Support for signing transactions outside of Bitcoin Core

Bitcoin Core can be launched with `-signer=<cmd>` where `<cmd>` is an external tool which can sign transactions and perform other functions. For example, it can be used to communicate with a hardware wallet.

Example usage

The following example is based on the HWI tool. Version 2.0 or newer is required. Although this tool is hosted under the Bitcoin Core GitHub organization and maintained by Bitcoin Core developers, it should be used with caution. It is considered experimental and has far less review than Bitcoin Core itself. Be particularly careful when running tools such as these on a computer with private keys on it.

When using a hardware wallet, consult the manufacturer website for (alternative) software they recommend. As long as their software conforms to the standard below, it should be able to work with Bitcoin Core.

Start Bitcoin Core:

```
$ bitcoind -signer=../HWI/hwi.py
```

Device setup

Follow the hardware manufacturers instructions for the initial device setup, as well as their instructions for creating a backup. Alternatively, for some devices, you can use the `setup`, `restore` and `backup` commands provided by HWI.

Create wallet and import keys

Get a list of signing devices / services:

```
$ bitcoin-cli enumeratesigners
{
  "signers": [
    {
      "fingerprint": "c8df832a"
    }
]
```

The master key fingerprint is used to identify a device.

Create a wallet, this automatically imports the public keys:

```
$ bitcoin-cli createwallet "hww" true true "" true true true
```

Verify an address

Display an address on the device:

```
$ bitcoin-cli -rpcwallet=<wallet> getnewaddress
$ bitcoin-cli -rpcwallet=<wallet> walletdisplayaddress <address>
```

Replace `<address>` with the result of `getnewaddress`.

Spending

Under the hood this uses a Partially Signed Bitcoin Transaction.

```
$ bitcoin-cli -rpcwallet=<wallet> sendtoaddress <address> <amount>
```

This prompts your hardware wallet to sign, and fail if it's not connected. If successful it automatically broadcasts the transaction.

```
{"complete": true, "txid": <txid>}
```

Signer API

In order to be compatible with Bitcoin Core any signer command should conform to the specification below. This specification is subject to change. Ideally a BIP should propose a standard so that other wallets can also make use of it.

Prerequisite knowledge:

- Output Descriptors
- Partially Signed Bitcoin Transaction (PSBT)

`enumerate` (required)

Usage:

```
$ <cmd> enumerate
[
    {
        "fingerprint": "00000000"
    }
]
```

The command MUST return an (empty) array with at least a `fingerprint` field.

A future extension could add an optional return field with device capabilities. Perhaps a descriptor with wildcards. For example: `["pkh("44'/0'/$'/{0,1}/*"), sh(wpkh("49'/0'/$'/{0,1}/*")), wpkh("84'/0'/$'/{0,1}/*")]`. This would indicate the device supports legacy, wrapped SegWit and native SegWit. In addition it restricts the derivation paths that can used for those, to maintain compatibility with other wallet software. It also indicates the device, or the driver, doesn't support multisig.

A future extension could add an optional return field `reachable`, in case `<cmd>` knows a signer exists but can't currently reach it.

`signtransaction` (required)

Usage:

```
$ <cmd> --fingerprint=<fingerprint> (--testnet) signtransaction <psbt>
base64_encode_signed_psbt
```

The command returns a psbt with any signatures.

The `psbt` SHOULD include bip32 derivations. The command SHOULD fail if none of the bip32 derivations match a key owned by the device.

The command SHOULD fail if the user cancels.

The command MAY complain if `--testnet` is set, but any of the BIP32 derivation paths contain a coin type other than `1h` (and vice versa).

`getdescriptors` (optional)

Usage:

```
$ <cmd> --fingerprint=<fingerprint> (--testnet) getdescriptors <account>
<xpub>
```

Returns descriptors supported by the device. Example:

```
$ <cmd> --fingerprint=00000000 --testnet getdescriptors
{
  "receive": [
    "pkh([00000000/44h/0h/0h]xpub6C.../0/*)#fn95jwmg",
    "sh(wpkh([00000000/49h/0h/0h]xpub6B..../0/*))#j4r9hntt",
    "wpkh([00000000/84h/0h/0h]xpub6C.../0/*)#qw72dxa9"
  ],
  "internal": [
    "pkh([00000000/44h/0h/0h]xpub6C.../1/*)#c8q40mts",
    "sh(wpkh([00000000/49h/0h/0h]xpub6B..../1/*))#85dn0v75",
    "wpkh([00000000/84h/0h/0h]xpub6C..../1/*)#36mtsnda"
  ]
}
```

`displayaddress` (optional)

Usage:

```
<cmd> --fingerprint=<fingerprint> (--testnet) displayaddress --desc descriptor
```

Example, display the first native SegWit receive address on Testnet:

```
<cmd> --fingerprint=00000000 --testnet displayaddress --desc
"wpkh([00000000/84h/1h/0h]tpubDDUZ..../0/0)"
```

The command MUST be able to figure out the address type from the descriptor.

If contains a master key fingerprint, the command MUST fail if it does not match the fingerprint known by the device.

If contains an xpub, the command MUST fail if it does not match the xpub known by the device.

The command MAY complain if `--testnet` is set, but the BIP32 coin type is not `1h` (and vice versa).

How Bitcoin Core uses the Signer API

The `enumeratesigners` RPC simply calls `<cmd> enumerate`.

The `createwallet` RPC calls:

- `<cmd> --fingerprint=00000000 getdescriptors 0`

It then imports descriptors for all support address types, in a BIP44/49/84 compatible manner.

The `walletdisplayaddress` RPC reuses some code from `getaddressinfo` on the provided address and obtains the inferred descriptor. It then calls `<cmd> --fingerprint=00000000 displayaddress --desc=<descriptor>`.

`sendtoaddress` and `sendmany` check `inputs->bip32_derivs` to see if any inputs have the same `master_fingerprint` as the signer. If so, it calls `<cmd> --fingerprint=00000000 signtransaction <psbt>`. It waits for the device to return a (partially) signed psbt, tries to finalize it and broadcasts the transaction.