Getting started with blockchain wallet
(For Cypherock)

In blockchain system, unlike any centralized system, user has the responsibility to generate and protect their keys. In this system, a private key is used to send digital assets and public key is used to receive digital assets. It is recommended to use different private-public key pair for every transaction to maintain privacy. To make key management compatible, most of the wallets implement hierarchical key derivation. A hierarchically derived wallet generates keys in the following structure:

ENT-->Mnemonics-->BIP39 Seed-->Master Node-->Purpose Node-->Coin Node-->Account Node-->Change Node-->Address Index Node.

m/44’/1’/0’/0/i

Now lets deep dive on how each level in HD tree is derived and their significance.

ENT to Mnemonics

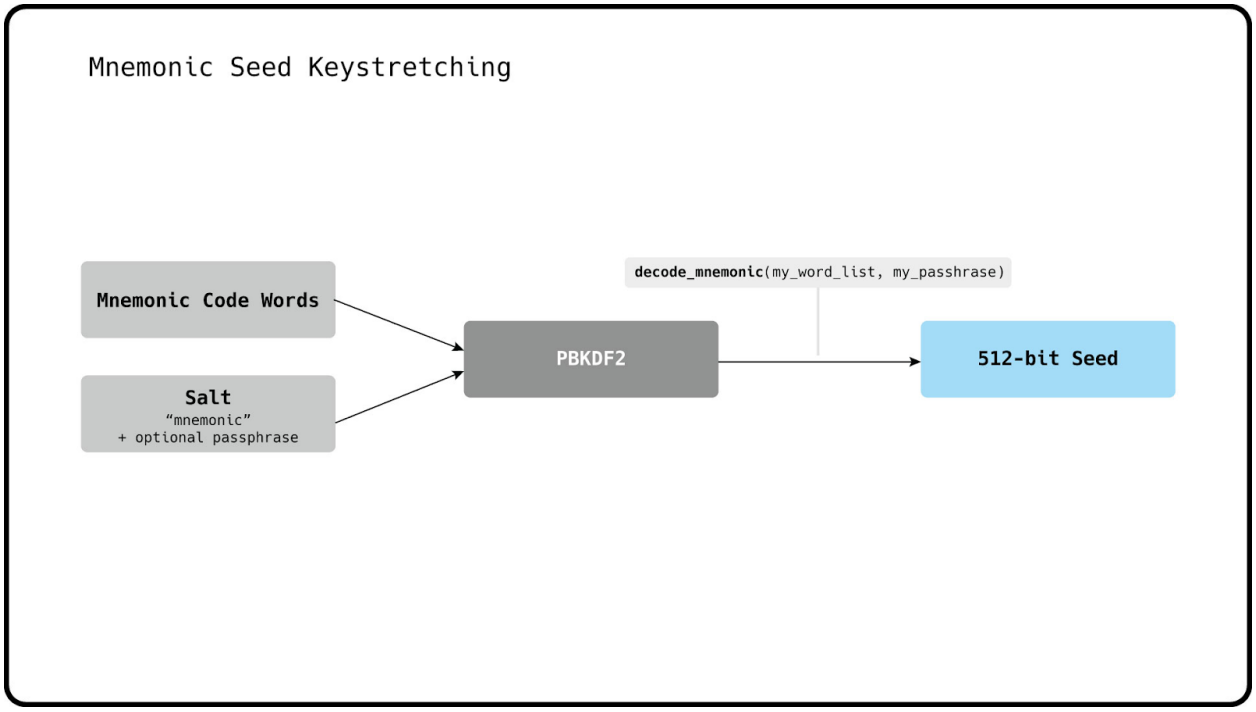
ENT stands for entropy bits and should be generated using a true random number generator. Its length may be chosen anywhere between 128 to 256 bits with increments of 32 bits.

ENT	**CS**	**ENT||CS**	**No. of words**
128	4	132	12
256	8	264	24

CS stands for checksum and represents first ENT/32 bits of sha256(ENT)). The CS bits are appended after ENT which is represented as ENT||CS. In ENT||CS each group of 11 bits represents a number between 0 to 2048 which represents a unique word in the BIP39 wordlist. The group of words represents mnemonics, also called as seed phrases is used to recover wallet in case user wants to regenerate keys on a different machine.

Mnemonics to BIP39 seed

To create a BIP39 seed from the mnemonics, we use the PBKDF2 function with a mnemonic sentence used as the password and the string "mnemonic" + passphrase used as the salt. The iteration count is set to 2048 and HMAC-SHA512 is used as the pseudo-random function. The length of the derived key is 512 bits

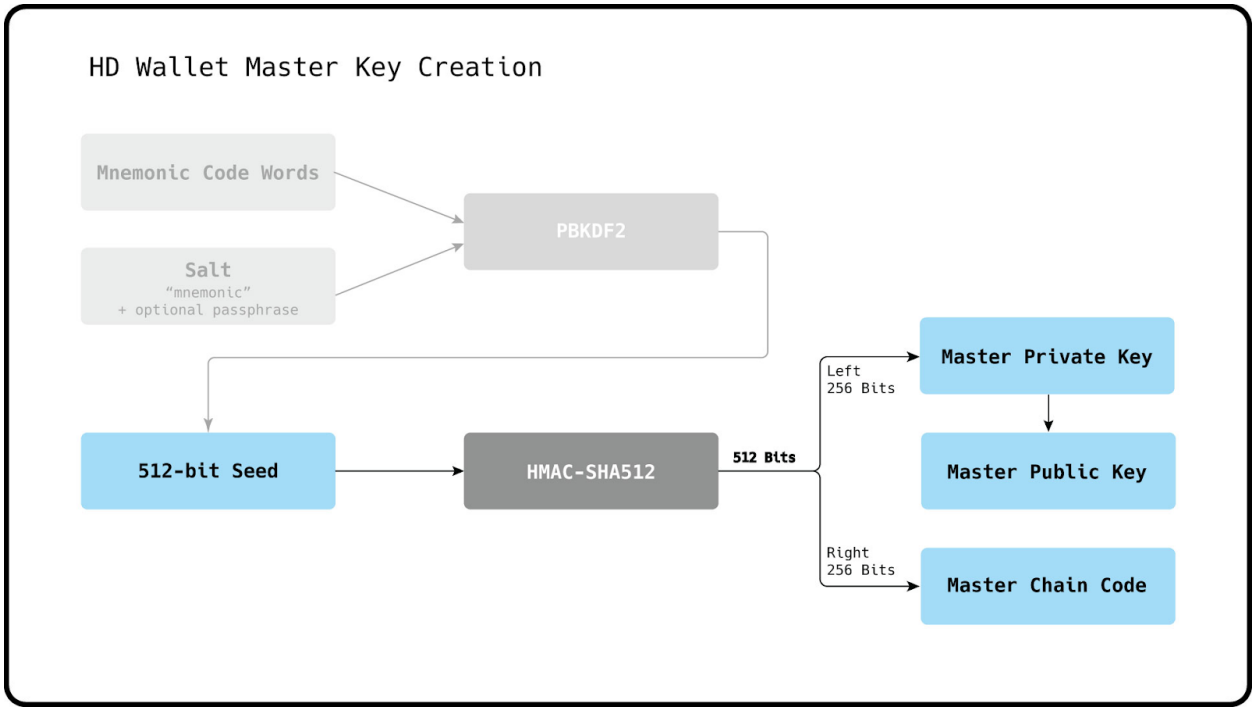


BIP39 seed to Master Node

A node consists of Public or Private key along with a parameter called “Chain Code”. Master Node acts as a root in the key derivation tree. The 512 bit HD root seed is hashed by the HMAC-SHA512 algorithm to create a 512 bit digest, which is split into two separate parts:

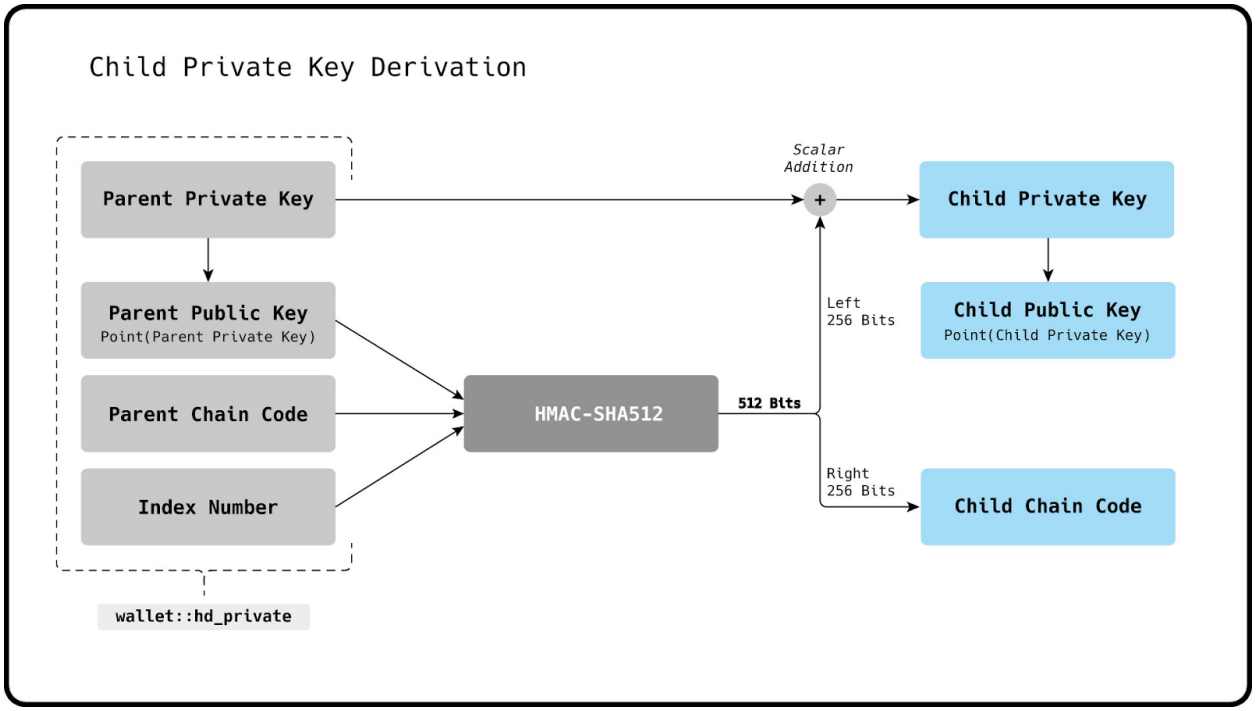
- Left 256 bits: Master Private key
- Right 256 bits: Master Chain Code

The master public key is simply derived from the master private key.



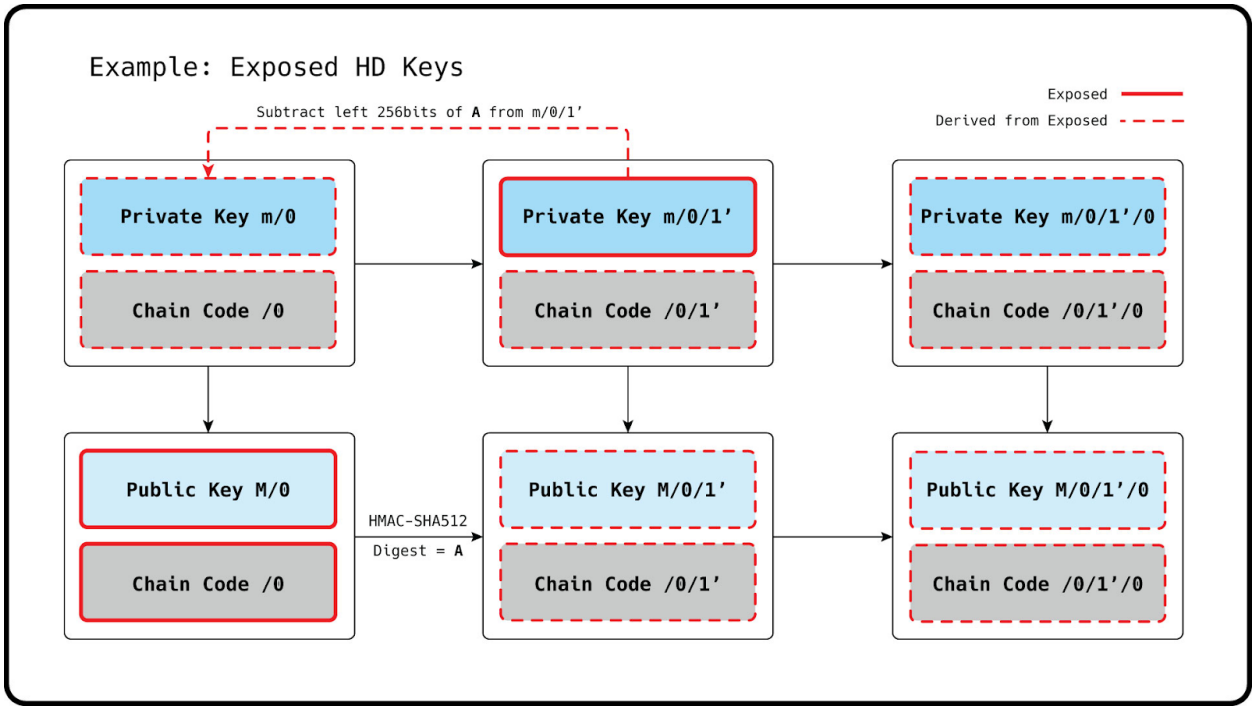
Before we dive further we need to know about hardened and non-hardened derivation

Non hardened Derivation

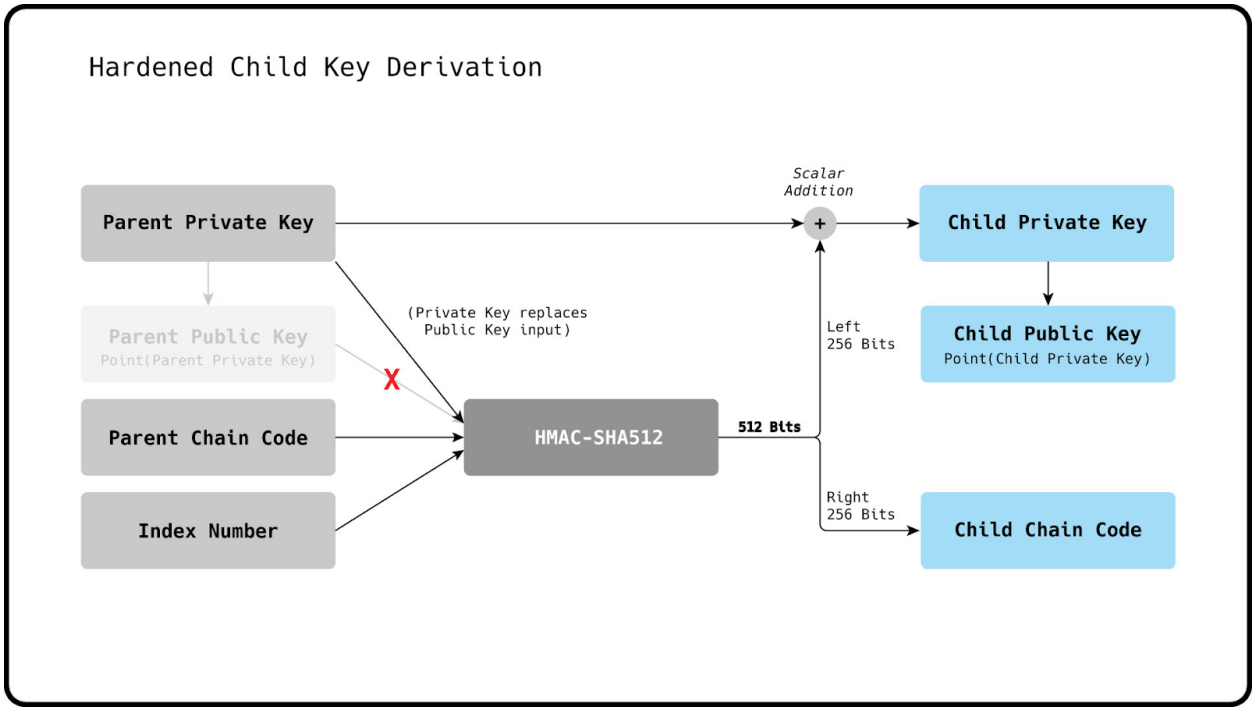


The index is incremented to generate additional child private key from the same parent private key. Indices between 0 and 2^31-1 (0x and 0x7FFFFFFF) are used for non hardened child derivation as described above.

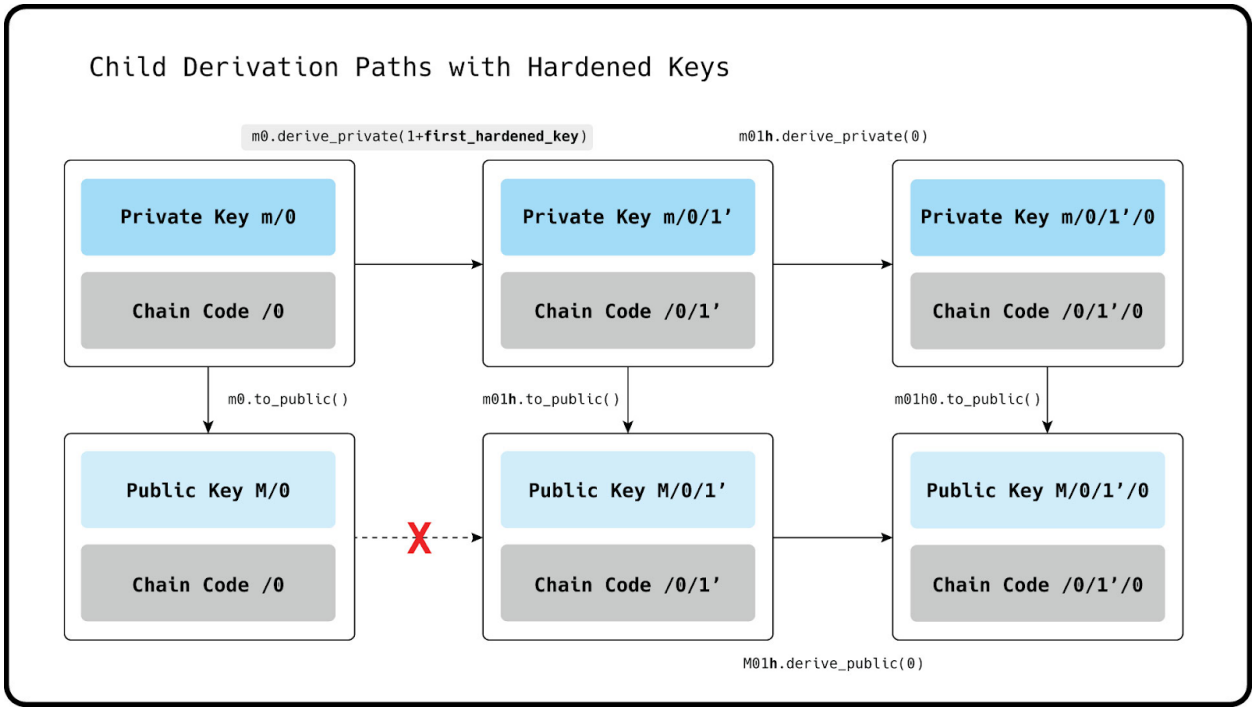
In the case that both the public extended key and a descendent child private key are exposed, it is possible for a malicious actor to derive both the private extended key as well as all descendent children.



Hardened Derivation



A hardened child private key can be derived using the parent private key instead of the parent public key as an input to the HMAC-SHA512 function. This breaks the derivation path between a hardened public key and its public key parent. Indices between 2^31 and 2^32-1 (0x80000000 and 0xFFFFFFFE) are used for hardened child derivation as described above



Path Representation

We define the following 5 levels in path:

`m / purpose' / coin_type' / account' / change / address_index`

The apostrophe in the path indicates that hardened derivation is used

Now lets get back to key derivation tree.

Master Node to Purpose Node

Purpose Node indicates that the subtree of this node is used according to this specification. Hardened derivation is used at this level.

Index	Hex	Transaction type
44	0x8000002C	P2PKH

Purpose Node to Coin Node

This level creates a separate subtree for every cryptocoin, avoiding reusing addresses across cryptocoins and improving privacy issues. Coin type is a constant, set for each cryptocoin. Hardened derivation is used at this level.

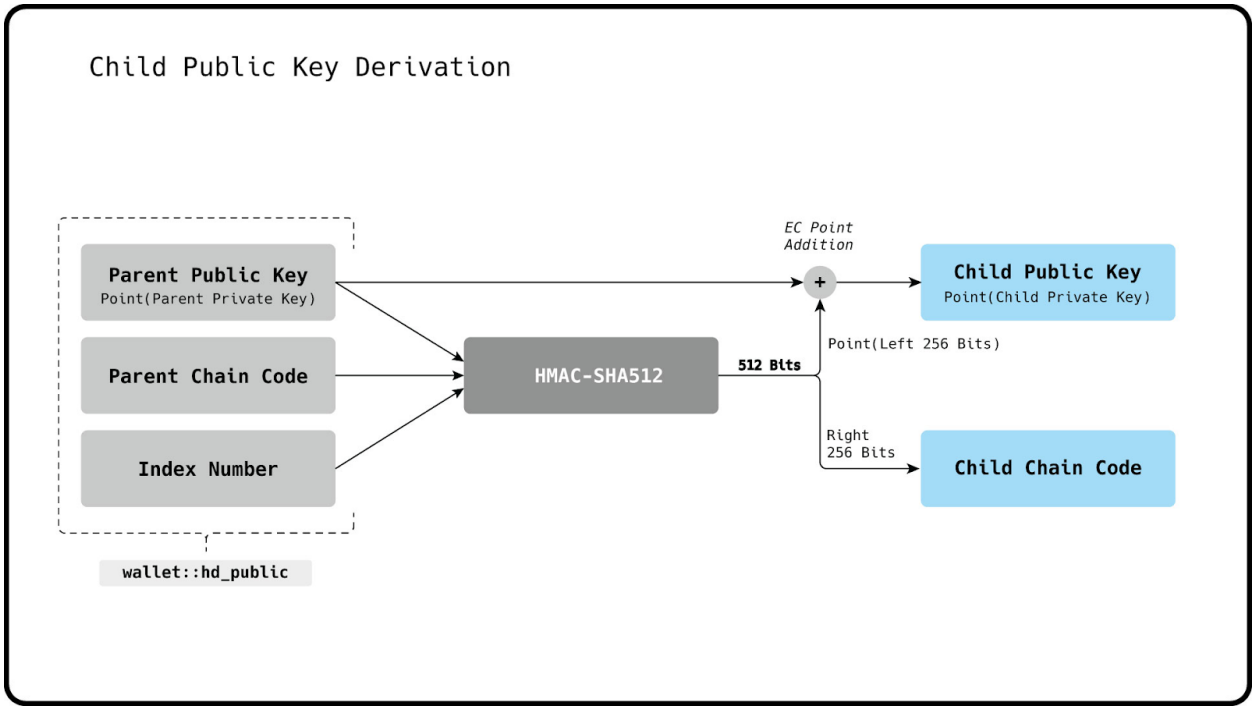
Index	Hex	Coin
0	0x80000000	Bitcoin Mainnet
1	0x80000001	Bitcoin Testnet

Coin Node to Account Node

Account node could be used to create a watching only wallets on different machine. Users can use these accounts to organize the funds in the same fashion as bank accounts; for donation purposes (where all addresses are considered public), for saving purposes, for common expenses, etc. Accounts are numbered from index 0 in sequentially increasing manner. Software should prevent a creation of an account if a previous account does not have a transaction history (meaning none of its addresses have been used before). Hardened derivation is used at this level.

Account Node to Change Node

Constant 0 is used for external chain and constant 1 for internal chain (also known as change addresses). External chain is used for addresses that are meant to be visible outside of the wallet (e.g. for receiving payments). Internal chain is used for addresses which are not meant to be visible outside of the wallet and is used for return transaction change. Public key derivation is used in watching wallet setup. Non hardened derivation is used in signing wallet setup.



Change Node to Address Index Node

Addresses are numbered from index 0 in a sequentially increasing manner. Public key derivation is used in watching wallet setup and non hardened derivation is used in signing wallet setup. When the mnemonics are imported from an external source the software should start to discover the account node in the following manner:

1. derive the first account's node (index = 0)
2. derive the external chain node of this account
3. scan addresses of the external chain; respect the gap limit (=20) described below
4. if no transactions are found on the external chain, stop discovery
5. if there are some transactions, increase the account index and go to step 1

Node Serialization Format

Parameter	**Length**
Version (mainnet: 0x0488B21E public, 0x0488ADE4 private; testnet: 0x043587CF public, 0x04358394 private)	4 bytes
Depth (0x00 for master nodes, 0x01 for level-1 derived keys, etc)	1 byte
Fingerprint (first 32 bits of hash160 of parent public key)	4 byte
Child Index	4 byte
Chain Code	32 byte
Private or Public Key	33 byte
Checksum	4 byte

First 78 bytes can be encoded like other Bitcoin data in Base58, by first adding 32 checksum bits (derived from the double SHA-256 checksum), and then converting to the Base58 representation. This results in a Base58-encoded string of up to 112 characters. Because of the choice of the version bytes, the Base58 representation will start with "xprv" or "xpub" on mainnet, "tprv" or "tpub" on testnet

To do : Now lets create an HD wallet.

Now we have our bitcoin testnet account extended public key and the first receive address we will start with wallet transactions.
In broader terms making transaction involves the following steps:

1. Create an unsigned transaction using previous transactions on the network
2. Signing the above unsigned transaction securely.

3. Create final transaction and broadcast it on the network

Since its a new wallet there should be no transaction history, we can also confirm that from blockexplorers like Blockcypher.com by typing any receive address on search bar. Now we will request a testnet faucet to send some bitcoins on our first receive address. This transaction will act as previous transaction as mentioned in step 1 above.

Bitcoin Unsigned Transaction Format

A simple 1 input - 2 output unsigned transaction contain following parameters:

Parameter	Description	Length in bytes
Network Version		4
Input Count		1
Input 1 Previous Transaction Hash		32
Input 1 Previous Output Index		4
Input 1 Script Public Key Length		1
Input 1 Script Public Key		25
Input 1 Sequence		4
Output Count		1
Output 1 value		8
Output 1 Script Public Key Length		1
Output 1 Script Public Key		25
Output 2 value		8
Output 2 Script Public Key Length		1
Output 2 Script Public Key		25
Locktime		4
SigHash		4

To create unsigned transaction using the previous transaction we would have to query the blockchain, to make that process simple, since our main goal is to secure the signing process, we will use Electrum to talk to blockchain network and create unsigned raw transaction for us. To do that we have to create a watching only wallet on electrum using the account extended public key.

To do: add steps to get decoded unsigned raw transaction from electrum

Electrum Unsigned Transaction Format

Parameter	Description	Length in bytes
Electrum specific		6
Network Version		4
Input Count		1
Input 1 Previous Transaction Hash		32
Input 1 Previous Output Index		4
Input 1 Script Length		1
Electrum Specific		4
Extended public key type		1
Input 1 Extended Public Key		78
Input 1 change node		2
Input 1 address index		2
Input 1 Sequence		4
Output Count		1
Output 1 value		8
Output 1 Script Public Key Length		1
Output 1 Script Public Key		25
Output 2 value		8
Output 2 Script Public Key		1

Length		
Output 2 Script Public Key		25
Locktime		4

Since the export option in Electrum is to implement cold signing functionality the unsigned transaction format here is different. Beside some electrum specifics it introduces extended public key in place of script public key. It also add path derivation suffix for change node and address index to indicate which private key and script public key has to be used in the signing process.

To do: Add following steps

1. Convert electrum unsigned transaction to standard unsigned transaction
2. Derive private key
3. Sign and create final transaction
4. Create signed.txn file
5. Broadcast signed.txn through electrum

Extra information

Integrating Trezor for Wallet:

> We are using the crypto folder of the repository https://github.com/trezor/trezor-firmware.

> Theory (Feel free to skip):

When we need to compile a single file, say `code.c`, we run `gcc code.c -o code.out`. If we have a large project spread out across multiple files, it is best to first make `.o` files of each file by running`gcc -c file.c`(creates `file.o`). The `-c` flag causes the compiler to skip the final linking step. One project can have at max `main` function. Let's say our main function is in `main.c`. To compile `main.c` which may use functions defined in `file.c1`, `file2.c`, ..., we need to tell the compiler where to find the functions(`main.c` only includes the `.h` files so it only knows that the functions exist, not where to find them) . To do that we provide `.o` files as arguments. So, we'll do `gcc file1.o file2.o main.c`. We can also say `gcc *.o main.c` so that each `.o` file in the current directory is included. Now if we change `file1.c`, we only need to recompile it and the main file and we can leave `file2.c` as it is. To automate this task of recompiling and providing the arguments to compiler, we use Makefile. Here's a resource to learn about them: http://nuclear.mutantstargoat.com/articles/make/

> How I set the things up: To integrate trezor-firmware's `crypto`, we keep a copy of it in our source folder. Parallel to it, we have a folder named `Wallet` where we keep our code. In theory we described how we need to provide `.o` files as arguments to the compiler if we want to use the functions in the corresponding `.c` files. Since there will be a lot of `.o` files in the `crypto`

folder when we `make` it, and we need all of them, we can archive them and create a `.a` file. Here's the resource from where I learned this:
https://www.howtogeek.com/427086/how-to-use-linuxs-ar-command-to-create-static-libraries/

We need only two commands:

`ar -crs crypto.a crypto/*.o` // creates a file named crypto.a which is like a zip file of all the `.o` files in crypto folder

`ar -t crypto.a` // to see what all files are there in an archive

Now you can just provide `crypto.a` as an argument to the compiler and it'll have access to all the functions defined in `crypto` folder.

> **All you need to know:** Just run `make` in the source directory. It will adjust everything accordingly. If something's breaking, first run `make clean` and then run `make`, it should work. For the specifics, you'd need to read and understand the *Makefile*s in each of the folders.