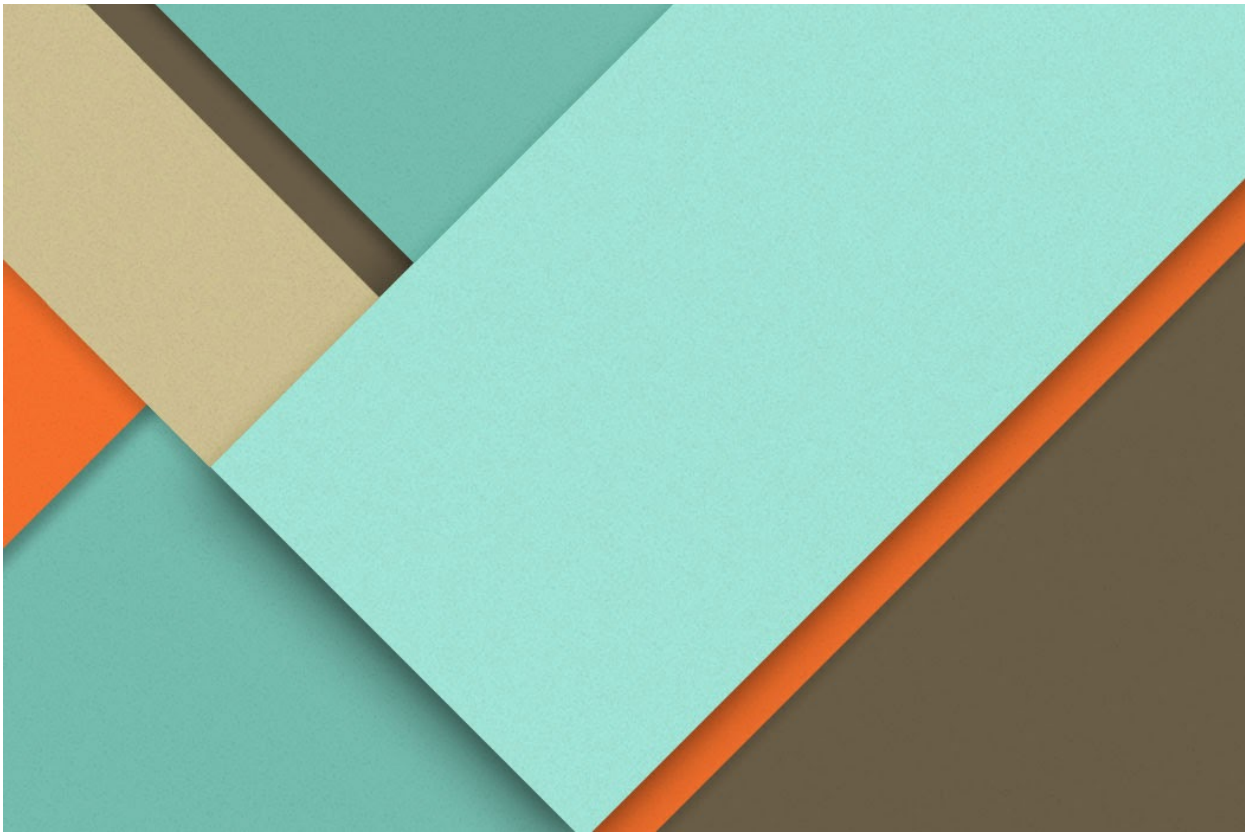

Bip39

26.01.2021

—

Team CPP {Maria Usman, Mahnoor Ghous, Umar Farooq}
MediaPark SMC Pvt Ltd.
ISE Tower Islamabad.

Overview

This BIP describes the implementation of a mnemonic code or mnemonic sentence.
A group of easy to remember words -- for the generation of deterministic wallets.
It consists of two parts: generating the mnemonic and converting it into a binary seed. This seed can be later used to generate deterministic wallets using BIP-0032 or similar methods.

Goals

1. Build the complete understanding of Bip39
2. Proper flow of the algorithm considering c++ language.

Milestones

I. Generate Entropy

It starts with entropy generation. With more entropy security is improved but the sentence length increases. It is allowed to be 128–256 bits to generate 12–24 phrases. We will take an example of 128 bits which will generate 12 phrases. In our example, below is the entropy generated , in hex & binary:

In hex: 063679ca1b28b5cfda9c186b367e271e

In Bin:
00000110001101100111100111001010000110110010100001011010111001111111011010100111
00000110000110101100110110011111100010011100011110

Reference to function:
blockchain-core-proto-cpp/src/crypto/bip39/bip39.cpp/generateSecureEntropy()

II. Validate Entropy:

In validate Entropy we make sure that the given hexadecimal string when converted to binary it must lie within one of the following entropyBits Set { 128, 160, 192, 224, 256}.

In hex: 063679ca1b28b5cfda9c186b367e271e

entropyBits = 32*4=128 bits
Because the above string in hex is of length 32 and 1 hex value is of 1 nibble(4bits) so expected no. of bits will be 128 which lies within the given range of entropyBits Set.

Reference to function: blockchain-core-proto-cpp/src/crypto/bip39/bip39.cpp/validateEntropy()

III. Generate Checksum

Third step is about generating checksum.
`checksum = first (length of entropy in bits/32) bits of SHA256 of entropy.`
In our case, it is 128/32 = 4 bits. Let's assume, it is 0110 (6 in decimal) in our case. Append this checksum at the end of initial entropy. So, after concatenation, it will be:

00000110001101100111100111001010000110110010100001011010111001111111011010100111
000001100001101011001101100111111000100111000111110110

Reference to function: blockchain-core-proto-cpp/src/crypto/bip39/bip39.cpp/checksum()

IV. Split

Now we need to split it into groups of 11 bits. Right now total bits are 128 + 4 = 132

After split, it will look like:

`00000110001 10110011110 01110010100 00110110010 10001011010 11100111111 01101010011 10000011000 01101011001 10110011111 10001001110 00111100110`

Reference to function: blockchain-core-proto-cpp/src/crypto/bip39/bip39.cpp/useEntropy()

V. Convert to Decimal

Now we need to convert these bits into its decimal representation as:

00000110001	10110011110	01110010100	00110110010	10001011010
49	1438	916	434	1114

11100111111	01101010011	10000011000	01101011001	10110011111
1855	851	1048	857	1439

10001001110	00111100110
1102	486

These decimal representations vary from 0–2047. These work as an index to mnemonic word list. This word list can be found here.

Reference to function: blockchain-core-proto-cpp/src/crypto/bip39/bip39.cpp/mnemonics()

VI. Find out words

Now words will be chosen from the wordlist. In our case, with English language, they will be as:

49	1438	916	434	1114	1855
alert	record	income	curve	mercy	tree

851	1048	857	1439	1102	486
heavy	loan	hen	recycle	mean	devote

So, this way final generated mnemonic phrases will be:

alert record income curve mercy tree heavy loan hen recycle mean devote

No. of mnemonic words generated depends on the size of initial entropy. It follows as:

Bits of Entropy	Number of words
128	12
160	15
192	18
224	21
256	24

Reference to function: blockchain-core-proto-cpp/src/crypto/bip39/bip39.cpp/mnemonics()

VII. Seed

A user may decide to protect their mnemonic with a passphrase. If a passphrase is not present, an empty string "" is used instead.

To create a binary seed from the mnemonic, we use the PBKDF2 function with a mnemonic sentence (in UTF-8 NFKD) used as the password and the string "mnemonic" + passphrase (again in UTF-8 NFKD) used as the salt. The iteration count is set to 2048 and HMAC-SHA512 is used as the pseudo-random function. The length of the derived key is 512 bits (= 64 bytes).

This seed can be later used to generate deterministic wallets using BIP-0032 or similar methods.

Reference to function: blockchain-core-proto-cpp/src/crypto/bip39/bip39.cpp/seed()

```
std::vector<uint8_t> BIP39::seed( Algo algo, const std::string &pass, const std::string &salt, uint32_t iterations, size_t outKeySize)
{
    std::vector<uint8_t> outKey(outKeySize);
    outKey=hashPbkdf2(algo, pass, salt, iterations, outKeySize);
    return outKey;
}
```

Note: Reference to function hashPbkdf2() : blockchain-core-proto-cpp/src/crypto/pbkdf2.cpp