Published in Better Programming

Laszlo Fazekas Follow
Jan 10 · 5 min read · Listen

Save

Zero-Knowledge Proofs Using SnarkJS and Circom

A JavaScript tutorial

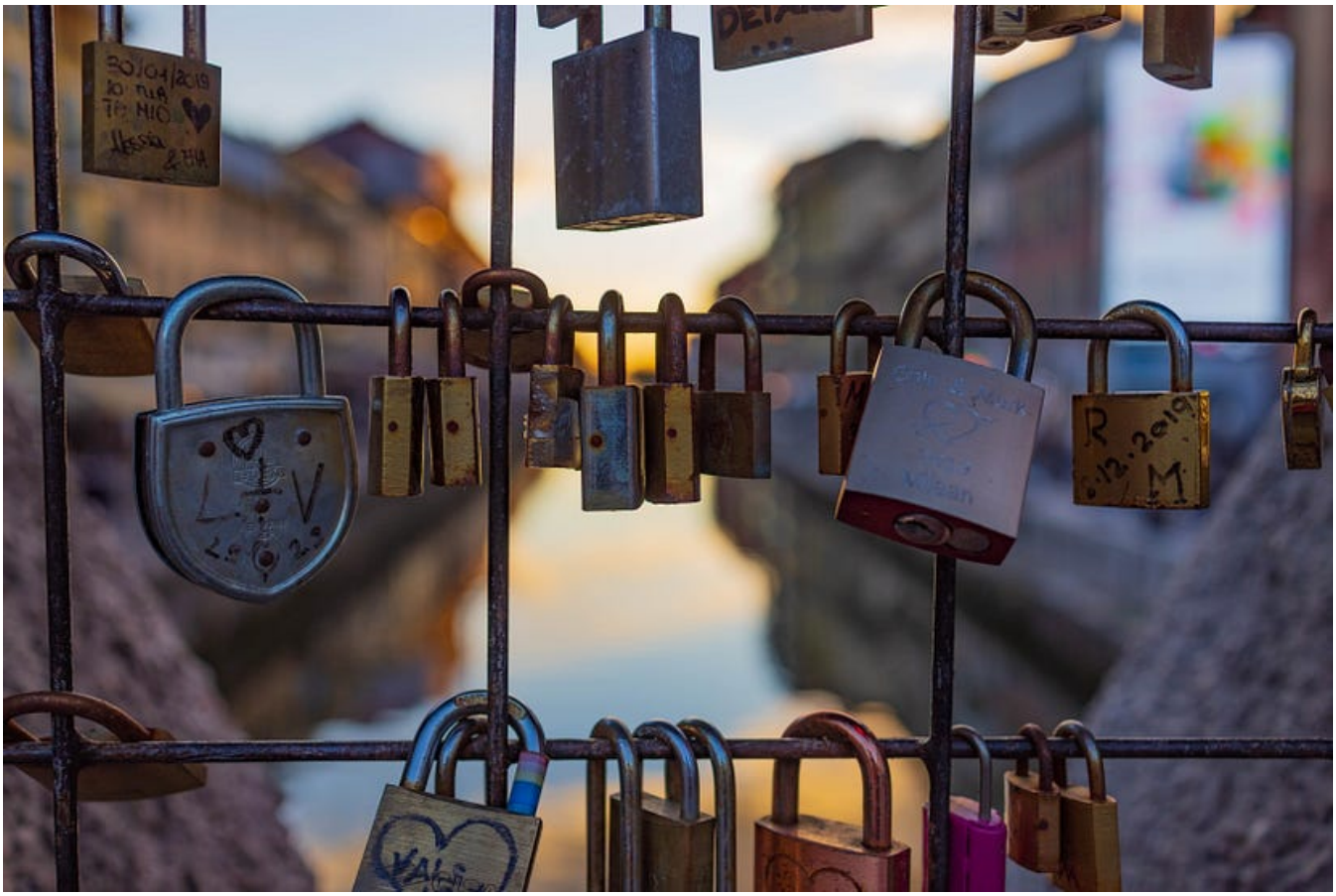

Photo by Maria Cappelli on Unsplash

The technology of zero-knowledge proof and especially zk-SNARK is one of the most exciting ones in the field of crypto because of the following reasons:

- You can prove you k for example, you can use it for anonymou
- The proof is small and easy to verify on the blockchain, so it can be used for rollups.

> *Rollup is a blockchain scaling solution where the computation is done off-chain, and after a given number of transactions the state is synchronized back to the blockchain. This solution gives you the security of the blockchain (after the synchronization), but the proof needs much less space (and less gas) than the original transactions. So zk-rollups are the ideal scaling solutions for the blockchain.*

I have a previous article where I show how zero-knowledge proofs work through the source code of the Tornado Cash coin mixer. If you are not familiar with the technology, it is strongly recommended to read that article before this one.

In this article, I will show you how you can use zk-SNARK in your JavaScript project.

If you have read my previous article, you know that you need a circuit to generate a zero-knowledge proof. A circuit is a huge mathematical expression used by the system to calculate the outputs and the proof. The zero-knowledge proof itself is proof that you have successfully done the calculation.

A circuit can be really complex, but fortunately, there are circuit programming languages and libraries that make it easy to write your own circuits. We will use Circom. Circom is written in Rust. To install it, you have to install the Rust environment with the following command:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

After the Rust installation, clone the Circom repo and build the compiler:

```
git clone https://github.com/iden3/circom.git
cd circom
```

```
cargo build --release
cargo install --pat
```


If everything went well, now you have the Circom compiler installed.

The other thing we need is the circomlib. Circomlib is a programming library with many useful predefined circuits. So, create an empty project, and install circomlib using this code:

```
npm init
npm i circomlib
```

Now, everything is ready to create our circuit. Here's what that looks like:

```
pragma circom 2.0.0;

include "node_modules/circomlib/circuits/poseidon.circom";

template PoseidonHasher() {
    signal input in;
    signal output out;

    component poseidon = Poseidon(1);
    poseidon.inputs[0] <== in;
    out <== poseidon.out;
}

component main = PoseidonHasher();
```

This simple circuit has a private input and an output signal. We are using the poseidon hasher from circomlib to generate the input `hash`. Using this circuit, we can prove that we know the original data for the given `hash` without revealing it.

In the first step, we compile the circuit by the circom compiler that will generate a wasm and an r1cs file.

```
circom poseidon_has
```


The generated wasm and r1cs files are available in the build folder. To generate the proof, we need a proving key file, and to generate this file, we need a ptau file. This ptau file can be generated by snarkjs, or you can download a pregenerated one (you can find the links in the snarkjs repo). For testing, the generated one is good for us, but in your production app, it's recommended to do the ceremony and generate your own ptau file. (You can read about it in my previous article.)

```
wget https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_12.ptau
```

Now we can generate the proving key (zkey file) by using the circuit and the ptau file:

```
npx snarkjs groth16 setup build/poseidon_hasher.r1cs powersOfTau28_hez_final_12
```

It is not recommended to use this zkey file for production, but for testing, it will be good for us (for more info, please check the snarkjs documentation).

Now, everything is ready to generate the proof. We will use snarkjs, so install it with this command:

```
npm i snarkjs
```

The generation of proof looks like this:

```
  "circuit_0000.zkey",
console.log(publicSignals);
console.log(proof);
```

The input signals are passed in the first parameter of the `fullProve` function. The second parameter is the compiled circuit, and the last parameter is the generated proving key. The function returns the outputs of the circuit and the proof.

We need a verification key that can be generated from the proving key to verify the proof. Here's how to get that:

```
npx snarkjs zkey export verificationkey circuit_0000.zkey verification_key.jsor
```

The verification code looks like this:

```
const vKey = JSON.parse(fs.readFileSync("verification_key.json"));
const res = await snarkjs.groth16.verify(vKey, publicSignals, proof);

if (res === true) {
  console.log("Verification OK");
} else {
  console.log("Invalid proof");
}
```

The verification key is the first parameter of the `verify` function, and the outputs and proof are the second and third parameters. The result of the function is a simple boolean.

In this example, we used the circuit to calculate the `hash`, but it is not always possible because the `hash` can be a partial result, or our circuit would look like this:

```
pragma circom 2.0.0
include "node_modules
template PoseidonHasher() {
    signal input in;
    signal input hash;

    component poseidon = Poseidon(1);
    poseidon.inputs[0] <== in;
    hash === poseidon.out;
}

component main {public [hash]} = PoseidonHasher();
```


This circuit has no outputs, only two inputs. The first input is the data, and the second is the hash of it. In the last line of the template, we check the `hash`. The circuit will run successfully only if the given `hash` is the poseidon `hash` of the given data. But how can we calculate the poseidon `hash` in JS?

Circomlib has a JS implementation that can be used for this. Let's install it:

```
npm i circomlibjs
```

Now we can calculate the `hash` by using the following code:

```
const poseidon = await circomlibjs.buildPoseidon();
const hash = poseidon.F.toString(poseidon([10]));
console.log(hash);
```

The result of the `poseidon` function is a `Buffer`, which we have to convert to a number. In zk-SNARK, every calculation is done in a finite field, so we have to use `poseidon.F.toString` for the conversion.

Circomlibjs and snarkjs work well in Node.js and the browser, so you can generate or verify proofs on the client side. It is also possible to generate a smart contract for

verification that you can use in your Solidity code to verify the proof. (For more info, please check the s

Circomlibjs also has sm if you want to generate a poseidon `hash` on-chain, you can do it by the generated code.

This was my very short tutorial on using zk-SNARK in JavaScript. It's not a full course, and you probably have many questions, but I hope I helped you to start your journey. Circom and snarkjs are well documented, and you can also learn a lot from existing projects like Tornado Cash.

The source codes for this tutorial are available in this GitHub repo.

Programming JavaScript Blockchain Ethereum Web Development