The SLIP-39 Digest: Detecting a Bad Share Set

November 4, 2026 · Document #418 · bip39-phrase.com



Photo by Eloi Smith on Unsplash

How SLIP-39 uses a reserved digest point to detect inconsistent shares, and what its 32-bit recovery check can and cannot prove.

Contents

How SLIP-39 splits bytes over GF(256)	2
Why the digest sits at x=254	2
How recovery detects an inconsistent set	3
What a successful check actually proves	3
FAQ	4
Sources	4
Example BIP39 word sequences	4
Reading a BIP39 phrase	6
See also	6

A set of SLIP-39 shares can contain enough points for Shamir interpolation and still reconstruct the wrong secret. Interpolation alone cannot detect this: any threshold-sized set of points with distinct x-coordinates defines a polynomial of degree below the threshold. For thresholds greater than one, SLIP-39 includes a **4-byte digest** so recovery can test whether the reconstructed bytes pass an internal consistency check before accepting them. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

This check is separate from each mnemonic's checksum. The mnemonic checksum helps detect transcription errors within one share; the digest checks the relationship among the shares used for reconstruction. Neither check detects every possible error. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

How SLIP-39 splits bytes over GF(256)

SLIP-39 applies Shamir's secret-sharing scheme to a byte string. Each byte is an element of **GF(256)**, the finite field with 256 elements. Addition and subtraction are XOR; multiplication and division follow finite-field arithmetic, rather than ordinary integer arithmetic modulo 256.

The field uses the irreducible polynomial `x^8 + x^4 + x^3 + x + 1`, represented by `0x11B`. Every nonzero element has a multiplicative inverse, which makes the divisions in polynomial interpolation possible. ([github.com](https://github.com/trezor/python-shamir-mnemonic/blob/master/shamir_mnemonic/shamir.py))

For a threshold `t`, each byte position has a polynomial of degree at most `t - 1`. A share contains an x-coordinate and the polynomial's value at that coordinate for every byte position. All bytes in that share use the same x-coordinate, but each byte position is interpolated separately.

Consequently, every share's unpadded data payload has the same byte length as the secret being split. The mnemonic also carries metadata, padding bits and a checksum. A member index supplies the member share's x-coordinate; a group index supplies the group share's x-coordinate. Ordinary share coordinates run from `0` through `15`.
([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

Why the digest sits at x=254

For thresholds greater than one, SLIP-39 reserves two coordinates outside that ordinary share range:

- At `x=255`, the polynomial evaluates to the secret byte string.
- At `x=254`, it evaluates to a digest followed by random bytes.

For a secret of `L` bytes and a threshold greater than one, the splitter generates `L - 4` uniformly random bytes, called `R` here. It computes `D = HMAC-SHA256(key=R, message=secret)[:4]`. The value at `x=254` is the concatenation `D || R`, exactly `L` bytes long.

The random suffix therefore serves as the HMAC key. The secret is the HMAC message. Only the first four bytes of the HMAC output are retained. ([github.com](https://github.com/trezor/python-shamir-mnemonic/blob/master/shamir_mnemonic/shamir.py))

To construct the sharing polynomial, the splitter combines these two reserved points with `t - 2` independently and uniformly random share values at distinct ordinary coordinates. Those `t` points determine the polynomial at every byte position. It then evaluates that polynomial at the remaining coordinates needed for the distributed shares. The random values must come from a source suitable for generating cryptographic keys.
([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

For a threshold of two, the reserved secret and digest points suffice; there are no additional random anchor points. The random suffix in the digest point still supplies randomness.

The reserved coordinates keep the secret and its checking value separate from distributed share coordinates. Because both are values on the same polynomial, recovery can reconstruct both from the same threshold set. The digest adds no extra bytes to each share's data payload. ([github.com](https:/

/github.com/trezor/python-shamir-mnemonic/blob/master/shamir_mnemonic/shamir.py))

How recovery detects an inconsistent set

Recovery performs the following check at each applicable sharing layer:

1. Decode the mnemonics, verify their checksums, and check their metadata, lengths, padding and indexes. Select exactly the required number of groups and exactly the required number of distinct member shares within each selected group.
2. Interpolate the selected share values at `x=255` to obtain a candidate secret.
3. Interpolate the same values at `x=254` to obtain a candidate digest string.
4. Split that string into its first four bytes, `D`, and its remaining bytes, `R`.
5. Recompute `HMAC-SHA256(key=R, message=candidate_secret)[:4]` and compare it with `D`.

A mismatch means recovery must reject the reconstruction. Without this comparison, interpolation of a threshold-sized set would simply return bytes, with no indication that the shares were inconsistent. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

SLIP-39 uses two sharing layers: member shares reconstruct group shares, and group shares reconstruct the encrypted master secret. The digest mechanism applies at each layer whose threshold exceeds one. A threshold-one split simply copies its input and has no digest check at that layer. ([github.com](https://github.com/trezor/python-shamir-mnemonic/blob/master/shamir_mnemonic/shamir.py))

What a successful check actually proves

For an accidentally inconsistent reconstruction, the probability of a matching 32-bit digest is approximately `2^-32` under the usual random-output assumption: about one in 4.3 billion per check. This provides strong error detection, but it is not the probability that an accepted backup is wrong, or proof that the shares came from the intended backup. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

The digest is **not authentication** against someone able to construct a replacement share set. Its HMAC key is reconstructed from the shares themselves. An attacker creating a complete replacement set can calculate a matching digest. ([github.com](https://github.com/trezor/python-shamir-mnemonic/blob/master/shamir_mnemonic/shamir.py))

Nor does the check identify which share is wrong or automatically correct errors. If recovery reports a digest failure, check the backup's identity, group and member indexes, thresholds, and transcription. Do not bypass the failure to obtain candidate wallet bytes. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

The outer reconstruction recovers an encrypted master secret. Passphrase-based decryption happens separately, so this digest does not validate the recovery passphrase or guarantee that the resulting wallet is the intended one. ([github.com](https://github.com/trezor/python-shamir-mnemonic/blob/master/shamir_mnemonic/shamir.py))

FAQ

Can a valid mnemonic checksum hide an incompatible share?

Yes. A share can pass its own checksum while being incompatible with the reconstruction set. Metadata checks and the digest address different parts of that problem, but passing them does not authenticate the backup. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

Why use only four digest bytes?

Four bytes provide a compact 32-bit consistency check. They occupy part of the reserved digest value, leaving the remaining bytes for the random HMAC key without increasing the share payload length. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

Does a passing digest confirm my wallet?

No. It means the reconstructed values satisfy the digest relation; an accidentally inconsistent reconstruction is unlikely to pass. Confirm the intended wallet separately using trusted wallet records, such as previously recorded receiving addresses. ([github.com](https://github.com/satoshilabs/slips/blob/master/slip-0039.md))

Sources

- SLIP-0039: Shamir's Secret-Sharing for Mnemonic Codes (github.com)
- SLIP-39 Reference Implementation: Shamir Sharing and Recovery (github.com)

Continue reading on bip39-phrase.com: SLIP-39 vs BIP39: Shamir Backup and 20-Word Shares Explained
More on this topic: BIP Standards

Example BIP39 word sequences

Randomly generated sequences of words from the BIP39 English wordlist. None of them passes the BIP39 checksum, so no wallet will accept them: they show the format only. Never store funds with a phrase you found in a document.

Example 1					
1	common	2	void	3	ankle
4	layer	5	surprise	6	purpose
7	army	8	pattern	9	diagram
10	need	11	pencil	12	cradle

Example 2

favorite	ripple	pencil
current	pole	picnic
column	episode	stone
lab	shoe	apple

Example 3

drum	submit	sunset
farm	uncover	buzz
kangaroo	melody	fancy
rib	hockey	vote

Example 4

dream	cattle	cattle
inspire	lion	grain
inhale	wool	grass
slight	hamster	cargo

Example 5

diagram	inflict	search
kit	tower	fetch
spoil	disorder	slide
unable	bicycle	tray

Example 6

finish	crash	slam
flat	smile	invest
ecology	syrup	old
notable	click	acquire

Example 7

attract	punch	tired
mouse	limit	brave
badge	orange	owner
twin	various	evidence

Example 8

pen	gentle	adult
minimum	donate	dance
subway	dad	toast
lesson	enable	trade

Example 9		
1 clog	2 trick	3 trophy
4 august	5 joke	6 crunch
7 able	8 human	9 cool
10 open	11 noble	12 news

Reading a BIP39 phrase

Every word stands for an 11-bit number: its position in the 2048-word list, counted from 0. Here is Example 1 decoded word by word. Each word links to its page on bip39-phrase.com, where the list is numbered from 1.

#	Word	Index (0-2047)	Word no.	11 bits
1	common	371	372	00101110011
2	void	1964	1965	11110101100
3	ankle	73	74	00001001001
4	layer	1010	1011	01111110010
5	surprise	1747	1748	11011010011
6	purpose	1395	1396	10101110011
7	army	96	97	00001100000
8	pattern	1290	1291	10100001010
9	diagram	487	488	00111100111
10	need	1182	1183	10010011110
11	pencil	1301	1302	10100010101
12	cradle	399	400	00110001111

The 12 words carry 132 bits: 128 bits of entropy followed by a 4-bit checksum. The checksum must equal the first 4 bits of the SHA-256 hash of the entropy.

`Entropy (hex): 2e7eb024bf2da75cc3050a3cf27a8a98`

This sequence ends in the checksum bits 1111, but SHA-256 of its entropy starts with 0010, so wallets reject it as an invalid phrase. That is the point of the checksum: about 15 of every 16 random or mistyped sequences fail it.

See also

- BIP-39 Seed Phrases: The Most Dangerous Backup You'll Ever Own · June 18, 2025
- Mnemonic Keys to the Kingdom: Why BIP-39 Still Matters in 2025 · June 15, 2025

Educational document. The example sequences are random and fail the BIP39 checksum: they are not wallets and must never hold funds. Security notes: bip39-phrase.com/security/