Abstract

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.

Motivation

A mnemonic code or sentence is superior for human interaction compared to the handling of raw binary or hexadecimal representations of a wallet seed. The sentence could be written on paper or spoken over the telephone.

This guide is meant to be a way to transport computer-generated randomness with a human readable transcription. It's not a way to process user-created sentences (also known as brainwallets) into a wallet seed.

Generating the mnemonic

The mnemonic must encode entropy in a multiple of 32 bits. With more entropy security is improved but the sentence length increases. We refer to the initial entropy length as ENT. The allowed size of ENT is 128-256 bits.

First, an initial entropy of ENT bits is generated. A checksum is generated by taking the first

```
ENT / 32
```

bits of its SHA256 hash. This checksum is appended to the end of the initial entropy. Next, these concatenated bits are split into groups of 11 bits, each encoding a number from 0-2047, serving as an index into a wordlist. Finally, we convert these numbers into words and use the joined words as a mnemonic sentence.

The following table describes the relation between the initial entropy length (ENT), the checksum length (CS) and the length of the generated mnemonic sentence (MS) in words.

```
CS = ENT / 32
MS = (ENT + CS) / 11

|  ENT  | CS | ENT+CS |  MS  |
+-------+----+--------+------+
|  128  |  4 |   132  |  12  |
|  160  |  5 |   165  |  15  |
|  192  |  6 |   198  |  18  |
|  224  |  7 |   231  |  21  |
|  256  |  8 |   264  |  24  |
```

Wordlist

An ideal wordlist has the following characteristics:

a) smart selection of words

```
- the wordlist is created in such way that it's enough to type the first four
  letters to unambiguously identify the word
```

b) similar words avoided

```
- word pairs like "build" and "built", "woman" and "women", or "quick" and "quickly"
  not only make remembering the sentence difficult, but are also more error
  prone and more difficult to guess
```

c) sorted wordlists

```
- the wordlist is sorted which allows for more efficient lookup of the code words
  (i.e. implementations can use binary search instead of linear search)
- this also allows trie (a prefix tree) to be used, e.g. for better compression
```

The wordlist can contain native characters, but they must be encoded in UTF-8 using Normalization Form Compatibility Decomposition (NFKD).

From mnemonic to 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.

The conversion of the mnemonic sentence to a binary seed is completely independent from generating the sentence. This results in rather simple code; there are no constraints on sentence structure and clients are free to implement their own wordlists or even whole sentence generators, allowing for flexibility in wordlists for typo detection or other purposes.

Although using a mnemonic not generated by the algorithm described in "Generating the mnemonic" section is possible, this is not advised and software must compute a checksum for the mnemonic sentence using a wordlist and issue a warning if it is invalid.

The described method also provides plausible deniability, because every passphrase generates a valid seed (and thus a deterministic wallet) but only the correct one will make the desired wallet available.