

Bitcoin Wallet Testnet — Wallet Creation & Transaction Lab

By: Fábio Vieira

Introduction

This lab aims to provide a practical, hands-on understanding of how a cryptocurrency wallet works — from wallet creation to transferring coins. The focus is Bitcoin (testnet) and the goal is to gain technical knowledge in Python (language used for the exercises) and Bitcoin standards such as BIP-39, BIP-32/BIP-84, and WIF/BIP-58 handling.

Objective

- Create a deterministic wallet from a BIP-39 mnemonic (12 words).
- Derive a native SegWit address (BIP-84) on Bitcoin testnet.
- Receive testnet BTC from a faucet and verify the receipt in a public explorer.
- Build, sign, and broadcast a real testnet transaction using the derived private key, without exposing the private key in cleartext.
- Document evidence (screenshots, TXID) and explain security best practices.

Bitcoin Wallet Testnet — Wallet Creation & Transaction Lab.....	1
Introduction.....	1
Objective.....	1
Tools & Environment	3
Commands to prepare environment:	3
High-level concepts	4
Step 1 Wallet Creation	5
What this script does:.....	5
Confirmation	6
Step 2 Sending.....	8
How the script sends Bitcoin:.....	10
Security Notes	12
Mnemonic safety	12
Private key handling	12
Secure execution	12
Network and API use	12
Testnet usage	12
Recovery principle	13
Conclusion:.....	13

Tools & Environment

- OS: Kali Linux (virtual machine)
- Python 3 (inside a virtual environment)
- Libraries used:
 - bip-utils (BIP-39/BIP-84 derivation)
 - bit (transaction building & signing)
 - requests (Blockstream API calls)
 - base58 (WIF encoding)
- Explorer used: <https://blockstream.info/testnet/>
- Optional GUI wallet for demonstration: Electrum (testnet)

Commands to prepare environment:

```
python3 -m venv venv
```

```
source venv/bin/activate
```

```
pip install -U pip
```

```
pip install bip-utils bit base58 requests
```

High-level concepts

- Mnemonic / BIP-39: human readable seed phrase. A 12-word mnemonic represents 128 bits of entropy and can deterministically recreate the master private key.
- Derivation / BIP-32 / BIP-84: hierarchical deterministic derivation that derives child keys and addresses from the master key. BIP-84 defines the derivation path for native SegWit (bech32) outputs (m/84'/{coin}'/account'/change/index).
- WIF / base58: Wallet Import Format encodes the private key in base58check; for testnet the prefix differs (0xEF).
- SegWit addresses: native SegWit addresses (bech32) start with tb1 on testnet and use P2WPKH.
- Signing: transactions are signed with the private key — the signature authorizes the spending of UTXOs without revealing the private key.

Step 1 Wallet Creation

After verifying that we have the necessary libraries and are in a secure environment, we create the script **wallet_testnet.py**:

```
#import
from bip_utils import(Bip39MnemonicGenerator, Bip39SeedGenerator, Bip84,Bip84Coins,Bip44Changes)

#generate seed
mnemonic = Bip39MnemonicGenerator().FromWordsNumber(12)

print("SEED (mnemonic) 12 words:\n", mnemonic)
#generate seed bytes
seed = Bip39SeedGenerator(mnemonic).Generate()

#derive BIP84 (Native SegWit) wallet on BITCOIN TESTNET
#path: m/84'/1'/0'/0/0 (1' = testnet)
ctx = Bip84.FromSeed(seed, Bip84Coins.BITCOIN_TESTNET) \
    .Account(0).Change(Bip44Changes.CHAIN_TEXT).AddressIndex(0)

address = ctx.PublicKey().ToAddress()
print("\nADDRESS TESTNET (tb1..):\n", address)
```

What this script does:

- Imports bip_utils helpers for BIP-39 (mnemonic + seed) and BIP-84 (SegWit derivation).
- Generates a 12-word mnemonic (BIP-39). This is the human-readable backup for the wallet.
- Creates the seed bytes from the mnemonic.
- Derives a native SegWit (BIP-84) testnet address using the path m/84'/1'/0'/0/0:
- 84' → BIP-84 (bech32 / native SegWit)
- 1' → testnet (mainnet would be 0')
- 0' → account 0
- 0 → external (receiving) chain
- 0 → first address index

Run the script **wallet_testnet.py**

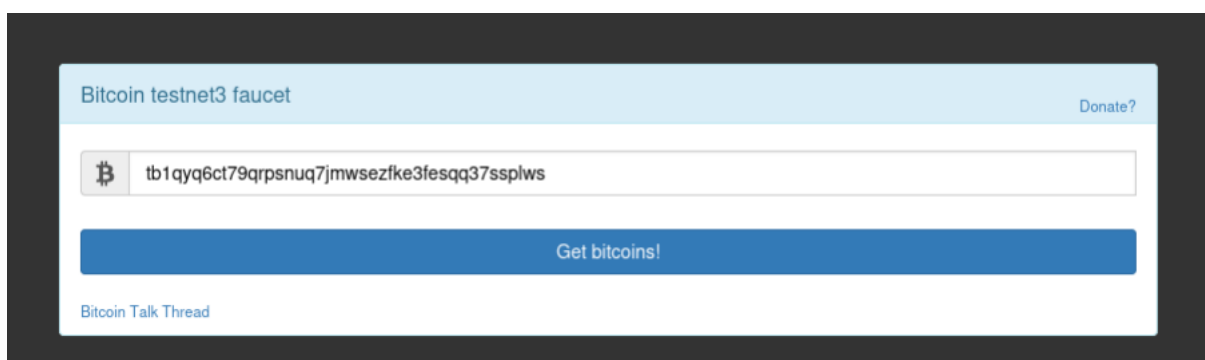
```
(venv)-(joe@vbox)-[~/Labs/bitcoin_wallet_lab]
$ python wallet_testnet.py
SEED (mnemonic) 12 words:
search author crush casual news measure later client because ivory panda dove

ADDRESS TESTNET (tb1..):
tb1qq6ct79qrpsnuq7jmwsezfke3fesqq37ssplws
```

Now with our seed and address we can receive BTC, this is a lab for education in a real environment remember that you should never share the seed as it is with it that you control the master private key and the private keys that follow.

Confirmation

I used the website <https://coinfaucet.eu/> to send tBTC to the wallet:



And used the <https://blockstream.info/testnet/> to confirm the transaction.

1 Transaction

b9773f972e1c65b5f3c662f6724966677dae3ed05b9bfc5b652693b0f0e3a9d3

Details 

#0 aea121a6f5587aaf0345b03222c7307f3b22ec10cc8 19.33208729 tBTC
01cba91733a02c435311:1



#0 tb1qyq6ct79qrpsnuq7jmwsezfke3fesqq37ssplws 0.00154581 tBTC

#1 tb1q2pzg7hvhfk4gvqqnfujhyjeu89u24e0009nec 19.33054007 tBTC

Step 2 Sending

After installing "bit requests base58" i create a simple and practical tBTC sending method that securely requests the seed (it doesn't appear on the screen), validates it, derives the source address, and then requests the destination/amount. I used GetPass for the seed and BIP-39 validation to avoid obvious errors.

```
from bip_utils import (
    Bip39SeedGenerator, Bip84, Bip84Coins, Bip44Changes,
    Bip39MnemonicValidator, Bip39Languages
)
from bit import PrivateKeyTestnet
from bit.network.meta import Unspent
from bit.network import NetworkAPI
from bit.network.services import BlockstreamAPI
import hashlib, base58, sys, getpass, requests

BLOCKSTREAM_API = "https://blockstream.info/testnet/api"

def mnemonic_to_ctx(mnemonic: str):
    seed_bytes = Bip39SeedGenerator(mnemonic).Generate()
    return (Bip84.FromSeed(seed_bytes, Bip84Coins.BITCOIN_TESTNET)
        .Purpose().Coin().Account(0)
        .Change(Bip44Changes.CHAIN_EXT).AddressIndex(0))

def bech32_script_pubkey(pubkey_hash_hex: str) -> str:
    return "0014" + pubkey_hash_hex

def get_utxos_blockstream(bech32_addr: str):
    r = requests.get(f"{BLOCKSTREAM_API}/address/{bech32_addr}/utxo", timeout=15)
    r.raise_for_status()
    return r.json()

def pubkey_hash_from_ctx(ctx) -> str:
    pubkey_bytes = ctx.PublicKey().RawCompressed().ToBytes()
    ripemd160 = hashlib.new('ripemd160')
    ripemd160.update(hashlib.sha256(pubkey_bytes).digest())
    return ripemd160.hexdigest()

def read_amount(amt_str: str) -> int:
    s = amt_str.strip()
    if "." in s:
        return int(round(float(s) * 100_000_000))
    return int(s)

def main():
    print("\n≡ Testnet sender (native SegWit tb1_, Blockstream UTXOs) ≡\n")
    mnemonic = getpass.getpass("Enter TESTNET mnemonic (12/24 words): ").strip()
    mnemonic = " ".join(mnemonic.split())

    try:
        Bip39MnemonicValidator(Bip39Languages.ENGLISH).Validate(mnemonic)
    except Exception as e:
        print("Invalid mnemonic:", e); sys.exit(1)

    ctx = mnemonic_to_ctx(mnemonic)
    bech32_addr = ctx.PublicKey().ToAddress()

    priv_bytes = ctx.PrivateKey().Raw().ToBytes()
    payload = b'\xEF' + priv_bytes + b'\x01'
    checksum = hashlib.sha256(hashlib.sha256(payload).digest()).digest()[:4]
    wif = base58.b58encode(payload + checksum).decode()

    key = PrivateKeyTestnet(wif)
    key.segwit = True

    print("Bech32 (funded) address:", bech32_addr)

    try:
        utxos_json = get_utxos_blockstream(bech32_addr)
    except Exception as e:
        print("Failed to fetch UTXOs from Blockstream:", e); sys.exit(1)
```



```

pkh_hex = pubkey_hash_from_ctx(ctx)
script_hex = bech32_script_pubkey(pkh_hex)

unspents = []
for u in utxos_json:
    unspents.append(Unspent(
        amount=int(u["value"]),
        confirmations=1 if u.get("status", {}).get("confirmed") else 0,
        script=script_hex,
        txid=u["txid"],
        txindex=int(u["vout"]),
    ))

balance = sum(u.amount for u in unspents)
print("Current balance (satoshis):", balance)

if balance <= 0:
    print("No UTXOs found (wait for 1 confirmation or check address).")
    sys.exit(1)

dest = input("\nDestination address (testnet) → ").strip()
amt_in = input("Amount (e.g., '1000' sats or '0.00001' BTC) → ").strip()
amount_sat = read_amount(amt_in)

MIN_FEE_EST = 250
if amount_sat + MIN_FEE_EST > balance:
    print("Insufficient funds for amount + fee."); sys.exit(1)

print("\nSummary")
print("  From:", bech32_addr)
print("  To:  ", dest)
print("  Amt: ", amount_sat, "sats")

if input("Confirm send? (y/N) → ").strip().lower() != "y":
    print("Aborted."); sys.exit(0)

try:
    tx_hex = key.create_transaction(
        outputs=[(dest, amount_sat, "satoshi")],
        unspents=unspents
    )
    NetworkAPI.BROADCAST_SERVICES_TESTNET = [BlockstreamAPI.broadcast_tx_testnet]
    txid = BlockstreamAPI.broadcast_tx_testnet(tx_hex)
    print("\n✅ Broadcast OK")
    print("TXID:", txid)
    print("Explorer:", f"https://blockstream.info/testnet/tx/{txid}")
except Exception as e:
    print("\n❌ Send/broadcast error:", e)
    sys.exit(1)

if __name__ == "__main__":
    main()

```

How the script sends Bitcoin:

1. Secure mnemonic input

- a. The script prompts the user for the BIP-39 mnemonic using `getpass()` so the seed is not shown on screen. It normalizes spaces and validates the mnemonic with `bip-utils`.

2. Derive BIP-84 context (native SegWit)

- a. Using `bip-utils` it derives the BIP-84 key context at the path `m/84'/1'/0'/0/0` (native SegWit, testnet).
- b. From that context it obtains the compressed public key and the bech32 address (`tb1...`).

3. Build TESTNET WIF (private key format)

- a. The script converts the derived private key bytes into the testnet WIF format (prefix `0xEF`, compressed flag, checksum, `base58check`).
- b. The WIF is used only in memory to construct a `bit.PrivateKeyTestnet` object for signing.

4. Query Blockstream for UTXOs

- a. It calls Blockstream's testnet API `/address/<tb1>/utxo` to fetch the unspent outputs for the derived bech32 address.
- b. Rationale: some libraries do not handle bech32 UTXO lookup properly, so the script fetches the authoritative UTXOs directly.

5. Convert Blockstream UTXOs to `bit.Unspent` objects

- a. For each Blockstream UTXO the script builds a `Unspent` object for `bit`, setting:
 - i. `amount` (value in satoshis),
 - ii. `txid` and `txindex` (vout),
 - iii. `script` = P2WPKH scriptPubKey in hex (`0014` + 20-byte `pubKeyHash`).
- b. The `pubKeyHash` is computed from the compressed public key (SHA-256 then RIPEMD-160).

6. Calculate balance and check funds

- a. The script sums the Unspent amounts to get the available balance (satoshis).
- b. It checks that requested amount + a small fee margin is \leq balance.

7. Create unsigned transaction and sign it locally

- a. It calls `key.create_transaction(outputs=[(dest, amount, 'satoshi')], unspents=unspents)` where `key` is the `PrivateKeyTestnet(wif)` object.
- b. `bit` uses the provided unspents and the private key in memory to create and sign the raw transaction bytes (signing occurs locally — the private key is never sent over the network).

8. Broadcast using Blockstream API

- a. The script sends the signed raw transaction hex to Blockstream's broadcast endpoint.
- b. If broadcast succeeds the script prints the `txid`.

9. Verification

- a. The user can verify the transaction and confirmations on Blockstream explorer:

<https://blockstream.info/testnet/tx/<txid>>.

Security Notes

Mnemonic safety

- The 12-word BIP-39 seed gives full control of the wallet.
- Never share or store it in plaintext or screenshots.
- In this lab, only Bitcoin testnet coins were used (no real value).
- For real use, store the seed offline or in a hardware wallet.

Private key handling

- The private key is generated only in memory and never saved to disk.
- It's used only to sign the transaction; the key itself is never sent online.
- The WIF format was used only internally to initialize the signing library.

Secure execution

- The script uses `getpass()` to hide the mnemonic input.
- Everything runs inside a Python virtual environment (venv) for isolation.

Network and API use

- All communication with the Blockstream API happens over HTTPS.

Testnet usage

- All operations were performed on Bitcoin testnet.

- On mainnet, stronger protections like hardware wallets or multisig are required.

Recovery principle

- The same mnemonic can recreate the entire wallet (BIP-39 + BIP-84).
- If it's lost, funds are lost; if exposed, anyone can steal them.

Conclusion:

This lab demonstrated how a Bitcoin wallet works from creation to transaction.

Using Python and public APIs, we generated a BIP-39 mnemonic, derived a SegWit address, received testnet coins, and successfully sent a transaction.

It provided practical understanding of wallet structure, key derivation, and secure transaction handling on the Bitcoin testnet.