Context

Until now, we're understood

1. Hashing (SHA-256)
2. Encryption (EDDSA, ECDSA), Public and Private Keys
3. HD Wallets

We haven't practically built a web based wallet yet (Some of you have).

Here is a good example -

https://wallet-kosh.vercel.app/

https://kitsunekode-creepy-wallet-sand.vercel.app/

https://github.com/Fardeen26/eth-wallet-generator

What we're covering today

1. Keccak-256
2. RPC, JSON-RPC, Some common RPC methods on ETH/SOL
3. Try to create a simple web based wallet that shows you your balance

keccak-256

Try it out here - https://emn178.github.io/online-tools/keccak_256.html

Properties of `keccak-256` hashing algorithm

1. **Collision resistance**: Keccak256 is designed to be collision-resistant, meaning finding two inputs that produce the same hash output should be extremely difficult. However, it's important to note that collision resistance is not absolute, and there is always a small chance of collision. Therefore, it's recommended to use a combination of unique input parameters (e.g., block hash + block timestamp + contract nonce) for a lower probability of collision.
2. **Pre-image resistance**: Keccak256 is also designed to be pre-image resistant, meaning it should be nearly impossible to determine the original input from the hash output. However, it's important to note that brute-force attacks can still be attempted, and stronger passwords or keys will increase security.
3. **Key length**: Keccak256 outputs a 256-bit hash value, which means that it has a very large output space. This makes it resistant to brute-force attacks, but it's important to ensure that the key length is also sufficient for the application.
4. **Implementation**: It's important to ensure that the implementation of Keccak256 used is secure and free from vulnerabilities. Additionally, the implementation should be updated regularly to ensure that any discovered vulnerabilities are patched.

ETH

Ethereum public addresses are 20 bytes (0x8BCd4591F46e809B15A490F5C6eD031FDDE0bee0)

When generating the `public key` for an ETH address

- Initially, a public key is generated using elliptic curve cryptography.
- The public key is then hashed using the Keccak-256 algorithm.
- After hashing the public key with Keccak-256, you get a 32-byte hash. The Ethereum address is derived from this hash by taking only the last 20 bytes of the hash output.
- The resulting 20-byte value is then converted into hexadecimal format and prefixed with '0x' to form the Ethereum address. This is the address that users use to send and receive ETH and interact with smart contracts.

How backpack does it - https://github.com/coral-xyz/backpack/blob/master/packages/secure-background/src/services/evm/util.ts#L3

How ethers does it under the hood - https://github.com/ethers-io/ethers.js/blob/main/src.ts/transaction/address.ts#L12

Solana

Solana public keys are 32 bytes (5W4oGgDHqir3KNEcmiMn6tNHmbWjC7PgW11sk4AwWbpe). No need for hashing/chopping

Frontend vs Backend, HTTP Servers

Backend servers in full stack are servers which run your backend logic.

Your frontend talks to them through HTTP calls.

For example - https://jsonplaceholder.typicode.com/posts/1

Postman

Postman is an application that lets you send requests to Backend servers without using the browser

RPC, JSON-RPC

JSON-RPC is a remote procedure call (RPC) protocol encoded in JSON (JavaScript Object Notation). It allows for communication between a client and a server over a network. JSON-RPC enables a client to invoke methods on a server and receive responses, similar to traditional RPC protocols but using JSON for data formatting.

As a user, you interact with the blockchain for two purposes -

1. To send a `transction`
2. To fetch some details from the blockchain (balances etc)

In both of these, the way to interact with the blockchain is using JSON-RPC

JSON RPC Spec - https://www.jsonrpc.org/specification

There are other ways to do RPC's like GRPC, TRPC

Wei, Lamports

ETH

Wei:

- **Definition**: Wei is the smallest unit of cryptocurrency in the Ethereum network. It's similar to how a cent is to a dollar.
- **Value**: 1 Ether (ETH) = 10^18 Wei.

Gwei

- **Definition**: A larger unit of Ether commonly used in the context of gas prices.
- **Conversion**: 1 Ether = 10^9 gwei

Lamports

- In the Solana blockchain, the smallest unit of currency is called a **lamport**. Just as wei is to Ether in Ethereum, lamports are to SOL (the native token of Solana).
- 1 SOL = 10 ^ 9 Lamports

```
const { LAMPORTS_PER_SOL } = require("@solana/web3.js")

console.log(LAMPORTS_PER_SOL)
```

RPC Server

An RPC server provides a way for external clients to interact with the blockchain network. RPC stands for **Remote Procedure Call**, and an RPC server exposes an API that allows clients to send requests and receive responses from the blockchain.

An RPC server is a service that listens for JSON-RPC requests from clients, processes these requests, and returns the results. It acts as an intermediary between the blockchain and external applications or services.

> 💡 An RPC (Remote Procedure Call) server is not inherently part of the blockchain network itself, nor does it participate in staking or consensus mechanisms

You can grab your own RPC server from one of many providers -

1. Quicknode
2. Alchemy
3. Helius
4. Infura

Common RPC calls on Solana

Get account info

Retrieves information about a specific account.

```
{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "getAccountInfo",
 "params": ["Eg4F6LW8DD3SvFLLigYJBFvRnXSBiLZYYJ3KEePDL95Q"]
}
```

Get Balance

Gets the balance for a given account

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "getBalance",
  "params": ["Eg4F6LW8DD3SvFLLigYJBFvRnXSBiLZYYJ3KEePDL95Q"]
}
```

Get Transaction count

```
{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "getTransactionCount"
}
```

Common RPC calls on ETH

Get balance

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "eth_getBalance",
  "params": ["0xaeaa570b50ad00377ff8add27c50a7667c8f1811", "latest"]
}
```

Get latest block

```
{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "eth_blockNumber"
}
```

Get block by number

```
{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "eth_getBlockByNumber",
 "params": ["0x1396d66", true]
}
```

Creating a web based wallet

Pre-requisites - React

Let's try to create a web based wallet similar to https://wallet-kosh.vercel.app/

https://github.com/keshav-exe/projekt-kosh/

- Initialize an empty React Project

```
npm create vite@latest
```

- Install dependencies

```
npm install
```

- Add node-pollyfills

```
npm install vite-plugin-node-polyfills
```

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import { nodePolyfills } from 'vite-plugin-node-polyfills'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react(), nodePolyfills()],
})
```

- Clean up App.jsx


```
import { useState } from 'react'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

  return (
    <>

    </>
  )
}

export default App
```

- Create a mnemonics state variable

```
  const [mnemonic, setMnemonic] = useState("");
```

- Add a button that lets the user generate a fresh mnemonic phrase. Ref - https://projects.100xdevs.com/tracks/public-private-keys/Public-Key-Cryptography-9

```
npm install bip39
```

```
import { generateMnemonic } from "bip39";

<button onClick={async function() {
  const mn = await generateMnemonic();
  setMnemonic(mn)
}}>
  Create Seed Phrase
</button>
```

Reference - https://github.com/hujiulong/web-bip39

- Display the `mnemonic` in an input box

```
<input type="text" value={mnemonic}></input>
```

- Add a `SolanaWallet` component

Ref - https://projects.100xdevs.com/tracks/public-private-keys/Public-Key-Cryptography-9

```
import { useState } from "react"
import { mnemonicToSeed } from "bip39";
import { derivePath } from "ed25519-hd-key";
import { Keypair } from "@solana/web3.js";
import nacl from "tweetnacl"

export function SolanaWallet({ mnemonic }) {
  const [currentIndex, setCurrentIndex] = useState(0);
  const [publicKeys, setPublicKeys] = useState([]);

  return <div>
    <button onClick={function() {
      const seed = mnemonicToSeed(mnemonic);
      const path = `m/44'/501'/${currentIndex}'/0'`;
      const derivedSeed = derivePath(path, seed.toString("hex")).key;
      const secret = nacl.sign.keyPair.fromSeed(derivedSeed).secretKey;
      const keypair = Keypair.fromSecretKey(secret);
      setCurrentIndex(currentIndex + 1);
      setPublicKeys([...publicKeys, keypair.publicKey]);
    }}>
      Add wallet
    </button>
    {publicKeys.map(p => <div>
      {p.toBase58()}
    </div>)}
  </div>
}
```

- Create ETH wallet
  Ref - https://projects.100xdevs.com/tracks/public-private-keys/Public-Key-Cryptography-10

```
import { useState } from "react";
import { mnemonicToSeed } from "bip39";
import { Wallet, HDNodeWallet } from "ethers";

export const EthWallet = ({mnemonic}) => {
  const [currentIndex, setCurrentIndex] = useState(0);
```

```
    const [addresses, setAddresses] = useState([]);

    return (
      <div>
        <button onClick={async function() {
          const seed = await mnemonicToSeed(mnemonic);
          const derivationPath = `m/44'/60'/${currentIndex}'/0`;
           const hdNode = HDNodeWallet.fromSeed(seed);
           const child = hdNode.derivePath(derivationPath);
           const privateKey = child.privateKey;
           const wallet = new Wallet(privateKey);
           setCurrentIndex(currentIndex + 1);
          setAddresses([...addresses, wallet.address]);
        }}>
          Add ETH wallet
        </button>

        {addresses.map(p => <div>
          Eth - {p}
        </div>)}
      </div>
    )
  }
```


Assignment

Can you add logic to show the ETH/SOL balances for these accounts?