UTXO Consolidation (3 inputs, 1 output) - Legacy P2PKH

TIP

To follow along this tutorial

- Clone the Github repository
- `cd code`
- `npm install` or `yarn install`
- Execute the transaction code by typing `node tx_filename.js`
- Alternatively you can enter the commands step-by-step by `cd` into `./code` then type `node` in a terminal to open the Node.js REPL
- Open the Bitcoin Core GUI console or use `bitcoin-cli` for the Bitcoin Core commands
- Use `bx` aka `Libbitcoin-explorer` as a handy complement

Let's do UTXO consolidation, also called an aggregating transaction.

The idea is that a single signer aggregates many small UTXOs into a single big UTXO. This represents the real-world equivalent of exchanging a pile of coins and currency notes for a single larger note.

In legacy P2PKH transaction, each input contains a signature, which quickly increases the size of your transactions if you need to spend multiple UTXOs, resulting in high fees. Consolidation allows to group multiple UTXOs in one, which will result in less fees when spent.

Reducing the number of UTXOs also frees the UTXOs database (chainstate), making it easier to run a full node, marginally improving Bitcoin's decentralisation and overall security, which is always nice.

Finally, consolidation gives an opportunity to update the addresses you use for your UTXOs, for example to roll keys over, switch to multisig, or switch to Segwit bech32 addresses.

For this example we will consolidate three P2PKH UTXOs (referencing each of them with three inputs) to one Segwit P2WPKH UTXO.

Creating UTXOs to spend

First we need to create three UTXOs.

UTXO consolidation makes sense if we have multiple payments on a single address, or multiple payments on different addresses controlled by a single person or entity.

Let's use different P2PKH addresses, all controlled by Alice. We will have one payment to alice_1 P2PKH address, one to alice_2 P2PKH address and one to alice_3 P2PKH address.

Let's send this three payments, of 0.2 BTC each, in a single transaction using sendmany.

```
sendmany "" '{"n4SvybJicv79X1Uc4o3fYXWGwXadA53FSq":0.33, "my
```

We have now three UTXOs locked with Alice public keys hash. In order to spend it, we refer to it with the transaction id (txid) and the output index (vout), also called **outpoint**.

Get the output indexes of the transaction, so that we have their outpoint (txid / vout).

```
gettransaction "txid"
```

Find the output index (or vout) under **details** › **vout**.
Each UXTO appears twice, once in the 'send' category, once in the 'receive' category.
You may think that the indexes will be 0, 1 and 2. If so you are forgetting that there is also a change UTXO, that can be located at any position.
The command `gettransaction` doesn't display the change UTXO, but a `decoderawtransaction` does.

Creating the UTXO consolidation transaction

Import libraries, test wallets and set the network

```
const bitcoin = require('bitcoinjs-lib')
const { alice } = require('./wallets.json')
const network = bitcoin.networks.regtest
```

Now let's spend the UTXOs with BitcoinJS, consolidating them into a single P2WPKH UTXO.

Create Alice's keypairs.

```
const keyPairAlice1 = bitcoin.ECPair.fromWIF(alice[1].wif, netw
const keyPairAlice2 = bitcoin.ECPair.fromWIF(alice[2].wif, netw
const keyPairAlice3 = bitcoin.ECPair.fromWIF(alice[3].wif, n
```

Get the previous transaction as buffer from TX_HEX

```
const nonWitnessUtxo =  Buffer.from('TX_HEX', 'hex') ①
```

① Get TX_HEX with "getrawtransaction TX_ID"

Create the PSBT by filling TX_ID, TX_OUT and TX_HEX.

```
const psbt = new bitcoin.Psbt({network})
  .addInput({
    hash: 'TX_ID',
    index: TX_OUT,
    nonWitnessUtxo
  })
  .addInput({
    hash: 'TX_ID',
    index: TX_OUT,
    nonWitnessUtxo
  })
  .addInput({
    hash: 'TX_ID',
    index: TX_OUT,
    nonWitnessUtxo
  }) ①
  .addOutput({
    address: alice[1].p2wpkh,
    value: 989e5,
  }) ②
```

① Three inputs referencing the 0.33BTC UTXOs just created
② P2WPKH output with an amount of 0.99 BTC

NOTE

The miner fee is calculated by subtracting the outputs from the inputs.
(33 000 000 + 33 000 000 + 33 000 000) - 98 900 000 = 100 000 100 000 satoshis equals 0,001 BTC, this is the miner fee.

Alice adds the correct signature to each input.

For each input, we have to select the right keypair to produce signatures that satisfy the locking scripts of the three UTXOs we are spending.
We have this information by calling `gettransaction`.
This is tricky because the order of the UTXOs in a transaction varies.
For example
txb.sign(0, keyPairAlice1)
txb.sign(1, keyPairAlice3)
txb.sign(2, keyPairAlice2)

Sign the inputs and validate signatures

```
psbt.signInput(0, keyPairAlice1)
psbt.signInput(1, keyPairAlice2)
psbt.signInput(2, keyPairAlice3)

psbt.validateSignaturesOfInput(0)
psbt.validateSignaturesOfInput(1)
psbt.validateSignaturesOfInput(2)
```

Finalize the PSBT.

```
psbt.finalizeAllInputs()
```

Finally we can extract the transaction and get the raw hex serialization.

```
console.log('Transaction hexadecimal:')
console.log(psbt.extractTransaction().toHex())
```

Inspect the raw transaction with Bitcoin Core CLI, check that everything is correct.

```
decoderawtransaction TX_HEX
```

Broadcasting the transaction

It's time to broadcast the transaction via Bitcoin Core CLI.

```
sendrawtransaction TX_HEX
```

Inspect the transaction.

```
getrawtransaction TX_ID true
```

Observations

We note that each input contains a 71 bytes signature (on average), resulting in a large and costly transaction. Spending Segwit UTXOs results in much smaller transaction since the input scripts are empty and Segwit data are discounted. Also, the implementation of Schnorr signatures into Bitcoin will allow to aggregate all those signatures into a single one.