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/44'/60'/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 permanently lost. Therefore, it must be backed up reliably, and be readily accessible.

Therefore, we must:

- Ensure that nobody untrustworthy can recover the seed, but
- Store the seed in many places with several (some perhaps untrustworthy) 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 successfully 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.

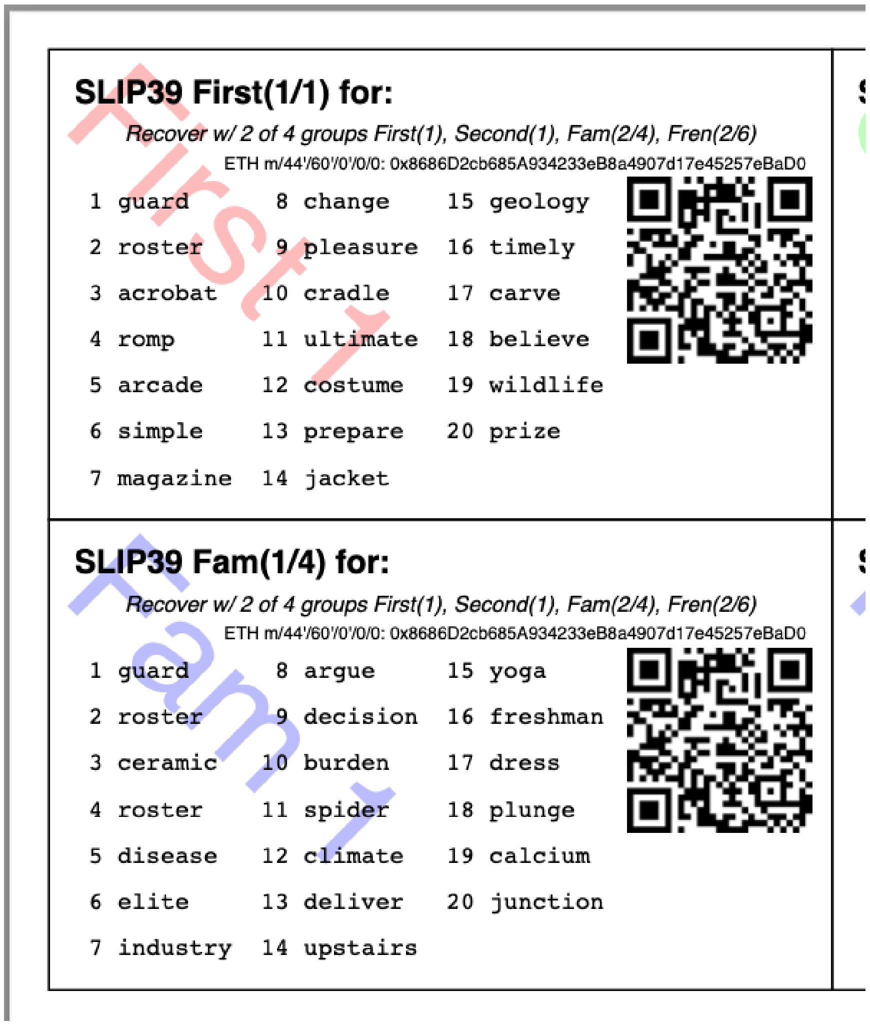
Then, we recover the master secret seed in hex with `slip39-recovery`, and finally send it to `slip39 --secret -` to re-generate the same wallet as we originally created.

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

```
2022-01-03 10:53:52 slip39           ETH m/44'/60'/0'/0/0    : 0x4AC7950dDc18479Cbe46b5F75a0001a08843d854
2022-01-03 10:53:53 slip39.recovery  Recovered 128-bit SLIP-39 secret with 4 (1st, 2nd, 7th, 8th) of 8 supplied mne
2022-01-03 10:53:53 slip39           ETH m/44'/60'/0'/0/0    : 0x4AC7950dDc18479Cbe46b5F75a0001a08843d854
```

2.2 Recovery Mnemonic Cards PDF

This is what the output SLIP-39 mnemonic cards PDF looks like:



SLIP39 First(1/1) for:

Recover w/ 2 of 4 groups First(1), Second(1), Fam(2/4), Fren(2/6)

ETH m/44'/60'/0'/0/0: 0x8686D2cb685A934233eB8a4907d17e45257eBaD0

1 guard	8 change	15 geology
2 roster	9 pleasure	16 timely
3 acrobat	10 cradle	17 carve
4 romp	11 ultimate	18 believe
5 arcade	12 costume	19 wildlife
6 simple	13 prepare	20 prize
7 magazine	14 jacket	

SLIP39 Fam(1/4) for:

Recover w/ 2 of 4 groups First(1), Second(1), Fam(2/4), Fren(2/6)

ETH m/44'/60'/0'/0/0: 0x8686D2cb685A934233eB8a4907d17e45257eBaD0

1 guard	8 argue	15 yoga
2 roster	9 decision	16 freshman
3 ceramic	10 burden	17 dress
4 roster	11 spider	18 plunge
5 disease	12 climate	19 calcium
6 elite	13 deliver	20 junction
7 industry	14 upstairs	

Figure 1: SLIP39 Mnemonic Cards PDF

2.3 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.

2.3.1 `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

#
# NOTE:
#
# We turn off randomness here during SLIP-39 generation to get deterministic phrases;
# during normal operation, secure entropy is used during mnemonic generation, yielding
# random phrases, even when the same seed is used multiple times.
#
import shamir_mnemonic
shamir_mnemonic.shamir.RANDOM_BYTES = lambda n: b'\00' * n

import eth_account
import slip39

paths               = [ "m/44'/60'/0'/0/0", "m/44'/60'/0'/0/1" ]
master_secret       = b'\xFF' * 16
passphrase          = b""
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, label=f"{g_name}({g_of}/{len(g_mnems)}) #{g_n+1}:" ))
]
```

0	1	2	3
Mine(1/1) #1:	1 academic	8 safari	15 standard
	2 acid	9 drug	16 angry
	3 acrobat	10 browser	17 similar
	4 easy	11 trash	18 aspect
	5 change	12 fridge	19 smug
	6 injury	13 busy	20 violence
	7 painting	14 finger	
Fam(2/3) #1:	1 academic	8 prevent	15 dwarf
	2 acid	9 mouse	16 dream
	3 beard	10 daughter	17 flavor
	4 echo	11 ancient	18 oral
	5 crystal	12 fortune	19 chest
	6 machine	13 ruin	20 marathon
	7 bolt	14 warmth	
Fam(2/3) #2:	1 academic	8 prune	15 briefing
	2 acid	9 pickup	16 often
	3 beard	10 device	17 escape
	4 email	11 device	18 sprinkle
	5 dive	12 peanut	19 segment
	6 warn	13 enemy	20 devote
	7 ranked	14 graduate	
Fam(2/3) #3:	1 academic	8 dining	15 intimate
	2 acid	9 invasion	16 satoshi
	3 beard	10 bumpy	17 hobo
	4 entrance	11 identify	18 ounce
	5 alarm	12 anxiety	19 both
	6 health	13 august	20 award
	7 discuss	14 sunlight	

Add the resultant HD Wallet addresses:

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

0	1
m/44'/60'/0'/0/0	0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1
m/44'/60'/0'/0/1	0x8D342083549C635C0494d3c77567860ee7456963

2.3.2 `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 )
```

2.3.3 `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.

Note that the passphrase is **not** checked; entering a different passphrase for the same set of mnemonics will recover a **different** wallet! This is by design; it allows the holder of the SLIP-39 mnemonic phrases to recover a "decoy" wallet by supplying a specific passphrase, while protecting the "primary" wallet.

Therefore, it is **essential** to remember any non-default (empty) passphrase used, separately and securely. Take great care in deciding if you wish to use a passphrase with your SLIP-39 wallet!

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

```
recoverydecoy       = slip39.recover(
    create_details.groups['Mine'][1][:] + create_details.groups['Fam'][1][:2],
    passphrase=b"wrong!"
)
recoverydecoyhex    = codecs.encode( recoverydecoy, 'hex_codec' ).decode( 'ascii' )

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

[[ f"{len(recoverydecoy)*8}-bit secret w/ decoy password recovered:" ]] + [
 [ f"{recoverydecoyhex[b*32:b*32+32]}" ]
    for b in range( len( recoverydecoyhex ) // 32 )
] +  [[ f"{len(recoveryvalid)*8}-bit secret recovered:" ]] + [
 [ f"{recoveryvalidhex[b*32:b*32+32]}" ]
    for b in range( len( recoveryvalidhex ) // 32 )
]
```

0
128-bit secret w/ decoy password recovered:
2e522cea2b566840495c220cf79c756e
128-bit secret recovered:
ffffffffffffffffffffffffffffffff

3 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!

3.1 BIP-39 vs. SLIP-39 Incompatibility

Unfortunately, it is **not possible** to cleanly 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.

The best we can do is to preserve the 512-bit **output** of the BIP-39 mnemonic phrase as a set of 512-bit SLIP-39 mnemonics.

3.1.1 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. Here's a 128-bit (12-word) example using some fixed "entropy" `0xFFFF..FFFF`:

```
from eth_account.hdaccount.mnemonic import Mnemonic
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:
6.28%
~ 1/15.9
```

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?

3.1.2 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 SHA512. The normalized **text** (*not the entropy bytes*) 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
```

3.1.3 BIP-39 Seed to Address

Finally, this 512-bit seed is used to derive HD wallet(s). The HD Wallet key derivation process consumes whatever seed entropy is provided (512 bits in the case of BIP-39), and uses HMAC SHA512 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/44'/60'/0'/0/0"
key                 = eth_account.hdaccount.key_from_seed( seed, path )
```

```
keyhex              = codecs.encode( key, 'hex_codec' ).decode( 'ascii' )
eth_hd              = eth_account.Account.from_key( keyhex )

[[ f"{len(key)*8}-bit derived key at path {path!r}:" ]] + [
 [ f"{keyhex}" ]] + [
 [ "... yields ..." ]] + [
 [ f"Ethereum address: {eth_hd.address}" ]
]
```

```
 0
 256-bit derived key at path "m/44'/60'/0'/0/0":
 7af65ba4dd53f23495dcb04995e96f47c243217fc279f10795871b725cd009ae
 ... yields ...
 Ethereum address: 0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E
```

Thus, we see that while the 12-word BIP-39 mnemonic careful preserves the original 128-bit entropy, this data is not directly used to derive the wallet private key and address. Also, since an irreversible hash is used to derive the seed from the mnemonic, we can't reverse the process on the seed to arrive back at the BIP-39 mnemonic phrase.

3.1.4 SLIP-39 Entropy to Mnemonic

Just like BIP-39 carefully preserves the original 128-bit entropy bytes in a single 12-word mnemonic phrase, SLIP-39 preserves the original 128-bit entropy in a *set* of 30-word mnemonic phrases.

```
name,thrs,grps,acct = 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 grps.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 academic	8 safari	15 standard
	2 acid	9 drug	16 angry
	3 acrobat	10 browser	17 similar
	4 easy	11 trash	18 aspect
	5 change	12 fridge	19 smug
	6 injury	13 busy	20 violence
	7 painting	14 finger	
Fam(2/3) #1:	1 academic	8 prevent	15 dwarf
	2 acid	9 mouse	16 dream
	3 beard	10 daughter	17 flavor
	4 echo	11 ancient	18 oral
	5 crystal	12 fortune	19 chest
	6 machine	13 ruin	20 marathon
	7 bolt	14 warmth	
Fam(2/3) #2:	1 academic	8 prune	15 briefing
	2 acid	9 pickup	16 often
	3 beard	10 device	17 escape
	4 email	11 device	18 sprinkle
	5 dive	12 peanut	19 segment
	6 warn	13 enemy	20 devote
	7 ranked	14 graduate	
Fam(2/3) #3:	1 academic	8 dining	15 intimate
	2 acid	9 invasion	16 satoshi
	3 beard	10 bumpy	17 hobo
	4 entrance	11 identify	18 ounce
	5 alarm	12 anxiety	19 both
	6 health	13 august	20 award
	7 discuss	14 sunlight	

Since there is some randomness used in the SLIP-39 mnemonics generation process, we would get a **different** set of words each time for the fixed "entropy" `0xFFFF..FF` used in this example (if we hadn't manually disabled entropy for `shamir_mnemonic`, above), but we will **always** derive the same Ethereum account `0x824b..19a1` at the specified HD Wallet derivation path.

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

0	1
HD Wallet Path:	Ethereum Address:
m/44'/60'/0'/0/0	0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1

3.1.5 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 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

3.1.6 SLIP-39 Seed to Address

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

```
reckey              = eth_account.hdaccount.key_from_seed( recseed, path )
reckeyhex           = 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/44'/60'/0'/0/0":
6a2ec39aab88ec0937b79c8af6aaf2fd3c909e9a56c3ddd32ab5354a06a21a2b
... yields ...
Ethereum address: 0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1

And we see that we obtain the same Ethereum address `0x824b..1a2b` as we originally got from `slip39.create` above. However, this is **not** the Ethereum wallet address obtained from BIP-39 with exactly the same `0xFFFF...FF` entropy, which was `0xfc20..1B5E`. This is due to the fact that BIP-39 does not use the recovered entropy to produce the seed like SLIP-39 does, but applies additional one-way hashing of the mnemonic to produce the seed.

3.2 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.

3.2.1 BIP-39 Backup via SLIP-39

There is one approach which can preserve an original BIP-39 wallet address, using SLIP-39 mnemonics.

It is clumsy, as it preserves the BIP-39 **output** 512-bit stretched seed, and the resultant 59-word SLIP-39 mnemonics cannot be used (at present) with the Trezor hardware wallet. They can, however, be used to recover the HD wallet private keys without access to the original BIP-39 mnemonic phrase – you could generate and distribute a set of more secure SLIP-39 mnemonic phrases, instead of trying to secure the original BIP-39 mnemonic.

We'll use `slip39.recovery --bip39 ...` to recover the 512-bit stretched seed from BIP-39:

```
   ( python3 -m slip39.recovery --bip39 \
         --mnemonic "zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong"
    ) 2>&1
```

```
2022-01-03 10:54:09 slip39.recovery  Recovered 512-bit BIP-39 secret from english mnemonic
b6a6d8921942dd9806607ebc2750416b289adea669198769f2e15ed926c3aa92bf88ece232317b4ea463e84b0fcd3b53577812ee449ccc448eb
```

Then we can generate a 59-word SLIP-39 mnemonic set from the 512-bit secret:

```
   ( python3 -m slip39.recovery --bip39 \
         --mnemonic "zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong" \
       | python3 -m slip39 --secret - --no-card -q ) 2>&1
```

```
2022-01-03 10:54:10 slip39.recovery  Recovered 512-bit BIP-39 secret from english mnemonic
2022-01-03 10:54:10 slip39           ETH m/44'/60'/0'/0/0    : 0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E
```

This `0xfc20..1B5E` address is the same Ethereum address as is recovered on a Trezor using this BIP-39 mnemonic phrase.

4 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.

4.1 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
```

4.2 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'
```