# SLIP-39 Wallet "Seed" Generation & Backup

Perry Kundert

2021-12-20 10:55:00

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

A 12- or 24-word BIP-39 seed recovery Mmnemonic Phrase helps, but a **single** lapse in security dooms the account (and all derived accounts, in fact). If someone finds your recovery phrase (or you lose it), the accounts derived from that seed are *gone*.

The SLIP-39 standard allows you to split the seed between 1, 2, or more groups of several mnemonic recovery phrases. This is better, but creating such accounts is difficult; presently, only the Trezor supports these directly, and they can only be created "manually". Writing down 5 or more sets of 20 words is difficult, error-prone and time consuming.

## Contents

# 1 Hardware Wallet "Seed" Configuration

> Your keys, your Bitcoin. Not your keys, not your Bitcoin.
>
> —Andreas Antonopoulos

The python-slip39 project (and the SLIP-39 macOS/win32 App) exists to assist in the safe creation, backup and documentation of Hierarchical Deterministic (HD) Wallet seeds and derived accounts, with various SLIP-39 sharing parameters. It generates the new random wallet seed, and generates the expected standard Ethereum account(s) (at derivation path **m/44'/60'/0'/0/0** by default) and Bitcoin accounts (at Bech32 derivation path **m/84'/0'/0'/0/0** by default), with wallet address and QR code (compatible with Trezor and Ledger derivations). It produces the required SLIP-39 phrases, and outputs a single PDF containing all the required printable cards to document the seed (and the specified derived accounts).

On an secure (ideally air-gapped) computer, new seeds can *safely* be generated (**without trusting this program**) and the PDF saved to a USB drive for printing (or directly printed without the file being saved to disk.). Presently, `slip39` can output example ETH, BTC, LTC, DOGE, BSC, and XRP addresses derived from the seed, to *illustrate* what accounts are associated with the backed-up seed. Recovery of the seed to a Trezor "Model T" or a (newer, less costly) "Model One" is simple, by entering the mnemonics right on the device.

We also support the backup of existing insecure and unreliable 12- or 24-word BIP-39 Mnemonic Phrases as SLIP-39 Mnemonic cards, for existing BIP-39 hardware wallets like the Ledger Nano, etc.! Recover from your existing BIP-39 Seed Phrase Mnemonic, select "Using BIP-39" (and enter your BIP-39 passphrase), and generate a set of SLIP-39 Mnemonic cards. Later, use the SLIP-39 App to recover from your SLIP-39 Mnemonic cards, click "Using BIP-39" to get your BIP-39 Mnemonic back, and use it (and your passphrase) to recover your accounts to your Ledger (or other) hardware wallet.

Output of BIP-38 or JSON encrypted Paper Wallets is also supported, for import into standard software cryptocurrency wallets.

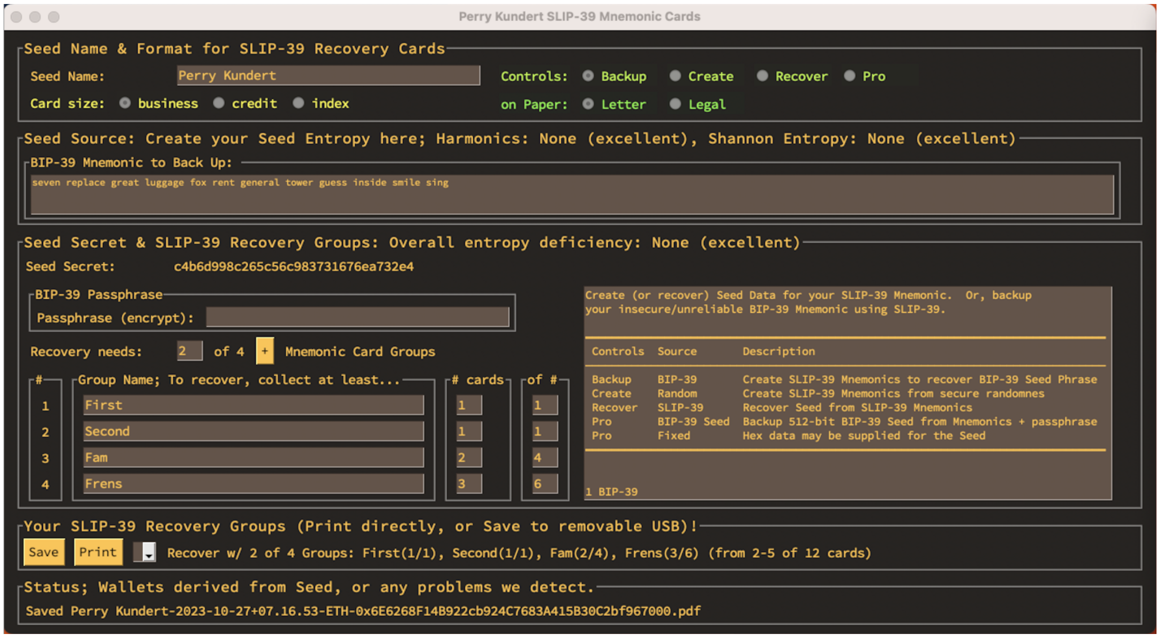



Figure 1: SLIP-39 App GUI

## 1.1 TL;DR Backup and Recover your BIP-39 Mnemonic

Here's a full round-trip demonstration of:

- Creating new (or "Backing Up" existing) Seed Entropy as a BIP-39 Mnemonic
- Recovering the Seed Entropy from SLIP-39 (via `https://iancoleman.io/slip39/`)
- Recovering the original BIP-39 (via `https://iancoleman.io/bip39/`)

First, we generate SLIP-39 Cards representing a BIP-39 Mnemonic seed. Remember, your BIP-39 Mnemonic simply encodes your 128- or 256-bit Seed Entropy. So, we're not backing up your Mnemonic phrase – we're backing up the raw seed data that is encoded into your BIP-39 Mnemonic.

```
# python3 -m pip install slip39; slip39 -q --using-bip39  # to generate one from scratch, or
slip39 -q --secret "seven replace great luggage fox rent general tower guess inside smile sing"
```

SLIP39-2024-05-15+10.04.19-ETH-0x6E6268F14B922cb924C7683A415B30C2bf967000.pdf

If you look at the generated SLIP39 PDF, you'll see that the cover page contains the original BIP-39 Mnemonic phrase (for confirmation), and generates a number of SLIP-39 Mnemonic cards. These cards encode the original Seed Entropy, and are what you use to recover the BIP-39 Mnemonic whenever you need it.

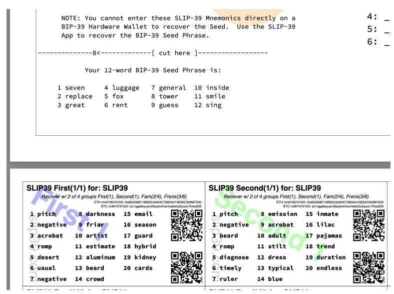



Figure 2: SLIP-39 Backup

I **recommend** that you *tear off and destroy* the BIP-39 Mnemonic from the cover sheet, once you've confirmed you can recover it anytime you want, and you've set up your hardware wallet, and confirmed that it contains the same cryptocurrency addresses displayed in the PDF.

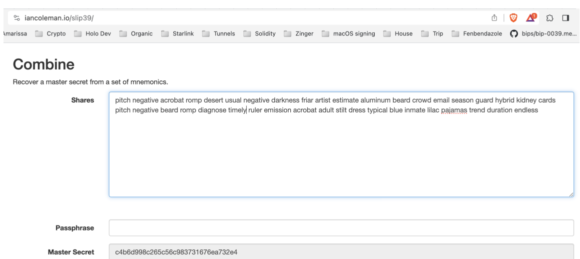



Figure 3: BIP-39 Entropy

Practice this full round-trip several times with a bad BIP-39 Mnemonic like "zoo zoo ... wrong". This is the only way to become comfortable with your ability to recover your original seed data, and (hence) your BIP-39 Mnemonic.

Later, when you need to recover your BIP-39 Seed Entropy and Mnemonic, use this SLIP-39 App or `https://iancoleman.io/slip39/` and enter some of your SLIP-39 Mnemonic Cards. These may need to be collected from friends and family.

In this case, we're using the First and Second cards, intended for you to secure, separately from each other; for example, in two safes or other secure locations like locked filing cabinets, at 2 locations known to you and your partner(s):

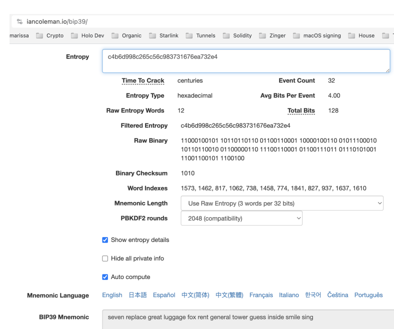



Figure 4: BIP-39 Mnemonic

Finally, convert the recovered Seed Entropy back to your BIP-39 Mnemonic. This requires 2 steps if you use `https://iancoleman.io/bip39/`

In this step, we're simply converting the recovered Seed Entropy back into its BIP-39 Mnemonic. You need to select the "[X] show entropy details" checkbox in order to enter the raw Seed Entropy we've recovered in the last step:

Alternatively, you can use the SLIP-39 App or the `slip39-recovery` command-line tool, and do it all in one step. This illustrates recovering your BIP-39 Mnemonic from the SLIP-39 Cards generated in the first step:

```
python3 -m slip39.recovery --using-bip39 \
  -m "pitch negative acrobat romp desert usual negative darkness friar artist estimate aluminum beard crowd email season g
  -m "pitch negative beard romp diagnose timely ruler emission acrobat adult stilt dress typical blue inmate lilac pajamas
```

seven replace great luggage fox rent general tower guess inside smile sing

## 2 Security with Availability

For both BIP-39 and SLIP-39, a 128- or 256-bit random "seed" is the source of an unlimited sequence of Ethereum and Bitcoin Heirarchical Deterministic (HD) derived Wallet accounts. Anyone who can obtain this seed gains control of all Ethereum, Bitcoin (and other) accounts derived from it, so it must be securely stored.

Losing this seed means that all of the HD Wallet accounts are permanently lost. It must be *both* backed up securely, *and* be readily accessible.

Therefore, we must:

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

How can we address these conflicting requirements?

### 2.1 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. This selection is both insecure (easy to accidentally disclose) and unreliable (easy to accidentally lose), but since most hardware wallets **only** accept BIP-39 phrases, we also provide a way to *backup your BIP-39 phrase* using SLIP-39!

For SLIP-39, you specify a "group_threshold" of *how many* of your groups must be successfully collected, to recover the seed; this seed is (conceptually) split between 1 or more groups (though not in reality – each group's data *alone* gives away *no information* about the seed).

For example, you might have First, Second, Fam and Frens groups, and decide that any 2 groups can be combined to recover the seed. 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 | 3 | / | 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 are 1-card groups.)

If the Seed 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 all of the HD Wallet accounts.

Only 2 Fam group member's cards must be collected to recover the Fam group's data. So, if the HD Wallet owner loses their home (and the one and only First group card) in a fire, they could get the one Second group card from the office safe, and also 2 cards from Fam group members, and recover the Seed and all of their wallets.

If catastrophe strikes and the wallet owner dies, and the heirs don't have access to either the First (at home) or Second (at the office) cards, they can collect 2 Fam cards and 3 Frens cards (at the funeral, for example), completing the Fam and Frens groups' data, and recover the Seed, and all derived HD Wallet accounts.

Since Frens are less likely to persist long term, we'll produce more (6) of these cards. Depending on how trustworthy the group is, adjust the Fren group's Required number higher (less trustworthy, more likely to know each-other, need to collect more to recover the group), or lower (more trustworthy, less likely to collude, need less to recover).

# 3 SLIP-39 Account Creation, Recovery and Generation

Generating a new SLIP-39 encoded Seed is easy, with results available as PDF and text. Any number of derived HD wallet account addresses can be generated from this Seed, and the Seed (and all derived HD wallets, for all cryptocurrencies) can be recovered by collecting the desired groups of recover card phrases. The default recovery groups are as described above.

## 3.1 Creating New SLIP-39 Recoverable Seeds

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

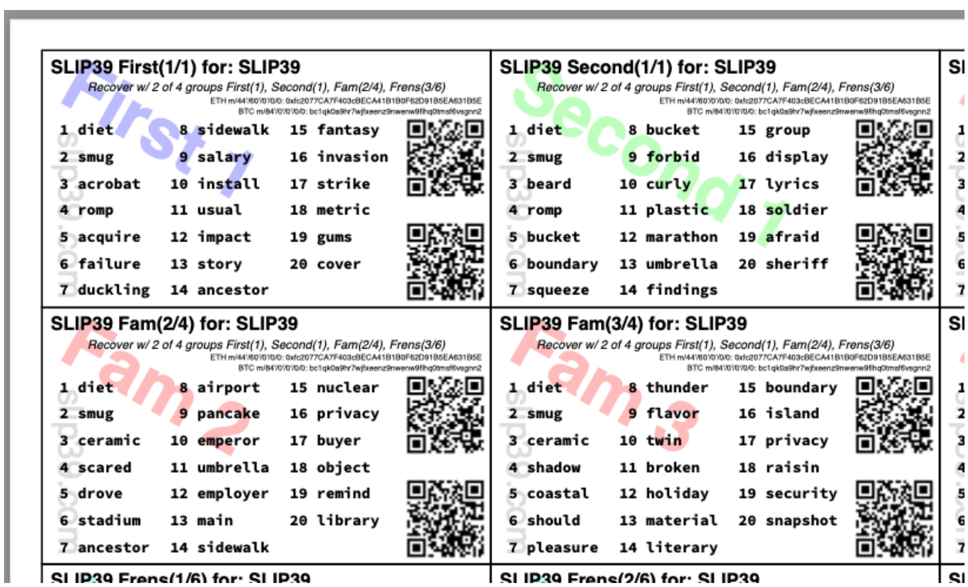



Figure 5: SLIP-39 Cards PDF (from `--secret ffff...`)

Run the following to obtain a PDF file containing business cards with the default SLIP-39 groups for a new account Seed named "Personal" (usable with any hardware wallet with SLIP-39 support, such as the Trezor "Model T") ; 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"
2022-11-22 05:35:21 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0x0F04cab1855CE275bd098c918075373EB3944Ba3
2022-11-22 05:35:21 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1qszvts5vyxy265er6ngk3ew4utx5sll2ck2m7m2
2022-11-22 05:35:22 slip39.layout    Writing SLIP39-encoded wallet for 'Personal' to:\
  Personal-2022-11-22+05.35.22-ETH-0x0F04cab1855CE275bd098c918075373EB3944Ba3.pdf
```

The resultant PDF will be output into the designated file.

This PDF file contains business card sized SLIP-39 Mnemonic cards, and will print on a single page of 8-1/2"x11" paper or card stock, and the cards can be cut out (`--card index`, `credit`, `half` (page), `third` and `quarter` are also available, as well as 4x6 `photo` and custom `"(<h>,<w>),<margin>"`).

To get the data printed on the terminal as in this example (so you could write it down on cards instead), add a `-v` (to see it logged in a tabular format), or `--text` to have it printed to stdout in full lines (ie. for pipelining to other programs).

### 3.1.1 BIP-39 Mnemonic Phrase Backup using SLIP-39

To obtain the Seed in BIP-39 format, with its original "entropy" backed up using SLIP-39 (supporting any BIP-39 hardware wallet, and recoverable from the Mnemonic cards using SLIP-39), use the `--using-bip39` option:

```
$ slip39 --using-bip39 Personal-BIP-39
2022-11-22 05:47:13 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0x927232296120343A89DeAb15F108a420087a2Ef3
2022-11-22 05:47:13 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1qgs6xg5kvrrxp4579y22a4tf0d8me4dslwxjr9x
2022-11-22 05:47:15 slip39.layout    Writing SLIP39 backup for BIP-39-encoded wallet for 'Personal-BIP-39' to:\
  Personal-BIP-39-2022-11-22+05.47.15-ETH-0x927232296120343A89DeAb15F108a420087a2Ef3.pdf
```

This is the best approach, if you want a new Seed and need to support a BIP-39-only Hardware Wallet. (If you already have a BIP-39 Mnemonic Phrase, see **??**)

### 3.1.2 Paper Wallets for Software Wallet Support

The Trezor hardware wallet natively supports the input of SLIP-39 Mnemonics. However, most software wallets do not (yet) support SLIP-39. So, how do we load the Crypto wallets produced from our Seed into software wallets such as the Metamask plugin or the Brave browser, for example?

The `slip39.gui` (and the macOS/win32 SLIP-39.App) support output of standard BIP-38 encrypted wallets for Bitcoin-like cryptocurrencies such as BTC, LTC and DOGE. It also outputs encrypted Ethereum JSON wallets for ETH. Here is how to produce them (from a test secret Seed; exclude `--secret ffff...` for yours!):

```
slip39 -c ETH -c BTC -c DOGE -c LTC --secret ffffffffffffffffffffffffffffffff \
    --no-card --wallet password --wallet-hint 'bad:pass...' 2>&1
```

```
2024-05-15 10:04:22 slip39          It is recommended to not use '-s|--secret <hex>'; specify '-' to read from input
2024-05-15 10:04:22 slip39          It is recommended to not use '-w|--wallet <password>'; specify '-' to read from input
2024-05-15 10:04:22 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1
2024-05-15 10:04:22 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1q9yscq3l2yfxlvnlk3cszpqefparrv7tk24u6pl
2024-05-15 10:04:22 slip39.layout    DOGE   m/44'/3'/0'/0/0     : DN8PNN3dipSJpLmyxtGe4EJH38EhqF8Sfy
2024-05-15 10:04:22 slip39.layout    LTC    m/84'/2'/0'/0/0     : ltc1qe5m2mst9kjcqtfpapaanaty40qe8xtusmq4ake
2024-05-15 10:04:26 slip39.layout    Writing SLIP39-encoded wallet for 'SLIP39' to: SLIP39-2024-05-15+10.04.24-ETH-0x824b1
SLIP39-2024-05-15+10.04.24-ETH-0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1.pdf
```

And what they look like:

Figure 6: Paper Wallets (from `--secret ffff...`)

To recover your real SLIP-39 Seed Entropy and print wallets, use the SLIP-39 App's "Recover" Controls, or to do so on the command-line, use `slip39-recover`:

```
slip39-recovery -v \
    --mnemonic "material leaf acrobat romp charity capital omit skunk change firm eclipse crush fancy best tracks flip gro
    --mnemonic "material leaf beard romp disaster duke flame uncover group slice guest blue gums duckling total suitable t
        2>&1

2024-05-15 10:04:27 slip39.recovery  Recovered 128-bit SLIP-39 Seed Entropy with 2 (all) of 2 supplied mnemonics; Seed dec
2024-05-15 10:04:27 slip39.recovery  Recovered SLIP-39 secret; To re-generate SLIP-39 wallet, send it to: python3 -m slip3
ffffffffffffffffffffffffffffffff
```

You can run this as a command-line pipeline. Here, we use some SLIP-39 Mnemonics that encode the `ffff...` Seed Entropy; note that the wallets match those output above:

```
slip39-recovery \
    --mnemonic "material leaf acrobat romp charity capital omit skunk change firm eclipse crush fancy best tracks flip gro
    --mnemonic "material leaf beard romp disaster duke flame uncover group slice guest blue gums duckling total suitable t
| slip39 -c ETH -c BTC -c DOGE -c LTC --secret - \
    --no-card --wallet password --wallet-hint 'bad:pass...' \
        2>&1
```

```
2024-05-15 10:04:28 slip39          It is recommended to not use '-w|--wallet <password>'; specify '-' to read from input
2024-05-15 10:04:28 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1
2024-05-15 10:04:28 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1q9yscq3l2yfxlvnlk3cszpqefparrv7tk24u6pl
2024-05-15 10:04:28 slip39.layout    DOGE   m/44'/3'/0'/0/0     : DN8PNN3dipSJpLmyxtGe4EJH38EhqF8Sfy
2024-05-15 10:04:28 slip39.layout    LTC    m/84'/2'/0'/0/0     : ltc1qe5m2mst9kjcqtfpapaanaty40qe8xtusmq4ake
2024-05-15 10:04:33 slip39.layout    Writing SLIP39-encoded wallet for 'SLIP39' to: SLIP39-2024-05-15+10.04.31-ETH-0x824b1
SLIP39-2024-05-15+10.04.31-ETH-0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1.pdf
```

#### 3.1.3 Supported Cryptocurrencies

While the SLIP-39 Seed is not cryptocurrency-specific (any wallet for any cryptocurrency can be derived from it), each type of cryptocurrency has its own standard derivation path (eg. `m/44'/3'/0'/0/0` for DOGE), and its own address representation (eg. Bech32 at `m/84'/0'/0'/0/0` for BTC eg. `bc1qcupw7k8enymvvsa7w35j5hq4ergtvus3zk8a8s`).

When you import your SLIP-39 Seed into a Trezor, you gain access to all derived HD cryptocurrency wallets supported directly by that hardware wallet, and **indirectly**, to any coin and/or blockchain network supported by any wallet software (eg. Metamask).

| Crypto | Semantic | Path | Address | Support |
|---|---|---|---|---|
| ETH | Legacy | m/44'/60'/0'/0/0 | 0x... | |
| BSC | Legacy | m/44'/60'/0'/0/0 | 0x... | Beta |
| BTC | Legacy | m/44'/ 0'/0'/0/0 | 1... | |
| | SegWit | m/49'/ 0'/0'/0/0 | 3... | |
| | Bech32 | m/84'/ 0'/0'/0/0 | bc1... | |
| LTC | Legacy | m/44'/ 2'/0'/0/0 | L... | |
| | SegWit | m/49'/ 2'/0'/0/0 | M... | |
| | Bech32 | m/84'/ 2'/0'/0/0 | ltc1... | |
| DOGE | Legacy | m/44'/ 3'/0'/0/0 | D... | |

1. ETH, BTC, LTC, DOGE

   These coins are natively supported both directly by the Trezor hardware wallet, and by most software wallets and "web3" platforms that interact with the Trezor, or can import the BIP-38 or Ethereum JSON Paper Wallets produced by `python-slip39`.

2. Binance Smart Chain (BSC): binance.com

   The Binance Smart Chain uses standard Ethereum addresses; support for the BSC is added directly to the wallet software; here are the instructions for adding BSC support for the Trezor hardware wallet, using the Metamask software wallet. In `python-slip39`, BSC is simply an alias for ETH, since the wallet addresses and Ethereum JSON Paper Wallets are identical.

### 3.2 The macOS/win32 `SLIP-39.app` GUI App

If you prefer a graphical user-interface, try the macOS/win32 SLIP-39.App. You can run it directly if you install Python 3.9+ from python.org/downloads or using homebrew `brew install python-tk@3.10`. Then, start the GUI in a variety of ways:

```
slip39-gui
python3 -m slip39.gui
```

Alternatively, download and install the macOS/win32 GUI App .zip, .pkg or .dmg installer from github.com/pjkundert/python-slip-39/releases.

### 3.3 The Python `slip39` CLI

From the command line, you can create SLIP-39 Seed Mnemonic card PDFs.

#### 3.3.1 `slip39` Synopsis

The full command-line argument synopsis for `slip39` is:

```
slip39 --help 2>&1               | sed 's/^/: /' # (just for output formatting)

usage: slip39 [-h] [-v] [-q] [-o OUTPUT] [-t THRESHOLD] [-g GROUP] [-f FORMAT]
              [-c CRYPTOCURRENCY] [-p PATH] [-j JSON] [-w WALLET]
              [--wallet-hint WALLET_HINT] [--wallet-format WALLET_FORMAT]
              [-s SECRET] [--bits BITS] [--using-bip39]
              [--passphrase PASSPHRASE] [-C CARD] [--no-card] [--paper PAPER]
              [--cover] [--no-cover] [--text] [--watermark WATERMARK]
              [--double-sided] [--no-double-sided] [--single-sided]
              [names ...]

Create and output SLIP-39 encoded Seeds and Paper Wallets to a PDF file.

positional arguments:
  names                 Account names to produce; if --secret Entropy is
                        supplied, only one is allowed.

options:
  -h, --help            show this help message and exit
  -v, --verbose         Display logging information.
  -q, --quiet           Reduce logging output.
  -o OUTPUT, --output OUTPUT
                        Output PDF to file or '-' (stdout); formatting w/
                        name, date, time, crypto, path, address allowed
  -t THRESHOLD, --threshold THRESHOLD
                        Number of groups required for recovery (default: half
                        of groups, rounded up)
  -g GROUP, --group GROUP
                        A group name[[<require>/]<size>] (default: <size> = 1,
                        <require> = half of <size>, rounded up, eg.
                        'Frens(3/5)' ).
  -f FORMAT, --format FORMAT
                        Specify crypto address formats: legacy, segwit,
                        bech32; default: ETH:legacy, BTC:bech32, LTC:bech32,
                        DOGE:legacy, BSC:legacy, XRP:legacy
  -c CRYPTOCURRENCY, --cryptocurrency CRYPTOCURRENCY
                        A crypto name and optional derivation path (eg.
                        '../<range>/<range>'); defaults: ETH:m/44'/60'/0'/0/0,
                        BTC:m/84'/0'/0'/0/0, LTC:m/84'/2'/0'/0/0,
                        DOGE:m/44'/3'/0'/0/0, BSC:m/44'/60'/0'/0/0,
                        XRP:m/44'/144'/0'/0/0
  -p PATH, --path PATH  Modify all derivation paths by replacing the final
                        segment(s) w/ the supplied range(s), eg. '.../1/-'
                        means .../1/[0,...)
  -j JSON, --json JSON  Save an encrypted JSON wallet for each Ethereum
                        address w/ this password, '-' reads it from stdin
                        (default: None)
  -w WALLET, --wallet WALLET
                        Produce paper wallets in output PDF; each wallet
```

```
                        private key is encrypted this password (use
                        --wallet="" for empty password)
  --wallet-hint WALLET_HINT
                        Paper wallets password hint
  --wallet-format WALLET_FORMAT
                        Paper wallet size; half, third, quarter or
                        '(<h>,<w>),<margin>' (default: quarter)
  -s SECRET, --secret SECRET
                        Use the supplied BIP-39 Mnemonic or 128-, 256- or
                        512-bit hex value as the secret seed; '-' reads it
                        from stdin (eg. output from slip39.recover)
  --bits BITS           Ensure that the seed is of the specified bit length;
                        128, 256, 512 supported.
  --using-bip39         Generate Seed from secret Entropy using BIP-39
                        generation algorithm (encode as BIP-39 Mnemonics,
                        encrypted using --passphrase)
  --passphrase PASSPHRASE
                        Encrypt the master secret w/ this passphrase, '-'
                        reads it from stdin (default: None/'')
  -C CARD, --card CARD  Card size; business, credit, index, half, third,
                        quarter, photo or '(<h>,<w>),<margin>' (default:
                        business)
  --no-card             Disable PDF SLIP-39 mnemonic card output
  --paper PAPER         Paper size (default: Letter)
  --cover               Produce PDF SLIP-39 cover page
  --no-cover            Disable PDF SLIP-39 cover page
  --text                Enable textual SLIP-39 mnemonic output to stdout
  --watermark WATERMARK
                        Include a watermark on the output SLIP-39 mnemonic
                        cards
  --double-sided        Enable double-sided PDF (default)
  --no-double-sided     Disable double-sided PDF
  --single-sided        Enable single-sided PDF
```

### 3.4 Recovery & Re-Creation

Later, if you need to recover the wallet seed, keep entering SLIP-39 mnemonics into `slip39-recovery` 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, re-create the wallet seed, perhaps including an encrypted JSON Paper Wallet for import of some accounts into a software wallet (use `--json password` to output encrypted Ethereum JSON wallet files):

```
slip39 --secret 383597fd63547e7c9525575decd413f7 --wallet password --wallet-hint bad:pass... 2>&1

2024-05-15 10:04:35 slip39           It is recommended to not use '-s|--secret <hex>'; specify '-' to read from input
2024-05-15 10:04:35 slip39           It is recommended to not use '-w|--wallet <password>'; specify '-' to read from input
2024-05-15 10:04:35 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0xb44A2011A99596671d5952CdC22816089f142FB3
2024-05-15 10:04:35 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1qcupw7k8enymvvsa7w35j5hq4ergtvus3zk8a8s
2024-05-15 10:04:39 slip39.layout    Writing SLIP39-encoded wallet for 'SLIP39' to: SLIP39-2024-05-15+10.04.37-ETH-0xb44A2
SLIP39-2024-05-15+10.04.37-ETH-0xb44A2011A99596671d5952CdC22816089f142FB3.pdf
```

### 3.4.1 `slip39.recovery` Synopsis

```
python3 -m slip39.recovery --help 2>&1              | sed 's/^/: /' # (just for output formatting)
```

```
usage: __main__.py [-h] [-v] [-q] [-m MNEMONIC] [-e] [--no-entropy] [-b] [-u]
                   [--binary] [--language LANGUAGE] [-p PASSPHRASE]

Recover and output secret Seed from SLIP-39 or BIP-39 Mnemonics

options:
  -h, --help            show this help message and exit
  -v, --verbose         Display logging information.
  -q, --quiet           Reduce logging output.
  -m MNEMONIC, --mnemonic MNEMONIC
                        Supply another SLIP-39 (or a BIP-39) mnemonic phrase
  -e, --entropy         Return the BIP-39 Mnemonic Seed Entropy instead of the
                        generated Seed (default: True if --using-bip39 w/o
                        passphrase)
  --no-entropy          Return the BIP-39 Mnemonic generated Seed
  -b, --bip39           Recover Entropy and generate 512-bit secret Seed from
                        BIP-39 Mnemonic + passphrase
  -u, --using-bip39     Recover Entropy from SLIP-39, generate 512-bit secret
                        Seed using BIP-39 Mnemonic + passphrase
  --binary              Output seed in binary instead of hex
  --language LANGUAGE   BIP-39 Mnemonic language (default: english)
  -p PASSPHRASE, --passphrase PASSPHRASE
                        Decrypt the SLIP-39 or BIP-39 master secret w/ this
                        passphrase, '-' reads it from stdin (default: None/'')

If you obtain a threshold number of SLIP-39 mnemonics, you can recover the original
secret Seed Entropy, and then re-generate one or more wallets from it.

Enter the mnemonics when prompted and/or via the command line with -m |--mnemonic "...".

The secret Seed Entropy can then be used to generate a new SLIP-39 encoded wallet:

    python3 -m slip39 --secret = "ab04...7f"

SLIP-39 Mnemonics may be encrypted with a passphrase; this is *not* Ledger-compatible, so it rarely
recommended!  Typically, on a Trezor "Model T", you recover using your SLIP-39 Mnemonics, and then
use the "Hidden wallet" feature (passwords entered on the device) to produce alternative sets of
accounts.

BIP-39 Mnemonics can be backed up as SLIP-39 Mnemonics, in two ways:

1) The actual BIP-39 standard 512-bit Seed can be generated by supplying --passphrase, but only at
the cost of 59-word SLIP-39 mnemonics.  This is because the *output* 512-bit BIP-39 Seed must be
stored in SLIP-39 -- not the *input* 128-, 160-, 192-, 224-, or 256-bit entropy used to create the
original BIP-39 mnemonic phrase.

2) The original BIP-39 12- or 24-word, 128- to 256-bit Seed Entropy can be recovered by supplying
--entropy.  This modifies the BIP-39 recovery to return the original BIP-39 Mnemonic Entropy, before
decryption and seed generation.  It has no effect for SLIP-39 recovery.
```

### 3.4.2 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 \
    | ( sort -r  ; echo "...later, after recovering SLIP-39 mnemonics..." 1>&2 ) \
    | python3 -m slip39.recovery \
    | python3 -m slip39 --secret - --no-card \
) 2>&1
```

```
2024-05-15 10:04:40 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0x7A4655ba56e605e4e22e378a4A092356Faf9836B
2024-05-15 10:04:40 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1qydzzgflwavtnmfftrtjhy30xz99luwd7ad9rgv
...later, after recovering SLIP-39 mnemonics...
2024-05-15 10:04:40 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0x7A4655ba56e605e4e22e378a4A092356Faf9836B
2024-05-15 10:04:40 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1qydzzgflwavtnmfftrtjhy30xz99luwd7ad9rgv
SLIP39-2024-05-15+10.04.40-ETH-0x7A4655ba56e605e4e22e378a4A092356Faf9836B.pdf
```

### 3.4.3 Pipelining Backup of a BIP-39 Mnemonic Phrase

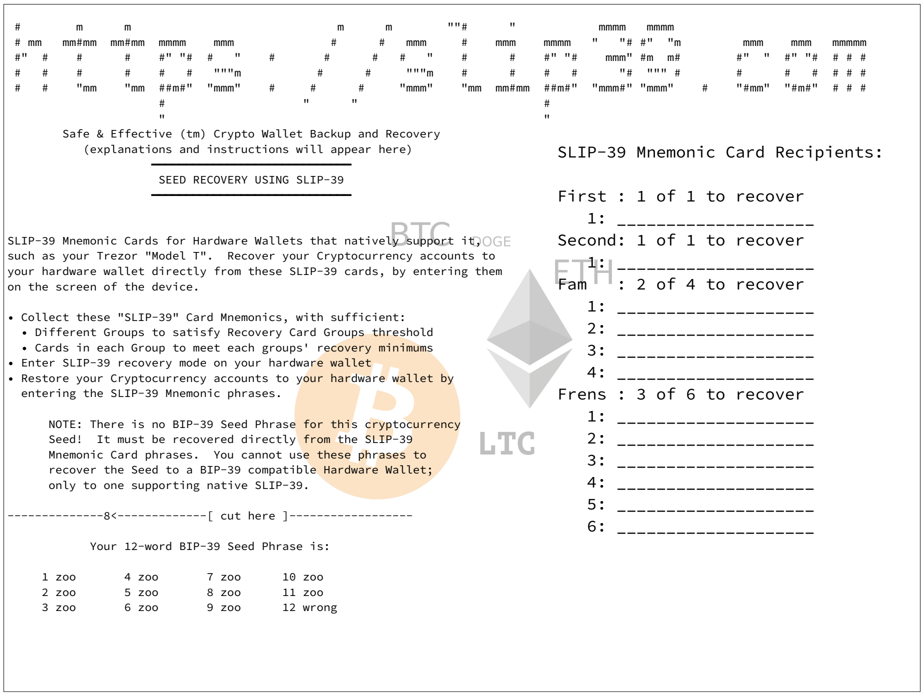
A primary use case for `python-slip39` will be to backup an existing BIP-39 Mnemonic Phrase to SLIP-39 cards, so here it is. Suppose you have some (arbitrary) way to recover (or generate) some Entropy; for example, by recovering the original seed entropy used to generate a BIP-39 Mhemonic:

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

```
2024-05-15 10:04:41 slip39           Assuming BIP-39 seed entropy: Ensure you recover and use via a BIP-39 Mnemonic
2024-05-15 10:04:41 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E
2024-05-15 10:04:41 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1qk0a9hr7wjfxeenz9nwenw9flhq0tmsf6vsgnn2
2024-05-15 10:04:44 slip39.layout    Writing SLIP39 backup for BIP-39-encoded wallet for 'SLIP39' to: SLIP39-2024-05-15+10
SLIP39-2024-05-15+10.04.44-ETH-0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E.pdf
```

Better yet, if you already have a BIP-39 Mnemonic, you can just use that directly (we'll use a bit of "wrapping" around the filename output, so the first page shows up here):

```
echo -n "[[./$( \
python3 -m slip39 -q --secret "zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong" \
)]]"
```




Note the presence of the BIP-39 recovery phrase on the cover sheet; this is recovered by round-tripping the original BIP-39 seed entropy, through SLIP-39, and re-encoding back to BIP-39.

## 3.5 Generation of Addresses

For systems that require a stream of groups of wallet Addresses (eg. for preparing invoices for clients, with a choice of cryptocurrency payment options), `slip-generator` can produce a stream of groups of addresses.

### 3.5.1 `slip39-generator` Synopsis

```
slip39-generator --help --version        | sed 's/^/: /' # (just for output formatting)

usage: slip39-generator [-h] [-v] [-q] [-s SECRET] [-f FORMAT] [--xpub]
                        [--no-xpub] [-c CRYPTOCURRENCY] [--path PATH]
                        [-d DEVICE] [--baudrate BAUDRATE] [-e ENCRYPT]
                        [--decrypt ENCRYPT] [--enumerated] [--no-enumerate]
                        [--receive] [--corrupt CORRUPT]

Generate public wallet address(es) from a secret seed

options:
  -h, --help            show this help message and exit
  -v, --verbose         Display logging information.
  -q, --quiet           Reduce logging output.
  -s SECRET, --secret SECRET
                        Use the supplied 128-, 256- or 512-bit hex value as
                        the secret seed; '-' (default) reads it from stdin
                        (eg. output from slip39.recover)
  -f FORMAT, --format FORMAT
                        Specify crypto address formats: legacy, segwit,
                        bech32; default: ETH:legacy, BTC:bech32, LTC:bech32,
```

```
                        DOGE:legacy, BSC:legacy, XRP:legacy
  --xpub                Output xpub... instead of cryptocurrency wallet
                        address (and trim non-hardened default path segments)
  --no-xpub             Inhibit output of xpub (compatible w/ pre-v10.0.0)
  -c CRYPTOCURRENCY, --cryptocurrency CRYPTOCURRENCY
                        A crypto name and optional derivation path (default:
                        "ETH:{Account.path_default('ETH')}"), optionally w/
                        ranges, eg: ETH:../0/-
  --path PATH           Modify all derivation paths by replacing the final
                        segment(s) w/ the supplied range(s), eg. '.../1/-'
                        means .../1/[0,...)
  -d DEVICE, --device DEVICE
                        Use this serial device to transmit (or --receive)
                        records
  --baudrate BAUDRATE   Set the baud rate of the serial device (default:
                        115200)
  -e ENCRYPT, --encrypt ENCRYPT
                        Secure the channel from errors and/or prying eyes with
                        ChaCha20Poly1305 encryption w/ this password; '-'
                        reads from stdin
  --decrypt ENCRYPT
  --enumerated          Include an enumeration in each record output (required
                        for --encrypt)
  --no-enumerate        Disable enumeration of output records
  --receive             Receive a stream of slip.generator output
  --corrupt CORRUPT     Corrupt a percentage of output symbols

Once you have a secret seed (eg. from slip39.recovery), you can generate a sequence
of HD wallet addresses from it.  Emits rows in the form:

    <enumeration> [<address group(s)>]

If the output is to be transmitted by an insecure channel (eg. a serial port), which may insert
errors or allow leakage, it is recommended that the records be encrypted with a cryptographic
function that includes a message authentication code.  We use ChaCha20Poly1305 with a password and a
random nonce generated at program start time.  This nonce is incremented for each record output.

Since the receiver requires the nonce to decrypt, and we do not want to separately transmit the
nonce and supply it to the receiver, the first record emitted when --encrypt is specified is the
random nonce, encrypted with the password, itself with a known nonce of all 0 bytes.  The plaintext
data is random, while the nonce is not, but since this construction is only used once, it should be
satisfactory.  This first nonce record is transmitted with an enumeration prefix of "nonce".
```

### 3.5.2 Producing Addresses

Addresses can be produced in plaintext or encrypted, and output to stdout or to a serial port.

```
echo ffffffffffffffffffffffffffffffff | slip39-generator --secret - --path '../-3' 2>&1
```

```
0: [["ETH", "m/44'/60'/0'/0/0", "0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1"], ["BTC", "m/84'/0'/0'/0/0", "bc1q9yscq3l2yfx
1: [["ETH", "m/44'/60'/0'/0/1", "0x8D342083549C635C0494d3c77567860ee7456963"], ["BTC", "m/84'/0'/0'/0/1", "bc1qnec684yvuhf
2: [["ETH", "m/44'/60'/0'/0/2", "0x52787E24965E1aBd691df77827A3CfA90f0166AA"], ["BTC", "m/84'/0'/0'/0/2", "bc1q2snj0zcg23d
3: [["ETH", "m/44'/60'/0'/0/3", "0xc2442382Ae70c77d6B6840EC6637dB2422E1D44e"], ["BTC", "m/84'/0'/0'/0/3", "bc1qxwekjd46aa5
```

To produce accounts from a BIP-39 or SLIP-39 seed, recover it using slip39-recovery.

Here's an example of recovering a test BIP-39 seed; note that it yields the well-known ETH `0xfc20...1B5E` and BTC `bc1qk0...gnn2` accounts associated with this test Mnemonic:

```
( python3 -m slip39.recovery --bip39 --mnemonic 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
    | python3 -m slip39.generator --secret - --path '../-3' --format 'BTC:segwit' --crypto 'DOGE' ) 2>&1
```

```
0: [["DOGE", "m/44'/3'/0'/0/0", "DTMaJd8wqye1fymnjxZ5Cc5QkN1w4pMgXT"], ["BTC", "m/49'/0'/0'/0/0", "3CfyLSjYFFV6MUAMh3auTK9
1: [["DOGE", "m/44'/3'/0'/0/1", "DGkL2LD5FfccAaKtx8G7TST5iZwrNkecTY"], ["BTC", "m/49'/0'/0'/0/1", "31nD3MEioUDchu7bVaHUCdC
2: [["DOGE", "m/44'/3'/0'/0/2", "DQa3SpFZH3fFpEFAJHTXZjam4hWiv9muJX"], ["BTC", "m/49'/0'/0'/0/2", "32pqj8rgW1BdXK2Cygwn2JV
3: [["DOGE", "m/44'/3'/0'/0/3", "DTW5tqLwspMY3NpW3RrgMfjWs5gnpXtfwe"], ["BTC", "m/49'/0'/0'/0/3", "3CimS2PfrNykKtJe1uxM4Qt
```

We can encrypt the output, to secure the sequence (and due to integrated MACs, ensures no errors occur over an insecure channel like a serial cable):

```
( slip39-recovery --bip39 --mnemonic 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
    | slip39-generator --secret - --path '../-3' --encrypt 'password' ) 2>&1 \
        | sed -E 's/^(.{100})(.{1,})$/\1.../'  # (shorten output)
```

```
nonce: adb1bfde768d15093523c3758d04c2a8f99c3e5e1d95de60519fbef4
    0: 90b6ee95dd1d1e9def21604ed0ca84c50ca9c838e22a53f12789f8ff918c2d654d755db08b32da6502ff58c3b4801...
    1: fc077ceed116248b919fc79a043c1302ffaec1103b9b8a3a3f3d92ffa5b81a2b9d61a3c87f39e1c075775df7c4e7f...
    2: e0f9ba34c4c826352bdcf845d6844cbe61ef25efb8c34f3f9606b2dc071305083bf9764e105b8cc1955c15800b5bf...
    3: 39d27e07d747da1db70796c7b309fb4b6add6ff7c6b63d82910a422cd58b3a5cd62b3e4a32aef3d23fa3df1ad5e95...
```

On the receiving computer, we can decrypt and recover the stream of accounts from the wallet seed; any rows with errors are ignored:

```
( slip39-recovery --bip39 --mnemonic 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
    | slip39-generator --secret - --path '../-3' --encrypt 'password' \
    | slip39-generator --receive --decrypt 'password' ) 2>&1
```

```
0: [["ETH", "m/44'/60'/0'/0/0", "0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E"], ["BTC", "m/84'/0'/0'/0/0", "bc1qk0a9hr7wjfx
1: [["ETH", "m/44'/60'/0'/0/1", "0xd1a7451beB6FE0326b4B78e3909310880B781d66"], ["BTC", "m/84'/0'/0'/0/1", "bc1qkd33yck74lg
2: [["ETH", "m/44'/60'/0'/0/2", "0x578270B5E5B53336baC354756b763b309eCA90Ef"], ["BTC", "m/84'/0'/0'/0/2", "bc1qvr7e5aytd0h
3: [["ETH", "m/44'/60'/0'/0/3", "0x909f59835A5a120EafE1c60742485b7ff0e305da"], ["BTC", "m/84'/0'/0'/0/3", "bc1q6t9vhestkcf
```

### 3.5.3 X Public Keys

If you prefer, you can output "xpub..." format public keys, instead of account addresses. By default, this will elide the non-hardened portion of the default addresses – use the "xpub..." keys to produce the remaining non-hardened portion of the HD wallet paths locally.

For example, assume you must produce a sequence of accounts for each client client of your company to deposit into. Your highly secure serial-connected "key enclave" system (which must know your HD wallet seed) emits a sequence of xpubkeys for each new client over a serial cable, to your accounting system:

```
( echo 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
    | python3 -m slip39.generator --secret - --xpub --path "../-2'"  --encrypt 'password' \
    | python3 -m slip39.generator -v --receive --decrypt 'password' ) 2>&1
```

```
2024-05-15 10:04:53 slip39.generator Decrypting accountgroups with nonce: 56ce6e3c37dc1269221752de
    0: [["ETH", "m/44'/60'/0'", "xpub6C2y6te3rtGg9SspDDFbjGEgn7yxc5ZzzkBk62yz3GRKvuqdaMDS7NUbesTJ44FprxAE7hvm5ZQjDMbYWehdJ
    1: [["ETH", "m/44'/60'/1'", "xpub6C2y6te3rtGgCPb4Gi89Qin7Da2dvnnHSuR9rLQV6bWQKiyfKyjtVzr2n9mKmTEHzr4rzK78LmdSXLSzvpZqV
    2: [["ETH", "m/44'/60'/2'", "xpub6C2y6te3rtGgENnaK62SyPawqKvbde17wc2ndMGFWi2yAkk3piwEY9QK8egtE9ye9uoqiqs5WV3MTNCCP2qjU
```

As required (throttled by hardward the serial cable RTS/CTS signals) your accounting system receives these "xpub..." addresses:

```
( echo 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
    | python3 -m slip39.generator --secret - --xpub --path "../-2'"  --encrypt 'password' \
    | python3 -m slip39.generator -v --receive --decrypt 'password' \
    | while IFS=':' read num json; do \
        echo "--- $(( num ))"; \
        echo "$json" | jq -c '.[]'; \
    done \
) 2>&1
```

```
2024-05-15 10:04:54 slip39.generator Decrypting accountgroups with nonce: 9714b178caed99cf2ea2b0ec
--- 0
["ETH","m/44'/60'/0'","xpub6C2y6te3rtGg9SspDDFbjGEgn7yxc5ZzzkBk62yz3GRKvuqdaMDS7NUbesTJ44FprxAE7hvm5ZQjDMbYWehdJQsyBCP
["BTC","m/84'/0'/0'","zpub6rD5AGSXPTDMSnpmczjENMT3NvVF7q5MySww6uxitUsBYgkZLeBywrcwUWhW5YkeY2aS7xc45APPgfA6s6wWfG2gnfAB
--- 1
["ETH","m/44'/60'/1'","xpub6C2y6te3rtGgCPb4Gi89Qin7Da2dvnnHSuR9rLQV6bWQKiyfKyjtVzr2n9mKmTEHzr4rzK78LmdSXLSzvpZqVs4ussU
["BTC","m/84'/0'/1'","zpub6rD5AGSXPTDMUaSe3aGDqWk4uMTwcrFwytkKuDGmi3ofUkJ4dQxXHZwiXWbHHrELJAor8xGs61F8sbKS2JdQkLZRnu5P
--- 2
["ETH","m/44'/60'/2'","xpub6C2y6te3rtGgENnaK62SyPawqKvbde17wc2ndMGFWi2yAkk3piwEY9QK8egtE9ye9uoqiqs5WV3MTNCCP2qjUNDb8cm
["BTC","m/84'/0'/2'","zpub6rD5AGSXPTDMYx2sQPuZgceniniRXDK5tELiREjxfSGJENNxuQD3u2yfpRqnNE1JeH14Pa7MVGrofDJtyXw252ws9HgR
```

Then, it generates each client's sequence of addresses locally: you are creating HD wallet accounts from each "xpub. . . " key, and adding the remaining non-hardened HD wallet path segments:

```
( echo 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
    | python3 -m slip39.generator --secret - --xpub --path "../-2'"  --encrypt 'password' \
    | python3 -m slip39.generator -v --receive --decrypt 'password' \
    | while IFS=':' read num json; do \
        echo "--- $(( num ))"; \
        echo "$json" | jq -cr '.[]|"--crypto " + .[0] + " --secret " + .[2]' | while read command; do \
            python3 -m slip39.cli -v --no-json addresses $command --paths m/0/-2; \
        done; \
    done \
) 2>&1
```

```
2024-05-15 10:04:55 slip39.generator Decrypting accountgroups with nonce: 5eb856a16408264b86e49da9
--- 0
ETH   m/0/0               0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E
ETH   m/0/1               0xd1a7451beB6FE0326b4B78e3909310880B781d66
ETH   m/0/2               0x578270B5E5B53336baC354756b763b309eCA90Ef
BTC   m/0/0               bc1qk0a9hr7wjfxeenz9nwenw9flhq0tmsf6vsgnn2
BTC   m/0/1               bc1qkd33yck74lg0kaq4tdcmu3hk4yruhjayxpe9ug
BTC   m/0/2               bc1qvr7e5aytd0hpmtaz2d443k364hprvqpm3lxr8w
--- 1
ETH   m/0/0               0x9176A747BA67C1d7F80AaDC930180b4183AfB5c4
ETH   m/0/1               0xa1409B655aC3e09eF261de00BAa4e85bD2820AA4
ETH   m/0/2               0xae22C13Ef5891Ed835C24Ed5090542DFa748c21F
BTC   m/0/0               bc1q8pqnqs573vx3qdp0xp6qdqzvnvy8px24rxh9lp
BTC   m/0/1               bc1qwtc58u4mmnxa29u8j07e6lmqpnrs38vefy3y24
BTC   m/0/2               bc1qg9s8qzm0lcetfv6umhlm3evtca5zsqv7elqd5s
--- 2
ETH   m/0/0               0x32A8b066c5dbD37147766491A32A612d313fda25
ETH   m/0/1               0xff8b88b975f9C296531C1E93d5e4f28757b4571A
ETH   m/0/2               0xc95Bdf50CA542E1B689f5C06e2D8bAd0625Dfa23
BTC   m/0/0               bc1q09zpchmkcnny90ghkg76gd69dvaf57qwcsrhes
BTC   m/0/1               bc1qjytdyw6zramwt4nvvpte93hfry2d4xhhqn0xg4
BTC   m/0/2               bc1qcummre0pxv5xj4gvyut0t84vfwjd6eu7r387v4
```

You'll notice that, after this elaborate exercise of generating xpubkeys, encrypted transmission and recovery, generating accounts from the xpubkeys, and producing multiples addresses using the remainder of the original HD wallet paths: the output addresses are identical to those generated directly from the BIP-39 Mnemonic Phrase:

```
secret='zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong'
for crypto in BTC ETH; do
    python3 -m slip39.cli -v --no-json addresses --secret "$secret" --crypto $crypto --paths "../-2"
done
```

```
BTC   m/84'/0'/0'/0/0      bc1qk0a9hr7wjfxeenz9nwenw9flhq0tmsf6vsgnn2
BTC   m/84'/0'/0'/0/1      bc1qkd33yck74lg0kaq4tdcmu3hk4yruhjayxpe9ug
BTC   m/84'/0'/0'/0/2      bc1qvr7e5aytd0hpmtaz2d443k364hprvqpm3lxr8w
ETH   m/44'/60'/0'/0/0     0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E
ETH   m/44'/60'/0'/0/1     0xd1a7451beB6FE0326b4B78e3909310880B781d66
ETH   m/44'/60'/0'/0/2     0x578270B5E5B53336baC354756b763b309eCA90Ef
```

### 3.5.4 Serial Port Connected Secure Seed Enclave

What if you or your company wants to accept Crypto payments, and needs to generate a sequence of wallets unique to each client? You **can** use an xpubkey and then generate a sequence of unique addresses from that, which doesn't disclose any of your private key material:

```
( python3 -m slip39.generator -q --secret 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
    --xpub --path "../-2'" --crypto BTC
) 2>&1
```

```
0: [["BTC", "m/84'/0'/0'", "zpub6rD5AGSXPTDMSnpmczjENMT3NvVF7q5MySww6uxitUsBYgkZLeBywrcwUWhW5YkeY2aS7xc45APPgfA6s6wWfG2gnf
1: [["BTC", "m/84'/0'/1'", "zpub6rD5AGSXPTDMUaSe3aGDqWk4uMTwcrFwytkKuDGmi3ofUkJ4dQxXHZwiXWbHHrELJAor8xGs61F8sbKS2JdQkLZRnu
2: [["BTC", "m/84'/0'/2'", "zpub6rD5AGSXPTDMYx2sQPuZgceniniRXDK5tELiREjxfSGJENNxuQD3u2yfpRqnNE1JeH14Pa7MVGrofDJtyXw252ws9H
```

Since you have to generate such an xpubkey from a "hardened" path, such as with `slip39.generate --xpub ...`, you **still** need to run that tool chain on some secure "air gapped" computer. So, how do you do that safely, knowing that you need to input your SLIP-39 or BIP-39 Mnemonics on that computer? Especially, if you want to do this under any kind of automation, and deliver the output xpubkey to your insecure business computer systems?

One solution is to have the computer hosting your Seed or Mnemonic private key material **only** connected to your business computer systems with a guaranteed **safe** mechanism. Definitely **not** with any kind of general purpose network system!

The solution: **The RS-232 Serial Port**

With USB to DB-9 female to DB-9 male serial adapters, any small computer with USB ports (such as the Raspberry Pi 400) can be connected serially and serve as your "secure" computer, storing your Seed Mnemonic.

Remember to disable all other wired and wireless networking!

The RS-232 port on the "secure" computer can be protected from all incoming data transmissions, make an exploit effectively impossible, while still allowing outgoing data (the generated xpubkeys).

A DB-9 serial breakout board or custom serial adapter be easily constructed that disconnects pin 3 (TXD) on the "business" side from pin 2 (RXD) on the "secure" side, eliminating any chance of data being sent to the "secure" side. The only electronic connection that transmits data to the "secure" side is the hardware flow control pin 7 (RTS) to pin 8 (CTS). An exploit using this single-bit approach vector is ... unlikely. :)

## 3.6 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.

### 3.6.1 `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: "") |
| using_bip39 | Produce wallet Seed from master_secret Entropy using BIP-39 generation |
| iteration_exponent | For encrypted secret, exponentially increase PBKDF2 rounds (default: 1) |
| cryptopaths | A number of crypto names, and their derivation paths ] |
| strength | Desired master_secret strength, in bits (default: 128) |

Outputs a `slip39.Details` namedtuple containing:

| Key | Description |
|---|---|
| name | (same) |
| group_threshold | (same) |
| groups | Like groups, w/ <members> = ["<mnemonics>", ...] |
| accounts | Resultant list of groups of accounts |
| using_bip39 | Seed produced from entropy using BIP-39 generation |

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

```
import codecs
import random
from tabulate import tabulate
```

```
#
# 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 slip39

cryptopaths         = [("ETH","../-2"), ("BTC","../-2")]
master_secret       = b'\xFF' * 16
master_secret       = 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong'
passphrase          = b""
create_details      = slip39.create(
    "Test", 2, { "Mine": (1,1), "Fam": (2,3) },
    master_secret=master_secret, passphrase=passphrase, cryptopaths=cryptopaths )

[
    [
        "Card", "Mnemonics 1 ", "Mnemonics 2", "Mnemonics 3"
    ],
    None,
] + [
    [
        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}:" ))
]
```

Add the resultant HD Wallet addresses:

```
[
    [ account.path, account.address ]
    for group in create_details.accounts
    for account in group
]
```

### 3.6.2 `slip39.produce_pdf`

| Key | Description |
|---|---|
| name | (same as `slip39.create`) |
| group_threshold | (same as `slip39.create`) |
| groups | Like groups, w/ <members> = ["<mnemonics>", ...] |
| accounts | Resultant { "path": Account, ...} |
| using_bip39 | Generate Seed from Entropy via BIP-39 generation algorithm |
| card_format | 'index', '(<h>,<w>),<margin>', ... |
| paper_format | 'Letter', ... |
| orientation | Force an orientation (default: portrait, landscape) |
| cover_text | Produce a cover page w/ the text (and BIP-39 Phrase if using_bip39) |

Layout and produce a PDF containing all the SLIP-39 details on cards for the crypto accounts, on the paper_format provided. Returns the paper (orientation,format) used, the FPDF, and passes through the supplied cryptocurrency accounts derived.

```
(paper_format,orientation),pdf,accounts = slip39.produce_pdf( *create_details )
pdf_binary = pdf.output()
```

```
[
    [ "Orientation:",       orientation ],
    [ "Paper:",             paper_format ],
    [ "PDF Pages:",         pdf.pages_count ],
    [ "PDF Size:",          len( pdf_binary )],
]
```

### 3.6.3 slip39.write_pdfs

| Key | Description |
|---|---|
| names | A sequence of Seed names, or a dict of { name: <details> } (from slip39.create) |
| master_secret | A Seed secret (only appropriate if exactly one name supplied) |
| passphrase | A SLIP-39 passphrase (not Trezor compatible; use "hidden wallet" phrase on device instead) |
| using_bip39 | Generate Seed from Entropy via BIP-39 generation algorithm |
| group | A dict of {"<group>":(<required>, <members>), ...} |
| group_threshold | How many groups are required to recover the Seed |
| cryptocurrency | A sequence of [ "<crypto>", "<crypto>:<derivation>", ... ] w/ optional ranges |
| edit | Derivation range(s) for each cryptocurrency, eg. "../0-4/-9" is 9 accounts first 5 change addresses |
| card_format | Card size (eg. "credit"); False specifies no SLIP-39 cards (ie. only BIP-39 or JSON paper wallets) |
| paper_format | Paper size (eg. "letter") |
| filename | A filename; may contain "...{name}..." formatting, for name, date, time, crypto path and address |
| filepath | A file path, if PDF output to file is desired; empty implies current dir. |
| printer | A printer name (or True for default), if output to printer is desired |
| json_pwd | If password supplied, encrypted Ethereum JSON wallet files will be saved, and produced into PDF |
| text | If True, outputs SLIP-39 phrases to stdout |
| wallet_pwd | If password supplied, produces encrypted BIP-38 or JSON Paper Wallets to PDF (preferred vs. json_pwd) |
| wallet_pwd_hint | An optional passphrase hint, printed on paper wallet |
| wallet_format | Paper wallet size, (eg. "third"); the default is 1/3 letter size |
| wallet_paper | Other paper format (default: Letter) |
| cover_page | A bool indicating whether to produce a cover page (default: True) |

For each of the names provided, produces a separate PDF containing all the SLIP-39 details and optionally encrypted BIP-38 paper wallets and Ethereum JSON wallets for the specified cryptocurrency accounts derived from the seed, and writes the PDF and JSON wallets to the specified file name(s).

```
slip39.write_pdfs( ... )
```

### 3.6.4 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 `Account` data.

Note that the SLIP-39 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 (non-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 Seed Entropy |
| using_bip39 | Use BIP-39 Seed generation from recover Entropy |

```
# Recover with the wrong password (on purpose, as a decoy wallet w/ a small amount)
recoverydecoy       = slip39.recover(
    create_details.groups['Mine'][1][:] + create_details.groups['Fam'][1][:2],
    passphrase=b"wrong!"
)
recoverydecoy_hex   = codecs.encode( recoverydecoy, 'hex_codec' ).decode( 'ascii' )

# But, recovering w/ correct passphrase yields our original Seed Entropy
recoveryvalid       = slip39.recover(
    create_details.groups['Mine'][1][:] + create_details.groups['Fam'][1][:2],
    passphrase=passphrase
```

```
)
recoveryvalid_hex   = codecs.encode( recoveryvalid, 'hex_codec' ).decode( 'ascii' )

[
  [ f"{len(recoverydecoy)*8}-bit secret (decoy):", f"{recoverydecoy_hex}" ],
  [ f"{len(recoveryvalid)*8}-bit secret recovered:", f"{recoveryvalid_hex}" ]
]
```

### 3.6.5 `slip39.recover_bip39`

Generate the 512-bit Seed from a BIP-39 Mnemonic + passphrase. Or, return the original 128- to 256-bit Seed Entropy, if `as_entropy` is specified.

| Key | Description |
|---|---|
| mnemonic | "<mnemonic>" |
| passphrase | Optional passphrase to decrypt secret Seed Entropy |
| as_entropy | Return the BIP-39 Seed Entropy, not the generated Seed |

### 3.6.6 `slip39.produce_bip39`

Produce a BIP-39 Mnemonic from the supplied 128- to 256-bit Seed Entropy.

| Key | Description |
|---|---|
| entropy | The `bytes` of Seed Entropy |
| strength | Or, the number of bits of Entropy to produce (Default: 128) |
| language | Default is "english" |

# 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!

Fortunately, **we can** do this! It takes a bit of practice to become comfortable with the process, but once you do – you can confidently discard your original insecure and unreliable BIP-39 Mnemonic backups.

## 4.1 BIP-39 vs. SLIP-39 Incompatibility

Unfortunately, it is **not possible** to cleanly convert a BIP-39 *generated* wallet Seed into a SLIP-39 wallet. Both BIP-39 and SLIP-39 preserve the original 128- to 256-bit Seed Entropy (random) bits, but these bits are used **very differently** – and incompatibly – to generate the resultant wallet Seed.

In native SLIP-39, the original, recovered Seed Entropy (128- or 256-bits) is used directly by the BIP-44 wallet derivation. In BIP-39, the Seed entropy is not directly used *at all*! It is only **indirectly** used; the BIP-39 Seed Phrase (which contains the exact, original entropy) is used, as normalized text, as input to a hashing function, along with some other fixed text, to produce a 512-bit Seed, which is then fed into the BIP-44 wallet derivation process.

The least desirable method is to preserve the 512-bit **output** of the BIP-39 mnemonic phrase as a set of 512-bit (59-word) SLIP-39 Mnemonics. But first, lets review how BIP-39 works.

### 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 Seed Entropy. Here's a 128-bit (12-word) example using some fixed "entropy" `0xFFFF..FFFF`. You'll note that, from the BIP-39 Mnemonic, we can either recover the original 128-bit Seed Entropy, **or** we can generate the resultant 512-bit Seed w/ the correct passphrase:

```
from mnemonic import Mnemonic
bip39_english     = Mnemonic("english")
entropy           = b'\xFF' * 16
entropy_hex       = codecs.encode( entropy, 'hex_codec' ).decode( 'ascii' )
entropy_mnemonic  = bip39_english.to_mnemonic( entropy )

recovered         = slip39.recover_bip39( entropy_mnemonic, as_entropy=True )
recovered_hex     = codecs.encode( recovered, 'hex_codec' ).decode( 'ascii' )
```

```
recovered_seed    = slip39.recover_bip39( entropy_mnemonic, passphrase=passphrase )
recovered_seed_hex= codecs.encode( recovered_seed, 'hex_codec' ).decode( 'ascii' )

[
    [ "Original Entropy", entropy_hex ],
    [ "BIP-39 Mnemonic", entropy_mnemonic ],
    [ "Recovered Entropy", recovered_hex ],
    [ "Recovered Seed", f"{recovered_seed_hex:.50}..." ],
]
```

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.check( m )
    for i,m in enumerate( random_words( 12, 10000 ))) / 100

[
    [ "Valid random 12-word mnemonics:", f"{successes}%" ],
    [ "Or, about: ", f"1 / {100/successes:.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, BIP-39 does **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 512-bit seed:

```
seed                = bip39_english.to_seed( entropy_mnemonic )
seed_hex            = codecs.encode( seed, 'hex_codec' ).decode( 'ascii' )
[
    [ f"{len(seed)*8}-bit seed:", f"{seed_hex:.50}..." ]
]
```

### 4.1.3 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"
bip39_eth_hd        = slip39.account( seed, 'ETH', path )
[
    [ f"{len(bip39_eth_hd.key)*4}-bit derived key path:", f"{path}" ],
    [ "Produces private key: ", f"{bip39_eth_hd.key}" ],
    [ "Yields Ethereum address:", f"{bip39_eth_hd.address}" ],
]
```

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.

### 4.1.4 SLIP-39 Entropy to Mnemonic

Just like BIP-39 carefully preserves the original 128-bit Seed Entropy bytes in a single 12-word mnemonic phrase, SLIP-39 preserves the original 128- or 256-bit Seed Entropy in a *set* of 20- or 33-word Mnemonic phrases.

```
name,thrs,grps,acct,ub39 = slip39.create(
    "Test", 2, { "Mine": (1,1), "Fam": (2,3) }, entropy )
[
    [ 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}:" ))
]
```

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.

```
[
    [ "Crypto", "HD Wallet Path:", "Ethereum Address:" ],
    None,
] + [
    [ account.crypto, account.path, account.address ]
    for group in create_details.accounts
    for account in group
]
```

### 4.1.5 SLIP-39 Mnemonic to Seed

Lets prove that we can actually recover the **original** Seed 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 the Fam group:

```
_,mnem_mine         = grps['Mine']
_,mnem_fam          = grps['Fam']
recseed             = slip39.recover( mnem_mine + mnem_fam[:2] )
recseed_hex         = codecs.encode( recseed, 'hex_codec' ).decode( 'ascii' )
[
    [ f"{len(recseed)*8}-bit Seed:", f"{recseed_hex}" ]
]
```

### 4.1.6 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:

```
slip39_eth_hd       = slip39.account( recseed, 'ETH', path )
[
    [ f"{len(slip39_eth_hd.key)*4}-bit derived key path:", f"{path}" ],
    [ "Produces private key: ", f"{slip39_eth_hd.key}" ],
    [ "Yields Ethereum address:", f"{slip39_eth_hd.address}" ],
]
```

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 same** Ethereum wallet address obtained from BIP-39 with exactly the same `0xFFFF...FF` Seed Entropy, which was `0xfc20..1B5E`!

This is due to the fact that BIP-39 does not use the recovered Seed Entropy to produce the seed like SLIP-39 does, but applies additional one-way hashing of the Mnemonic to produce a 512-bit Seed.

## 4.2 BIP-39 vs SLIP-39 Key Derivation Summary

At no time in BIP-39 account derivation is the original 128-bit Seed Entropy used (directly) in the derivation of the wallet key. This differs from SLIP-39, which directly uses the 128-bit Seed 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 Seed Entropy to account generation where we **could** introduce a known 128-bit seed and produce a known Ethereum wallet from it, other than at the very beginning.

Therefore, our BIP-39 Backup via SLIP-39 strategy must focus on backing up the original 128- to 256-bit Seed *Entropy*, **not** the output Seed data!

## 4.3 BIP-39 Backup via SLIP-39

Here are the two available methods for backing up insecure and unreliable BIP-39 Mnemonic phrases, using SLIP-39.

The first "Emergency Recovery" method allows you to recover your BIP-39 generated wallets **without the passphrase**, but does not support recovery using hardware wallets; you must output "Paper Wallets" and use them to recover the Cryptocurrency funds.

The second "Best Recovery: Using Recovered BIP-39 Mnemonic Phrase" allows us to recover the accounts to *any* standard BIP-39 hardware wallet! However, the SLIP-39 Mnemonics are **not** compatible with standard SLIP-39 wallets like the Trezor "Model T" – you have to use the recovered BIP-39 Mnemonic phrase to recover the hardware wallet.

### 4.3.1 Emergency Recovery: Using Recovered Paper Wallets

There is one approach which can preserve an original BIP-39 *generated* wallet addresses, 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 *or passphrase* – you could generate and distribute a set of more secure SLIP-39 Mnemonic phrases, instead of trying to secure the original BIP-39 mnemonic + passphrase – without abandoning your existing BIP-39 wallets.

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

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

```
2024-05-15 10:05:07 slip39.recovery  BIP-39 Language detected: english
2024-05-15 10:05:07 slip39.recovery  Recovered 512-bit BIP-39 secret from english mnemonic
2024-05-15 10:05:07 slip39.recovery  Recovered BIP-39 secret; To re-generate SLIP-39 wallet, send it to: python3 -m slip39
b6a6d8921942dd9806607ebc2750416b289adea669198769f2e15ed926c3aa92bf88ece232317b4ea463e84b0fcd3b53577812ee449ccc448eb45e6f54
```

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 -v
) 2>&1 | tail -20
```

```
    2024-05-15 10:05:08 slip39               8 blind     20 trip      32 raisin    44 window    56 declare
    2024-05-15 10:05:08 slip39               9 enlarge   21 library   33 vexed     45 bracelet  57 walnut
    2024-05-15 10:05:08 slip39              10 amuse     22 smirk     34 vexed     46 wrap      58 prayer
    2024-05-15 10:05:08 slip39              11 peaceful  23 envy      35 item      47 acrobat   59 course
    2024-05-15 10:05:08 slip39              12 exhaust   24 rainbow   36 pipeline  48 penalty
    2024-05-15 10:05:08 slip39          6th  1 smart     13 already   25 agency    37 scout     49 elite
    2024-05-15 10:05:08 slip39               2 lungs     14 cylinder  26 dance     38 cradle    50 story
    2024-05-15 10:05:08 slip39               3 decision  15 listen    27 seafood   39 debut     51 paid
    2024-05-15 10:05:08 slip39               4 spider    16 viral     28 knit      40 genre     52 editor
    2024-05-15 10:05:08 slip39               5 acid      17 thorn     29 desire    41 large     53 welcome
    2024-05-15 10:05:08 slip39               6 cowboy    18 railroad  30 hormone   42 fridge    54 wildlife
    2024-05-15 10:05:08 slip39               7 primary   19 tracks    31 cylinder  43 email     55 paces
    2024-05-15 10:05:08 slip39               8 work      20 jewelry   32 keyboard  44 tadpole   56 violence
    2024-05-15 10:05:08 slip39               9 husky     21 gravity   33 dance     45 peanut    57 decent
```

```
2024-05-15 10:05:08 slip39              10 company   22 inherit   34 decorate  46 dance     58 jewelry
2024-05-15 10:05:08 slip39              11 prisoner  23 unknown   35 flash     47 science   59 license
2024-05-15 10:05:08 slip39              12 plastic   24 hormone   36 acrobat   48 amazing
2024-05-15 10:05:08 slip39.layout    ETH    m/44'/60'/0'/0/0     : 0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E
2024-05-15 10:05:08 slip39.layout    BTC    m/84'/0'/0'/0/0      : bc1qk0a9hr7wjfxeenz9nwenw9flhq0tmsf6vsgnn2
SLIP39-2024-05-15+10.05.08-ETH-0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E.pdf
```

This `0xfc20..1B5E` address is the same Ethereum address as is recovered on a Trezor using this BIP-39 mnemonic phrase. Thus, we can generate "Paper Wallets" for the desired Cryptocurrency accounts, and recover the funds.

So, this does the job:

- Uses our original BIP-39 Mnemonic
- Does not require remembering the BIP-39 passphrase
- Preserves all of the original wallets

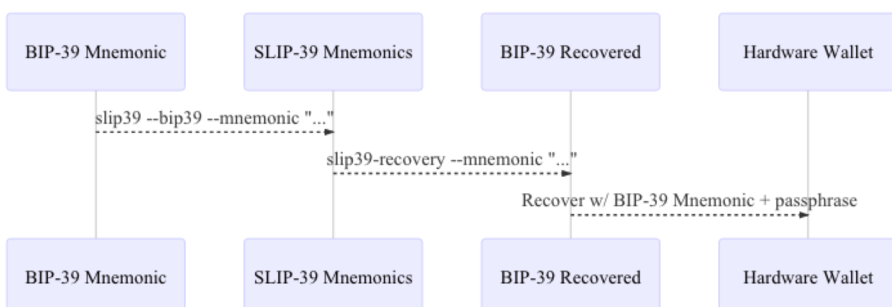
But:

- The 59-word SLIP-39 Mnemonics cannot (yet) be imported into the Trezor "Model T"
- The original BIP-39 Mnemonic phrase cannot be recovered, for any hardware wallet
- Must use the SLIP-39 App to generate "Paper Wallets", to recover the funds

So, this is a good "emergency backup" solution; you or your heirs would be able to recover the funds with a very high level of security and reliability.

### 4.3.2 Best Recovery: Using Recovered BIP-39 Mnemonic Phrase

The best solution is to use SLIP-39 to back up the original BIP-39 Seed *Entropy* (*not* the generated Seed), and then later recover that Seed Entropy and re-generate the BIP-39 Mnemonic phrase. You will continue to need to remember and use your original BIP-39 passphrase:




First, observe that we can recover the 128-bit Seed Entropy from the BIP-39 Mnemonic phrase (not the 512-bit generated Seed): 3

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

```
2024-05-15 10:05:08 slip39.recovery  BIP-39 Language detected: english
2024-05-15 10:05:08 slip39.recovery  Recovered 128-bit BIP-39 entropy from english mnemonic (no passphrase supported)
2024-05-15 10:05:08 slip39.recovery  Recovered BIP-39 secret; To re-generate SLIP-39 wallet, send it to: python3 -m slip39
ffffffffffffffffffffffffffffffff
```

Now we generate SLIP-39 Mnemonics to recover the 128-bit Seed Entropy. Note that these are 20-word Mnemonics. However, these are **NOT** the wallets we expected! These are the well-known native SLIP-39 wallets from the `0xFFFF...FF` Seed Entropy; not the well-known native BIP-39 wallets from that Seed Entropy, which generate the Ethereum wallet address `0xfc20..1B5E`! Why not?

3

```
( python3 -m slip39.recovery --bip39 --entropy \
    --mnemonic 'zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong' \
  | python3 -m slip39 --secret - --no-card -v
) 2>&1 | tail -20
```

```
    2024-05-15 10:05:09 slip39                5 dream     12 leaf      19 expand
    2024-05-15 10:05:09 slip39                6 away      13 unfair    20 royal
    2024-05-15 10:05:09 slip39                7 trip      14 fridge
    2024-05-15 10:05:09 slip39           5th  1 mortgage   8 fiction   15 discuss
    2024-05-15 10:05:09 slip39                2 shadow     9 twice     16 usual
    2024-05-15 10:05:09 slip39                3 decision  10 climate   17 paper
    2024-05-15 10:05:09 slip39                4 snake     11 analysis  18 reward
    2024-05-15 10:05:09 slip39                5 depict    12 flexible  19 trip
    2024-05-15 10:05:09 slip39                6 galaxy    13 sharp     20 improve
    2024-05-15 10:05:09 slip39                7 welfare   14 superior
    2024-05-15 10:05:09 slip39           6th  1 mortgage   8 spirit    15 order
    2024-05-15 10:05:09 slip39                2 shadow     9 timber    16 birthday
    2024-05-15 10:05:09 slip39                3 decision  10 beaver    17 union
    2024-05-15 10:05:09 slip39                4 spider    11 float     18 skin
    2024-05-15 10:05:09 slip39                5 bulge     12 spill     19 clinic
    2024-05-15 10:05:09 slip39                6 fatigue   13 punish    20 findings
    2024-05-15 10:05:09 slip39                7 analysis  14 together
    2024-05-15 10:05:09 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1
    2024-05-15 10:05:09 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1q9yscq3l2yfxlvnlk3cszpqefparrv7tk24u6pl
    SLIP39-2024-05-15+10.05.09-ETH-0x824b174803e688dE39aF5B3D7Cd39bE6515A19a1.pdf
```

Because we must tell `slip39` to that we're using the BIP-39 Mnemonic and Seed generation process to derived the wallet addresses from the Seed Entropy (not the SLIP-39 standard). So, we add the `-using-bip39` option:

3

```
( python3 -m slip39.recovery --bip39 --entropy \
    --mnemonic "zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo zoo wrong" \
  | python3 -m slip39 --secret - --no-card -v --using-bip39
) 2>&1 | tail -20
```

```
    2024-05-15 10:05:10 slip39                5 dining    12 disease   19 mailman
    2024-05-15 10:05:10 slip39                6 item      13 firm      20 unfair
    2024-05-15 10:05:10 slip39                7 sweater   14 theory
    2024-05-15 10:05:10 slip39           5th  1 island     8 evoke     15 ordinary
    2024-05-15 10:05:10 slip39                2 cause      9 exchange  16 mild
    2024-05-15 10:05:10 slip39                3 decision  10 favorite  17 erode
    2024-05-15 10:05:10 slip39                4 snake     11 alpha     18 twin
    2024-05-15 10:05:10 slip39                5 aquatic   12 work      19 sidewalk
    2024-05-15 10:05:10 slip39                6 income    13 tricycle  20 cricket
    2024-05-15 10:05:10 slip39                7 buyer     14 similar
    2024-05-15 10:05:10 slip39           6th  1 island     8 research  15 shame
    2024-05-15 10:05:10 slip39                2 cause      9 equation  16 shaped
    2024-05-15 10:05:10 slip39                3 decision  10 watch     17 slow
    2024-05-15 10:05:10 slip39                4 spider    11 taxi      18 exhaust
    2024-05-15 10:05:10 slip39                5 ancient   12 jump      19 rumor
    2024-05-15 10:05:10 slip39                6 warn      13 dryer     20 husband
    2024-05-15 10:05:10 slip39                7 kernel    14 bundle
    2024-05-15 10:05:10 slip39.layout    ETH    m/44'/60'/0'/0/0    : 0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E
    2024-05-15 10:05:10 slip39.layout    BTC    m/84'/0'/0'/0/0     : bc1qk0a9hr7wjfxeenz9nwenw9flhq0tmsf6vsgnn2
    SLIP39-2024-05-15+10.05.10-ETH-0xfc2077CA7F403cBECA41B1B0F62D91B5EA631B5E.pdf
```

And, there we have it – we've recovered exactly the same Ethereum and Bitcoin wallets as would a native BIP-39 hardware wallet like a Ledger Nano.

1. Using SLIP-39 App "Backup" Controls

   In the SLIP-39 App, the default Controls presented are to "Backup" a BIP-39 recovery phrase.

   In "Seed Source", enter your existing BIP-39 recovery phrase. In "Seed Secret", make sure "Using BIP-39" is selected, and enter your BIP-39 passphrase. This allows us to display the proper wallet addresses – we do **not** store your password, or save it as part of the SLIP-39 cards! You will need to remember and use your passphrase whenever you use your BIP-39 phrase to initialize a hardware wallet.

   Check that the Recovery needs . . . Mnemonic Card Groups are correct for your application, and hit Save!

   Later, use the "Recover" Controls to get your BIP-39 recovery phrase back, from your SLIP-39 cards, whenever you need it.

   Practice this a few times (using the "zoo zoo . . . wrong" 12-word or "zoo zoo . . . vote" 24-word phrase) until you're confident. Then, back up your real BIP-39 recovery phrase.

   Once you're convinced you can securely and reliably recover your BIP-39 phrase any time you need it, we recommend that you destroy your original BIP-39 recovery phrase backup(s). They are dangerous and unreliable, and only serve to make your Cryptocurrency accounts **less** secure!

# 5 Cryptocurrency Invoicing and Licensing

A valuable use of Cryptocurrency accounts is to send or receive payments for goods, services or remittances/donations. The global monetary system makes this very difficult (or even impossible), especially if any of the corporations or governments involved in the transaction are hostile to you or any of the other individuals attempting to transact business.

Worse yet, your business or family can be arbitrarily ejected from the financial system by an of the many intermediary banking industry and government parties involved in any traditional financial transaction, even if you are not convicted of a crime.

Even if such payments are allowed, and none of the counterparties are actively hostile to you, the complexity and expense of quoting a price, signing a client, invoicing for payment, confirming the validity of the invoice and making the payment, monitoring for incoming payments and associating them with the correct invoices, conforming amounts paid are correct, issuing a receipt, book-keeping the incoming payment, converting currencies, retaining the correct taxes for each counterparty jurisdiction, reconciling books, and finally preparing and filing taxes, and then (perhaps years later) defending your accounting decisions against a hostile tax inspector with infinite funds to prosecute – all this makes it virtually impossible for a small business to survive. Furthermore, you must accomplish all this, **without error**, while attempting to defend yourself against business adversaries with preferential tax treatment, office-towers full of lawyers and accountants, for whom the total percentage of gross revenues paid to accomplish compliance is less than 1%, while the small business is likely to spend 10% to 25% of their entire gross revenue **just** to financial and regulatory compliance overheads.

Fortunately, DeFi (Decentralized Finance) provides you with the capability to receive payments, quickly and efficiently, from anyone on the planet who wishes to pay you for your services.

Your software can use a variety of means to verify payment, and then license use of the various functionality of your software.

## 5.1 Using Plain HD Wallet Accounts

If your needs are simple, you can securely generate an `xpub...` key using a unique HD Wallet derivation path for each separate enterprise you wish to receive funds for. If this is done in a secure (ie. air-gapped) environment, then this `xpub...` key can safely be used to generate a sequence of HD Wallet addresses for each "client" you wish to charge.

The client deposits cryptocurrency, and you (later) transfer or aggregate it as you wish – using your normal, secure (ie. Hardware Wallet) transfer process.

This is simplest for the client, as they can buy Cryptocurrency on any exchange, and simply "withdraw" the correct amount of Cryptocurrency to the given account. However, these is no other information attached to the transaction – all "licensing" verification takes place manually, outside the cryptocurrency system.

So, generating a sequence of plain (Externally Owned Account, or EOA) client addresses from an `xpub...` is a worthwhile solution to consider. However, it does have some drawbacks:

### 5.1.1 Fee Distribution

Collecting up all the clients' payments later is a manual process. If you wish to distribute the payments (say, to pay partners, or a licensor whose software you sub-licensed, or to ensure that **you** get paid should someone license your software), this must all be done manually.

### 5.1.2 ERC-20 Tokens

If ERC-20 tokens are accepted into the generated EOA addresses, then you must transfer Ethereum into the account first, before transferring out the ERC-20 tokens. This requires at least one additional transfer fee, and since Gas fees are now variable, may result in a small amount of Ethereum abandoned in the account.

### 5.1.3 Distribution Failures

In any payment system with many clients paying for product(s), and fees being distributed to various payees, there are normally many trusted partners involved, and many manual (or automated) processes that can fail.

For example, what if a piece of software is created and distributed by some organization, and this software uses your licensed module? If the organization fails, or is deplatformed by a hostile government or corporation, or the software is abandoned, then clients who find the software and install it and want to pay for a license will probably not be able to pay the organization for it – and your license fees also will go unpaid.

Surely, there must be a way to deploy a sequence of interconnected Smart Contracts that can ensure that:

- Any new client can uniquely allocate (and optionally store some data with) a "Forwarder" address and pay for the product,
- Payment into that account address (and perhaps validation of its unique data) constitutes proof of licensing,
- Payees (direct and indirect) can flush any such payments through to themselves (and others)

These goals **can** be achieved using per-client "Forwarder" account addresses.

## 5.2 "Forwarder" Account Addresses

The solution to this problem is to use accounts that **nobody** has "private key" access to – including you (the software issuer), or any client.

These accounts host an Ethereum Smart Contract (the account address is actually the pre-computed address of a future instance of a "Forwarder" contract). They are not Externally Owned Account (EOA) addresses, and have no Private Key; they can only do what their Smart Contract defines.

The **only** function of these "Forwarder" Contracts is to collect the address' Ethereum and ERC-20 tokens into the product's "Fee Distribution" `MultiPayoutERC20` Contract account, optionally storing an immutable value associated with the "salt" value from which the "Forwarder" account address is derived.

### 5.2.1 The Client "Forwarder" Contract

Once the product's `MultiPayoutERC20` "Fee Distribution" contract address is identified, the act of obtaining a unique client payee "Forwarder" address is simple.

1. A "salt" value unique to the client is deduced, usually consisting of "something they know" (eg. a Public Key) plus "something they have" (eg. a Machine ID, or a User Name).
2. The salt is used to deduce the client's unique "Forwarder" address

1. The Product Owner's MultiPayoutERC20 Contract

   The creation of a `MultiPayoutERC20` contract is very simple.

   The product owner must know the Ethereum addresses of the payees, and each payee's proportion of the product revenue. A payee may be another MultiPayoutERC20 contract (eg. for a product module sub-license), which may in turn have its own payees.

   An Ethereum account containing sufficient funds to establish the MultiPayoutERC20 contract must be available. Here's an example, using the Ethereum "Goerli" testnet.

   Since contract creation is expensive, we'll determine if we've already deployed a MultiPayoutERC20 contract, so we don't need to spend funds to executed this example. We'll use any `GOERLI_SEED` or `GOERLI_XPRVKEY` defined in your environment, as well as `ALCHEMY_API_TOKEN` and `ETHERSCAN_API_TOKEN` available.

   First, lets get the Ethereum account private key details, and see if we've already deployed a MultiPayoutERC20:

```
#
# To create a new MultiPayoutERC20contract:
#
#     Provide yourself with a Goerli testnet account under your control;
# provide an "xpub..." key for it, or the BIP-39 Mnemonic phrase to
# derive its HD wallet.  Use the https://goerlifaucet.com to fund the
# account with some Goerli test Ethereum; requires you to set up an
# https://alchemy.com account, and put your API token in the
# ALCHEMY_API_TOKEN environment variable.
#
#     If you have an Etherscan API token, put it in ETHERSCAN_API_TOKEN.
# This will be used to scan for any existing contracts already deployed by
# your Goerli testnet Ethernet address.
#
import os
import logging

from web3 import Web3
import slip39
from slip39.invoice import MultiPayoutERC20, ethereum, Chain

goerli_xprvkey                 = os.getenv( 'GOERLI_XPRVKEY' )
if not goerli_xprvkey:
    goerli_seed                = os.getenv( 'GOERLI_SEED' )
    print(f"Using Ethereum seed: {goerli_seed}")
    if goerli_seed:
        try:
            # why m/44'/1'/... instead of m/44'/60'/...?  Dunno;
            # That's the derivation path that Trezor Suite uses for
            # Goerli testnet wallets...
             goerli_xprvkey = slip39.account(
                 goerli_seed, crypto="ETH", path="m/44'/1'/0'"
             ).xprvkey
        except Exception as exc:
            print(f"Failed to deduce XPRVKEY from seed: {exc}")

contract                       = None
mp_found                       = None
if goerli_xprvkey:
    # We have the means to authorize a transaction on an Ethereum account!
    # Get the Account.address public from the xprvkey
    goerli_src                 = slip39.account(
        goerli_xprvkey, crypto='ETH', path="m/0/0"
    )
    print(f"Using Ethereum address: {goerli_src}")

    # eg. f"wss://eth-goerli.g.alchemy.com/v2/{os.getenv( 'ALCHEMY_API_TOKEN' )}"
    provider_url               = ethereum.alchemy_url( ethereum.Chain.Goerli )
    provider                   = Web3.WebsocketProvider( provider_url )

    # Lets scan the address's transactions for any existing contract creations,
    # and see if any match our MultiPayoutERC20 API.  Will automatically use the
    # ETHERSCAN_API_TOKEN environment variable, if defined.
    try:
        for tx in reversed(ethereum.ethertx( chain=Chain.Goerli, address=goerli_src.address )):
            if not ( contract := tx.get( 'contractAddress' )):
                continue
            try:
                mp_found = MultiPayoutERC20(
                    provider,
                    address     = Web3.to_checksum_address( contract ),
                )
            except Exception as exc:
```

```
                print( f"Contract {contract} is not a MultiPayoutERC20: {exc}" )
            else:
                print( f"Contract {contract} IS a MultiPayoutERC20" )
                break
        else:
            print( f"No MultiPayoutERC20 contracts found for Goerli address {goerli_src}" )
    except Exception as exc:
        print( f"Failed to scan {goerli_src!r} for contracts: {exc}" )

f"MultiPayoutERC20 found: {contract}\n\n{mp_found}"
```

If we didn't find a MultiPayoutERC20, and have the means to deploy and have not already, do so! We'll always send the same proportion to the next 3 accounts in our HD wallet for this example.

```
#
# Deploy a new MultiPayoutERC20, if necessary
#
mp_created = None
if goerli_xprvkey and not contract:
    destination                 = tuple(
        a.address
        for a in slip39.accounts( goerli_xprvkey, crypto="ETH", paths=f"m/0/1-3" )
    )

    payees                      = {
        address: share + 1
        for share,address in enumerate(destination)
    }

    tokens                      = list(
        ethereum.tokeninfo(
            t,
            chain               = ethereum.Chain.Goerli,
            w3_url              = provider_url,
            use_provider        = Web3.WebsocketProvider
        )
        for t in (
            "0xe802376580c10fE23F027e1E19Ed9D54d4C9311e", # USDT
            "0xde637d4C445cA2aae8F782FFAc8d2971b93A4998", # USDC
            "0xaFF4481D10270F50f203E0763e2597776068CBc5", # WEENUS
            "0x1f9061B953bBa0E36BF50F21876132DcF276fC6e", # ZEENUS
        )
    )

    # print( tabulate( [
    #     [
    #         [ "Token", "Decimals", "Contract"]
    #     ] + [
    #         [ t.name, t.decimals, t.contract ]
    #         for t in tokens
    #     ]
    # ], tablefmt='orgtbl' ))

    erc20s                      = list(
        t.contract
        for t in tokens
    )

    try:
        mp_created              = MultiPayoutERC20(
            provider,
```

```
            agent       = goerli_src.address,
            agent_prvkey= goerli_src.prvkey,
            payees      = payees,
            erc20s      = erc20s,
        )
    except Exception as exc:
        print( f"Failed to deploy a new MultiPayoutERC20: {exc}" )
    else:
        print( f"Success deploying a new MultiPayoutERC20: {mp_created}" )
        contract                = mp_created._address

        print("MultiPayoutERC20 Newly Deployed Contract Details:")

f"MultiPayoutERC20 deployed: {mp_created}"
```

Finally, if we found or deployed a MultiPayoutERC20 contract, lets generate a "Forwarder" for some unique user-identifying data (we'll use a pre-existing contract, if necessary).

```
if not contract:
    # We haven't been able to create a contract; just show a pre-defined one.
    contract                    = "0xbE69793974Fc55cD8B94Dac6b410827740Cc6d68"

#
# To examine an existing MultiPayoutERC20 contract, use:
#     https://goerli.etherscan.io/address/<contract>
mp                              = MultiPayoutERC20(
    provider,
    address         = Web3.to_checksum_address( contract ),
)

import hashlib
import uuid
username                        = "perry@kundert.ca"
machine_id                      = uuid.uuid4()
salt                            = hashlib.sha256(
    f"{username}-{machine_id}".encode()
)
salt_int = int.from_bytes( salt.digest(), byteorder='big' )
forwarder                       = mp.forwarder_address( salt_int )

(
f"MultiPayoutERC20 Contract & Forwarder Details:\n\n{mp}\n\n"
+ tabulate( [
    [ "MultiPayoutERC20 Contract:", f"{contract}" ],
    [ "Unique Client Salt:", f"{salt.hexdigest()}" ],
    [ "Their Forwarder Contract:", f"{forwarder}" ],
], tablefmt='orgtbl' ))
```

### 5.2.2 Distributing Funds Deterministically

One (or more) Smart Contracts move the funds from this account, into the payees' accounts/Contracts.

If multiple independent parties must be paid out of the proceeds from each client, receiving payments to a plain Account (for which you hold the private key) may not be acceptable to all parties involved. A product owner providing a licensee the capability to sub-licensing their product may, for example, charge a much better fee, if the licensee can prove that payments will **automatically** flow back to the license owner, every time the licensee sells their product which contains the sub-license.

There are ways to ensure that each client payment **must** be distributed to each payee, as agreed, using cryptocurrencies which implement Smart Contracts.

A Smart Contract can be created which guarantees that Cryptocurrency funds from a source address are distributed in a fixed proportion to several destination addresses.

A Contract is created that is unique to each set of payee accounts and (fractional) distribution of assets, containing a function something like this (in Solidity):

```
function payout_internal() private nonReentrant {
    move_but_x10000_to(  9310, payable( address( 0x7F7458EF9A583B95DFD90C048d4B2d2F09f6dA5b ))); //   6.900%
    move_but_x10000_to(  5703, payable( address( 0x94Da50738E09e2f9EA0d4c15cf8DaDfb4CfC672B ))); //  40.000%
    move_but_x10000_to(     0, payable( address( 0xa29618aBa937D2B3eeAF8aBc0bc6877ACE0a1955 ))); //  53.100%
}
```

(you can see the remaining Smart Contract code in the slip-39 source.)
This function can be executed in various ways.

### 5.2.3 Disbursement Directly from Random Account

The most expensive and least flexible method constructs, executes and selfdestructs this `payout_internal` Smart Contract function.

```
constructor() payable {
    payout_internal();
    selfdestruc
}
```

### 5.2.4 Client "Forwarder" Account Creation and Licensing

Guaranteeing that each client's payments always flow through to the designated tree of payees is the responsibility of the `MultiPayoutERC20Forwarder` (client "Forwarder" account) and `MultiPayoutERC20` (product's "Fee Distribution") Smart Contracts.

Confirmation of Licensing is the responsibility of the client software. At software runtime, some checks are completed. At minimum, two pieces of data are loaded from storage:

- The Machine ID, and
- A Public Key and a Signature of the Machine ID (previously generated), or
- A Private Key (from which the Machine ID Signature can be generated)

Then, assuming:

- Accounts can be created uniquely for some pseudo-random client-specific identifying key
  - A public key for example
- Some data can be stored and later retrieved using that client key
  - The signature of some License text or the client-unique Machine ID (or a significant portion of it)
- Payments to the account can be queried from on the blockchain

The client software can check the blockchain to confirm payment to the account, and the saved data (eg. Signature) can be checked to confirm that this client is indeed the licensee. For example, the client's Private Key generates the Public Key, and the retrieved signature matches the Sgnature of the Machine ID.

If all of these tests pass, then the client software has confirmed that it is licensed.

If not, a licensing offer (invoice) can be generated, to allow the client to obtain a license.

1. Licensing Attacks

   An attacker can attempt to re-use some pre-existing license payment; it can inspect the blockchain history to obtain the prior allocation of a paid "Forwarder" account, and recover the client key (the client's public key), and the associated data stored for that key (the signature). However, when the software attempts to confirm that the public key signs the machine ID, it will fail, because the attacker doesn't have the original payee's machine ID – which was not included in the original blockchain transaction.

   Only if the client software is also under the control of the attacker can this attack succeed; but, in that case, the attacker can simply remove the entire license check from the software.

## 5.3 Single-Use Accounts using Pre-Signed Transfers

One tempting (but ultimately fragile) solution is to use Ethereum "transfer" transactions that you don't actually have the private key to sign.

One artifact of how Ethereum (and similar) Cryptocurrency systems create and validate transactions, is that the "source" address may be deduced from the transaction and its accompanying signature. Normally, one already "has" a source address (and its private key) containing funds, and then creates a transaction moving some of those funds (or executing a Smart Contract call) to some destination, and finally signs it using their private key. However, one **can** create the exact same transaction performing the same actions – and then **provide a random signature**, and deduce what Ethereum Account it **must** have originated from. This will be a pseudo-random (unpredictable) source Address, assuming that some bit(s) in the transaction and/or signature differ.

This "signed" transaction from this random Ethereum account may do anything – so long as (when it's finally executed) there is sufficient Ethereum available to pay the "Gas" fees, and to supply whatever value (in ETH, ERC-20 tokens, etc.) is required by the transaction.

### 5.3.1 Collection Failure

A number of failure modes exist that can result in cryptocurrency lost in this client address:

- Only ETH supported at reasonable cost
- If an ERC-20 token transfer is invoked, the exact token must be known in advance.
  - Any other token or Ethereum deposited would be lost
- If the exact correct amount of Ethereum to pay for Gas is not deposited, the transaction will fail and will not be re-executable, resulting in loss of all funds at the address.

### 5.3.2 Not Supported for Bitcoin

This idea is **not possible** using Bitcoin, due to its lack of general-purpose smart contracts, and the fact that one cannot "sign" transactions in advance to generate the "source" Account address: the transaction must contain specific information about the source UTXOs (Unspent TransaXion Outputs) being spent, which is of course unavailable in advance.

## 5.4 Testing `MultiPayoutERC20`

Put some TGOR (Test Goerli Ethereeum) tokens into the "zoo zoo ... wrong" Ethereum account on the Goerli testnet. This is (of course) a well-known account, and the funds will disappear pretty quickly, but should give you time to run the tests successfully.

You can mint TGOR for free, at:

```
https://faucet.quicknode.com/ethereum/goerli
https://goerlifaucet.com/
```

Transfer about 0.1 TGOR to the "zoo zoo ... wrong" test account:

```
0x667AcC3Fc27A8EbcDA66E7E01ceCA179d407ce00
```

Then, run:

```
make test # or: make unit-test_multipayout_ERC20_web3_tester
```

# 6 Building & Installing

The `python-slip39` project is tested under both homebrew:

```
$ brew install python-tk@3.9
```

and using the official python.org/downloads installer.

Either of these methods will get you a `python3` executable running version 3.9+, usable for running the `slip39` module, and the `slip39.gui` GUI.

## 6.1 The slip39 Module

To build the wheel and install `slip39` manually:

```
$ git clone git@github.com:pjkundert/python-slip39.git
$ make -C python-slip39 install
```

To install from Pypi, including the optional requirements to run the PySimpleGUI/tkinter GUI, support serial I/O, and to support creating encrypted BIP-38 and Ethereum JSON Paper Wallets:

```
$ python3 -m pip install slip39[gui,wallet,serial]
```

## 6.2 The slip39 GUI

To install from Pypi, including the optional requirements to run the PySimpleGUI/tkinter GUI:

```
$ python3 -m pip install slip39[gui]
```

Then, there are several ways to run the GUI:

```
$ python3 -m slip39.gui     # Execute the python slip39.gui module main method
$ slip39-gui                # Run the main function provided by the slip39.gui module
```

### 6.2.1 The macOS/win32 SLIP-39.app GUI

You can build the native macOS and win32 `SLIP-39.app` App.

This requires the official python.org/downloads installer; the homebrew python-tk@3.9 will not work for building the native app using either `PyInstaller`. (The `py2app` approach doesn't work in either version of Python).

```
$ git clone git@github.com:pjkundert/python-slip39.git
$ make -C python-slip39 app
```

### 6.2.2 The Windows 10 SLIP-39 GUI

Install Python from `https://python.org/downloads`, and the Microsoft C++ Build Tools via the Visual Studio Installer (required for installing some `slip39` package dependencies).

To run the GUI, just install `slip39` package from Pypi using pip, including the `gui` and `wallet` options. Building the Windows `SLIP-39` executable GUI application requires the `dev` option.

```
PS C:\Users\IEUser> pip install slip39[gui,wallet,dev]
```

To work with the python-slip39 Git repo on Github, you'll also need to install Git from git-scm.com. Once installed, run "Git bash", and

```
$ ssh-keygen.exe -t ed25519
```

to create an `id_ed25519.pub` SSH identity, and import it into your Git Settings SSH keys. Then,

```
$ mkdir src
$ cd src
$ git clone git@github.com:pjkundert/python-slip39.git
```

1. Code Signing

   The MMC (Microsoft Management Console) is used to store your code-signing certificates. See stackoverflow.com for how to enable its Certificate management.

# 7 Licensing

Each installation of the SLIP-39 App requires an Ed25519 "Agent" identity, and cryptographically signed license(s) to activate various python-slip39 features. No license is required to use basic features; advanced features require a license.

## 7.1 Create an Ed25519 "Agent" Key

The Ed25519 signing "Agent" identity is loaded at start-up, and (if necessary) is created automatically on first execution. This is similar to the `ssh-keygen -t ed25519` procedure.

Each separate installation must have a ~/.crypto-licensing/python-slip39.crypto-keypair. This contains the licensing "Agent" credentials: a passphrase-encrypted Ed25519 private key, and a self-signed public key. This shows that we actually had access to the private key and used it to create a signature for the claimed public key and the supplied encrypted private key – proving that the public key is valid, and associated with the encrypted private key.

## 7.2 Validating an Advanced Feature License

When an advanced feature is used, all available `python-slip39.crypto-license` files are loaded. They are examined, and if a license is found that is:

- Assigned to this Agent and Machine-ID
- Contains the required license authorizations

then the functionality is allowed to proceed.

If no license is found, instructions on how to obtain a license for this Agent on this Machine-ID will be displayed.

If you've already obtained a "master" license on your primary machine's SLIP-39 installation, you can use it to issue a sub-license to this installation (eg. for your air-gapped cryptocurrency management machine).

Otherwise, a URL is displayed at which the required "master" license can be issued.

### 7.2.1 Get a sub-license From Your "master" License

Typically, you'll be using python-slip39's advanced features on an air-gapped computer. You do not want to visit websites from this computer. So, you obtain a sub-license from your primary computer's python-slip39 installation, and place it on your secure air-gapped computer (eg. using a USB stick).

Take note of the secondary machine's Agent ID (pubkey) and Machine ID. On your primary computer (with the "master" license), run:

```
python3 -m slip39.sublicense <agent-pubkey> <machine-id>
```

Take the output, and place it in the file `~/.crypto-licensing/python-slip39.crypto-license` on your air-gapped computer.

### 7.2.2 Obtaining an Advanced Feature "master" License

On your primary computer, open the provided URL in a browser. The URL contains the details of the advanced feature desired.

This URL's web page will request an Ed25519 "Agent" public key to issue your "master" license to. This should be your primary user account's Ed25519 "Agent" public key – this master "Agent" will be issuing sub-licenses to any of your other SLIP-39 installations. You will be redirected to a URL that is unique to the advanced feature plus your Agent ID.

An invoice will be generated with unique Bitcoin, Ethereum and perhaps other cryptocurrency addresses. Pay the required amount of cryptocurrency to one of the provided wallet addresses. Within a few seconds, the cryptocurrency transfer will be confirmed.

Once the payment for the advanced feature is confirmed, the URL including your agent ID will always allow you to re-download the license. It is only usable by your Agent ID to issue sub-licenses to your python-slip39 installations on your machines.

# 8 Dependencies

Internally, python-slip39 project uses Trezor's python-shamir-mnemonic to encode the seed data to SLIP-39 phrases, python-hdwallet to convert seeds to ETH, BTC, LTC and DOGE wallets, and the Ethereum project's eth-account to produce encrypted JSON wallets for specified Ethereum accounts.

## 8.1 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 location r
Group 2 of 4 - 1 of 1 shares required:
merit aluminum beard romp briefing email member flavor disaster exercise cinema subject perfect facility genius bike inclu
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 garlic en
merit aluminum ceramic scared beam findings expand broken smear cleanup enlarge coding says destroy agency emperor hairy d
merit aluminum ceramic shadow cover smith idle vintage mixture source dish squeeze stay wireless likely privacy impulse to
merit aluminum ceramic sister duke relate elite ruler focus leader skin machine mild envelope wrote amazing justice mornin
merit aluminum ceramic smug buyer taxi amazing marathon treat clinic rainbow destroy unusual keyboard thumb story literary
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 fantasy extra
merit aluminum decision scatter carpet spine ruin location forward priest cage security careful emerald screw adult jerky
merit aluminum decision shaft arcade infant argue elevator imply obesity oral venture afraid slice raisin born nervous uni
merit aluminum decision skin already fused tactics skunk work floral very gesture organize puny hunting voice python trial
merit aluminum decision snake cage premium aide wealthy viral chemical pharmacy smoking inform work cubic ancestor clay ge
merit aluminum decision spider boundary lunar staff inside junior tendency sharp editor trouble legal visual tricycle auct
```