1. Multisig Tutorial

Currently, it is possible to create a multisig wallet using Bitcoin Core only.

Although there is already a brief explanation about the multisig in the Descriptors documentation, this tutorial proposes to use the signet (instead of regtest), bringing the reader closer to a real environment and explaining some functions in more detail.

This tutorial uses jq JSON processor to process the results from RPC and stores the relevant values in bash variables. This makes the tutorial reproducible and easier to follow step by step.

Before starting this tutorial, start the bitcoin node on the signet network.

```
./src/bitcoind -signet -daemon
```

This tutorial also uses the default WPKH derivation path to get the xpubs and does not conform to BIP 45 or BIP 87.

At the time of writing, there is no way to extract a specific path from wallets in Bitcoin Core. For this, an external signer/xpub can be used.

PR #22341, which is still under development, introduces a new wallet RPC `getxpub`. It takes a BIP32 path as an argument and returns the xpub, along with the master key fingerprint.

1.1 Basic Multisig Workflow

1.1 Create the Descriptor Wallets

For a 2-of-3 multisig, create 3 descriptor wallets. It is important that they are of the descriptor type in order to retrieve the wallet descriptors. These wallets contain HD seed and private keys, which will be used to sign the PSBTs and derive the xpub.

These three wallets should not be used directly for privacy reasons (public key reuse). They should only be used to sign transactions for the (watch-only) multisig wallet.

```
for ((n=1;n<=3;n++))
do
 ./src/bitcoin-cli -signet createwallet "participant_${n}"
done
```

Extract the xpub of each wallet. To do this, the `listdescriptors` RPC is used. By default, Bitcoin Core single-sig wallets are created using path `m/44'/1'/0'` for PKH, `m/84'/1'/0'` for WPKH, `m/49'/1'/0'` for P2WPKH-nested-in-P2SH and `m/86'/1'/0'` for P2TR based accounts. Each of them uses the chain 0 for external addresses and chain 1 for internal ones, as shown in the example below.

```
wpkh([1004658e/84'/1'/0']tpubDCBEcmVKbfC9KfdydyLbJ2gfNL88grZu1XcWSW9ytTM6fitvaRmVyr8Ddf7SjZ2ZfMx9RicjYAXhuh3fmLiVLPodPEqnQQURUfrBKi


wpkh([1004658e/84'/1'/0']tpubDCBEcmVKbfC9KfdydyLbJ2gfNL88grZu1XcWSW9ytTM6fitvaRmVyr8Ddf7SjZ2ZfMx9RicjYAXhuh3fmLiVLPodPEqnQQURUfrBKi
```

The suffix (after #) is the checksum. Descriptors can optionally be suffixed with a checksum to protect against typos or copy-paste errors. All RPCs in Bitcoin Core will include the checksum in their output.

```
declare -A xpubs

for ((n=1;n<=3;n++))
do
 xpubs["internal_xpub_${n}"]=$(./src/bitcoin-cli -signet -rpcwallet="participant_${n}" listdescriptors | jq '.descriptors | [.[]
| select(.desc | startswith("wpkh") and contains("/1/*"))][0] | .desc' | grep -Po '(?<=\().*(?=\))')

 xpubs["external_xpub_${n}"]=$(./src/bitcoin-cli -signet -rpcwallet="participant_${n}" listdescriptors | jq '.descriptors | [.[]
| select(.desc | startswith("wpkh") and contains("/0/*") )][0] | .desc' | grep -Po '(?<=\().*(?=\))')
done
```

`jq` is used to extract the xpub from the `wpkh` descriptor.

The following command can be used to verify if the xpub was generated correctly.

```
for x in "${!xpubs[@]}"; do printf "[%s]=%s\n" "$x" "${xpubs[$x]}" ; done
```

As previously mentioned, this step extracts the `m/84'/1'/0'` account instead of the path defined in BIP 45 or BIP 87, since there is no way to extract a specific path in Bitcoin Core at the time of writing.

1.2 Define the Multisig Descriptors

Define the external and internal multisig descriptors, add the checksum and then, join both in a JSON array.

```
external_desc="wsh(sortedmulti(2,${xpubs["external_xpub_1"]},${xpubs["external_xpub_2"]},${xpubs["external_xpub_3"]}))"
internal_desc="wsh(sortedmulti(2,${xpubs["internal_xpub_1"]},${xpubs["internal_xpub_2"]},${xpubs["internal_xpub_3"]}))"

external_desc_sum=$(./src/bitcoin-cli -signet getdescriptorinfo $external_desc | jq '.descriptor')
internal_desc_sum=$(./src/bitcoin-cli -signet getdescriptorinfo $internal_desc | jq '.descriptor')
```

```
multisig_ext_desc="{\"desc\": $external_desc_sum, \"active\": true, \"internal\": false, \"timestamp\": \"now\"}"
multisig_int_desc="{\"desc\": $internal_desc_sum, \"active\": true, \"internal\": true, \"timestamp\": \"now\"}"

multisig_desc="[$multisig_ext_desc, $multisig_int_desc]"
```

`external_desc` and `internal_desc` specify the output type (`wsh` , in this case) and the xpubs involved. They also use BIP 67 (`sortedmulti`), so the wallet can be recreated without worrying about the order of xpubs. Conceptually, descriptors describe a list of scriptPubKey (along with information for spending from it) [source].

Note that at least two descriptors are usually used, one for internal derivation paths and external ones. There are discussions about eliminating this redundancy, as can been seen in the issue #17190.

After creating the descriptors, it is necessary to add the checksum, which is required by the `importdescriptors` RPC.

The checksum for a descriptor without one can be computed using the `getdescriptorinfo` RPC. The response has the `descriptor` field, which is the descriptor with the checksum added.

There are other fields that can be added to the descriptors:

- `active` : Sets the descriptor to be the active one for the corresponding output type (`wsh` , in this case).
- `internal` : Indicates whether matching outputs should be treated as something other than incoming payments (e.g. change).
- `timestamp` : Sets the time from which to start rescanning the blockchain for the descriptor, in UNIX epoch time.

Documentation for these and other parameters can be found by typing `./src/bitcoin-cli help importdescriptors` .

`multisig_desc` concatenates external and internal descriptors in a JSON array and then it will be used to create the multisig wallet.

1.3 Create the Multisig Wallet

To create the multisig wallet, first create an empty one (no keys, HD seed and private keys disabled).

Then import the descriptors created in the previous step using the `importdescriptors` RPC.

After that, `getwalletinfo` can be used to check if the wallet was created successfully.

```
./src/bitcoin-cli -signet -named createwallet wallet_name="multisig_wallet_01" disable_private_keys=true blank=true

./src/bitcoin-cli  -signet -rpcwallet="multisig_wallet_01" importdescriptors "$multisig_desc"

./src/bitcoin-cli  -signet -rpcwallet="multisig_wallet_01" getwalletinfo
```

Once the wallets have already been created and this tutorial needs to be repeated or resumed, it is not necessary to recreate them, just load them with the command below:

```
for ((n=1;n<=3;n++)); do ./src/bitcoin-cli -signet loadwallet "participant_${n}"; done
```

1.4 Fund the wallet

The wallet can receive signet coins by generating a new address and passing it as parameters to `getcoins.py` script.

This script will print a captcha in dot-matrix to the terminal, using unicode Braille characters. After solving the captcha, the coins will be sent directly to the address or wallet (according to the parameters).

The url used by the script can also be accessed directly. At time of writing, the url is `https://signetfaucet.com` .

Coins received by the wallet must have at least 1 confirmation before they can be spent. It is necessary to wait for a new block to be mined before continuing.

```
receiving_address=$(./src/bitcoin-cli -signet -rpcwallet="multisig_wallet_01" getnewaddress)

./contrib/signet/getcoins.py -c ./src/bitcoin-cli -a $receiving_address
```

To copy the receiving address onto the clipboard, use the following command. This can be useful when getting coins via the signet faucet mentioned above.

```
echo -n "$receiving_address" | xclip -sel clip
```

The `getbalances` RPC may be used to check the balance. Coins with `trusted` status can be spent.

```
./src/bitcoin-cli -signet -rpcwallet="multisig_wallet_01" getbalances
```

1.5 Create a PSBT

Unlike singlesig wallets, multisig wallets cannot create and sign transactions directly because they require the signatures of the co-signers. Instead they create a Partially Signed Bitcoin Transaction (PSBT).

PSBT is a data format that allows wallets and other tools to exchange information about a Bitcoin transaction and the signatures necessary to complete it. [source]

The current PSBT version (v0) is defined in BIP 174.

For simplicity, the destination address is taken from the `participant_1` wallet in the code above, but it can be any valid bitcoin address.

The `walletcreatefundedpsbt` RPC is used to create and fund a transaction in the PSBT format. It is the first step in creating the PSBT.

```
balance=$(./src/bitcoin-cli -signet -rpcwallet="multisig_wallet_01" getbalance)

amount=$(echo "$balance * 0.8" | bc -l | sed -e 's/^\./0./' -e 's/^-\./-0./')

destination_addr=$(./src/bitcoin-cli -signet -rpcwallet="participant_1" getnewaddress)

funded_psbt=$(./src/bitcoin-cli -signet -named -rpcwallet="multisig_wallet_01" walletcreatefundedpsbt outputs="
{\"$destination_addr\": $amount}" | jq -r '.psbt')
```

There is also the `createpsbt` RPC, which serves the same purpose, but it has no access to the wallet or to the UTXO set. It is functionally the same as `createrawtransaction` and just drops the raw transaction into an otherwise blank PSBT. [source] In most cases, `walletcreatefundedpsbt` solves the problem.

The `send` RPC can also return a PSBT if more signatures are needed to sign the transaction.

1.6 Decode or Analyze the PSBT

Optionally, the PSBT can be decoded to a JSON format using `decodepsbt` RPC.

The `analyzepsbt` RPC analyzes and provides information about the current status of a PSBT and its inputs, e.g. missing signatures.

```
./src/bitcoin-cli -signet decodepsbt $funded_psbt

./src/bitcoin-cli -signet analyzepsbt $funded_psbt
```

1.7 Update the PSBT

In the code above, two PSBTs are created. One signed by `participant_1` wallet and other, by the `participant_2` wallet.

The `walletprocesspsbt` is used by the wallet to sign a PSBT.

```
psbt_1=$(./src/bitcoin-cli -signet -rpcwallet="participant_1" walletprocesspsbt $funded_psbt | jq '.psbt')

psbt_2=$(./src/bitcoin-cli -signet -rpcwallet="participant_2" walletprocesspsbt $funded_psbt | jq '.psbt')
```

1.8 Combine the PSBT

The PSBT, if signed separately by the co-signers, must be combined into one transaction before being finalized. This is done by `combinepsbt` RPC.

```
combined_psbt=$(./src/bitcoin-cli -signet combinepsbt "[$psbt_1, $psbt_2]")
```

There is an RPC called `joinpsbts`, but it has a different purpose than `combinepsbt`. `joinpsbts` joins the inputs from multiple distinct PSBTs into one PSBT.

In the example above, the PSBTs are the same, but signed by different participants. If the user tries to merge them using `joinpsbts`, the error `Input txid:pos exists in multiple PSBTs` is returned. To be able to merge different PSBTs into one, they must have different inputs and outputs.

1.9 Finalize and Broadcast the PSBT

The `finalizepsbt` RPC is used to produce a network serialized transaction which can be broadcast with `sendrawtransaction`.

It checks that all inputs have complete scriptSigs and scriptWitnesses and, if so, encodes them into network serialized transactions.

```
finalized_psbt_hex=$(./src/bitcoin-cli -signet finalizepsbt $combined_psbt | jq -r '.hex')

./src/bitcoin-cli -signet sendrawtransaction $finalized_psbt_hex
```

1.10 Alternative Workflow (PSBT sequential signatures)

Instead of each wallet signing the original PSBT and combining them later, the wallets can also sign the PSBTs sequentially. This is less scalable than the previously presented parallel workflow, but it works.

After that, the rest of the process is the same: the PSBT is finalized and transmitted to the network.

```
psbt_1=$(./src/bitcoin-cli -signet -rpcwallet="participant_1" walletprocesspsbt $funded_psbt | jq -r '.psbt')

psbt_2=$(./src/bitcoin-cli -signet -rpcwallet="participant_2" walletprocesspsbt $psbt_1 | jq -r '.psbt')

finalized_psbt_hex=$(./src/bitcoin-cli -signet finalizepsbt $psbt_2 | jq -r '.hex')

./src/bitcoin-cli -signet sendrawtransaction $finalized_psbt_hex
```