Proposal: Numeric Chunk Encoding for Deterministic Wallet Backups

Author: Zachary Smith

Status: Draft

Type: Standards Track

Created: 2024-12-26

Abstract

This proposal introduces a method for encoding entropy and checksum data as a series of numeric chunks for deterministic wallet generation and recovery, similar to BIP-39. The approach eliminates reliance on wordlists, providing a more compact and efficient format for backup and restoration.

The encoding system supports both backup and restoration of the original entropy, while maintaining a high degree of error resistance and compatibility with existing wallet standards.

Motivation

Existing wallet backup methods often rely on wordlists, which introduce inefficiencies and ambiguities:

1. Error-Prone Truncation

- Some systems only recognize the first four letters of words, leading to potential recovery errors.

2. Complexity

- Wordlists require additional encoding and decoding steps, adding complexity to backups and recovery.

3. Space and Input Limitations

- Words are verbose and may not fit well into compact or constrained backup systems.

This proposal eliminates these issues by representing entropy and checksum as a sequence of numeric values, resulting in a more compact and streamlined approach that is easier to store, transmit, and restore.

Specification

1. Encoding Process

Step 1: Generate Entropy

• Generate a random sequence of ENT bits of entropy.
Example values: 128, 160, 192, 224, 256.
ENT must be a multiple of 32.

Step 2: Calculate Checksum

• Compute the checksum as the first ENT / 32 bits of the SHA-256 hash of the entropy.

Step 3: Append Checksum

• Append the checksum bits to the entropy.
The resulting bit length becomes ENT + (ENT / 32).

Step 4: Split into 11-Bit Chunks

• Divide the combined entropy and checksum bit sequence into 11-bit chunks.
Each chunk represents a numeric value in the range [0, 2047].

Step 5: Output Numeric Chunks

• The resulting sequence of numbers constitutes the encoded backup.

Example: Encoding

For 128 bits of entropy:

1. Entropy:
1101011101100101... (128 bits).

2. Checksum:
0011 (first 4 bits of SHA-256 hash of the entropy).

3. Combined:
1101011101100101...0011 (132 bits).

4. Split into Chunks:
11010111011, 001011... (12 groups of 11 bits).

5. Numeric Encoding:
[1715, 645, 2046, 31, ...] (12 numbers).

2. Restoration Process

Step 1: Input Numeric Chunks

- Take the sequence of numbers [0–2047] that were used as the backup.

Step 2: Convert Numbers to Binary

- Convert each numeric value back into its 11-bit binary representation.
  Example: 1715 → 11010111011.

Step 3: Combine Binary Chunks

- Concatenate the binary sequences into a single bitstring representing the original entropy and checksum.

Step 4: Separate Entropy and Checksum

- Extract the first ENT bits as entropy.
- Extract the remaining ENT / 32 bits as the checksum.

Step 5: Validate Checksum

- Recompute the checksum by taking the SHA-256 hash of the entropy and extracting the first ENT / 32 bits.
- Compare the computed checksum with the extracted checksum:
- If they match, the entropy is valid.
- If they do not match, the input data contains an error.

Step 6: Use the Restored Entropy

- Once the entropy is validated, it can be used as input for seed generation or other deterministic wallet processes.

Example: Restoring

For the numeric backup [1715, 645, 2046, 31, …]:

1. Convert Numbers to Binary:

- Example: 1715 → 11010111011.
- Combine all binary chunks into a single sequence:
  11010111011 001011...0011 (132 bits).

2. Separate Entropy and Checksum:

- First 128 bits: Entropy.
- Last 4 bits: Checksum.

3. Validate Checksum:

- Compute SHA-256 hash of the entropy.
- Extract the first 4 bits of the hash and compare to the checksum.
- If valid, restoration is complete.

---

3. Seed Derivation

The restored entropy can be converted into a deterministic wallet seed using the standard PBKDF2 function:

(Python Pseudo Code)
seed = PBKDF2(HMAC-SHA512, entropy, "mnemonic" + passphrase, 2048, 64)

The resulting 512-bit seed is fully compatible with deterministic wallet systems.

Advantages

1. Compact Format

- Numeric chunks are shorter and less verbose than traditional word-based representations, making them ideal for constrained environments.

2.Error Resistance

- The checksum ensures that errors during recovery can be detected.

3. Universal Simplicity

- Numbers are easier to write, store, and input than wordlists or raw binary.

4. Hardware Compatibility

- Numeric backups can be engraved on steel plates, represented as QR codes, or stored in any medium that supports numeric data.

5. Interoperability

- Fully compatible with existing systems that use entropy-based seed recovery.

Compatibility

This encoding system is backward-compatible with any wallet or tool that accepts raw entropy for seed derivation. Minor software updates are required to support numeric chunk format for restoration.

Reference Implementation

Encoding (Pseudo Code)

```
import hashlib

def encode_entropy_to_numeric_chunks(entropy: bytes) -> list:
    ENT = len(entropy) * 8
    if ENT % 32 != 0 or not (128 <= ENT <= 256):
        raise ValueError("Invalid entropy length")

    checksum_length = ENT // 32
    checksum = int(hashlib.sha256(entropy).hexdigest(), 16) >> (256 - checksum_length)

    entropy_with_checksum = int.from_bytes(entropy, 'big') << checksum_length | checksum

    total_bits = ENT + checksum_length
    numeric_chunks = []
    for i in range(total_bits // 11):
        chunk = (entropy_with_checksum >> (total_bits - 11 * (i + 1))) & 0x7FF
        numeric_chunks.append(chunk)

    return numeric_chunks
```

Restoration (Pseudo Code)

```
def restore_entropy_from_numeric_chunks(numeric_chunks: list, entropy_bits: int) -> bytes:
    checksum_length = entropy_bits // 32
    total_bits = entropy_bits + checksum_length

    # Recombine binary chunks
    binary_sequence = 0
    for chunk in numeric_chunks:
        binary_sequence = (binary_sequence << 11) | chunk

    # Extract entropy and checksum
    entropy_with_checksum = binary_sequence
    entropy = entropy_with_checksum >> checksum_length
    checksum = entropy_with_checksum & ((1 << checksum_length) - 1)

    # Validate checksum
```

```
    computed_checksum = int(hashlib.sha256(entropy.to_bytes(entropy_bits // 8,
'big')).hexdigest(), 16) >> (256 - checksum_length)
    if checksum != computed_checksum:
        raise ValueError("Checksum validation failed")

    return entropy.to_bytes(entropy_bits // 8, 'big')
```

Conclusion

This proposal simplifies the wallet backup and restoration process by encoding entropy and checksum into numeric chunks. The system ensures compactness, error resistance, and compatibility with modern deterministic wallet standards while avoiding the complexities of word-based systems.