Support for Output Descriptors in Bitcoin Core

Since Bitcoin Core v0.17, there is support for Output Descriptors. This is a simple language which can be used to describe collections of output scripts. Supporting RPCs are: - `scantxoutset` takes as input descriptors to scan for, and also reports specialized descriptors for the matching UTXOs. - `getdescriptorinfo` analyzes a descriptor, and reports a canonicalized version with checksum added. - `deriveaddresses` takes as input a descriptor and computes the corresponding addresses. - `listunspent` outputs a specialized descriptor for the reported unspent outputs. - `getaddressinfo` outputs a descriptor for solvable addresses (since v0.18). - `importmulti` takes as input descriptors to import into the wallet (since v0.18). - `generatetodescriptor` takes as input a descriptor and generates coins to it (`regtest` only, since v0.19). - `utxoupdatepsbt` takes as input descriptors to add information to the psbt (since v0.19). - `createmultisig` and `addmultisigaddress` return descriptors as well (since v0.20)

This document describes the language. For the specifics on usage, see the RPC documentation for the functions mentioned above.

Features

Output descriptors currently support: - Pay-to-pubkey scripts (P2PK), through the `pk` function. - Pay-to-pubkey-hash scripts (P2PKH), through the `pkh` function. - Pay-to-witness-pubkey-hash scripts (P2WPKH), through the `wpkh` function. - Pay-to-script-hash scripts (P2SH), through the `sh` function. - Pay-to-witness-script-hash scripts (P2WSH), through the `wsh` function. - Pay-to-taproot outputs (P2TR), through the `tr` function. - Multisig scripts, through the `multi` function. - Multisig scripts where the public keys are sorted lexicographically, through the `sortedmulti` function. - Multisig scripts inside taproot script trees, through the `multi_a` (and `sortedmulti_a`) function. - Any type of supported address through the `addr` function. - Raw hex scripts through the `raw` function. - Public keys (compressed and uncompressed) in hex notation, or BIP32 extended pubkeys with derivation paths.

Examples

- `pk(0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)` describes a P2PK output with the specified public key.
- `pkh(02c6047f9441ed7d6d3045406e95c07cd85c778e4b8cef3ca7abac09b95c709ee5)` describes a P2PKH output with the specified public key.
- `wpkh(02f9308a019258c31049344f85f89d5229b531c845836f99b08601f113bce036f9)` describes a P2WPKH output with the specified public key.
- `sh(wpkh(03fff97bd5755eeea420453a14355235d382f6472f8568a18b2f057a1460297556))` describes a P2SH-P2WPKH output with the specified public key.
- `combo(0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)` describes any P2PK, P2PKH, P2WPKH, or P2SH-P2WPKH output with the specified public key.

- `sh(wsh(pkh(02e493dbf1c10d80f3581e4904930b1404cc6c13900ee0758474fa94abe8c4cd13)))` describes an (overly complicated) P2SH-P2WSH-P2PKH output with the specified public key.
- `multi(1,022f8bde4d1a07209355b4a7250a5c5128e88b84bddc619ab7cba8d569b240efe4,025cbdf0646e` describes a bare *1-of-2* multisig output with keys in the specified order.
- `sh(multi(2,022f01e5e15cca351daff3843fb70f3c2f0a1bdd05e5af888a67784ef3e10a2a01,03acd484e` describes a P2SH *2-of-2* multisig output with keys in the specified order.
- `sh(sortedmulti(2,03acd484e2f0c7f65309ad178a9f559abde09796974c57e714c35f110dfc27ccbe,022` describes a P2SH *2-of-2* multisig output with keys sorted lexicographically in the resulting redeemScript.
- `wsh(multi(2,03a0434d9e47f3c86235477c7b1ae6ae5d3442d49b1943c2b752a68e2a47e247c7,03774ae7` describes a P2WSH *2-of-3* multisig output with keys in the specified order.
- `sh(wsh(multi(1,03f28773c2d975288bc7d1d205c3748651b075fbc6610e58cddeeddf8f19405aa8,03499` describes a P2SH-P2WSH *1-of-3* multisig output with keys in the specified order.
- `pk(xpub661MyMwAqRbcFtXgS5sYJABqqG9YLmC4Q1Rdap9gSE8NqtwybGhePY2gZ29ESFjqJoCu1Rupje8YtGqs` describes a P2PK output with the public key of the specified xpub.
- `pkh(xpub68Gmy5EdvgibQVfPdqkBBCHxA5htiqg55crXYuXoQRKfDBFA1WEjWgP6LHhwBZeNK1VTsfTFUHCdrfp` describes a P2PKH output with child key *1/2* of the specified xpub.
- `pkh([d34db33f/44'/0'/0']xpub6ERApfZwUNrhLCkDtcHTcxd75RbzS1ed54G1LkBUHQVHQKqhMkhgbmJbZRk` describes a set of P2PKH outputs, but additionally specifies that the specified xpub is a child of a master with fingerprint `d34db33f`, and derived using path `44'/0'/0'`.
- `wsh(multi(1,xpub661MyMwAqRbcFW31YEwpkMuc5THy2PSt5bDMsktWQcFF8syAmRUapSCGu8ED9W6oDMSgv6Z` describes a set of *1-of-2* P2WSH multisig outputs where the first multisig key is the *1/0/`i`* child of the first specified xpub and the second multisig key is the *0/0/`i`* child of the second specified xpub, and `i` is any number in a configurable range (`0-1000` by default).
- `wsh(sortedmulti(1,xpub661MyMwAqRbcFW31YEwpkMuc5THy2PSt5bDMsktWQcFF8syAmRUapSCGu8ED9W6oD` describes a set of *1-of-2* P2WSH multisig outputs where one multisig key is the *1/0/`i`* child of the first specified xpub and the other multisig key is the *0/0/`i`* child of the second specified xpub, and `i` is any number in a configurable range (`0-1000` by default). The order of public keys in the resulting witnessScripts is determined by the lexicographic order of the public keys at that index.
- `tr(c6047f9441ed7d6d3045406e95c07cd85c778e4b8cef3ca7abac09b95c709ee5,{pk(fff97bd5755eeea` describes a P2TR output with the `c6...` x-only pubkey as internal key, and two script paths.
- `tr(c6047f9441ed7d6d3045406e95c07cd85c778e4b8cef3ca7abac09b95c709ee5,sortedmulti_a(2,2f8` describes a P2TR output with the `c6...` x-only pubkey as internal key, and a single `multi_a` script that needs 2 signatures with 2 specified x-only keys, which will be sorted lexicographically.

Reference

Descriptors consist of several types of expressions. The top level expression is either a `SCRIPT`, or `SCRIPT#CHECKSUM` where `CHECKSUM` is an 8-character alphanumeric descriptor checksum.

`SCRIPT` expressions: - `sh(SCRIPT)` (top level only): P2SH embed the argument. - `wsh(SCRIPT)` (top level or inside `sh` only): P2WSH embed the argument. - `pk(KEY)` (anywhere): P2PK output for the given public key. - `pkh(KEY)` (not inside `tr`): P2PKH output for the given public key (use `addr` if you only know the pubkey hash). - `wpkh(KEY)` (top level or inside `sh` only): P2WPKH output for the given compressed pubkey. - `combo(KEY)` (top level only): an alias for the collection of `pk(KEY)` and `pkh(KEY)`. If the key is compressed, it also includes `wpkh(KEY)` and `sh(wpkh(KEY))`. - `multi(k,KEY_1,KEY_2,...,KEY_n)` (not inside `tr`): k-of-n multisig script using OP_CHECKMULTISIG. - `sortedmulti(k,KEY_1,KEY_2,...,KEY_n)` (not inside `tr`): k-of-n multisig script with keys sorted lexicographically in the resulting script. - `multi_a(k,KEY_1,KEY_2,...,KEY_N)` (only inside `tr`): k-of-n multisig script using OP_CHECKSIG, OP_CHECKSIGADD, and OP_NUMEQUAL. - `sortedmulti_a(k,KEY_1,KEY_2,...,KEY_N)` (only inside `tr`): similar to `multi_a`, but the (x-only) public keys in it will be sorted lexicographically. - `tr(KEY)` or `tr(KEY,TREE)` (top level only): P2TR output with the specified key as internal key, and optionally a tree of script paths. - `addr(ADDR)` (top level only): the script which ADDR expands to. - `raw(HEX)` (top level only): the script whose hex encoding is HEX.

`KEY` expressions: - Optionally, key origin information, consisting of: - An open bracket `[` - Exactly 8 hex characters for the fingerprint of the key where the derivation starts (see BIP32 for details) - Followed by zero or more `/NUM` or `/NUM'` path elements to indicate unhardened or hardened derivation steps between the fingerprint and the key or xpub/xprv root that follows - A closing bracket `]` - Followed by the actual key, which is either: - Hex encoded public keys (either 66 characters starting with `02` or `03` for a compressed pubkey, or 130 characters starting with `04` for an uncompressed pubkey). - Inside `wpkh` and `wsh`, only compressed public keys are permitted. - Inside `tr`, x-only pubkeys are also permitted (64 hex characters). - WIF encoded private keys may be specified instead of the corresponding public key, with the same meaning. - `xpub` encoded extended public key or `xprv` encoded extended private key (as defined in BIP 32). - Followed by zero or more `/NUM` unhardened and `/NUM'` hardened BIP32 derivation steps. - Optionally followed by a single `/*` or `/*'` final step to denote all (direct) unhardened or hardened children. - The usage of hardened derivation steps requires providing the private key.

(Anywhere a `'` suffix is permitted to denote hardened derivation, the suffix `h` can be used instead.)

`TREE` expressions: - any `SCRIPT` expression - An open brace `{`, a `TREE` expression, a comma `,`, a `TREE` expression, and a closing brace `}`

`ADDR` expressions are any type of supported address: - P2PKH addresses (base58, of the form `1...` for mainnet or `[nm]...` for testnet). Note that P2PKH addresses in descriptors cannot be used for P2PK outputs (use the `pk` function instead). - P2SH addresses (base58, of the form `3...` for mainnet or `2...` for testnet, defined in BIP 13). - Segwit addresses (bech32 and bech32m, of the form `bc1...` for mainnet or `tb1...` for testnet, defined in BIP 173 and BIP 350).

Explanation

Single-key scripts

Many single-key constructions are used in practice, generally including P2PK, P2PKH, P2WPKH, and P2SH-P2WPKH. Many more combinations are imaginable, though they may not be optimal: P2SH-P2PK, P2SH-P2PKH, P2WSH-P2PK, P2WSH-P2PKH, P2SH-P2WSH-P2PK, P2SH-P2WSH-P2PKH.

To describe these, we model these as functions. The functions `pk` (P2PK), `pkh` (P2PKH) and `wpkh` (P2WPKH) take as input a `KEY` expression, and return the corresponding *scriptPubKey*. The functions `sh` (P2SH) and `wsh` (P2WSH) take as input a `SCRIPT` expression, and return the script describing P2SH and P2WSH outputs with the input as embedded script. The names of the functions do not contain "p2" for brevity.

Multisig

Several pieces of software use multi-signature (multisig) scripts based on Bitcoin's OP_CHECKMULTISIG opcode. To support these, we introduce the `multi(k,key_1,key_2,...,key_n)` and `sortedmulti(k,key_1,key_2,...,key_n)` functions. They represent a *k-of-n* multisig policy, where any *k* out of the *n* provided `KEY` expressions must sign.

Key order is significant for `multi()`. A `multi()` expression describes a multisig script with keys in the specified order, and in a search for TXOs, it will not match outputs with multisig scriptPubKeys that have the same keys in a different order. Also, to prevent a combinatorial explosion of the search space, if more than one of the `multi()` key arguments is a BIP32 wildcard path ending in `/*` or `*'`, the `multi()` expression only matches multisig scripts with the `i`th child key from each wildcard path in lockstep, rather than scripts with any combination of child keys from each wildcard path.

Key order does not matter for `sortedmulti()`. `sortedmulti()` behaves in the same way as `multi()` does but the keys are reordered in the resulting script such that they are lexicographically ordered as described in BIP67.

Basic multisig example For a good example of a basic M-of-N multisig between multiple participants using descriptor wallets and PSBTs, as well as a signing flow, see this functional test.

Disclaimers: It is important to note that this example serves as a quick-start and is kept basic for readability. A downside of the approach outlined here is that each participant must maintain (and backup) two separate wallets: a signer and the corresponding multisig. It should also be noted that privacy best-practices are not “by default” here - participants should take care to only use the signer to sign transactions related to the multisig. Lastly, it is not recommended to use anything other than a Bitcoin Core descriptor wallet to serve as your signer(s). Other wallets, whether hardware or software, likely impose additional checks and safeguards to prevent users from signing transactions that could lead to loss of funds, or are deemed security hazards. Conforming to various 3rd-party checks and verifications is not in the scope of this example.

The basic steps are:

1. Every participant generates an xpub. The most straightforward way is to create a new descriptor wallet which we will refer to as the participant’s signer wallet. Avoid reusing this wallet for any purpose other than signing transactions from the corresponding multisig we are about to create. Hint: extract the wallet’s xpubs using `listdescriptors` and pick the one from the `pkh` descriptor since it’s least likely to be accidentally reused (legacy addresses)
2. Create a watch-only descriptor wallet (blank, private keys disabled). Now the multisig is created by importing the two descriptors: wsh(sortedmulti(<M>,XPUB1/0/*,XPUB2/0/*,...,XPUBN/0/*)) and wsh(sortedmulti(<M>,XPUB1/1/*,XPUB2/1/*,...,XPUBN/1/*)) (one descriptor w/ `0` for receiving addresses and another w/ `1` for change). Every participant does this
3. A receiving address is generated for the multisig. As a check to ensure step 2 was done correctly, every participant should verify they get the same addresses
4. Funds are sent to the resulting address
5. A sending transaction from the multisig is created using `walletcreatefundedpsbt` (anyone can initiate this). It is simple to do this in the GUI by going to the `Send` tab in the multisig wallet and creating an unsigned transaction (PSBT)
6. At least `M` participants check the PSBT with their multisig using `decodepsbt` to verify the transaction is OK before signing it.
7. (If OK) the participant signs the PSBT with their signer wallet using `walletprocesspsbt`. It is simple to do this in the GUI by loading the PSBT from file and signing it
8. The signed PSBTs are collected with `combinepsbt`, finalized w/ `finalizepsbt`, and then the resulting transaction is broadcasted to the network. Note that any wallet (eg one of the signers or multisig) is capable of doing this.
9. Checks that balances are correct after the transaction has been included in a block

You may prefer a daisy chained signing flow where each participant signs the PSBT one after another until the PSBT has been signed `M` times and is "complete." For the most part, the steps above remain the same, except (6, 7) change slightly from signing the original PSBT in parallel to signing it in series. `combinepsbt` is not necessary with this signing flow and the last (`m`th) signer can just broadcast the PSBT after signing. Note that a parallel signing flow may be preferable in cases where there are more signers. This signing flow is also included in the test / Python example. The test is meant to be documentation as much as it is a functional test, so it is kept as simple and readable as possible.

BIP32 derived keys and chains

Most modern wallet software and hardware uses keys that are derived using BIP32 ("HD keys"). We support these directly by permitting strings consisting of an extended public key (commonly referred to as an *xpub*) plus derivation path anywhere a public key is expected. The derivation path consists of a sequence of 0 or more integers (in the range *0..231-1*) each optionally followed by `'` or `h`, and separated by `/` characters. The string may optionally end with the literal `/*` or `/*'` (or `/*h`) to refer to all unhardened or hardened child keys in a configurable range (by default `0-1000`, inclusive).

Whenever a public key is described using a hardened derivation step, the script cannot be computed without access to the corresponding private key.

Key origin identification

In order to describe scripts whose signing keys reside on another device, it may be necessary to identify the master key and derivation path an xpub was derived with.

For example, when following BIP44, it would be useful to describe a change chain directly as `xpub.../44'/0'/0'/1/*` where `xpub...` corresponds with the master key `m`. Unfortunately, since there are hardened derivation steps that follow the xpub, this descriptor does not let you compute scripts without access to the corresponding private keys. Instead, it should be written as `xpub.../1/*`, where xpub corresponds to `m/44'/0'/0'`.

When interacting with a hardware device, it may be necessary to include the entire path from the master down. BIP174 standardizes this by providing the master key *fingerprint* (first 32 bit of the Hash160 of the master pubkey), plus all derivation steps. To support constructing these, we permit providing this key origin information inside the descriptor language, even though it does not affect the actual scriptPubKeys it refers to.

Every public key can be prefixed by an 8-character hexadecimal fingerprint plus optional derivation steps (hardened and unhardened) surrounded by brackets, identifying the master and derivation path the key or xpub that follows was derived with.

Note that the fingerprint of the parent only serves as a fast way to detect parent and child nodes in software, and software must be willing to deal with collisions.

Including private keys

Often it is useful to communicate a description of scripts along with the necessary private keys. For this reason, anywhere a public key or xpub is supported, a private key in WIF format or xprv may be provided instead. This is useful when private keys are necessary for hardened derivation steps, or for dumping wallet descriptors including private key material.

Compatibility with old wallets

In order to easily represent the sets of scripts currently supported by existing Bitcoin Core wallets, a convenience function `combo` is provided, which takes as input a public key, and describes a set of P2PK, P2PKH, P2WPKH, and P2SH-P2WPKH scripts for that key. In case the key is uncompressed, the set only includes P2PK and P2PKH scripts.

Checksums

Descriptors can optionally be suffixed with a checksum to protect against typos or copy-paste errors.

These checksums consist of 8 alphanumeric characters. As long as errors are restricted to substituting characters in `0123456789()[],'/*abcdefgh@:$%{}` for others in that set and changes in letter case, up to 4 errors will always be detected in descriptors up to 501 characters, and up to 3 errors in longer ones. For larger numbers of errors, or other types of errors, there is a roughly 1 in a trillion chance of not detecting the errors.

All RPCs in Bitcoin Core will include the checksum in their output. Only certain RPCs require checksums on input, including `deriveaddress` and `importmulti`. The checksum for a descriptor without one can be computed using the `getdescriptorinfo` RPC.