Ethereum SLIP-39 Account Generation

Perry Kundert

2021-12-20 10:55:00

Creating Ethereum accounts is complex and fraught with potential for loss of funds.

A BIP-39 seed recovery phrase helps, but a **single** lapse in security dooms the account. If someone finds your recovery phrase, the account is *gone*.

The SLIP-39 standard allows you to split the seed between 1 or more groups of several mnemonic recovery phrases. This is better, but creating such accounts is difficult; presently, only the Trezor supports these, and they can only be created "manually". Writing down 5 or more sets of 20 words is difficult, error-prone and time consuming.

The python-slip39 project exists to assist in the safe creation and documentation of Ethereum HD Wallet accounts, with various SLIP-39 sharing parameters. It generates the new random wallet seed, generates standard Ethereum account(s) (at derivation path `m/66'/40'/0'/0/0` by default) with Ethereum wallet address and QR code, produces the required SLIP-39 phrases, and outputs a single PDF containing all the required printable cards to document the account.

On an secure (ideally air-gapped) computer, new accounts can safely be generated and the PDF saved to a USB drive for printing (or directly printed without the file being saved to disk.)

Contents

1 Security with Availability

For both BIP-39 and SLIP-39, a 128-bit random "seed" is the source of
an unlimited sequence of Ethereum HD Wallet accounts. Anyone who can
obtain this seed gains control of all Ethereum accounts derived from it, so it
must be securely stored.

Losing this seed means that all of the HD Wallet accounts are perma-
nently lost. Therefore, it must be backed up reliably, and be readily acces-
sible.

Therefore, we must:

- Ensure that nobody untrustworthy can recover the seed, but

- Store the seed in many places with several (some perhaps untrustwor-
  thy) people.

How can we address these conflicting requirements?

1.1 Shamir's Secret Sharing System (SSSS)

Satoshi Lab's (Trezor) SLIP-39 uses SSSS to distribute the ability to recover
the key to 1 or more "groups". Collecting the mnemonics from the required
number of groups allows recovery of the seed. For BIP-39, the number of
groups is always 1, and the number of mnemonics required for that group is
always 1.

For SLIP-39, a "group_threshold" of how many groups must bet success-
fully collected to recover the key. Then key is (conceptually) split between 1
or more groups (not really; each group's data alone gives away no information
about the key).

For example, you might have First, Second, Fam and Frens groups, and
decide that any 2 groups can be combined to recover the key. Each group
has members with varying levels of trust and persistence, so have different

number of Members, and differing numbers Required to recover that group's data:

Group	Required		Members	Description
First	1	/	1	Stored at home
Second	1	/	1	Stored in office safe
Fam	2	/	4	Distributed to family members
Frens	2	/	6	Distributed to friends and associates

The account owner might store their First and Second group data in their home and office safes. These are 1/1 groups (1 required, and only 1 member, so each of these are3 1-card groups.)

If the account needs to be recovered, collecting the First and Second cards from the home and office safe is sufficient to recover the seed, and re-generate the HD Wallet accounts.

Only 2 Fam member's cards must be collected to recover the Fam group's data. So, if the HD Wallet owner loses their home and First group card in a fire, they could get the Second group card from the office safe, and 2 cards from Fam group members, and recover the wallet.

If catastrophe strikes and the owner dies, and the heirs don't have access to either the First (at home) or Second (at the office), they can collect 2 Fam cards and 2 Frens cards (at the funeral, for example), completing the Fam and Frens groups' data, and recover the HD Wallet account. Since Frens are less likely to persist long term (and are also less likely to know each-other), we'll require a lower proportion of them to be collected.

2 SLIP-39 Account Generation and Recovery

Generating a new SLIP-39 encoded Ethereum wallet is easy, with results available as PDF and text. The default groups are as described above. Run the following to obtain a PDF file containing index cards with the default SLIP-39 groups for the account named "Personal"; insert a USB drive to collect the output, and run:

```
$ python3 -m pip install slip39   # Install slip39 in Python3
$ cd /Volumes/USBDRIVE/           # Change current directory to USB
$ python3 -m slip39 Personal      # Or just run "slip39 Personal"
2021-12-25 11:10:38 slip39            m/44'/60'/0'/0/0    : 0xb44A2011A99596671d5952CdC22816089f142FB3
2021-12-25 11:10:38 slip39            Wrote SLIP-39-encoded wallet for 'Personal' to:\
  Personal-2021-12-22+15.45.36-0xb44A2011A99596671d5952CdC22816089f142FB3.pdf
```

The resultant PDF will be output into the designated file.

This PDF file can be printed on 3x5 cards, or on regular paper or card stock and the cards can be cut out (`--card credit` or `--card business` are also available).

To get the data printed on the terminal as in this example (so you could write it down on cards instead), add a `-v`.

2.1 Recover & Regeneration

Later, if you need to recover the Ethereum wallet, keep entering SLIP-39 mnemonics until the secret is recovered (invalid/duplicate mnemonics will be ignored):

```
$ python3 -m slip39.recovery   # (or just "slip39-recovery")
Enter 1st SLIP-39 mnemonic: ab c
Enter 2nd SLIP-39 mnemonic: veteran guilt acrobat romp burden campus purple webcam uncover ...
Enter 3rd SLIP-39 mnemonic: veteran guilt acrobat romp burden campus purple webcam uncover ...
Enter 4th SLIP-39 mnemonic: veteran guilt beard romp dragon island merit burden aluminum worthy ...
2021-12-25 11:03:33 slip39.recovery  Recovered SLIP-39 secret; Use:  python3 -m slip39 --secret ...
383597fd63547e7c9525575decd413f7
```

Finally, regenerate the Ethereum wallet, perhaps including an encrypted JSON wallet file for import into a software wallet:

```
$ python3 -m slip39 --secret 383597fd63547e7c9525575decd413f7 --json -
2021-12-25 11:09:57 slip39              ETH(m/44'/60'/0'/0/0): 0xb44A2011A99596671d5952CdC22816089f142FB3
...
JSON key file password: <enter JSON wallet password>
2021-12-25 11:10:38 slip39              Wrote JSON encrypted wallet for '' to:\
   SLIP39-2021-12-25+11.09.57-0xb44A2011A99596671d5952CdC22816089f142FB3.json
2021-12-25 11:10:39 slip39              Wrote SLIP39-encoded wallet for '' to:\
   SLIP39-2021-12-25+11.09.57-0xb44A2011A99596671d5952CdC22816089f142FB3.pdf
```

2.1.1 Pipelining `slip39.recovery | slip39 --secret -`

The tools can be used in a pipeline to avoid printing the secret. Here we generate some mnemonics, sorting them in reverse order so we need more than just the first couple to recover. Observe the Ethereum wallet address generated.

Then, we recover the master secret seed in hex with `slip39-recovery`, and finally send it to `slip39 --secret -` to re-generate the same wallet as we originally created.

```
( python3 -m slip39 --text --no-card -q \
    | sort -r \
    | python3 -m slip39.recovery \
    | python3 -m slip39 --secret - --no-card -q ) 2>&1

2021-12-28 10:55:31 slip39              m/44'/60'/0'/0/0    : 0xc5BfB72a1487d509F466198704bca88dB1b3230e
2021-12-28 10:55:31 slip39.recovery Recovered SLIP-39 secret with 4 (1st, 2nd, 7th, 8th) of 8 supplied mnemonics
2021-12-28 10:55:31 slip39              m/44'/60'/0'/0/0    : 0xc5BfB72a1487d509F466198704bca88dB1b3230e
```

2.2 The `slip39` module API

Provide SLIP-39 Mnemonic set creation from a 128-bit master secret, and recovery of the secret from a subset of the provided Mnemonic set.

2.2.1 `slip39.create`

Creates a set of SLIP-39 groups and their mnemonics.

Key	Description
name	Who/what the account is for
group_threshold	How many groups' data is required to recover the account(s)
groups	Each group's description, as {"<group>":(<required>, <members>), ...}
master_secret	128-bit secret (default: from secrets.token_bytes)
passphrase	An optional additional passphrase required to recover secret (default: "")
iteration_exponent	For encrypted secret, exponentially increase PBKDF2 rounds (default: 1)
paths	A number of HD Wallet derivation paths (default: ["m/60'/44'/0'/0/0"]

Outputs a `slip39.Details` namedtuple containing:

Key	Description
name	(same)
group_threshold	(same)
groups	Like groups, w/ <members> = ["<mnemonics>", ...]
accounts	Resultant { "path": eth_account.Account, ...}

This is immediately usable to pass to `slip39.output`.

```
import codecs
import random

import eth_account
import slip39
paths = [ "m/60'/44'/0'/0/0", "m/60'/44'/0'/0/1" ]
master_secret,passphrase = bytes(range(16)),b"password!" # >8/
create_details = slip39.create(
    "Test", 2, { "Mine": (1,1), "Fam": (2,3) },
    master_secret=master_secret, passphrase=passphrase, paths=paths )
[
    [
        f"{g_name}({g_of}/{len(g_mnems)}) #{g_n+1}:" if l_n == 0 else ""
    ] + words
    for g_name,(g_of,g_mnems) in create_details.groups.items()
    for g_n,mnem in enumerate( g_mnems )
    for l_n,(line,words) in enumerate(slip39.organize_mnemonic(
            mnem, rows=7, cols=3, label=f"{g_name}({g_of}/{len(g_mnems)}) #{g_n+1}:" ))
]
```

0	1	2	3
Mine(1/1) #1:	1 genre	8 election	15 training
	2 lungs	9 liberty	16 military
	3 acrobat	10 warmth	17 woman
	4 easy	11 transfer	18 owner
	5 clinic	12 exercise	19 pencil
	6 hunting	13 frozen	20 type
	7 formal	14 silent	
Fam(2/3) #1:	1 genre	8 wavy	15 spew
	2 lungs	9 branch	16 robin
	3 beard	10 race	17 justice
	4 echo	11 railroad	18 tracks
	5 crucial	12 silent	19 smirk
	6 wrote	13 budget	20 deadline
	7 pickup	14 beam	
Fam(2/3) #2:	1 genre	8 hobo	15 deploy
	2 lungs	9 pickup	16 diagnose
	3 beard	10 revenue	17 episode
	4 email	11 ruin	18 satoshi
	5 album	12 library	19 fiction
	6 desktop	13 elder	20 parking
	7 fantasy	14 fiscal	
Fam(2/3) #3:	1 genre	8 soldier	15 cargo
	2 lungs	9 center	16 arena
	3 beard	10 phrase	17 network
	4 entrance	11 galaxy	18 primary
	5 devote	12 escape	19 acne
	6 identify	13 havoc	20 vexed
	7 express	14 olympic	

Add the resultant HD Wallet addresses:

```
[
    [ path, eth.address ]
    for path,eth in create_details.accounts.items()
]
```

0	1
m/60'/44'/0'/0/0	0x7A64E13f00a4502b62d45EED15d6254b9Cc10a34
m/60'/44'/0'/0/1	0xF7dc45001A1F3b26565B1364b03C76553FBEE931

2.2.2 `slip39.output`

Key	Description
name	(same as `slip39.create`)
group_threshold	(same as `slip39.create`)
groups	Like groups, w/ <members> = ["<mnemonics>", . . .]
accounts	Resultant { "path": eth_account.Account, . . . }
card_format	'index', . . .
paper_format	'Letter', . . .

Produce a PDF containing all the SLIP-39 details for the account.

```
slip32.output( *create_details )
```

2.2.3 `slip39.recover`

Takes a number of SLIP-39 mnemonics, and if sufficient `group_threshold` groups' mnemonics are present (and the options `passphrase` is supplied), the `master_secret` is recovered. This can be used with `slip39.accounts` to directly obtain any `eth_account.Account` data.

Key	Description
mnemonics	["<mnemonics>", ...]
passphrase	Optional passphrase to decrypt secret

```
recovery = slip39.recover(
    create_details.groups['Mine'][1] + create_details.groups['Fam'][1][:2],
    passphrase=passphrase
)
recoveryhex = codecs.encode( recovery, 'hex_codec' ).decode( 'ascii' )

[[ f"{len(recovery)*8}-bit secret recovered:" ]] + [
 [ f"{recoveryhex[b*32:b*32+32]}" ]
    for b in range( len( recoveryhex ) // 32 )
]
```

0
128-bit secret recovered:
000102030405060708090a0b0c0d0e0f

3 Dependencies

Internally, python-slip39 project uses Trezor's python-shamir-mnemonic to encode the seed data, and the Ethereum project's eth-account to convert seeds to Ethereum accounts.

3.1 The `python-shamir-mnemonic` API

To use it directly, obtain , and install it, or run `python3 -m pip install shamir-mnemonic`.

```
$ shamir create custom --group-threshold 2 --group 1 1 --group 1 1 --group 2 5 --group 3 6
Using master secret: 87e39270d1d1976e9ade9cc15a084c62
Group 1 of 4 - 1 of 1 shares required:
merit aluminum acrobat romp capacity leader gray dining thank rhyme escape genre havoc furl breathe class pitch loc
Group 2 of 4 - 1 of 1 shares required:
merit aluminum beard romp briefing email member flavor disaster exercise cinema subject perfect facility genius bik
Group 3 of 4 - 2 of 5 shares required:
merit aluminum ceramic roster already cinema knit cultural agency intimate result ivory makeup lobe jerky theory ga
merit aluminum ceramic scared beam findings expand broken smear cleanup enlarge coding says destroy agency emperor
merit aluminum ceramic shadow cover smith idle vintage mixture source dish squeeze stay wireless likely privacy imp
merit aluminum ceramic sister duke relate elite ruler focus leader skin machine mild envelope wrote amazing justice
merit aluminum ceramic smug buyer taxi amazing marathon treat clinic rainbow destroy unusual keyboard thumb story l
Group 4 of 4 - 3 of 6 shares required:
merit aluminum decision round bishop wrote belong anatomy spew hour index fishing lecture disease cage thank fantas
```

```
merit aluminum decision scatter carpet spine ruin location forward priest cage security careful emerald screw adult
merit aluminum decision shaft arcade infant argue elevator imply obesity oral venture afraid slice raisin born nerv
merit aluminum decision skin already fused tactics skunk work floral very gesture organize puny hunting voice pytho
merit aluminum decision snake cage premium aide wealthy viral chemical pharmacy smoking inform work cubic ancestor
merit aluminum decision spider boundary lunar staff inside junior tendency sharp editor trouble legal visual tricyc
```

3.2 The `eth-account` API

To creete Ethereum accounts from seed data, two steps are required.

First, derive a Private Key from the seed data plus a derivation path:

```
>>> seed=codecs.decode("dd0e2f02b1f6c92a1a265561bc164135", 'hex_codec')
>>> key=eth_account.hdaccount.key_from_seed(seed, "m/44'/60'/0'/0/0")
>>> keyhex=codecs.encode(key, 'hex_codec')
>>> keyhex
b'178870009416174c9697777b1d94229504e83f25b1605e7bb132aa5b88da64b6'
```

Then, use the private key to obtain the Ethereum account data:

```
>>> keyhex.decode('ascii')
'178870009416174c9697777b1d94229504e83f25b1605e7bb132aa5b88da64b6'
>>> keyhex = '0x'+keyhex.decode('ascii')
>>> keyhex
'0x178870009416174c9697777b1d94229504e83f25b1605e7bb132aa5b88da64b6'
>>> account = eth_account.Account.from_key(keyhex)
>>> account
<eth_account.signers.local.LocalAccount object at 0x7fba368ae670>
>>> account.address
'0x336cBeAB83aCCdb2541e43D514B62DC6C53675f4'
```

4 Conversion from BIP-39 to SLIP-39

If we already have a BIP-39 wallet, it would certainly be nice to be able to create nice, safe SLIP-39 mnemonics for it, and discard the unsafe BIP-39 mnemonics we have lying around, just waiting to be accidentally discovered and the account compromised!

4.1 BIP-39 vs. SLIP-39 Incompatibility

Unforunately, it is **not possible** to convert a BIP-39 derived wallet into a SLIP-39 wallet. Both of these techniques preserve "entropy" (random) bits, but these bits are used **differently** – and incompatibly – to derive the resultant Ethereum wallets.

4.1.1 BIP-39 Entropy to Mnemonic

BIP-39 uses a single set of 12, 15, 18, 21 or 24 BIP-39 words to carefully preserve a specific 128 to 256 bits of initial entropy.

"adult cattle ... remind"   0x9CE8...44   Normalized string Extended

```
from eth_account.hdaccount.mnemonic import Mnemonic
# bip39 = eth_account.hdaccount.generate_mnemonic( 12, "english" )
bip39_english = Mnemonic("english")
entropy = b'\xFF' * 16
entropy_mnemonic = bip39_english.to_mnemonic( entropy )
[[entropy_mnemonic]]
```

0
zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong

Each word is one of a corpus of 2048 words; therefore, each word encodes 11 bits (2048 = 2**11) of entropy. So, we provided 128 bits, but 12*11 = 132. So where does the extra 4 bits of data come from?

It comes from the first few bits of a SHA256 hash of the entropy, which is added to the end of the supplied 128 bits, to reach the required 132 bits: 132 / 11 == 12 words.

This last 4 bits (up to 8 bits, for a 256-bit 24-word BIP-39) is checked, when validating the BIP-39 mnemonic. Therefore, making up a random BIP-39 mnemonic will succeed only 1 / 16 times on average, due to an incorrect checksum 4-bit (16 == 2**4) . Lets check:

```
def random_words( n, count=100 ):
    for _ in range( count ):
        yield ' '.join( random.choice( bip39_english.wordlist ) for _ in range( n ))

successes = sum(
    bip39_english.is_mnemonic_valid( m )
    for i,m in enumerate( random_words( 12, 10000 ))) / 100
[[ f"Valid random 12-word mnemonics:" ]] + [
 [ f"{successes}%" ]] + [
 [ f"~ 1/{100/successes:.3}" ]]
```

0
Valid random 12-word mnemonics:
6.12%
~ 1/16.3

Sure enough, about 1/16 random 12-word phrases are valid BIP-39 mnemonics. OK, we've got the contents of the BIP-39 phrase dialed in. How is it used to generate accounts?

4.1.2 BIP-39 Mnemonic to Seed

Unfortunately, we do **not** use the carefully preserved 128-bit entropy to generate the wallet! Nope, it is stretched to a 512-bit seed using PBKDF2

HMAC SAH512. The normalized **text** of the 12-word mnemonic is then used (with a salt of "mnemonic" plus an optional passphrase, "" by default), to obtain the seed:

```
seed = bip39_english.to_seed( entropy_mnemonic )
seedhex = codecs.encode( seed, 'hex_codec' ).decode( 'ascii' )
[[ f"{len(seed)*8}-bit seed:" ]] + [
 [ f"{seedhex[b*32:b*32+32]}" ]
 for b in range( len( seedhex ) // 32 )
]
```

0
512-bit seed:
b6a6d8921942dd9806607ebc2750416b
289adea669198769f2e15ed926c3aa92
bf88ece232317b4ea463e84b0fcd3b53
577812ee449ccc448eb45e6f544e25b6

Then, this 512-bit seed is used to derive HD wallets. The HD Wallet key derivation process consumes whatever seed entropy is provided (512 bits in this case), and uses HMAC SAH512 with a prefix of b"Bitcoin seed" to stretch the supplied seed entropy to 64 bytes (512 bits). Then, the HD Wallet path segments are iterated through, permuting the first 32 bytes of this material as the key with the second 32 bytes of material as the chain node, until finally the 32-byte (256-bit) Ethereum account private key is produced. We then use this private key to compute the rest of the Ethereum account details, such as its public address.

```
path = "m/66'/40'/0'/0/0"
key = eth_account.hdaccount.key_from_seed( seed, path )
keyhex = '0x' + codecs.encode( key, 'hex_codec' ).decode( 'ascii' )
eth = eth_account.Account.from_key( keyhex )
[[ f"{len(key)*8}-bit derived key at path {path!r}:" ]] + [
 [ f"{keyhex}" ]] + [
 [ "... yields ..." ]] + [
 [ f"Ethereum address: {eth.address}" ]
]
```

0
256-bit derived key at path "m/66'/40'/0'/0/0":
0x9d291e4a972d86ee5a8381a7fc4cd99913a0ec3fd141995e33812c32da40c2fb
. . . yields . . .
Ethereum address: 0xeFCf257B01833cF14229Ed9be0a49b3E026409e9

4.1.3 SLIP-39 Entropy to Mnemonic

```
# We can turn off/on randomness during SLIP-39 generation to get deterministic phrases:
#import shamir_mnemonic
#shamir_mnemonic.shamir.RANDOM_BYTES = lambda n: b'\00' * n
#import secrets
#shamir_mnemonic.shamir.RANDOM_BYTES = secrets.token_bytes
```

```
name,threshold,entropy_slip39,accts = slip39.create(
    "Test", 2, { "Mine": (1,1), "Fam": (2,3) }, entropy, paths=[path] )
[[ f"{g_name}({g_of}/{len(g_mnems)}) #{g_n+1}:" if l_n == 0 else "" ] + words
 for g_name,(g_of,g_mnems) in entropy_slip39.items()
 for g_n,mnem in enumerate( g_mnems )
 for l_n,(line,words) in enumerate(slip39.organize_mnemonic(
        mnem, rows=7, cols=3, label=f"{g_name}({g_of}/{len(g_mnems)}) #{g_n+1}:" ))
]
```

0	1	2	3
Mine(1/1) #1:	1 false	8 benefit	15 receiver
	2 item	9 script	16 smoking
	3 acrobat	10 plot	17 employer
	4 easy	11 cubic	18 skunk
	5 adequate	12 detailed	19 husband
	6 safari	13 sidewalk	20 patent
	7 that	14 reward	
Fam(2/3) #1:	1 false	8 snapshot	15 award
	2 item	9 story	16 device
	3 beard	10 rebuild	17 enjoy
	4 echo	11 slice	18 bracelet
	5 beam	12 nail	19 fused
	6 firm	13 distance	20 pickup
	7 impulse	14 script	
Fam(2/3) #2:	1 false	8 envelope	15 describe
	2 item	9 headset	16 species
	3 beard	10 short	17 fantasy
	4 email	11 flea	18 forget
	5 cover	12 exhaust	19 hesitate
	6 reaction	13 training	20 multiple
	7 juice	14 slim	
Fam(2/3) #3:	1 false	8 valuable	15 kidney
	2 item	9 downtown	16 episode
	3 beard	10 exclude	17 hour
	4 entrance	11 timely	18 tolerate
	5 clogs	12 desert	19 remind
	6 math	13 force	20 memory
	7 average	14 stadium	

Since there is some randomess in the SLIP-39 mnemonics generation process, we'll get a different set of words each time for the fixed "entropy" 0xFFFF..FF used in this example, but we'll **always** derive the same Ethereum account 0x3224..56c5 at the specified HD Wallet derivation path.

```
[[ "HD Wallet Path:", "Ethereum Address:" ]] + [
 [ path, eth.address ]
 for path,eth in accts.items()
]
```

0	1
HD Wallet Path:	Ethereum Address:
m/66'/40'/0'/0/0	0x322408FCF0dAB471570038DEA08536780aAB56c5

4.1.4 SLIP-39 Mnemonic to Seed

Lets prove that we can actually recover the **original** entropy from the SLIP-39 recovery mnemonics; in this case, we've specified a SLIP-39 group_threshold of 2 groups, so we'll use all 1 mnemonic from Mine, and 2 from Fam:

```
_,mnem_mine = entropy_slip39['Mine']
_,mnem_fam =  entropy_slip39['Fam']
recseed = slip39.recover( mnem_mine + mnem_fam[:2] )
recseedhex = codecs.encode( recseed, 'hex_codec' ).decode( 'ascii' )

[[ f"{len(recseed)*8}-bit seed:" ]] + [
 [ f"{recseedhex[b*32:b*32+32]}" ]
    for b in range( len( recseedhex ) // 32 )
]
```

0
128-bit seed:
ffffffffffffffffffffffffffffffff

And we'll use the same style of code as for the BIP-39 example above, to derive the Ethereum address from this 128-bit seed:

```
reckey = eth_account.hdaccount.key_from_seed( recseed, path )
reckeyhex = '0x' + codecs.encode( reckey, 'hex_codec' ).decode( 'ascii' )
receth = eth_account.Account.from_key( reckeyhex )
[[ f"{len(reckey)*8}-bit derived key at path {path!r}:" ]] + [
 [ f"{reckeyhex}" ]] + [
 [ "... yields ..." ]] + [
 [ f"Ethereum address: {receth.address}" ]
]
```

0
256-bit derived key at path "m/66'/40'/0'/0/0":
0x738fc6d8dd28f75027ec04b1c64cead968bbae0d9b15de2dab664e5b59db04f3
... yields ...
Ethereum address: 0x322408FCF0dAB471570038DEA08536780aAB56c5

And we see that we obtain the same Ethereum address `0x3224..56c5` as we originally got from `slip39.create` above.

4.2 BIP-39 vs SLIP-39 Key Derivation Summary

At no time in BIP-39 account derivation is the original 128-bit mnemonic entropy used directly in the derivation of the wallet key. This differs from SLIP-39, which directly uses the 128-bit mnemonic entropy recovered from the SLIP-39 Shamir's Secret Sharing System recovery process to generate each HD Wallet account's private key.

Furthermore, there is no point in the BIP-39 entropy to account generation where we **could** introduce a known 128-bit seed and produce a known Ethereum wallet from it, other than as the very beginning.

Therefore, SLIP-39 and BIP-39 HD Wallet generation are fundamentally distinct; we cannot produce BIP-39 and SLIP-39 mnemonics that result in the same wallet. To convert your funds from a BIP-39 wallet to a SLIP-39 wallet will require **moving** the funds.