Problems with the PCG implementation from the npm package pcg

Eris TSX • 5 Feb 2024

Problems with the PCG implementation from `npm:pcg`

Reaction: 🤬🤬🤬🤬

TL;DR: It *will* DoS your app, **never** use it!

It's slow. It's slow as hell. It's **≈100x** slower than an optimized implementation. They say, a PRNG is unlikely to become a performance bottleneck, but this particular one **is** a performance bottleneck! And it also can DoS your app, see below.

It's slow, part 2. For a single-step state transition during the number generation, it uses the arbitrary length jump algorithm. Not something as simple and obvious as `y = x * MULT + c`, but more like `y = nextNthState(x, n=1)`.

It makes the bundler cry (due to Ramda).

It's bloated. It's **≈19 KiB minigzipped**! Well, it's due to Ramda, and if you use it and related, ahem, *bloatware*, the impact of this particluar module will not be that high, but it's still much heavier than alternatives, especially taking in account that it doesn't have any features except a single function to generate bounded integers, **but it is…**

It's incorrect

Here is the core function `randomInt(min, max)`:

```
// (referred as $b$)
const bound = max - min;

// (1)
```

```
if (
    bound < 0 ||
    bound >= pcg.algorithm.outputMaxRange
)
    throw new RangeError();

// (referred as $t$)
const threshold =
    (pcg.algorithm.outputMaxRange - bound) %
    bound;

// Uniformity guarantees that this loop will terminate
// (↑ oh dear…)
let n: Long;
let nextPcg = pcg;
// (2)
do {
    n = Long.fromValue(
        nextPcg.getOutput(pcg.state),
        //                (!!WRONG!!)
        //                (↑ `pcg` never changes!)
    );
    nextPcg = nextState(nextPcg);
} while (n.lt(threshold));

return [
    n.mod(bound).add(min).toNumber(),
    nextPcg,
];
```

Parenthesized comments are mine.

No full range integers!

LET'S IMAGINE the `(WRONG)` fragment is not wrong. In this case, the generator **cannot yield integers in the full range** $[\mathbf{0}, \mathbf{2^{32} - 1}]$, only $[0, 2^{32} - 2]$:

1. The check `(1)` throws if the bound $b \geq 2^{32}$, so you can't `randomInt(0,2³²)` to get values in $[0, 2^{32} - 1]$.
2. If the bound $b = 2^{32} - 1$, the threshold $t = 1$. The loop `(2)` continues while the generated value n is less than the threshold t. If the generator yields 0, the loop will continue (actually, it will continue *forever*). If the generator yields n in $[1, 2^{32} - 2]$, the loop terminates, and `randomInt(…)` returns n, but if $n = 2^{32} - 1$, **the result is $\mathbf{0}$**.

Infinite loop!

But the main problem is that it **loops infinitely**! (Or rather, either once **xor** infinitely.)

The problem is in the `(WRONG)` fragment. It calls `getOutput()` on `nextPcg`, but the method doesn't depend on the state of `nextPcg`, it just computes the output function of its *argument*. And, *yes*, the argument **never** changes! So if the *first* generated value is $n < t$, the code will **loop forever**! What a nice generator that DoSes your app at random!

Wait, **this is a critical vulnerability**, isn't it?

How to use it

If you absolutely need this particular generator for some reason, you should use it like this:

```
let handle = createPcg32(...params);

const rand = () => {
    let x = handle.getOutput(handle.state);
    handle = nextState(handle);
    return x;
};
```

The exploit

This margin is too narrow to contain it.

The GitHub issue: https://github.com/philihp/pcg/issues/137

Well, the probability to get infinite loop is $t/2^{32}$. E.g., for $b = 2^{32} - 1$ it is 2^{-32}, for a simple `d6` roll it's 2^{-30}, and for $b = 2^{31} + 1$ it's $(2^{31} - 1)/2^{32} \approx 1/2$.

The vulnerability can be exploited by finding an application, that

1. uses a vulnerable version either server-side **and** accepts user input for `min` and `max` bounds,
2. **or** client-side **and** accepts the data from **other** users for the bounds.

Everything that remains is to fill these fields to get the bound $b = 2^{31} + 1$, PROFIT, the app is DoSed.