BIP39 Mnemonics Made Easy (Part 2 - The Tech of Bits to Backups)

Author	josh
Date	2018-01-19 18:46:25

Overview

In the last article, we discussed a high level overview of BIP39 mnemonics and their value as a simplified backup tool. Mnemonics make it much easier to take a *single seed*, back it up, and ensure access to an entire wallet of private keys, addresses, and transactions. But how do we go from a random set of bits to a list of words? Let's discuss the technical side of BIP39.

Bits to Backups - The Steps for Generating a Mnemonic

First, Chaos

In order to generate a good seed, a fair amount of entropy or "randomness" is desirable. Good random number generators are hard to get, but modern OS's like Linux do a pretty good job of sourcing entropy from the user and hard drives, and something like /dev/urandom on a daily driver machine should be sufficiently secure for generating the entropy we need. Now how many random bits do we need? The BIP39 standard specifies 128-256 bits of entropy to be used for generating the seed. This will correspond to 12-24 words later on when we "map" the entropy to the words. First, a warning: ***DO NOT USE any of the examples in this article to generate a wallet - your funds will be stolen!*** With that out of the way, let's look at an example. First, let's generate 128 bits of entropy using os.urandom() in Python. Represented as binary, our entropy looks like this:

```
10111110011001010101110111001111010100011111011010110001110101111011110111000101101001100011111010001010001110100001101101110000
```

Next, a checksum

In order to better secure the seed, we'll add a checksum to the end of the entropy. This makes it easier for wallet software to validate a backup seed. To get the checksum, we'll first take the SHA-256 hash of our entropy. Then, we take the first `N/32` bits of the hash and append it to the entropy. In our case, 128/32 bits gives us a 4 bit checksum size. In our example, the 4 bit checksum will be `0101`. We'll append that to the entropy to give us a 132 bit value:

```
10111110011001010101110111001111010100011111011010110001110101111011110111000101101001100011111010001010001110100001101101110000000101
```

Dividing and our Dictionary

The final step of the process involves dividing our checksummed bits into "chunks" and mapping those chunks to the mnemonic words from the dictionary. The BIP39 standard specifies that the

chunks will always be 11 bits long. So, we divide our 132 bit checksummed entropy into 12 chunks of 11 bits each:

1. 10111110011
2. 00101010111
3. 01110011110

... Now, each of these 11 bit chunks can be interpreted as an unsigned 11 bit integer value ranging from 0-2047. This "maps" to a word from the dictionary of 2048 words directly! These are standardized and listed in alphabetic order. So, we can take the 11 bit chunk as an index in the dictionary to extract the words we need:

1. 10111110011 = 1523 -> salmon
2. 00101010111 = 343 -> cliff
3. 01110011110 = 926 -> inherit

... The overall mnemonic we generate is this example turns out to be:

1. salmon
2. cliff
3. inherit
4. physical
5. help
6. type
7. warfare
8. regular
9. dial
10. photo
11. asset
12. scheme

Mnemonics - from Entropy to Dictionary Entries

The process of generating a mnemonic seed is both ingenious and straightforward. One can easily create a secure wallet seed of 12-24 words by generating some entropy, checksumming the data, and mapping to a standard dictionary. I've written a project called MnemonicGen that generates mnemonics using these steps. Take a look at this project to see these steps implemented in Python. This code should be considered academic/experimental - use it to create wallets at your own risk. Other proven implementations such as Ian Coleman's BIP39 are also available to study. Happy generating!