



WORKSHOP

*Tout sur les wallets, des bases à
l'implémentation*



SOMMAIRE

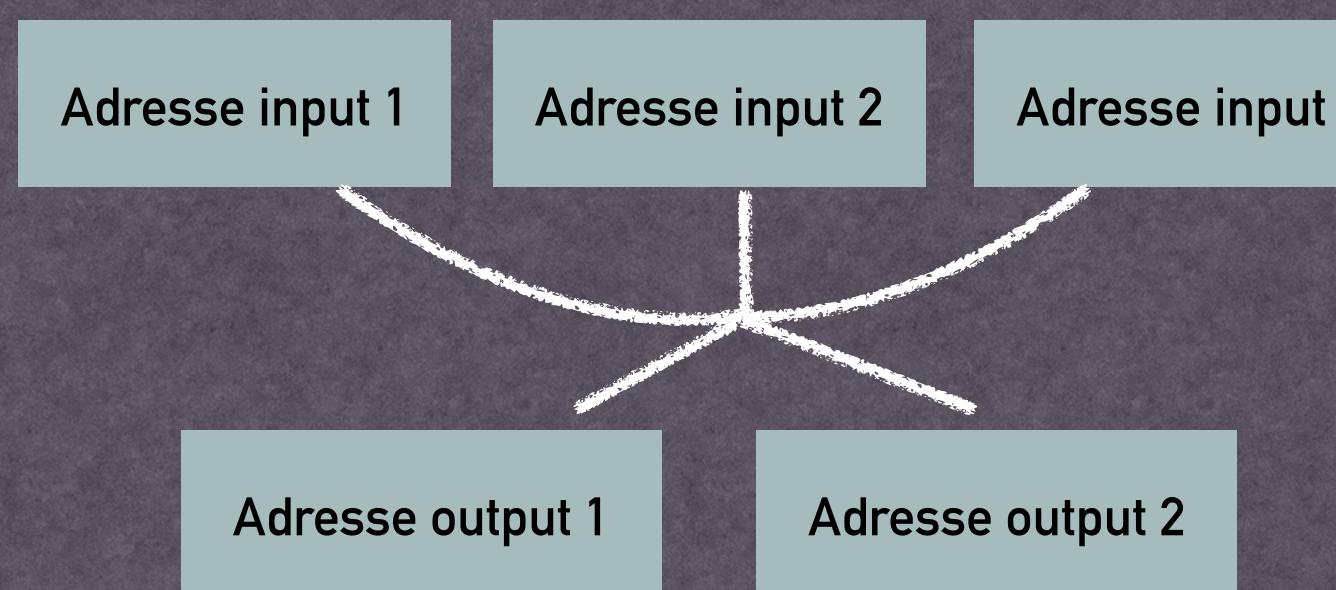
- Les adresses sur les blockchains EVM
- Wallets et Web 3
- Les seeds phrases : BIP-32, BIP-39 et BIP-44
- Sécurité des hardware wallet Ledger
- Format d'une transaction EVM
- Faire une whitelist centralisée en Solidity

LES ADRESSES SUR LES BLOCKCHAINS EVM

- Une clé privée, une clé publique, comme Bitcoin
- Algorithme de chiffrement basé sur les courbes elliptiques, le même que Bitcoin
- Format des adresses différent en revanche : 40 caractères hexadécimaux, prefixés par *0x*
- Contrairement à Bitcoin, une transaction EVM émane toujours d'une et une seule adresse

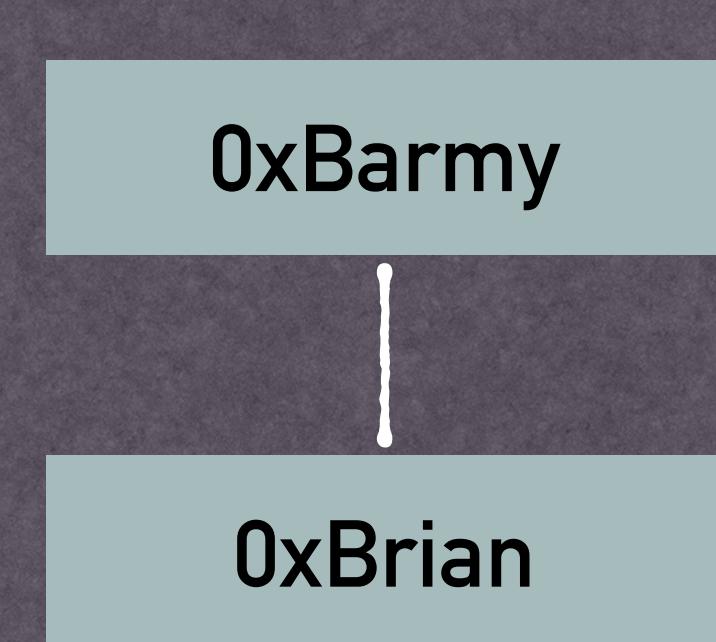
Transaction Bitcoin

“input 1 envoie 0.02 BTC, input 2 envoie 0.05 BTC, input 3 envoie 0.03 BTC, output 1 reçoit 0.07 BTC, output 2 reçoit 0.02 BTC, je laisse 0.01 BTC pour le mineur. Signé input 1, input 2, input 3.”



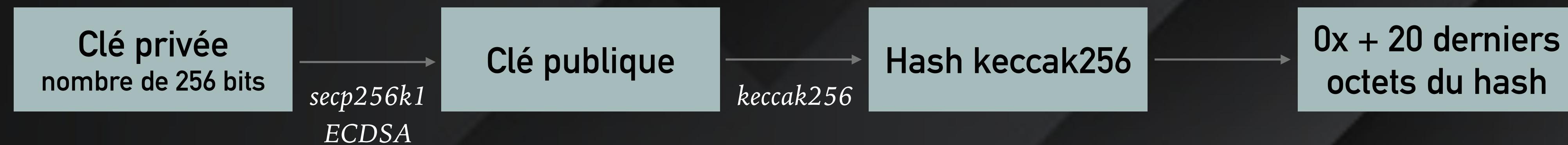
Transaction EVM

“J'envoie 1 BNB à 0xBrian. Pas de data à exécuter. J'accepte un maximum de 21,000 unités de gas et je paie 3 Gwei par unité de gas. Signé 0xBarmy”



LES ADRESSES SUR LES BLOCKCHAINS EVM

- Calculer l'adresse de wallet à partir de la clé publique



- Exemple d'implémentation en python
- Remarque : comme l'adresse n'est un extrait d'un hash, on ne peut pas avoir la clé publique d'une adresse EVM qui n'a jamais envoyé de transaction sur le réseau.
- Une transaction contient forcément la clé publique complète afin que les nodes puissent confirmer la validité de la signature.

LES ADRESSES SUR LES BLOCKCHAINS EVM

```
# source:  
# https://www.freecodecamp.org/news/how-to-create-an-ethereum-wallet-address-from-a-private-key-ae72b0eee27b/  
  
import codecs  
# pip3 install ecdsa  
import ecdsa  
# pip3 install pycryptodome  
from Crypto.Hash import keccak  
  
# Une clé privée est une chaîne de 64 caractères hexadécimaux = 256 bits de données  
private_key_bytes = codecs.decode("60cf347dbc59d31c1358c8e5cf5e45b822ab85b79cb32a9f3d98184779a9efc2", "hex")  
  
# Obtenir la clé publique en utilisant l'algorithme secp256k1  
key = ecdsa.SigningKey.from_string(private_key_bytes, curve=ecdsa.SECP256k1).verifying_key  
key_bytes = key.to_string()  
public_key = codecs.encode(key_bytes, "hex")  
print(public_key)  
  
# Générer un hash keccak de la clé publique  
public_key_bytes = codecs.decode(public_key, "hex")  
keccak_hash = keccak.new(digest_bits=256)  
keccak_hash.update(public_key_bytes)  
keccak_digest = keccak_hash.hexdigest()  
print(keccak_digest)  
  
# L'adresse publique correspond aux 20 octets les plus à droite = 40 caractères hexadécimaux  
wallet_len = 40  
wallet = "0x" + keccak_digest[-wallet_len:]  
print(wallet)
```

LES ADRESSES SUR LES BLOCKCHAINS EVM

- Deux types d'adresses : EOA (Externally Owned Account) et Contract
- Une adresse EOA correspond au wallet de n'importe quel utilisateur final
- Une adresse Contract est générée et assignée automatiquement lorsqu'un smart contract déployé
- Chaque Contract possède sa propre adresse unique
- Personne ne connaît la clé privée qui est derrière
- Les adresses Contract ne peuvent émettre aucune transaction

LES ADRESSES SUR LES BLOCKCHAINS EVM

- Une adresse Contract peut recevoir des transactions “simples”
- Une adresse Contract ouvre des points d’entrée qui peuvent être appelés en passant des données personnalisées : on “appelle” une méthode du contrat pour interagir avec lui, il peut lui-même interagir avec d’autres Contracts. Si une seule opération échoue, toute la transaction échoue.
- Exemple très vulgarisé, voir les détails dans les futurs ateliers Solidity :

Transaction avec un Contract

“J’envoie 0 BNB à 0xPancakeswap. J’appelle la méthode Deposit avec les paramètres suivants : token = CAKE, quantité = 100. Signé 0xBarmy”

0xBarmy

0xPancakeswap

Token CAKE

*“Envoie moi 100 CAKE de Barmy
Cordialement, Pancakeswap”*

REVERT: “he’s too poor”

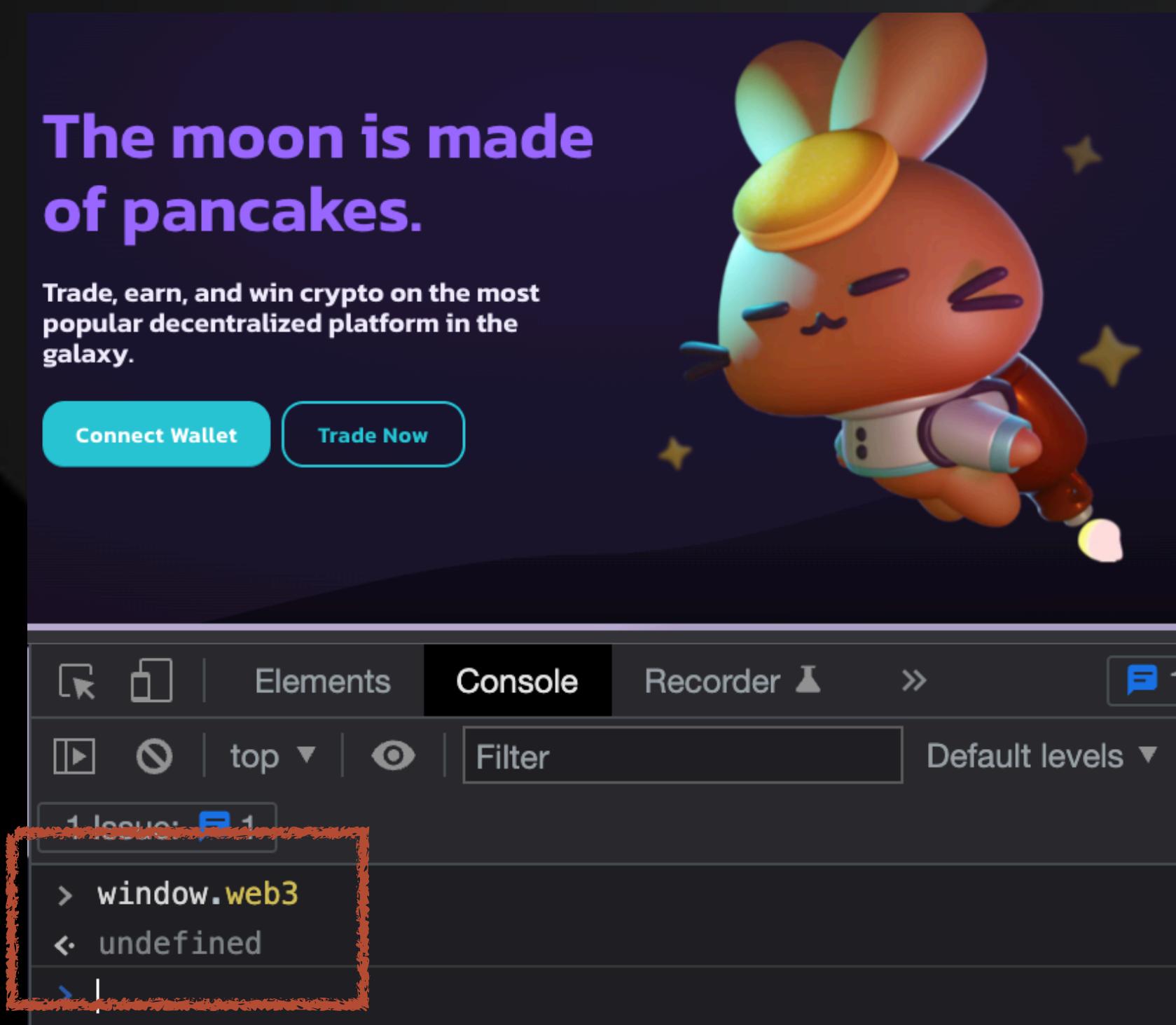
WALLETS ET WEB 3

- Un wallet n'est qu'un simple carnet d'adresses
- Le Web 3 est une convention permettant de développer des interfaces web pour communiquer avec des smart contracts
- Tous les wallets n'offrent pas cette intégration, exemple : Ledger Live
- MetaMask est à la fois un Wallet EVM, et une passerelle Web 3
- Communiquer avec la BNB Chain avec MetaMask : bnbchainlist.org

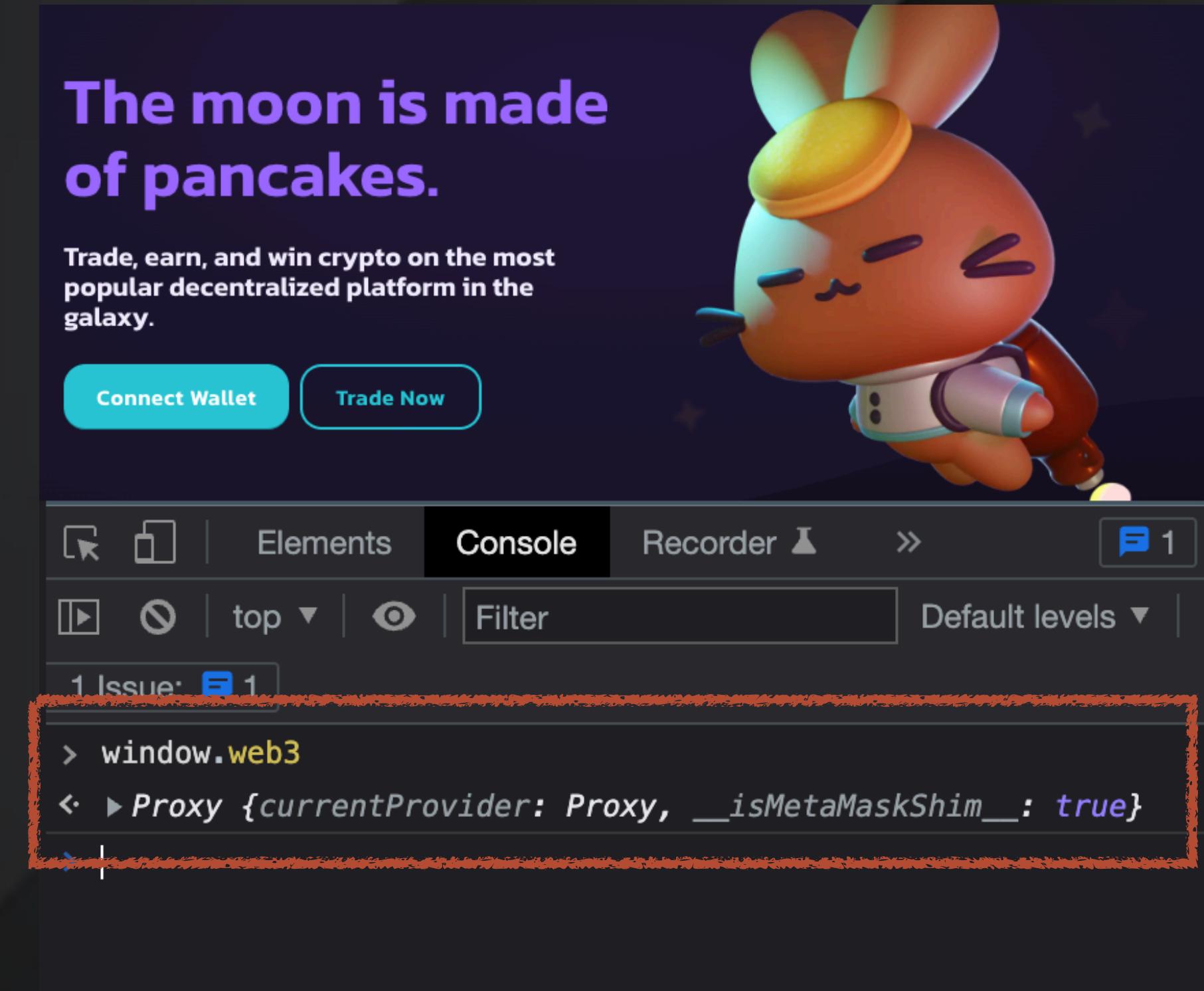
WALLETS ET WEB 3

- Les “Browser wallets” comme MetaMask injectent du JavaScript dans les pages, faisant comprendre au site Internet visité que le navigateur est compatible avec Web 3 et prêt à connecter une adresse et interagir si besoin.

Sans MetaMask

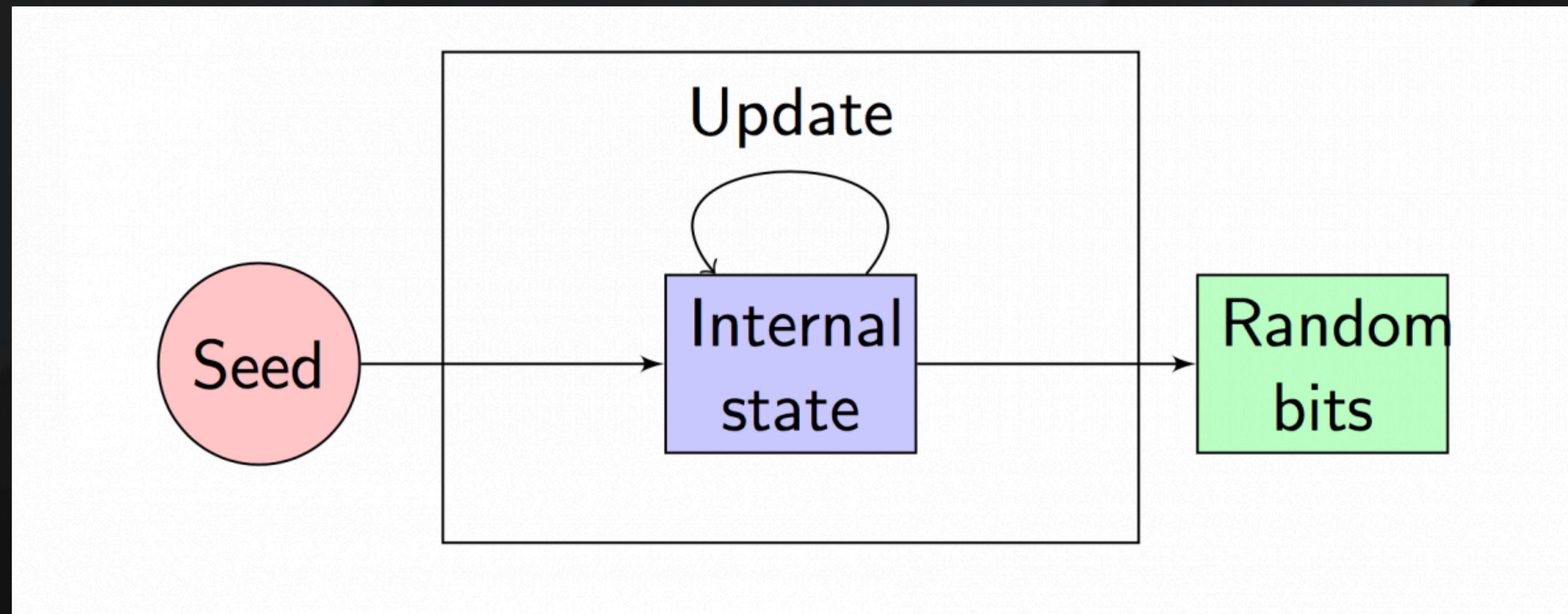


Avec MetaMask



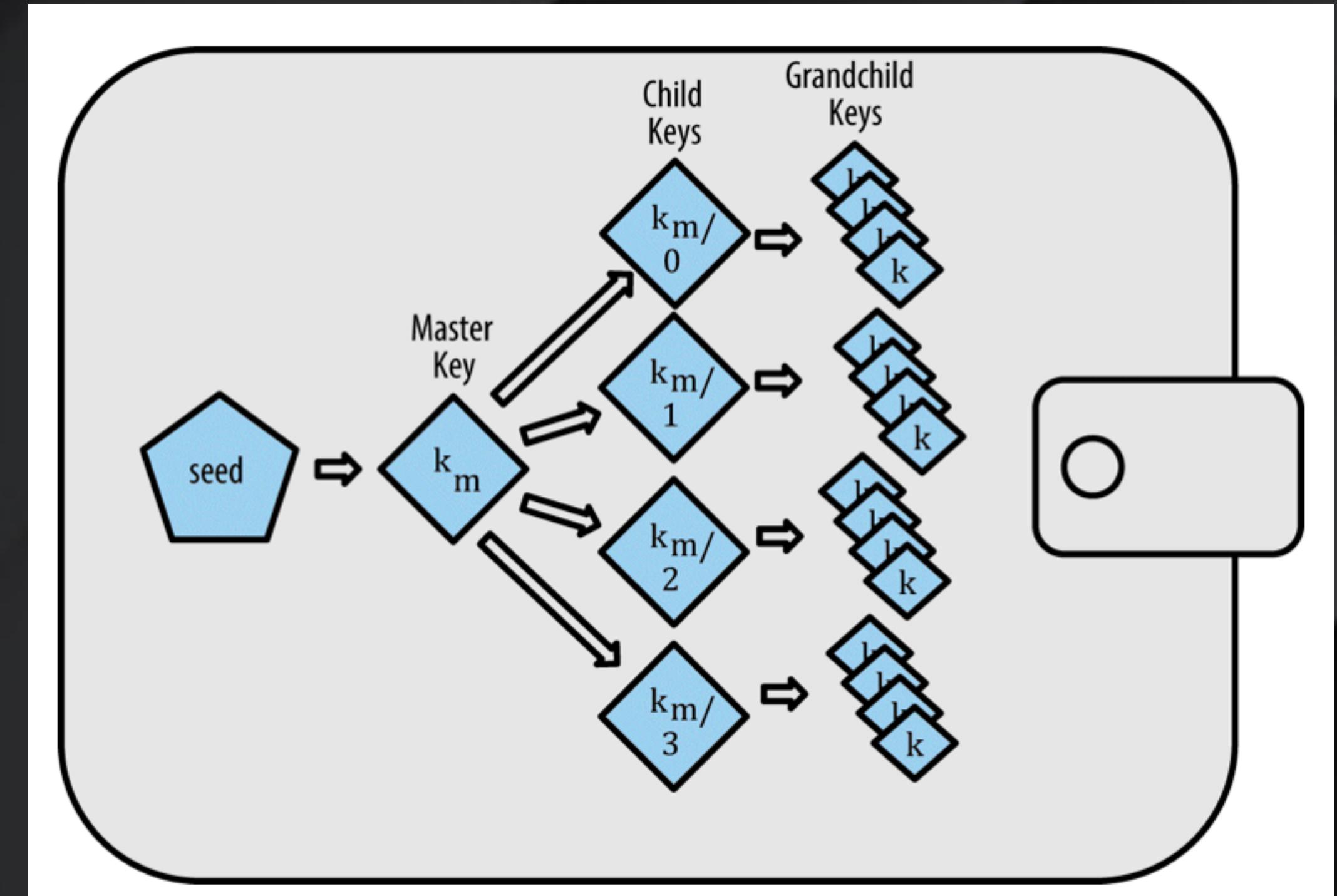
LES SEED PHRASES

- L'informatique est déterministe
- Une “seed” sert à initialiser un générateur de nombres pseudo-aléatoires (PRNG)
- Un PRNG donnera toujours les mêmes résultats avec une seed donnée



LES SEED PHRASES

- Problème : à chaque fois qu'on souhaite générer une nouvelle adresse, on doit récupérer et sauvegarder la clé privée en lieu sûr, cela pose des risques d'erreur humaine, de sécurité et demande du temps.
- Solution : on sécurise une seule donnée aléatoire de départ unique (“Seed”), qui permettra de générer toujours les mêmes adresses, et autant d’adresses qu’on souhaite.
- La communauté Bitcoin a créé le standard BIP-32, une Seed de 128 à 256 bits qui permet de dériver des clés privées : “Hierarchical Deterministic Wallet”



LES SEED PHRASES

- Problème : l’humain est très mauvais pour retenir ou noter de longues séries de nombres hexadécimaux. Imaginez devoir recopier 64 caractères hexadécimaux sur une feuille sans vous tromper.
- Solution : la communauté Bitcoin a créé les standards BIP-39 et BIP-44. On les utilise également sur les blockchains EVM et sur beaucoup d’autres blockchains.
- BIP-39 : les bits aléatoire de la Seed sont divisés en “mots” lisibles par des humains à partir d’un dictionnaire de 2048 mots.
- Ce n’est pas le mot qui compte mais sa position dans le dictionnaire, entre 0 et 2047.
- BIP-44 : convention permettant à tous les wallets d’utiliser la même méthode pour recréer des adresses à partir de la Seed. Une Seed sur MetaMask sait générer les mêmes adresses qu’une Ledger, etc...

LES SEED PHRASES

- Fun fact : les 2048 mots anglais ont été choisis de telle sorte qu'aucun mot ne partage les mêmes 4 premières lettres.
- Les 4 premières lettres de chaque mot suffisent à retrouver la Seed phrase complète.
- Personne ne doit jamais vous demander une Seed phrase. TOUT prétexte est bidon. Même en cas de problème, d'autres solutions existent, quel que ce soit le problème.
- **NE DONNEZ JAMAIS, SOUS AUCUN PRETEXTE, MÊME EN CAS D'ATTAQUE ZOMBIE, VOTRE SEED PHRASE À QUI QUE CE SOIT, MÊME UN ADMIN D'UN PROJET DEFI, MÊME UN EMPLOYÉ D'UN EXCHANGE, ET MÊME UN NOTAIRE.**

IMPLEMENTATION DE BIP-39

- Comment passer des bits aléatoire aux mots ?
- On va créer notre propre Seed phrase en PHP, parce que pourquoi pas
- Quizz : la probabilité de deviner une Seed phrase de 12 mots est équivalente à combien de fois celle de gagner à l'Euromillions ?

IMPLEMENTATION DE BIP-39

```
<?php
// array[0..2047]
$words = explode("\n", file_get_contents('wordlist_en.txt'));

// Sources:
// https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki
// https://github.com/satoshilabs/slips/blob/master/slip-0044.md
// https://iancoleman.io/bip39/

// Generate 128 bits of entropy
$f = fopen('/dev/random', 'r');
$ent = fread($f, 16); // 16 octets = 128 bits
fclose($f);

$strSha0fEnt = hash("sha256", $ent);
$binSha0fEnt = hex2bin($strSha0fEnt);
echo "sha256 of ENT = " . $strSha0fEnt . "\n";

// First, an initial entropy of ENT bits is generated.
// A checksum is generated by taking the first ENT / 32 bits (128/32 = 4) of its SHA256 hash.
// => Cela consiste tout simplement à prendre le premier chiffre hexadécimal puisque 1 chiffre = 4 bits
$strChecksum = substr($strSha0fEnt, 0, 1);
$strEntWithChecksum = bin2hex($ent) . $strChecksum;

echo "Entropy with checksum = $strEntWithChecksum\n";

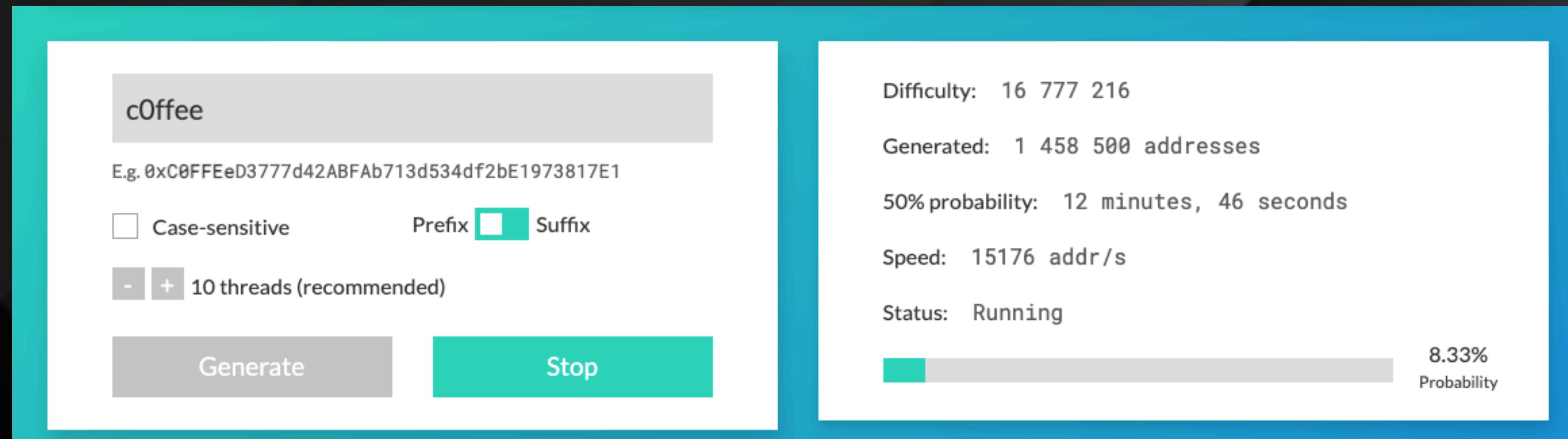
// Parce qu'on a un nombre impair de bits, les fonctions natives du style
// chunk_split, array_slice, etc... ne fonctionnent pas. On fait à l'ancienne avec des "0"/"1" en string.
$strbinEntWithChecksum = str_replace(
    ['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'],
    ['0000','0001','0010','0011','0100','0101','0110','0111','1000','1001','1010','1011','1100','1101','1110','1111'],
    $strEntWithChecksum
);
$splited = explode("\n", chunk_split($strbinEntWithChecksum, 11));

foreach($splited as $k => $split) {
    if (!empty($split)) {
        // Print chaque mot les uns après les autres
        echo $words[bindec($split)] . " ";
    }
}
echo "\n";
```

QU'EST-CE QU'UNE "VANITY" ADDRESS ?

- Une “vanity” address est une adresse qui contient un pattern particulier
- Une seule solution : générer des clés privées aléatoirement jusqu'à trouver une adresse qui correspond
- Librairie nodejs populaire pour le faire soi-même :

<https://github.com/bokub/vanity-eth>

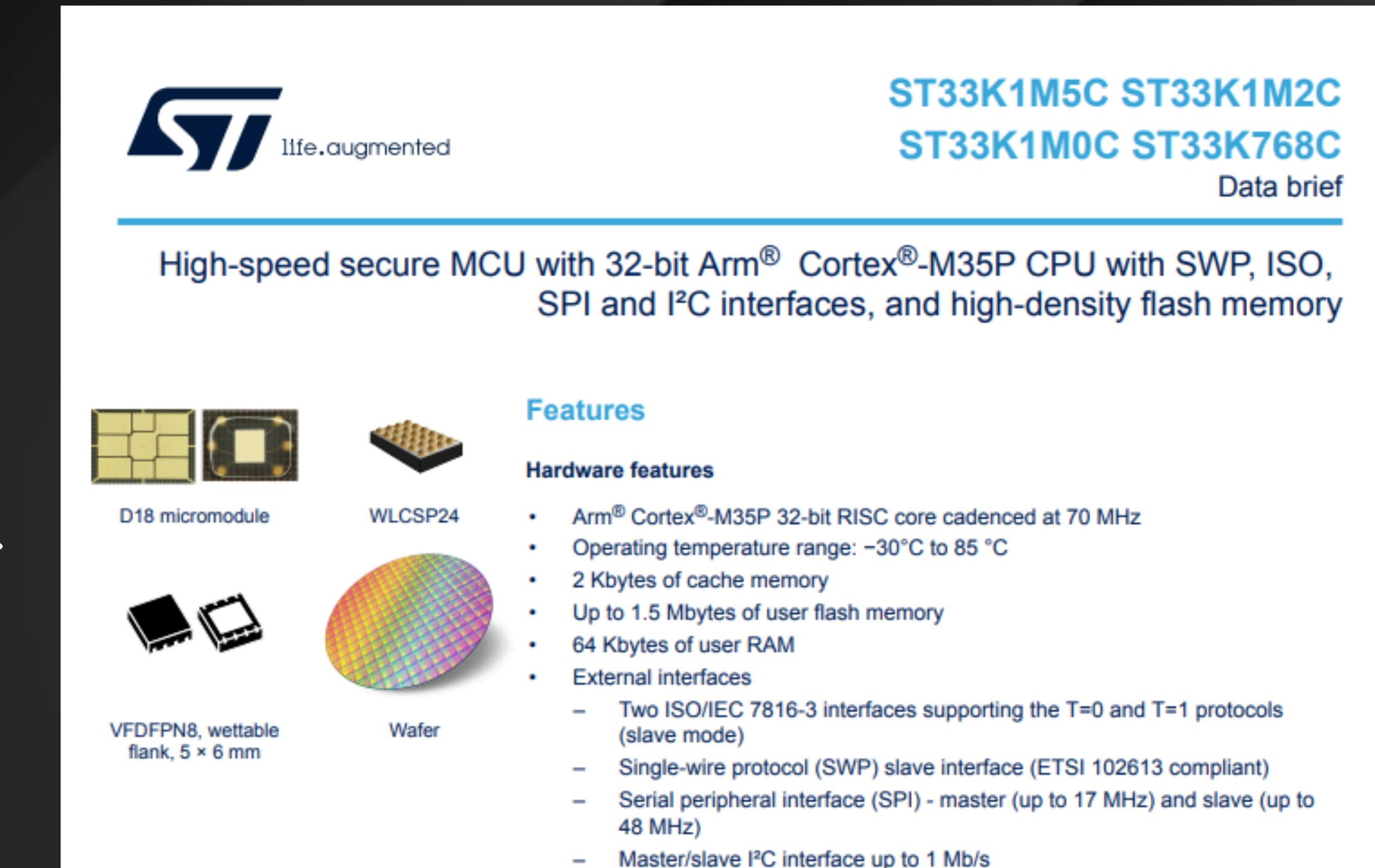


SÉCURITÉ DES HARDWARE WALLETS LEDGER

- Les hardware wallets intègrent des micro-contrôleurs et exécutent leur propre code sur leur propre mémoire, les mêmes que dans les cartes SIM ou dans les cartes bancaires à puce.
- Ces micro-contrôleurs ont un module de chiffrement intégré. Ils stockent les clés privées et les seed dans une mémoire interne, inaccessible depuis l'extérieur.
- On peut les voir comme des “ASICs” du chiffrement.
- Lorsqu'on signe une transaction, l'ordinateur ne reçoit de la part du hardware wallet que la transaction finale déjà signée. La clé privée ne sort JAMAIS de la mémoire secrète de la puce de chiffrement.
- Vous êtes safe, même si votre ordinateur est hacké.
- (!) Par contre : vérifiez toujours que l'adresse de destination sur l'écran du Ledger correspond au vrai destinataire. Une fausse transaction peut toujours être envoyée à la place de votre transaction légitime par un malware.

SÉCURITÉ DES HARDWARE WALLETS LEDGER

- Hardware embarqué dans les Ledger :
- Micro-contrôleur STM32WB de STMicroelectronics
- Processeur de chiffrement ST33 de STMicroelectronics. Il embarque un True Random Number Generator basé sur des événements physiques captés par la puce.
- Documentation top secrète, sous embargo et NDA industriel



The image shows a data brief for STMicroelectronics' ST33 series of microcontrollers. It features the ST logo and the tagline "life.augmented". The title "High-speed secure MCU with 32-bit Arm® Cortex®-M35P CPU with SWP, ISO, SPI and I²C interfaces, and high-density flash memory" is displayed. Below this, four physical forms of the chip are shown: D18 micromodule, WLCSP24, VFDFPN8, wettable flank, 5 × 6 mm, and Wafer. A detailed list of features is provided, including the Arm® Cortex®-M35P 32-bit RISC core, operating temperature range, cache memory, user flash memory, user RAM, and various external interfaces like ISO/IEC 7816-3, SWP, SPI, and I²C.

**ST33K1M5C ST33K1M2C
ST33K1M0C ST33K768C**

Data brief

High-speed secure MCU with 32-bit Arm® Cortex®-M35P CPU with SWP, ISO, SPI and I²C interfaces, and high-density flash memory

Features

Hardware features

- Arm® Cortex®-M35P 32-bit RISC core cadenced at 70 MHz
- Operating temperature range: -30°C to 85 °C
- 2 Kbytes of cache memory
- Up to 1.5 Mbytes of user flash memory
- 64 Kbytes of user RAM
- External interfaces
 - Two ISO/IEC 7816-3 interfaces supporting the T=0 and T=1 protocols (slave mode)
 - Single-wire protocol (SWP) slave interface (ETSI 102613 compliant)
 - Serial peripheral interface (SPI) - master (up to 17 MHz) and slave (up to 48 MHz)
 - Master/slave I²C interface up to 1 Mb/s

SÉCURITÉ DES HARDWARE WALLETS LEDGER

- Ces puces sont inviolables
- Même si le ST33 est dissout dans l'acide, la puce de silicium est recouverte d'un maillage métallique empêchant tout reverse engineering.
- Aucun circuit n'est visible au microscope, il faudrait détruire la puce pour la sonder et cela détruirait sa mémoire.
- Voir la vidéo de Deus Ex Silicium sur le Ledger Nano S qui embarque la puce ST31 :
<https://www.youtube.com/watch?v=ma3S7UTrwgo>



FORMAT D'UNE TRANSACTION EVM (PRÉ EIP-1559)

T_n	Nonce	0-32 octets
T_p	Prix consenti par unité de gas	0-32 octets
T_g	Quantité max de gas acceptée	0-32 octets
T_t	Adresse de destination	20 octets
T_v	Quantité de coins natifs envoyés (en wei)	0-32 octets
T_i (ou) T_d	Call data (ou) Contract creation data	illimité
T_w	v	illimité
T_r	r	32 octets
T_s	s	32 octets

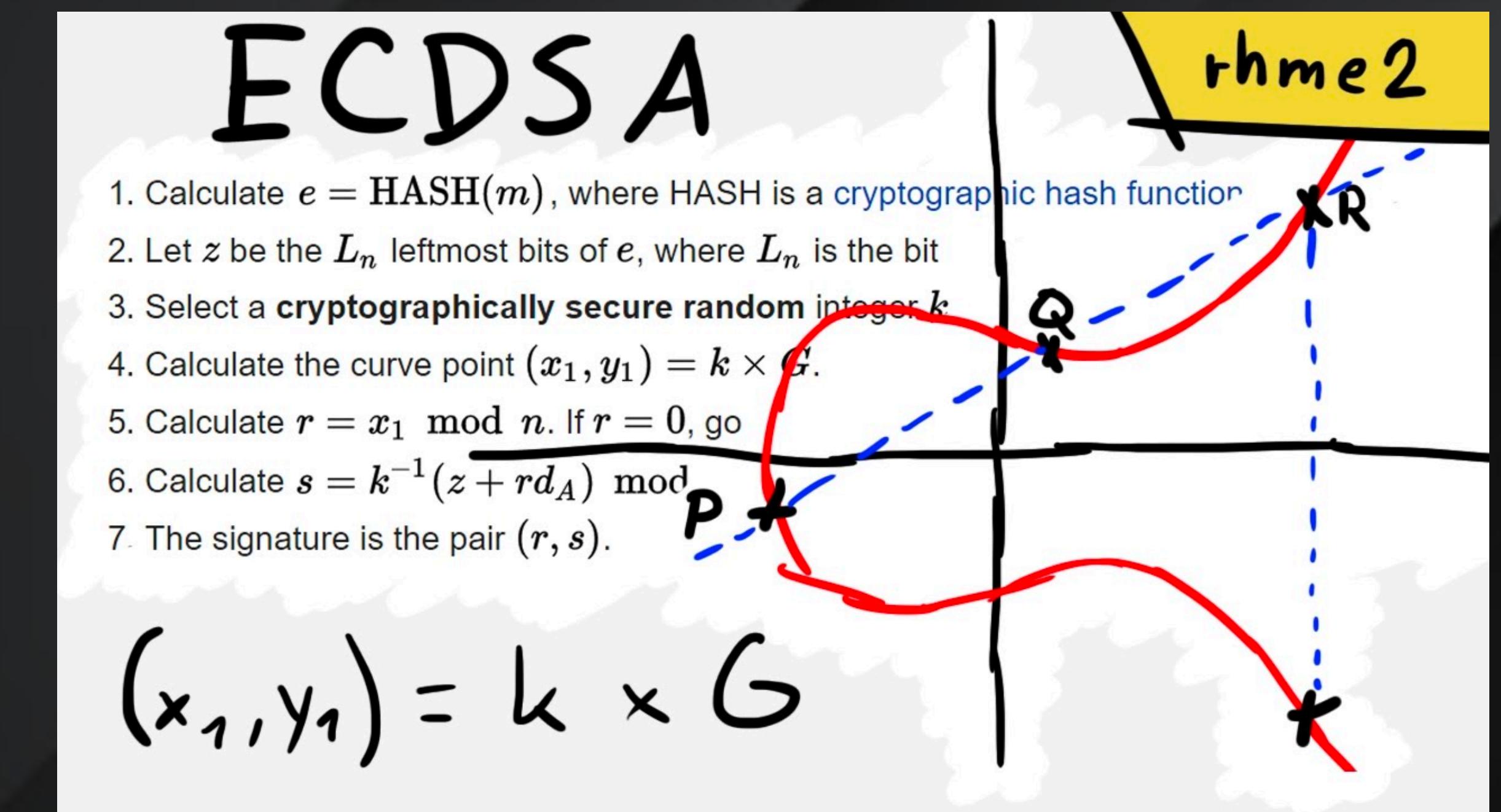
FORMAT D'UNE TRANSACTION EVM (PRÉ EIP-1559)

- Une signature ECDSA est vérifiée grâce au couple (r, s) + la clé publique
- Elle est reconstruite grâce au message + le champ de parité v
- Depuis 2016, l'EIP-1559 la valeur de v dépend également du Chain ID pour éviter les “*replay attacks*”

$v \quad \{0,1\} + \text{CHAIN_ID} * 2 + 35$

r Signature

s Signature



FORMAT D'UNE TRANSACTION EVM (PRÉ EIP-1559)

- $(message, r, s, v)$ = Reconstruction de la clé publique
- $(message, r, s)$ + Clé publique = Authenticité de la signature vérifiée
- (!) On ne réinvente JAMAIS la roue en cryptographie : c'est un domaine difficile qui nécessite des compétences mathématiques poussées. Il faut TOUJOURS utiliser les outils existants sous peine d'ajouter des failles de sécurités critiques.

Les libs web3 gèrent déjà très bien les signatures.

Ne cherchez pas à recalculer les signatures vous-mêmes.

CRÉER UNE WHITELIST CENTRALISÉE AVEC UNE SIGNATURE ECDSA

```
contract CentralizedWhitelist {  
  
    address constant public WHITELIST_SIGNER_ADDRESS = 0x4513218Ce2e31004348Fd374856152e1a026283C;  
  
    modifier onlyValidAccess(uint8 _v, bytes32 _r, bytes32 _s) {  
        bytes memory packedHash = abi.encodePacked(address(this), msg.sender);  
        bytes32 hash = keccak256(packedHash);  
        bytes memory packedString = abi.encodePacked("\x19Ethereum Signed Message:\n32", hash);  
        require(ecrecover(keccak256(packedString), _v, _r, _s) == WHITELIST_SIGNER_ADDRESS, "Invalid ECDSA signature");  
    }  
  
    constructor () public {}  
  
    function formMessage(address userAddress) external view returns (bytes32) {  
        return keccak256(abi.encodePacked(address(this), userAddress));  
    }  
  
    function coolFeatureForWhitelistedUser(uint8 _v, bytes32 _r, bytes32 _s) external onlyValidAccess(_v,_r,_s) {  
        // ...  
    }  
}
```

CRÉER UNE WHITELIST CENTRALISÉE AVEC UNE SIGNATURE ECDSA

```
process.env.NTBA_FIX_319 = 1;
const Web3 = require('web3')

const SIGNING_ADDRESS_PRIVATE_KEY = "60cf347dbc59d31c1358c8e5cf5e45b822ab85b79cb32a9f3d98184779a9efc2"

const main = async () => {

    // Init RPC
    const web3 = new Web3('http://127.0.0.1:8545/')

    // Signer account
    const account = web3.eth.accounts.privateKeyToAccount(SIGNING_ADDRESS_PRIVATE_KEY)
    web3.eth.accounts.wallet.add(account)
    web3.eth.defaultAccount = account.address

    // Generate message
    const message = "ici le hash généré par le smart contract avec sa méthode formMessage(adresseDuUserAWhitelister)"

    // Sign it with the private key
    const signed = await web3.eth.accounts.sign(message, SIGNING_ADDRESS_PRIVATE_KEY)

    console.log({
        signedBy: account.address,
        signedMessage: signed
    })
}

main()
```