases where `Math.random()` is the right answer. Don't use it, unless you've read this *entire* article, and determined that it's necessary for your case.
- **Don't use `crypto.getRandomBytes` directly.** While it's a CSPRNG, it's easy to bias the result when 'transforming' it, such that the output becomes more predictable.
- **If you want to generate random tokens or API keys:** Use `uuid`, specifically the `uuid.v4()` method. Avoid `node-uuid` - it's not the same package, and doesn't produce reliably secure random values.
- **If you want to generate random numbers in a range:** Use `random-number-csprng`.

You should seriously consider reading the entire article, though - it's not *that* long :)

Types of "random"

There exist roughly three types of "random":

- **Truly random:** Exactly as the name describes. True randomness, to which no pattern or algorithm applies. It's debatable whether this really exists.
- **Unpredictable:** Not *truly* random, but impossible for an attacker to predict. This is what you need for security-related code - it doesn't matter *how* the data is generated, as long as it can't be guessed.
- **Irregular:** This is what most people think of when they think of "random". An example is a game with a background of a star field, where each star is drawn in a "random" position on the screen. This isn't truly random, and it isn't even unpredictable - it just doesn't *look* like there's a pattern to it, visually.

Irregular data is fast to generate, but utterly worthless for security purposes - even if it doesn't seem like there's a pattern, there is almost always a way for an attacker to predict what the values are going to be. The only realistic usecase for irregular data is things that are represented visually, such as game elements or randomly generated phrases on a joke site.

Unpredictable data is a bit slower to generate, but still fast enough for most cases, and it's sufficiently hard to guess that it will be attacker-resistant. Unpredictable data is provided by what's called a **CSPRNG**.

Types of RNGs (Random Number Generators)

- **CSPRNG:** A *Cryptographically Secure Pseudo-Random Number Generator*. This is what produces *unpredictable* data that you need for security purposes.
- **PRNG:** A *Pseudo-Random Number Generator*. This is a broader category that includes CSPRNGs *and* generators that just return irregular values - in other words, you *cannot* rely on a PRNG to provide you with unpredictable values.
- **RNG:** A *Random Number Generator*. The meaning of this term depends on the context. Most people use it as an even *broader* category that includes PRNGs and *truly* random number generators.

Every random value that you need for security-related purposes (ie. anything where there exists the possibility of an "attacker"), should be generated using a **CSPRNG**. This includes verification tokens, reset tokens, lottery numbers, API keys, generated passwords, encryption keys, and so on, and so on.

Bias

In Node.js, the most widely available CSPRNG is the `crypto.randomBytes` function, but *you shouldn't use this directly*, as it's easy to mess up and "bias" your random values - that is, making it more likely that a specific value or set of values is picked.

A common example of this mistake is using the `%` modulo operator when you have less than 256 possibilities (since a single byte has 256 possible values). Doing so actually makes lower values *more likely* to be picked than higher values.

For example, let's say that you have 36 possible random values - `0-9` plus every lowercase letter in `a-z`. A naive implementation might look something like this:

```
let randomCharacter = randomByte % 36;
```

That code is broken and insecure. With the code above, you essentially create the following ranges (all inclusive):

- **0-35** stays 0-35.
- **36-71** becomes 0-35.
- **72-107** becomes 0-35.

- **108-143** becomes 0-35.
- **144-179** becomes 0-35.
- **180-215** becomes 0-35.
- **216-251** becomes 0-35.
- **252-255** becomes *0-3*.

If you look at the above list of ranges you'll notice that while there are **7 possible values** for each `randomCharacter` between 4 and 35 (inclusive), there are **8 possible values** for each `randomCharacter` between 0 and 3 (inclusive). This means that while there's a **2.64% chance** of getting a value between 4 and 35 (inclusive), there's a **3.02% chance** of getting a value between 0 and 3 (inclusive).

This kind of difference may *look* small, but it's an easy and effective way for an attacker to reduce the amount of guesses they need when bruteforcing something. And this is only *one* way in which you can make your random values insecure, despite them originally coming from a secure random source.

So, how do I obtain random values securely?

In Node.js:

- **If you need a sequence of random bytes:** Use `crypto.randomBytes`.
- **If you need individual random numbers in a certain range:** use `crypto.randomInt`.
- **If you need a random string:** You have two good options here, depending on your needs.
  1. Use a v4 UUID. Safe ways to generate this are `crypto.randomUUID`, and the `uuid` library (only the v4 variant!).
  2. Use a nanoid, using the `nanoid` library. This also allows specifying a custom alphabet to use for your random string.

Both of these use a CSPRNG, and 'transform' the bytes in an unbiased (ie. secure) way.

In the browser:

- When using the Node.js options, your bundler *should* automatically select equivalently safe browser implementations for all of these.
- If not using a bundler:
  - **If you need a sequence of random bytes:** Use `crypto.getRandomValues` with a `Uint8Array`. Other array types will get you numbers in different ranges.
  - **If you need a random string:** You have two good options here, depending on your needs.

1. Use a v4 UUID, with the `crypto.randomUUID` method.
2. Use a nanoid, using the **standalone build** of the `nanoid` library. This also allows specifying a custom alphabet to use for your random string.

However, it is **strongly** recommended that you use a bundler, in general.

Revision #1
Created 11 December 2024 01:09:55 by joepie91
Updated 11 December 2024 18:44:19 by joepie91