



Lecture 06: Create our Own Wallet



How to create a Digital wallet



Step 0: The True Beginning – Entropy (Randomness)

◆ What is Entropy?

A long string of 1s and 0s — the raw randomness that forms the foundation of your wallet.

- For a **12-word seed phrase** →  **128-bit number**.
 - For a **24-word seed phrase** →  **256-bit number**.
-

◆ How is it Generated?

✓ A **hardware wallet** has a built-in special chip called a:

 **True Random Number Generator (TRNG)**

This chip measures unpredictable physical phenomena (like tiny electrical noise) to create true randomness.

◆ Can You Contribute to Randomness?

👉 Yes! Some wallets let you add randomness by:

-  Rolling dice

-  Pressing or mashing buttons randomly

This ensures your seed is even harder to guess.

Example Entropy (128 bits)

```
plaintext
CopyEdit
10100101110101110010000110110101...
(and so on for 128 digits)
```

✨ **Why does this matter?**

The stronger and more random your entropy, the more secure your wallet — because it ensures no one can guess your private key or seed.

Step 1: Creating the Seed Phrase (Your Human-Friendly Backup)

Why a Seed Phrase?

Humans can't easily memorize a 128-bit binary number — so we convert it into a **human-readable list of words**.

This is done using the standard:

 **BIP-39 (Bitcoin Improvement Proposal 39)**

How It Works: The 4 Steps

1. The BIP-39 Wordlist

- There is an official list of **2048 unique words**.
- Examples:


"abandon" , "ability" , "able" , ... "zoo"


- Each word corresponds to a number between **0** and **2047** .
-

✓ 2. Chopping the Entropy

- The initial **128-bit entropy** is split into **11-bit chunks**.
 - Why?
Because $2^{11} = 2048$, which perfectly matches the number of words in the list.
 - Each 11-bit chunk maps directly to one word from the wordlist.
-

✓ 3. Adding a Checksum

 The checksum protects against mistakes when writing down the phrase.

- A few bits from the hash of the original entropy are appended at the end.
 - This is why if even *one* word is wrong, your wallet detects it and shows:
 "Invalid Seed Phrase"
 - Without a checksum, you might accidentally create a wrong — and empty — wallet.
-

✓ 4. Mapping to Words

- Each 11-bit chunk → Find the corresponding word in the list → Display the full **12-word (or 24-word) phrase**.
 - This is your **backup** — you must store it securely!
-

What is a Checksum?

A **checksum** is a small piece of calculated data that verifies the integrity of larger data.

 Real-world analogy: Credit card numbers

The last digit of a credit card isn't random — it's calculated from the others.

If you mistype another digit, the system instantly knows the number is invalid.

Similarly, your wallet uses the checksum to detect mistakes in your seed phrase.

Visual Flow

```
mathematica
CopyEdit
[ Initial 128-bit Entropy ]
    |
    ▼
Run through SHA-256
    |
Take first 4 bits → [ 4-bit Checksum ]
    |
Combine entropy + checksum → [ 132-bit Data ]
    |
Split into 12 × 11-bit chunks
    |
Map each chunk → [ Word 1 ], [ Word 2 ], ... [ Word 12 ]
```

✨ Summary:

- ✅ Your seed phrase is a **human-friendly, error-checked representation of your wallet's entropy**.
- ✅ It is your most important backup — **write it down, don't lose it!**



Step 2: From Seed Phrase to Master Private Key



How do we get the Master Key?

We use a cryptographic function called:

PBKDF2

(Password-Based Key Derivation Function 2)

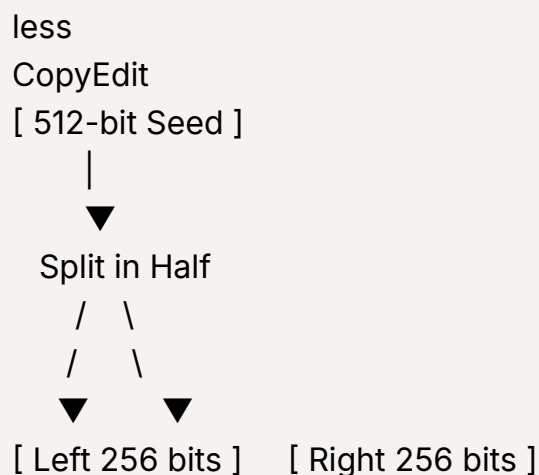
✓ This takes your **seed phrase** and derives a strong, cryptographically secure number.

◆ Output: The 512-bit Seed

- The output of PBKDF2 is:
 - 🔧 **512 bits (64 bytes)**
 - This 512-bit seed becomes the foundation of your wallet.
-

◆ Splitting the Seed

We split the **512-bit Seed** into two equal halves:



◆ What are the halves used for?

- ✓ Left 256 bits → MASTER PRIVATE KEY
 - ✓ Right 256 bits → MASTER CHAIN CODE
-

📌 Why do we need the Chain Code?

The **Master Chain Code** is used to derive child keys securely in a hierarchical structure — without revealing the master private key itself.

✨ **Summary:**

- Your **seed phrase** → PBKDF2 → 512-bit seed → split → **Master Private Key** + **Chain Code**
 - These two pieces allow your wallet to generate an infinite number of addresses in a secure and organized way.
-


Step 3: From Master Key to Specific Private Key

The "Engine Room" of Your Wallet

Your wallet takes the single **Master Private Key** and generates an **infinite number of specific keys & addresses** — in a secure, predictable, and organized way.

Why Generate Multiple Keys?

1. Privacy


- Using the same address every time makes all your transactions visible to anyone.
-  By using a *different address for each transaction*, your financial history stays much harder to track.


2. Organization

- Manage multiple cryptocurrencies (e.g., Bitcoin, Ethereum, Litecoin) with the **same seed phrase**.
 - Create separate accounts for "Savings" and "Spending," all derived securely.
-

The Solution: HD Wallets (BIP-32 & BIP-44)




Your wallet follows the **Hierarchical Deterministic (HD)** standard.

-  **Hierarchical:** Keys are organized in a tree — parent keys generate child keys.

-  **Deterministic:** The same seed always generates the same tree structure.

Decoding the Path: `m/44'/0'/0'/0/0`

Let's break it down step by step — like navigating folders in a file system:

 Path Component	 Name	 What It Means
m	Master	Root of the tree — your Master Private Key + Chain Code
/	Separator	Moves one level deeper
44'	Purpose	Signals that the path uses BIP-44 standard
0'	Coin Type	Which cryptocurrency? 0 = Bitcoin, 60 = Ethereum, 2 = Litecoin
0'	Account	Separate accounts within the same coin. Example: 0 = Savings, 1 = Spending
0	Change	Address type: 0 = External (for receiving), 1 = Internal (for change)
0	Address Index	Specific address within that branch. 0 , 1 , 2 , ... up to billions

✨ **Summary:**

✅ With HD wallets, you can derive and manage thousands of addresses — across coins and accounts — with just one seed phrase.

✨ **Step 4: From Private Key to Public Key**

The Cryptographic Magic

Your **Private Key** is transformed into a **Public Key** using a special one-way mathematical function:

Elliptic Curve Cryptography (ECC)

Why ECC?

✅ **One-way function:**

It's easy to compute a public key from a private key — but nearly impossible to reverse it and compute the private key from the public key.

✅ **Secure and efficient:**


ECC achieves high levels of security even with smaller keys (compared to RSA).

◆ **How does it work? (Conceptually)**

- The private key is just a very large random number.
- This number is used as a "scalar" to multiply a fixed point on an **elliptic curve**.
- The result is another point on the curve — your **Public Key**.

✏️ **Formula (simplified):**

```
vbnet
CopyEdit
Public Key = Private Key × Generator Point (G)
```

Here,  is a fixed point on the chosen elliptic curve, standardized by the blockchain protocol.

✨ **Summary:**

✅ Your Public Key is securely derived from your Private Key, making it safe to share publicly (while keeping your Private Key secret).

Step 5: Generate Public Address

After obtaining the **Public Key**, the final step is to convert it into a **Public Address** — the one you share to receive funds.

1. Bitcoin (Legacy Address – P2PKH)

✅ Bitcoin's process is **the most elaborate**, designed for maximum error detection.

- ✓ Goal: make it nearly impossible to mistype an address unnoticed.

◆ Input:

A 256-bit (33-byte compressed) Public Key

◆ Steps:

1 SHA-256 Hash:

```
scss
CopyEdit
SHA256(PublicKey) → 32-byte hash
```

2 RIPEMD-160 Hash:

```
arduino
CopyEdit
RIPEMD160(32-byte hash) → 20-byte hash
```

3 Add Version Byte:

- Prepend `0x00` (for mainnet) to the 20-byte hash:

```
arduino
CopyEdit
0x00 + 20-byte hash → 21-byte payload
```

4 Create Checksum:

- Double SHA-256 the 21-byte payload:

```
scss
CopyEdit
SHA256(SHA256(21-byte payload)) → first 4 bytes → checksum
```

5 Append Checksum:

arduino

CopyEdit

21-byte payload + 4-byte checksum → 25-byte full address

6 Base58Check Encode:

- Encode the 25 bytes using Base58Check to get the human-readable address.

Final Product:

✓ Starts with 1

✓ Example:

CopyEdit

1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2

✓ Built-in typo detection (via checksum).

2. Ethereum (and EVM-compatible chains)

✓ Ethereum chose a **simpler and shorter** approach — optional checksum via letter casing.

◆ Input:

A 512-bit (64-byte uncompressed) Public Key

◆ Steps:

1 Keccak-256 Hash:

scss

CopyEdit

Keccak256(PublicKey) → 32-byte hash

2 Take Last 20 Bytes:

arduino
CopyEdit
Last 20 bytes of the 32-byte hash → 20-byte address

3 (Optional) EIP-55 Checksum (via casing):

- Hash the lowercase address again with Keccak-256.
- Uppercase certain letters based on the hash result.
- Wallets can use this to verify correctness.

Final Product:

✓ Starts with `0x`

✓ Examples:

makefile
CopyEdit
Lowercase: 0x742d35af1d49f05829b3a0b14c33a94da8c5a4d0
Checksummed: 0x742d35AF1d49F05829B3a0B14C33A94da8C5a4d0

✓ Short, simple — relies on optional casing (EIP-55) for error checking.

3. Solana

✓ Solana's philosophy: **the key pair is the identity.**

◆ Input:

A 256-bit (32-byte) Ed25519 Public Key

◆ Steps:

1 Use Public Key directly:

No hashing is performed — the Public Key is the Address.

2 Base58 Encode:

- Encode the 32-byte Public Key using Base58 for readability.

Final Product:

✓ Example:

```
nginx
CopyEdit
So11111111111111111111111111111111111112
```

✓ Most direct — no built-in checksum. Applications usually check before sending.

Smallest Units of Cryptocurrency:

Blockchain	Smallest Unit	 In Honor Of	1 Coin = ... Smallest Units
Bitcoin (BTC)	Satoshi (sat)	Satoshi Nakamoto	100,000,000 (10^8)
Ethereum (ETH)	Wei	Wei Dai	1,000,000,000,000,000,000 (10^{18})
Solana (SOL)	Lamport	Leslie Lamport	1,000,000,000 (10^9)

How to Connect to a Blockchain Network (RPC Server)

✓ What is JSON-RPC?

A lightweight **remote procedure call (RPC) protocol** encoded in JSON.

- Enables clients to send method calls to a blockchain node (server) and get responses.
- Works over HTTP(S), WebSockets, or IPC.
- Used to send **transactions** or **fetch data (e.g., balances)** from the blockchain.

Two Main User Actions:

✓ 1. Send a Transaction

✓ 2. Fetch Blockchain Data

Lifecycle of an Ethereum Transaction: Step by Step



Step 1: The Transaction "Object"

An Ethereum transaction is a JSON object with these fields:



Field	Description
<code>to</code>	Recipient's <code>0x...</code> address
<code>value</code>	Amount of ETH (in Wei)
<code>gasLimit</code>	Max gas allowed
<code>gasPrice</code> / <code>maxFeePerGas</code>	Price per unit of gas
<code>nonce</code>	How many txs sent from this account
<code>data</code>	(optional) Instructions for smart contracts
<code>chainId</code>	Network identifier (Mainnet, Testnet, etc.)



Step 2: The Nonce (Not Like Bitcoin's Nonce)

- In **Ethereum**, the nonce is a **counter of transactions** sent by an account.
- Starts at `0` for the first tx, `1` for the second, etc.

✓ Why Nonce?

-  Prevents replay attacks (same tx cannot be reused).
-  Ensures strict transaction order.



Step 3: Fees = Gas

Unlike Bitcoin's size-based fee model, Ethereum uses a **computation-based model**:



Gas Analogy:

Term	Meaning
Gas Limit	Max fuel you're willing to spend
Gas Used	Actual fuel consumed
Gas Price	How much you pay per unit of gas
Fee	<code>Gas Used * Gas Price</code>

✓ **Simple ETH transfer = 21,000 gas.**

Unused gas is refunded.



Step 4: The Data Field

- Usually empty for ETH transfer.
- Filled with **function calls + parameters** for smart contracts.



Example:

```
less
CopyEdit
swapTokens(amount: 100, tokenAddress: 0x...)
```

Validators execute smart contract code using this `data`.



Step 5: Signing & Broadcasting

- 1 Assemble the transaction object.
- 2 Sign it with your **private key** (produces a signature).
- 3 Send the signed transaction to the network **mempool**.



Step 6: Validation (Proof-of-Stake)

Ethereum moved from **Proof-of-Work (PoW)** → **Proof-of-Stake (PoS)**.

PoS Key Points:

- ✓ No miners; only validators.
- ✓ Validators stake ETH as collateral.

- ✅ Validators propose and attest to blocks.
- ✅ More energy-efficient and secure.

When finalized:

- Sender balance decreases.
- Recipient balance increases.
- Smart contract states update.



Summary Table: Transaction Flow

Step	What Happens
✅ 1	Build transaction object
✅ 2	Add correct nonce
✅ 3	Specify gas & fees
✅ 4	(Optional) Include smart contract data
✅ 5	Sign transaction
✅ 6	Broadcast to mempool
✅ 7	Validator includes it in block
✅ 8	Block is finalized; state updated
