

Bitcoin Developer Examples

Find examples of how to build programs using Bitcoin.

Search the glossary, RPCs, and more

The following guide aims to provide examples to help you start building Bitcoin-based applications. To make the best use of this document, you may want to install the current version of Bitcoin Core, either from [source](#) or from a [pre-compiled executable](#).

Once installed, you'll have access to three programs: `bitcoind`, `bitcoin-qt`, and `bitcoin-cli`.

- `bitcoin-qt` provides a combination full Bitcoin peer and wallet frontend. From the Help menu, you can access a console where you can enter the RPC commands used throughout this document.
- `bitcoind` is more useful for programming: it provides a full peer which you can interact with through RPCs to port 8332 (or 18332 for testnet).
- `bitcoin-cli` allows you to send RPC commands to `bitcoind` from the command line. For example, `bitcoin-cli help`

All three programs get settings from `bitcoin.conf` in the `Bitcoin` application directory:

- Windows: `%APPDATA%\Bitcoin\`
- OSX: `$HOME/Library/Application Support/Bitcoin/`
- Linux: `$HOME/.bitcoin/`

To use `bitcoind` and `bitcoin-cli`, you will need to add a RPC password to your `bitcoin.conf` file. Both programs will read from the same file if both run on the same system as the same user, so any long random password will work:

```
rpcpassword=change_this_to_a_long_random_password
```

You should also make the `bitcoin.conf` file only readable to its owner. On Linux, Mac OSX, and other Unix-like systems, this can be accomplished by running the following command in the Bitcoin application directory:

```
chmod 0600 bitcoin.conf
```

For development, it's safer and cheaper to use Bitcoin's test network (testnet) or regression test mode (regtest) described below.

Questions about Bitcoin use are best sent to the [BitcoinTalk forum](#) and [IRC channels](#). Errors or suggestions related to documentation on Bitcoin.org can be [submitted as an issue](#) or posted to the [bitcoin-documentation mailing list](#).

In the following documentation, some strings have been shortened or wrapped: “[...]” indicates extra data was removed, and lines ending in a single backslash “\” are continued below. If you hover your mouse over a paragraph, cross-reference links will be shown in blue. If you hover over a cross-reference link, a brief definition of the term will be displayed in a tooltip.

Testing Applications

Bitcoin Core provides testing tools designed to let developers test their applications with reduced risks and limitations.

Testnet

When run with no arguments, all Bitcoin Core programs default to Bitcoin’s main network (**mainnet**). However, for development, it’s safer and cheaper to use Bitcoin’s test network (testnet) where the satoshis spent have no real-world value. Testnet also relaxes some restrictions (such as standard transaction checks) so you can test functions which might currently be disabled by default on mainnet.

To use testnet, use the argument `-testnet` with `bitcoin-cli`, `bitcoind` or `bitcoin-qt` or add `testnet=1` to your `bitcoin.conf` file as [described earlier](#). To get free satoshis for testing, use [Piotr Piasecki’s testnet faucet](#). Testnet is a public resource provided for free by members of the community, so please don’t abuse it.

Regtest Mode

For situations where interaction with random peers and blocks is unnecessary or unwanted, Bitcoin Core’s regression test mode (regtest mode) lets you instantly create a brand-new private block chain with the same basic rules as testnet—but one major difference: you choose when to create new blocks, so you have complete control over the environment.

Many developers consider regtest mode the preferred way to develop new applications. The following example will let you create a regtest environment after you first [configure bitcoind](#).

```
> bitcoind -regtest -daemon
Bitcoin server starting
```

Start `bitcoind` in regtest mode to create a private block chain.

```
## Bitcoin Core 0.10.1 and earlier
bitcoin-cli -regtest setgenerate true 101

## Bitcoin Core master (as of commit 48265f3)
bitcoin-cli -regtest generate 101
```

Generate 101 blocks using a special RPC which is only available in regtest mode. This takes about 30 seconds on a generic PC. Because this is a new block chain using Bitcoin’s default rules, the first blocks pay a block reward of 50 bitcoins. Unlike mainnet, in regtest mode only the first 150 blocks pay a reward of 50 bitcoins. However, a block must have 100 confirmations before that reward can be spent, so we generate 101 blocks to get access to the coinbase transaction from block #1.

```
bitcoin-cli -regtest getbalance
50.00000000
```

Verify that we now have 50 bitcoins available to spend.

You can now use Bitcoin Core RPCs prefixed with `bitcoin-cli -regtest`.

Regtest wallets and block chain state (chainstate) are saved in the `regtest` subdirectory of the Bitcoin Core configuration directory. You can safely delete the `regtest` subdirectory and restart Bitcoin Core to start a new regtest. (See the [Developer](#)

[Examples](#) [Introduction](#) for default configuration directory locations on various operating systems. Always back up mainnet wallets before performing dangerous operations such as deleting.)

Transactions

Transaction Tutorial

Creating transactions is something most Bitcoin applications do. This section describes how to use Bitcoin Core’s RPC interface to create transactions with various attributes.

Your applications may use something besides Bitcoin Core to create transactions, but in any system, you will need to provide the same kinds of data to create transactions with the same attributes as those described below.

In order to use this tutorial, you will need to setup [Bitcoin Core](#) and create a regression test mode environment with 50 BTC in your test wallet.

Simple Spending

Bitcoin Core provides several RPCs which handle all the details of spending, including creating change outputs and paying appropriate fees. Even advanced users should use these RPCs whenever possible to decrease the chance that satoshis will be lost by mistake.

```
> bitcoin-cli -regtest getnewaddress
mvbnrCX3bg1cDRUu8pkecrvP6vQkSLDSou

> NEW_ADDRESS=mvbnrCX3bg1cDRUu8pkecrvP6vQkSLDSou
```

Get a new Bitcoin address and save it in the shell variable `NEW_ADDRESS`.

```
> bitcoin-cli -regtest sendtoaddress NEW_ADDRESS 10.00
263c018582731ff54dc72c7d67e858c002ae298835501d80200f05753de0edf0
```

Send 10 bitcoins to the address using the `sendtoaddress` RPC. The returned hex string is the transaction identifier (txid).

The `sendtoaddress` RPC automatically selects an unspent transaction output (UTXO) from which to spend the satoshis. In this case, it withdrew the satoshis from our only available UTXO, the coinbase transaction for block #1 which matured with the creation of block #101. To spend a specific UTXO, you could use the `sendfrom` RPC instead.

```
> bitcoin-cli -regtest listunspent
[
]
```

Use the `listunspent` RPC to display the UTXOs belonging to this wallet. The list is empty because it defaults to only showing confirmed UTXOs and we just spent our only confirmed UTXO.

```
> bitcoin-cli -regtest listunspent 0
[
  {
    "txid" : "263c018582731ff54dc72c7d67e858c002ae298835501d\
      80200f05753de0edf0",
    "vout" : 0,
    "address" : "muhtvdmsnbQEPFuEmxcChX58fGvXaaUoVt",
    "scriptPubKey" : "76a9149ba386253ea698158b6d34802bb9b550\
      f5ce36dd88ac",
    "amount" : 40.00000000,
    "confirmations" : 0
  },
  {
    "txid" : "263c018582731ff54dc72c7d67e858c002ae298835501d\
      80200f05753de0edf0",
    "vout" : 1,
    "address" : "mvbnrCX3bg1cDRUu8pkecrvP6vQkSLDSou",
    "account" : "",
    "scriptPubKey" : "76a914a57414e5ffae9ef5074bacbe10a320bb\
      2614e1f388ac",
    "amount" : 10.00000000,
    "confirmations" : 0
  }
]
```

Re-running the `listunspent` RPC with the argument “0” to also display unconfirmed transactions shows that we have two UTXOs, both with the same txid. The first UTXO shown is a change output that `sendtoaddress` created using a new address from the key pool. The second UTXO shown is the spend to the address we provided. If we had spent those satoshis to someone else, that second transaction would not be displayed in our list of UTXOs.

```
## Bitcoin Core 0.10.1 and earlier
> bitcoin-cli -regtest setgenerate true 1

## Later versions of Bitcoin Core
> bitcoin-cli -regtest generate 1

> unset NEW_ADDRESS
```

Create a new block to confirm the transaction above (takes less than a second) and clear the shell variable.

Simple Raw Transaction

The raw transaction RPCs allow users to create custom transactions and delay broadcasting those transactions. However, mistakes made in raw transactions may not be detected by Bitcoin Core, and a number of raw transaction users have permanently lost large numbers of satoshis, so please be careful using raw transactions on mainnet.

This subsection covers one of the simplest possible raw transactions.

```
> bitcoin-cli -regtest listunspent
[
  {
    "txid" : "263c018582731ff54dc72c7d67e858c002ae298835501d\
      80200f05753de0edf0",
    "vout" : 0,
    "address" : "muhtvdmsnbQEPFuEmxcChX58fGvXaaUoVt",
    "scriptPubKey" : "76a9149ba386253ea698158b6d34802bb9b550\
```

```
        "f5ce36dd88ac",
        "amount" : 40.00000000,
        "confirmations" : 1
    },
    {
        "txid" : "263c018582731ff54dc72c7d67e858c002ae298835501d\
80200f05753de0edf0",
        "vout" : 1,
        "address" : "mvbnrCX3bg1cDRUu8pkecrvP6vQkSLDSou",
        "account" : "",
        "scriptPubKey" : "76a914a57414e5ffae9ef5074bacbe10a320bb\
2614e1f388ac",
        "amount" : 10.00000000,
        "confirmations" : 1
    },
    {
        "txid" : "3f4fa19803dec4d6a84fae3821da7ac7577080ef754512\
94e71f9b20e0able7b",
        "vout" : 0,
        "address" : "mwJTL1dZG8BAP6X7Be3CNNcuVKi7Qqt7Gk",
        "scriptPubKey" : "210260a275cccf0f4b106220725be516adba27\
52db1bec8c5b7174c89c4c07891f88ac",
        "amount" : 50.00000000,
        "confirmations" : 101
    }
]

> UTXO_TXID=3f4fa19803dec4d6a84fae3821da7ac7577080ef75451294e71f[...]
> UTXO_VOUT=0
```

Re-rerun `listunspent`. We now have three UTXOs: the two transactions we created before plus the coinbase transaction from block #2. We save the txid and output index number (vout) of that coinbase UTXO to shell variables.

```
> bitcoin-cli -regtest getnewaddress
mz6KvC4aoUeo6wSxtiVQTo7FDwPnkp6URG

> NEW_ADDRESS=mz6KvC4aoUeo6wSxtiVQTo7FDwPnkp6URG
```

Get a new address to use in the raw transaction.


```
## Outputs - inputs = transaction fee, so always double-check your math!
> bitcoin-cli -regtest createrawtransaction '' '
[
  {
    "txid": "'$UTXO_TXID'",
    "vout": '$UTXO_VOUT'
  }
]
...
{
  "'$NEW_ADDRESS'": 49.9999
}'

01000000017b1eabe0209b1fe794124575ef807057c77ada2138ae4fa8d6c4de\
0398a14f3f0000000000ffffffff01f0ca052a010000001976a914cbc20a7664\
f2f69e5355aa427045bc15e7c6c77288ac00000000

> RAW_TX=01000000017b1eabe0209b1fe794124575ef807057c77ada2138ae4[...]
```

Using two arguments to the `createrawtransaction` RPC, we create a new raw format transaction. The first argument (a JSON array) references the txid of the coinbase transaction from block #2 and the index number (0) of the output from that transaction

we want to spend. The second argument (a JSON object) creates the output with the address (public key hash) and number of bitcoins we want to transfer. We save the resulting raw format transaction to a shell variable.

 **Warning:** `createrawtransaction` does not automatically create change outputs, so you can easily accidentally pay a large transaction fee. In this example, our input had 50.0000 bitcoins and our output (`$NEW_ADDRESS`) is being paid 49.9999 bitcoins, so the transaction will include a fee of 0.0001 bitcoins. If we had paid `$NEW_ADDRESS` only 10 bitcoins with no other changes to this transaction, the transaction fee would be a whopping 40 bitcoins. See the Complex Raw Transaction subsection below for how to create a transaction with multiple outputs so you can send the change back to yourself.

```
> bitcoin-cli -regtest decoderawtransaction $RAW_TX
{
  "txid" : "c80b343d2ce2b5d829c2de9854c7c8d423c0e33bda264c4013\
            8d834aab4c0638",
  "version" : 1,
  "locktime" : 0,
  "vin" : [
    {
      "txid" : "3f4fa19803dec4d6a84fae3821da7ac7577080ef75\
                451294e71f9b20e0able7b",
      "vout" : 0,
      "scriptSig" : {
        "asm" : "",
        "hex" : ""
      },
      "sequence" : 4294967295
    }
  ],
  "vout" : [
    {
      "value" : 49.99990000,
      "n" : 0,
      "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 cbc20a7664f2f69e5355a\
                  a427045bc15e7c6c772 OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a914cbc20a7664f2f69e5355aa427045bc15e\
                  7c6c77288ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
          "mz6KvC4aoUeo6wSxtiVQTo7FDwPnkp6URG"
        ]
      }
    }
  ]
}
```

Use the `decoderawtransaction` RPC to see exactly what the transaction we just created does.

```
> bitcoin-cli -regtest signrawtransaction $RAW_TX
{
  "hex" : "01000000017b1eabe0209b1fe794124575ef807057c77ada213\
            8ae4fa8d6c4de0398a14f3f00000000494830450221008949f0\
            cb400094ad2b5eb399d59d01c14d73d8fe6e96dfa7150deb38\
            8ab8935022079656090d7f6bac4c9a94e0aad311a4268e082a7\
            25f8aeae0573fb12ff866a5f01fffffffff01f0ca052a010000\
            01976a914cbc20a7664f2f69e5355aa427045bc15e7c6c77288\
            ac00000000",
  "complete" : true
}
```



```
> SIGNED_RAW_TX=01000000017b1eabe0209b1fe794124575ef807057c77ada[...]
```

Use the `signrawtransaction` RPC to sign the transaction created by `createrawtransaction` and save the returned “hex” raw format signed transaction to a shell variable.

Even though the transaction is now complete, the Bitcoin Core node we’re connected to doesn’t know anything about the transaction, nor does any other part of the network. We’ve created a spend, but we haven’t actually spent anything because we could simply unset the `$SIGNED_RAW_TX` variable to eliminate the transaction.

```
> bitcoin-cli -regtest sendrawtransaction $SIGNED_RAW_TX
c7736a0a0046d5a8cc61c8c3c2821d4d7517f5de2bc66a966011aaa79965ffba
```

Send the signed transaction to the connected node using the `sendrawtransaction` RPC. After accepting the transaction, the node would usually then broadcast it to other peers, but we’re not currently connected to other peers because we started in regtest mode.

```
## Bitcoin Core 0.10.1 and earlier
> bitcoin-cli -regtest setgenerate true 1

## Later versions of Bitcoin Core
> bitcoin-cli -regtest generate 1

> unset UTXO_TXID UTXO_VOUT NEW_ADDRESS RAW_TX SIGNED_RAW_TX
```

Generate a block to confirm the transaction and clear our shell variables.

Complex Raw Transaction

In this example, we’ll create a transaction with two inputs and two outputs. We’ll sign each of the inputs separately, as might happen if the two inputs belonged to different people who agreed to create a transaction together (such as a CoinJoin transaction).

```
> bitcoin-cli -regtest listunspent
[
  {
    "txid" : "263c018582731ff54dc72c7d67e858c002ae298835501d\
80200f05753de0edf0",
    "vout" : 0,
    "address" : "muhtvdmsnbQEPFuEmxcChX58fGvXaaUoVt",
    "scriptPubKey" : "76a9149ba386253ea698158b6d34802bb9b550\
f5ce36dd88ac",
    "amount" : 40.00000000,
    "confirmations" : 2
  },
  {
    "txid" : "263c018582731ff54dc72c7d67e858c002ae298835501d\
80200f05753de0edf0",
    "vout" : 1,
    "address" : "mvbnrCX3bg1cDRUu8pkecrvP6vQkSLDSou",
    "account" : "",
    "scriptPubKey" : "76a914a57414e5ffae9ef5074bacbe10a320bb\
2614e1f388ac",
    "amount" : 10.00000000,
    "confirmations" : 2
  }
]
```

```
},
{
  "txid" : "78203a8f6b529693759e1917a1b9f05670d036fbb12911\
0ed26be6a36de827f3",
  "vout" : 0,
  "address" : "n2KprMQm4z2vmZnPMENfbp2P1LLdAEFRjS",
  "scriptPubKey" : "210229688a74abd0d5ad3b06ddff36fa9cd8ed\
d181d97b9489a6adc40431fb56e1d8ac",
  "amount" : 50.00000000,
  "confirmations" : 101
},
{
  "txid" : "c7736a0a0046d5a8cc61c8c3c2821d4d7517f5de2bc66a\
966011aaa79965ffba",
  "vout" : 0,
  "address" : "mz6KvC4aoUeo6wSxtiVQTo7FDwPnkp6URG",
  "account" : "",
  "scriptPubKey" : "76a914cbc20a7664f2f69e5355aa427045bc15\
e7c6c77288ac",
  "amount" : 49.99990000,
  "confirmations" : 1
}
]

> UTXO1_TXID=78203a8f6b529693759e1917a1b9f05670d036fbb129110ed26[... ]
> UTXO1_VOUT=0
> UTXO1_ADDRESS=n2KprMQm4z2vmZnPMENfbp2P1LLdAEFRjS

> UTXO2_TXID=263c018582731ff54dc72c7d67e858c002ae298835501d80200[... ]
> UTXO2_VOUT=0
> UTXO2_ADDRESS=muhtvdmsnbQEPFuEmxcChX58fGvXaaUoVt
```

For our two inputs, we select two UTXOs by placing the txid and output index numbers (vouts) in shell variables. We also save the addresses corresponding to the public keys (hashed or unhashed) used in those transactions. We need the addresses so we can get the corresponding private keys from our wallet.


```
> bitcoin-cli -regtest dumpprivkey $UTXO1_ADDRESS
cSp57iWuu5APuzrPGyGc4PGUeCg23PjenZPBPoUs24HtJawccHPm

> bitcoin-cli -regtest dumpprivkey $UTXO2_ADDRESS
cT26DX6Ctco7pxaUptJujRfbMS2PJvdqiSMaGaoSktHyon8kQUSg

> UTXO1_PRIVATE_KEY=cSp57iWuu5APuzrPGyGc4PGUeCg23PjenZPBPoUs24Ht[... ]

> UTXO2_PRIVATE_KEY=cT26DX6Ctco7pxaUptJujRfbMS2PJvdqiSMaGaoSktHy[... ]
```

Use the `dumpprivkey` RPC to get the private keys corresponding to the public keys used in the two UTXOs out inputs we will be spending. We need the private keys so we can sign each of the inputs separately.

 **Warning:** Users should never manually manage private keys on mainnet. As dangerous as raw transactions are (see warnings above), making a mistake with a private key can be much worse—as in the case of a HD wallet [cross-generational key compromise](#). These examples are to help you learn, not for you to emulate on mainnet.

```
> bitcoin-cli -regtest getnewaddress
n4puhBEeEWD2VvjdRC9kQuX2abKxSCMNqN
> bitcoin-cli -regtest getnewaddress
n4LWXU59yM5MzQev7Jx7VNeq1BqZ85ZbLj

> NEW_ADDRESS1=n4puhBEeEWD2VvjdRC9kQuX2abKxSCMNqN
> NEW_ADDRESS2=n4LWXU59yM5MzQev7Jx7VNeq1BqZ85ZbLj
```


For our two outputs, get two new addresses.

```
## Outputs - inputs = transaction fee, so always double-check your math!
> bitcoin-cli -regtest createrawtransaction '' '
[
  {
    "txid": "'$UTXO1_TXID'",
    "vout": '$UTXO1_VOUT'
  },
  {
    "txid": "'$UTXO2_TXID'",
    "vout": '$UTXO2_VOUT'
  }
]
...

{
  "'$NEW_ADDRESS1'": 79.9999,
  "'$NEW_ADDRESS2'": 10
}'''

0100000002f327e86da3e66bd20e1129b1fb36d07056f0b9a117199e75939652\
6b8f3a20780000000000000000000000000000000000000000000000000000\
e8677d2cc74df51f738285013c2600000000000000000000000000000000\
1976a914ffb035781c3c69e076d48b60c3d38592e7ce06a788ac00ca9a3b0000\
00001976a914fa5139067622fd7e1e722a05c17c2bb7d5fd6df088ac00000000

> RAW_TX=0100000002f327e86da3e66bd20e1129b1fb36d07056f0b9a117199[...]
```

Create the raw transaction using `createrawtransaction` much the same as before, except now we have two inputs and two outputs.

```
> bitcoin-cli -regtest signrawtransaction $RAW_TX '[' ']' ''

[
  "'$UTXO1_PRIVATE_KEY'"
]'''

{
  "hex" : "0100000002f327e86da3e66bd20e1129b1fb36d07056f0b9a11\
7199e759396526b8f3a20780000000049483045022100fce442\
ec52aa2792efc27fd3ad0eaf7fa69f097fdcefab017ea56d179\
9b10b2102207a6ae3eb61e11ffaba0453f173d1792f1b7bb8e7\
422ea945101d68535c4b474801fffffffff0ede03d75050f208\
01d50358829ae02c058e8677d2cc74df51f738285013c260000\
00000000000000000000000000000000000000000000000000000000\
c69e076d48b60c3d38592e7ce06a788ac00ca9a3b00000000019\
76a914fa5139067622fd7e1e722a05c17c2bb7d5fd6df088ac0\
00000000",
  "complete" : false
}

> PARTLY_SIGNED_RAW_TX=0100000002f327e86da3e66bd20e1129b1fb36d07[...]
```

Signing the raw transaction with `signrawtransaction` gets more complicated as we now have three arguments:

1. The unsigned raw transaction.
2. An empty array. We don't do anything with this argument in this operation, but some valid JSON must be provided to get access to the later positional arguments.
3. The private key we want to use to sign one of the inputs.

The result is a raw transaction with only one input signed; the fact that the transaction isn’t fully signed is indicated by value of the `complete` JSON field. We save the incomplete, partly-signed raw transaction hex to a shell variable.

```
> bitcoin-cli -regtest signrawtransaction $PARTLY_SIGNED_RAW_TX '[]' ''

{
  "complete": false,
  "hex": "0100000002f327e86da3e66bd20e1129b1fb36d07056f0b9a11\
7199e759396526b8f3a20780000000049483045022100fce442\
ec52aa2792efc27fd3ad0eaf7fa69f097fdcefab017ea56d179\
9b10b2102207a6ae3eb61e11ffaba0453f173d1792f1b7bb8e7\
422ea945101d68535c4b474801fffffffff0ede03d75050f208\
01d50358829ae02c058e8677d2cc74df51f738285013c260000\
00006b483045022100b77f935ff366a6f3c2fdeb83589c79026\
5d43b3d2cf5e5f0047da56c36de75f40220707ceda75d8dcf2c\
caebc506f7293c3dcb910554560763d7659fb202f8ec324b012\
102240d7d3c7aad57b68aa0178f4c56f997d1bfab2ded3c2f94\
27686017c603a6d6fffffffff02f028d6dc010000001976a914f\
fb035781c3c69e076d48b60c3d38592e7ce06a788ac00ca9a3b\
000000001976a914fa5139067622fd7e1e722a05c17c2bb7d5f\
d6df088ac00000000",
  "complete": true
}
```

To sign the second input, we repeat the process we used to sign the first input using the second private key. Now that both inputs are signed, the `complete` result is *true*.


```
> unset PARTLY_SIGNED_RAW_TX RAW_TX NEW_ADDRESS1 [...]
```

Clean up the shell variables used. Unlike previous subsections, we’re not going to send this transaction to the connected node with `sendrawtransaction`. This will allow us to illustrate in the Offline Signing subsection below how to spend a transaction which is not yet in the block chain or memory pool.

Offline Signing

We will now spend the transaction created in the Complex Raw Transaction subsection above without sending it to the local node first. This is the same basic process used by wallet programs for offline signing—which generally means signing a transaction without access to the current UTXO set.

Offline signing is safe. However, in this example we will also be spending an output which is not part of the block chain because the transaction containing it has never been broadcast. That can be unsafe:

 **Warning:** Transactions which spend outputs from unconfirmed transactions are vulnerable to transaction malleability. Be sure to read about transaction malleability and adopt good practices before spending unconfirmed transactions on mainnet.

```
> OLD_SIGNED_RAW_TX=0100000002f327e86da3e66bd20e1129b1fb36d07056\
f0b9a117199e759396526b8f3a20780000000049483045022100fce442\
ec52aa2792efc27fd3ad0eaf7fa69f097fdcefab017ea56d1799b10b21\
02207a6ae3eb61e11ffaba0453f173d1792f1b7bb8e7422ea945101d68\
535c4b474801fffffffff0ede03d75050f20801d50358829ae02c058e8\
677d2cc74df51f738285013c26000000006b483045022100b77f935ff3\
66a6f3c2fdeb83589c790265d43b3d2cf5e5f0047da56c36de75f40220\
707ceda75d8dcf2ccaebc506f7293c3dcb910554560763d7659fb202f8\
```

```
ec324b012102240d7d3c7aad57b68aa0178f4c56f997d1bfab2ded3c2f\
9427686017c603a6d6fffffffff02f028d6dc010000001976a914ffb035\
781c3c69e076d48b60c3d38592e7ce06a788ac00ca9a3b000000001976\
a914fa5139067622fd7e1e722a05c17c2bb7d5fd6df088ac00000000
```

Put the previously signed (but not sent) transaction into a shell variable.

```
> bitcoin-cli -regtest decoderawtransaction $OLD_SIGNED_RAW_TX
{
  "txid" : "682cad881df69cb9df8f0c996ce96ecad758357ded2da03bad\
40cf18ffbb8e09",
  "version" : 1,
  "locktime" : 0,
  "vin" : [
    {
      "txid" : "78203a8f6b529693759e1917a1b9f05670d036fbb1\
29110ed26be6a36de827f3",
      "vout" : 0,
      "scriptSig" : {
        "asm" : "3045022100fce442ec52aa2792efc27fd3ad0ea\
f7fa69f097fdcefab017ea56d1799b10b210220\
7a6ae3eb61e11ffaba0453f173d1792f1b7bb8e\
7422ea945101d68535c4b474801",
        "hex" : "483045022100FCE442ec52aa2792efc27fd3ad0\
eaf7fa69f097fdcefab017ea56d1799b10b2102\
207a6ae3eb61e11ffaba0453f173d1792f1b7bb\
8e7422ea945101d68535c4b474801"
      },
      "sequence" : 4294967295
    },
    {
      "txid" : "263c018582731ff54dc72c7d67e858c002ae298835\
501d80200f05753de0edf0",
      "vout" : 0,
      "scriptSig" : {
        "asm" : "3045022100b77f935ff366a6f3c2fdeb83589c7\
90265d43b3d2cf5e5f0047da56c36de75f40220\
707ceda75d8dcf2ccaebc506f7293c3dcb91055\
4560763d7659fb202f8ec324b01\
02240d7d3c7aad57b68aa0178f4c56f997d1bfa\
b2ded3c2f9427686017c603a6d6",
        "hex" : "483045022100b77f935ff366a6f3c2fdeb83589\
c790265d43b3d2cf5e5f0047da56c36de75f402\
20707ceda75d8dcf2ccaebc506f7293c3dcb910\
554560763d7659fb202f8ec324b012102240d7d\
3c7aad57b68aa0178f4c56f997d1bfab2ded3c2\
f9427686017c603a6d6"
      },
      "sequence" : 4294967295
    }
  ],
  "vout" : [
    {
      "value" : 79.99990000,
      "n" : 0,
      "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 ffb035781c3c69e076d48\
b60c3d38592e7ce06a7 OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a914ffb035781c3c69e076d48b60c3d38592e\
7ce06a788ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
          "n4puhBEeEWD2VvjdRC9kQuX2abKxSCMNqN"
        ]
      }
    }
  ]
}
```

```
    ]
  },
  {
    "value" : 10.00000000,
    "n" : 1,
    "scriptPubKey" : {
      "asm" : "OP_DUP OP_HASH160 fa5139067622fd7e1e722\
a05c17c2bb7d5fd6df0 OP_EQUALVERIFY OP_CHECKSIG",
      "hex" : "76a914fa5139067622fd7e1e722a05c17c2bb7d\
5fd6df088ac",
      "reqSigs" : 1,
      "type" : "pubkeyhash",
      "addresses" : [
        "n4LWXU59yM5MzQev7Jx7VNeq1BqZ85ZbLj"
      ]
    }
  }
]
```

```
> UTXO_TXID=682cad881df69cb9df8f0c996ce96ecad758357ded2da03bad40[...]  
> UTXO_VOUT=1  
> UTXO_OUTPUT_SCRIPT=76a914fa5139067622fd7e1e722a05c17c2bb7d5fd6[...]
```

Decode the signed raw transaction so we can get its txid. Also, choose a specific one of its UTXOs to spend and save that UTXO’s output index number (vout) and hex pubkey script (scriptPubKey) into shell variables.

```
> bitcoin-cli -regtest getnewaddress  
mfdCHEFL2tW9eEUpizk7XLZJcnFM4hrp78  
  
> NEW_ADDRESS=mfdCHEFL2tW9eEUpizk7XLZJcnFM4hrp78
```

Get a new address to spend the satoshis to.

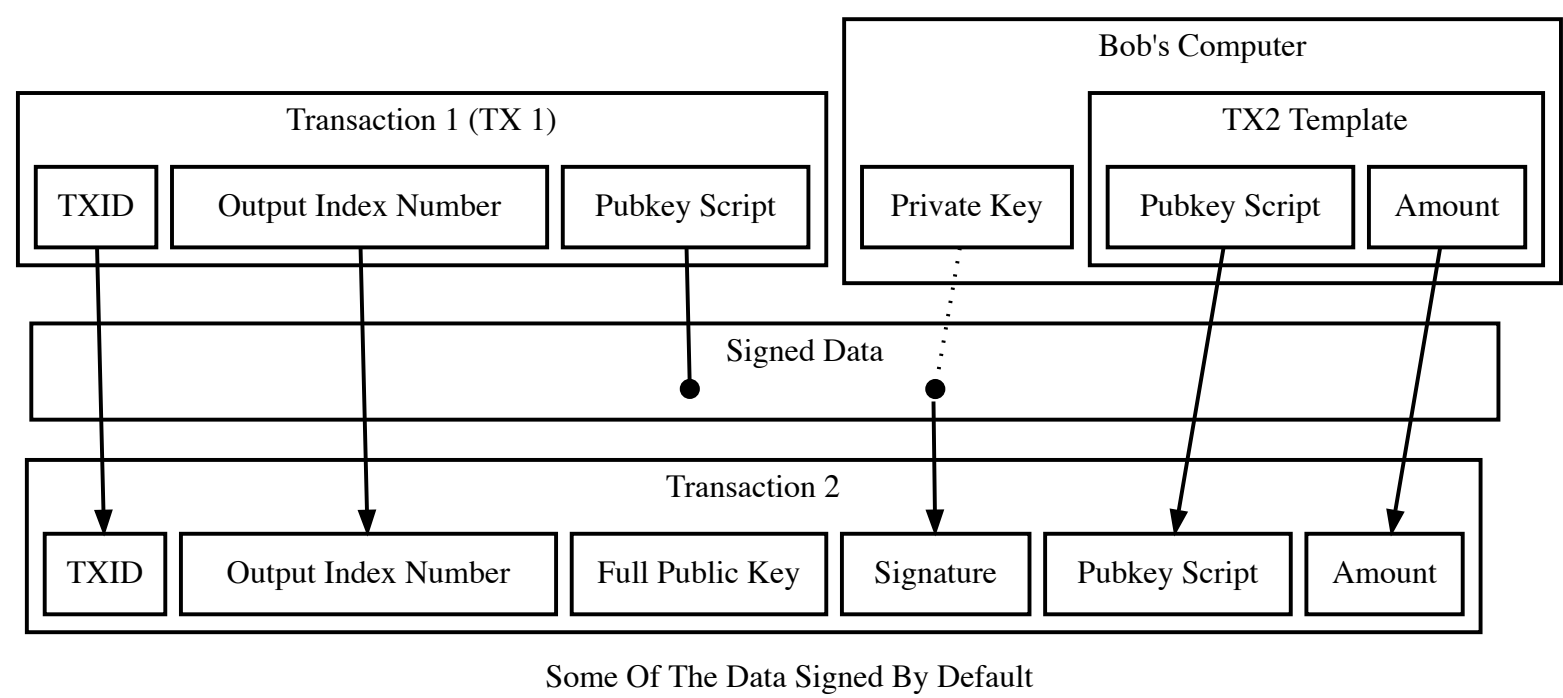
```
## Outputs - inputs = transaction fee, so always double-check your math!  
> bitcoin-cli -regtest createrawtransaction ''  
[  
  {  
    "txid": "'$UTXO_TXID'",  
    "vout": '$UTXO_VOUT'  
  }  
]  
,, , ,  
{  
  "'$NEW_ADDRESS'": 9.9999  
}''  
0100000001098ebbbff18cf40ad3ba02ded7d3558d7ca6ee96c990c8fd6b99cf6\  
1d88ad2c680100000000ffffffff01f0a29a3b000000001976a914012e2ba6a0\  
51c033b03d712ca2ea00a35eac1e7988ac00000000  
  
> RAW_TX=0100000001098ebbbff18cf40ad3ba02ded7d3558d7ca6ee96c990c8[...]
```

Create the raw transaction the same way we’ve done in the previous subsections.

```
> bitcoin-cli -regtest signrawtransaction $RAW_TX  
{  
  "hex" : "0100000001098ebbbff18cf40ad3ba02ded7d3558d7ca6ee\  
96c990c8fd6b99cf61d88ad2c680100000000ffffffff01\  
f0a29a3b000000001976a914012e2ba6a051c033b03d712ca2ea00a35eac1e7988ac00000000"
```

```
f0a29a3b00000001976a914012e2ba6a051c033b03d712\
ca2ea00a35eac1e7988ac00000000",
"complete" : false
}
```

Attempt to sign the raw transaction without any special arguments, the way we successfully signed the the raw transaction in the Simple Raw Transaction subsection. If you’ve read the [Transaction section](#) of the guide, you may know why the call fails and leaves the raw transaction hex unchanged.



As illustrated above, the data that gets signed includes the txid and vout from the previous transaction. That information is included in the `createrawtransaction` raw transaction. But the data that gets signed also includes the pubkey script from the previous transaction, even though it doesn’t appear in either the unsigned or signed transaction.

In the other raw transaction subsections above, the previous output was part of the UTXO set known to the wallet, so the wallet was able to use the txid and output index number to find the previous pubkey script and insert it automatically.

In this case, you’re spending an output which is unknown to the wallet, so it can’t automatically insert the previous pubkey script.

```
> bitcoin-cli -regtest signrawtransaction $RAW_TX ''

[
  {
    "txid": "'$UTXO_TXID'",
    "vout": '$UTXO_VOUT',
    "scriptPubKey": "'$UTXO_OUTPUT_SCRIPT'"
  }
]''

{
  "hex" : "0100000001098ebbf18cf40ad3ba02ded7d3558d7ca6ee96c9\
90c8fdfb99cf61d88ad2c68010000006b483045022100c3f92f\
b74bfa687d76ebe75a654510bb291b8aab6f89ded4fe26777c2\
eb233ad02207f779ce2a181cc4055cb0362aba7fd7a6f72d5db\
b9bd863f4faaf47d8d6c4b500121028e4e62d25760709806131\
b014e2572f7590e70be01f0ef16bfbd51ea5f389d4dfffffff\
01f0a29a3b000000001976a914012e2ba6a051c033b03d712ca\
2ea00a35eac1e7988ac00000000",
  "complete" : true
}

> SIGNED_RAW_TX=0100000001098ebbf18cf40ad3ba02ded7d3558d7ca6ee9[...]
```

Successfully sign the transaction by providing the previous pubkey script and other required input data.

This specific operation is typically what offline signing wallets do. The online wallet creates the raw transaction and gets the previous pubkey scripts for all the inputs. The user brings this information to the offline wallet. After displaying the transaction

details to the user, the offline wallet signs the transaction as we did above. The user takes the signed transaction back to the online wallet, which broadcasts it.

```
> bitcoin-cli -regtest sendrawtransaction $SIGNED_RAW_TX
error: {"code":-22,"message":"TX rejected"}
```

Attempt to broadcast the second transaction before we’ve broadcast the first transaction. The node rejects this attempt because the second transaction spends an output which is not a UTXO the node knows about.

```
> bitcoin-cli -regtest sendrawtransaction $OLD_SIGNED_RAW_TX
682cad881df69cb9df8f0c996ce96ecad758357ded2da03bad40cf18ffbb8e09
> bitcoin-cli -regtest sendrawtransaction $SIGNED_RAW_TX
67d53afala8167ca093d30be7fb9dcb8a64a5fdecacec9d93396330c47052c57
```

Broadcast the first transaction, which succeeds, and then broadcast the second transaction—which also now succeeds because the node now sees the UTXO.

```
> bitcoin-cli -regtest getrawmempool
[
  "67d53afala8167ca093d30be7fb9dcb8a64a5fdecacec9d93396330c47052c57",
  "682cad881df69cb9df8f0c996ce96ecad758357ded2da03bad40cf18ffbb8e09"
]
```

We have once again not generated an additional block, so the transactions above have not yet become part of the regtest block chain. However, they are part of the local node’s memory pool.

```
> unset OLD_SIGNED_RAW_TX SIGNED_RAW_TX RAW_TX [...]
```

Remove old shell variables.

P2SH Multisig

In this subsection, we will create a P2SH multisig address, spend satoshis to it, and then spend those satoshis from it to another address.

Creating a multisig address is easy. Multisig outputs have two parameters, the *minimum* number of signatures required (*m*) and the *number* of public keys to use to validate those signatures. This is called m-of-n, and in this case we’ll be using 2-of-3.

```
> bitcoin-cli -regtest getnewaddress
mhAXF4Eq7iRyvbYk1mpDVBiGdLP3YbY6Dm
> bitcoin-cli -regtest getnewaddress
moaCrnRfP5zzyhW8k65f6Rf2z5QpvJzSKe
> bitcoin-cli -regtest getnewaddress
mk2QpYatsKicvFVuTAQLBryyccRXMUaGHP

> NEW_ADDRESS1=mhAXF4Eq7iRyvbYk1mpDVBiGdLP3YbY6Dm
> NEW_ADDRESS2=moaCrnRfP5zzyhW8k65f6Rf2z5QpvJzSKe
> NEW_ADDRESS3=mk2QpYatsKicvFVuTAQLBryyccRXMUaGHP
```

Generate three new P2PKH addresses. P2PKH addresses cannot be used with the multisig redeem script created below. (Hashing

each public key is unnecessary anyway—all the public keys are protected by a hash when the redeem script is hashed.) However, Bitcoin Core uses addresses as a way to reference the underlying full (unhashed) public keys it knows about, so we get the three new addresses above in order to use their public keys.

Recall from the Guide that the hashed public keys used in addresses obfuscate the full public key, so you cannot give an address to another person or device as part of creating a typical multisig output or P2SH multisig redeem script. You must give them a full public key.

```
> bitcoin-cli -regtest validateaddress $NEW_ADDRESS3
{
  "isvalid" : true,
  "address" : "mk2QpYatsKicvFVuTAQLBryyccRXMUaGHP",
  "ismine" : true,
  "isscript" : false,
  "pubkey" : "029e03a901b85534ff1e92c43c74431f7ce72046060fcf7a\
             95c37e148f78c77255",
  "iscompressed" : true,
  "account" : ""
}

> NEW_ADDRESS3_PUBKEY=029e03a901b85534ff1e92c43c74431f7ce720[...]
```

Use the `validateaddress` RPC to display the full (unhashed) public key for one of the addresses. This is the information which will actually be included in the multisig redeem script. This is also the information you would give another person or device as part of creating a multisig output or P2SH multisig redeem script.


We save the address returned to a shell variable.

```
> bitcoin-cli -regtest createmultisig 2 ''
[
  '$NEW_ADDRESS1',
  '$NEW_ADDRESS2',
  '$NEW_ADDRESS3_PUBKEY'
]''
{
  "address" : "2N7NaqSKYQUeM8VNgBy8D9xQQbiA8yiJayk",
  "redeemScript" : "522103310188e911026cf18c3ce274e0ebb5f95b00\
                   7f230d8cb7d09879d96dbeab1aff210243930746e6ed6552e03359db521b\
                   088134652905bd2d1541fa9124303a41e95621029e03a901b85534ff1e92\
                   c43c74431f7ce72046060fcf7a95c37e148f78c7725553ae"
}

> P2SH_ADDRESS=2N7NaqSKYQUeM8VNgBy8D9xQQbiA8yiJayk
> P2SH_REDEEM_SCRIPT=522103310188e911026cf18c3ce274e0ebb5f95b007[...]
```

Use the `createmultisig` RPC with two arguments, the number (*n*) of signatures required and a list of addresses or public keys. Because P2PKH addresses can’t be used in the multisig redeem script created by this RPC, the only addresses which can be provided are those belonging to a public key in the wallet. In this case, we provide two addresses and one public key—all of which will be converted to public keys in the redeem script.

The P2SH address is returned along with the redeem script which must be provided when we spend satoshis sent to the P2SH address.

 **Warning:** You must not lose the redeem script, especially if you don’t have a record of which public keys you used to create the P2SH multisig address. You need the redeem script to spend any bitcoins sent to the P2SH address. If you lose the redeem script, you can recreate it by running the same command above, with the public keys listed in the same order. However, if you lose

both the redeem script and even one of the public keys, you will never be able to spend satoshis sent to that P2SH address.

Neither the address nor the redeem script are stored in the wallet when you use `createmultisig`. To store them in the wallet, use the `addmultisigaddress` RPC instead. If you add an address to the wallet, you should also make a new backup.

```
> bitcoin-cli -regtest sendtoaddress $P2SH_ADDRESS 10.00
7278d7d030f042ebe633732b512bcb31fff14a697675a1fe1884db139876e175

> UTXO_TXID=7278d7d030f042ebe633732b512bcb31fff14a697675a1fe1884[...]
```

Paying the P2SH multisig address with Bitcoin Core is as simple as paying a more common P2PKH address. Here we use the same command (but different variable) we used in the Simple Spending subsection. As before, this command automatically selects an UTXO, creates a change output to a new one of our P2PKH addresses if necessary, and pays a transaction fee if necessary.

We save that txid to a shell variable as the txid of the UTXO we plan to spend next.

```
> bitcoin-cli -regtest getrawtransaction $UTXO_TXID 1
{
  "hex" : "0100000001f0ede03d75050f20801d50358829ae02c058e8677\
d2cc74df51f738285013c26010000006a47304402203c375959\
2bf608ab79c01596c4a417f3110dd6eb776270337e575cdafc6\
99af20220317ef140d596cc255a4067df8125db7f349ad94521\
2e9264a87fa8d777151937012102a92913b70f9fb15a7ea5c42\
df44637f0de26e2dad97d6d54957690b94cf2cd05fffffffff01\
00ca9a3b0000000017a9149af61346ce0aa2dffcf697352b4b7\
04c84dcbaff8700000000",
  "txid" : "7278d7d030f042ebe633732b512bcb31fff14a697675a1fe18\
84db139876e175",
  "version" : 1,
  "locktime" : 0,
  "vin" : [
    {
      "txid" : "263c018582731ff54dc72c7d67e858c002ae298835\
501d80200f05753de0edf0",
      "vout" : 1,
      "scriptSig" : {
        "asm" : "304402203c3759592bf608ab79c01596c4a417f\
3110dd6eb776270337e575cdafc699af2022031\
7ef140d596cc255a4067df8125db7f349ad9452\
12e9264a87fa8d77715193701\
02a92913b70f9fb15a7ea5c42df44637f0de26e\
2dad97d6d54957690b94cf2cd05",
        "hex" : "47304402203c3759592bf608ab79c01596c4a41\
7f3110dd6eb776270337e575cdafc699af20220\
317ef140d596cc255a4067df8125db7f349ad94\
5212e9264a87fa8d777151937012102a92913b7\
0f9fb15a7ea5c42df44637f0de26e2dad97d6d5\
4957690b94cf2cd05"
      },
      "sequence" : 4294967295
    }
  ],
  "vout" : [
    {
      "value" : 10.00000000,
      "n" : 0,
      "scriptPubKey" : {
        "asm" : "OP_HASH160 9af61346ce0aa2dffcf697352b4b\
704c84dcbaff OP_EQUAL",
        "hex" : "a9149af61346ce0aa2dffcf697352b4b704c84d\
"
```

```
        "cbaff87",
        "reqSigs" : 1,
        "type" : "scripthash",
        "addresses" : [
            "2N7NaqSKYQUeM8VNgBy8D9xQQbiA8yiJayk"
        ]
    }
}

]
}

> UTXO_VOUT=0
> UTXO_OUTPUT_SCRIPT=a9149af61346ce0aa2dfcf697352b4b704c84dcbaff87
```

We use the `getrawtransaction` RPC with the optional second argument (*true*) to get the decoded transaction we just created with `spendtoaddress`. We choose one of the outputs to be our UTXO and get its output index number (vout) and pubkey script (scriptPubKey).

```
> bitcoin-cli -regtest getnewaddress
mxCNLtKxzgjjg8yyNHeuFSXvxCvagkWdfGU

> NEW_ADDRESS4=mxCNLtKxzgjjg8yyNHeuFSXvxCvagkWdfGU
```

We generate a new P2PKH address to use in the output we’re about to create.

```
## Outputs - inputs = transaction fee, so always double-check your math!
> bitcoin-cli -regtest createrawtransaction '' '
[
  {
    "txid": "'$UTXO_TXID'",
    "vout": '$UTXO_VOUT'
  }
]
...

{
  "'$NEW_ADDRESS4'": 9.998
}'''

010000000175e1769813db8418fea17576694af1ff31cb2b512b7333e6eb42f0\
30d0d778720000000000000000000000000000000000000000000000000000\
e38f25ead28817df7929c06fe847ee88ac00000000


> RAW_TX=010000000175e1769813db8418fea17576694af1ff31cb2b512b733[...]
```

We generate the raw transaction the same way we did in the Simple Raw Transaction subsection.

```
> bitcoin-cli -regtest dumpprivkey $NEW_ADDRESS1
cVinshabsALz5Wg4tGDtBuqEGq4i6WCKWXRQdM8RFxLbALvNSHw7
> bitcoin-cli -regtest dumpprivkey $NEW_ADDRESS3
cNmbnwwGzEghMMelvBwH34DFHShEj5bcXD1QpFRPHgG9Mj1xc5hq

> NEW_ADDRESS1_PRIVATE_KEY=cVinshabsALz5Wg4tGDtBuqEGq4i6WCKWXRQd[...]
> NEW_ADDRESS3_PRIVATE_KEY=cNmbnwwGzEghMMelvBwH34DFHShEj5bcXD1Qp[...]
```

We get the private keys for two of the public keys we used to create the transaction, the same way we got private keys in the Complex Raw Transaction subsection. Recall that we created a 2-of-3 multisig pubkey script, so signatures from two private keys are needed.

 **Reminder:** Users should never manually manage private keys on mainnet. See the warning in the [complex raw transaction section](#).

```
> bitcoin-cli -regtest signrawtransaction $RAW_TX ''

[
  {
    "txid": "'$UTXO_TXID'",
    "vout": '$UTXO_VOUT',
    "scriptPubKey": "'$UTXO_OUTPUT_SCRIPT'",
    "redeemScript": "'$P2SH_REDEEM_SCRIPT'"
  }
]
...

[
  "'$NEW_ADDRESS1_PRIVATE_KEY'"
]'

{
  "hex" : "01000000175e1769813db8418fea17576694af1ff31cb2b512\
b7333e6eb42f030d0d7787200000000b5004830450221008d5e\
c57d362ff6ef6602e4e756ef1bdeee12bd5c5c72697ef1455b3\
79c90531002202ef3ea04dfbeda043395e5bc701e4878c15baa\
b9c6ba5808eb3d04c91f641a0c014c69522103310188e911026\
cf18c3ce274e0ebb5f95b007f230d8cb7d09879d96dbeab1aff\
210243930746e6ed6552e03359db521b088134652905bd2d154\
1fa9124303a41e95621029e03a901b85534ff1e92c43c74431f\
7ce72046060fcf7a95c37e148f78c7725553aefffffffff01c0b\
c973b000000001976a914b6f64f5bf3e38f25ead28817df7929\
c06fe847ee88ac00000000",
  "complete" : false
}

> PARTLY_SIGNED_RAW_TX=010000000175e1769813db8418fea17576694af1f[...]
```

We make the first signature. The input argument (JSON object) takes the additional redeem script parameter so that it can append the redeem script to the signature script after the two signatures.

```
> bitcoin-cli -regtest signrawtransaction $PARTLY_SIGNED_RAW_TX ''

[
  {
    "txid": "'$UTXO_TXID'",
    "vout": '$UTXO_VOUT',
    "scriptPubKey": "'$UTXO_OUTPUT_SCRIPT'",
    "redeemScript": "'$P2SH_REDEEM_SCRIPT'"
  }
]
...

[
  "'$NEW_ADDRESS3_PRIVATE_KEY'"
]'

{
  "hex" : "010000000175e1769813db8418fea17576694af1ff31cb2b512\
b7333e6eb42f030d0d7787200000000fdfd0000483045022100\
8d5ec57d362ff6ef6602e4e756ef1bdeee12bd5c5c72697ef14\
55b379c90531002202ef3ea04dfbeda043395e5bc701e4878c1\
5baab9c6ba5808eb3d04c91f641a0c0147304402200bd8c62b9\
38e02094021e481b149fd5e366a212cb823187149799a68cfa7\
652002203b52120c5cf25ceab5f0a6b5cdb8eca0fd2f386316c\
9721177b75ddca82a4ae8014c69522103310188e911026cf18c\
3ce274e0ebb5f95b007f230d8cb7d09879d96dbeab1aff21024\
3930746e6ed6552e03359db521b088134652905bd2d1541fa91\
24303a41e95621029e03a901b85534ff1e92c43c74431f7ce72\
046060fcf7a95c37e148f78c7725553aefffffffff01c0bc973b\
```

```
000000001976a914b6f64f5bf3e38f25ead28817df7929c06fe\
847ee88ac00000000",
  "complete" : true
}

> SIGNED_RAW_TX=010000000175e1769813db8418fea17576694af1ff31cb2b[...]
```

The `signrawtransaction` call used here is nearly identical to the one used above. The only difference is the private key used. Now that the two required signatures have been provided, the transaction is marked as complete.

```
> bitcoin-cli -regtest sendrawtransaction $SIGNED_RAW_TX
430a4cee3a55efb04cbb8718713cab18dea7f2521039aa660ffb5aae14ff3f50
```

We send the transaction spending the P2SH multisig output to the local node, which accepts it.

Payment Processing

Payment Protocol

To request payment using the payment protocol, you use an extended (but backwards-compatible) `bitcoin:` URI. For example:

```
bitcoin:mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN\
?amount=0.10\
&label=Example+Merchant\
&message=Order+of+flowers+%26+chocolates\
&r=https://example.com/pay.php/invoice%3Dda39a3ee
```

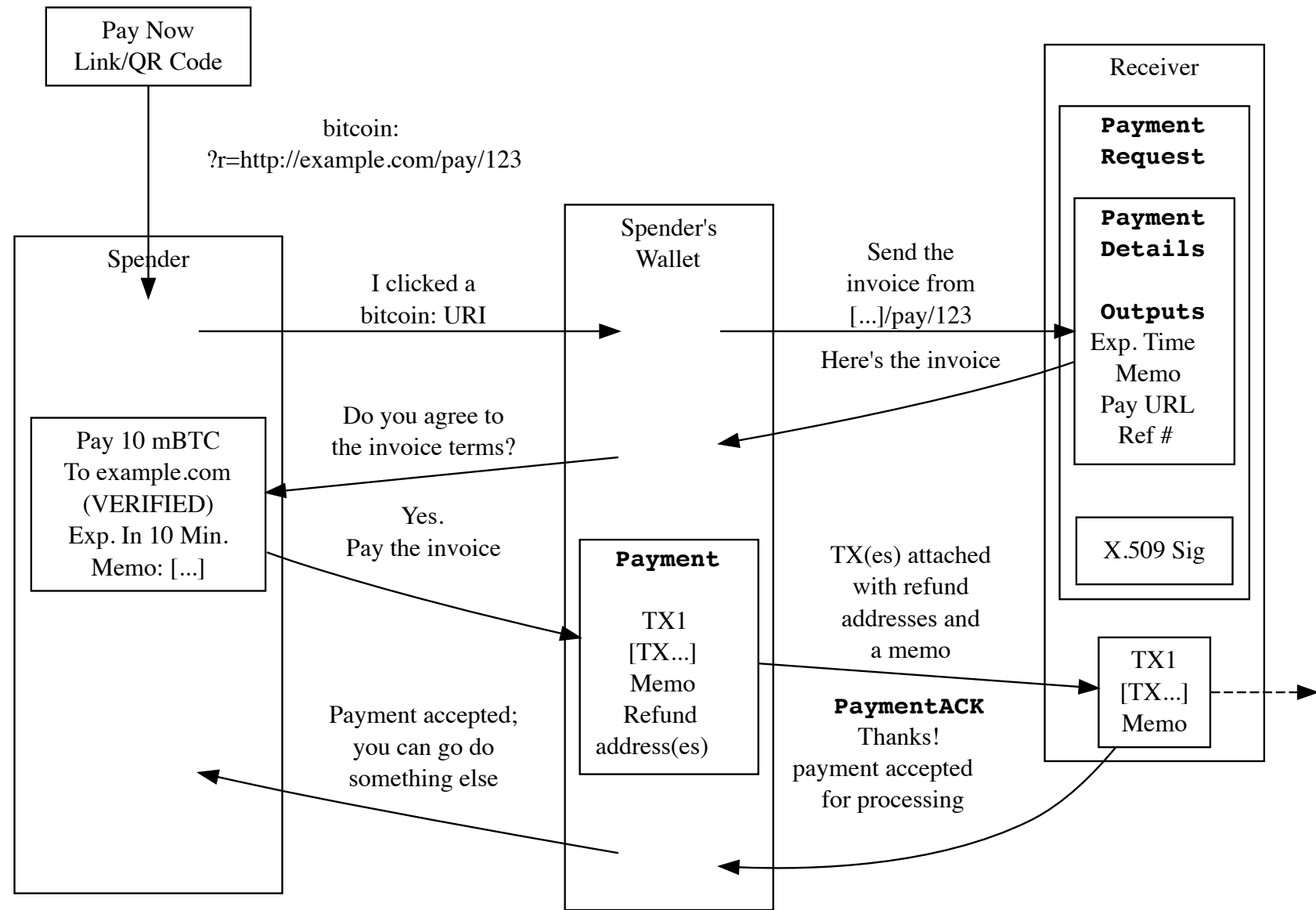
The browser, QR code reader, or other program processing the URI opens the spender’s Bitcoin wallet program on the URI. If the wallet program is aware of the payment protocol, it accesses the URL specified in the `r` parameter, which should provide it with a serialized `PaymentRequest` served with the [MIME](#) type `application/bitcoin-paymentrequest`.

Resource: Gavin Andresen’s [Payment Request Generator](#) generates custom example URIs and payment requests for use with testnet.

PaymentRequest & PaymentDetails

The `PaymentRequest` is created with data structures built using Google’s Protocol Buffers. BIP70 describes these data structures in the non-sequential way they’re defined in the payment request protocol buffer code, but the text below will describe them in a more linear order using a simple (but functional) Python CGI program. (For brevity and clarity, many normal CGI best practices are not used in this program.)

The full sequence of events is illustrated below, starting with the spender clicking a `bitcoin:` URI or scanning a `bitcoin:` QR code.



The Bitcoin Payment Protocol As Described In BIP70

For the script to use the protocol buffer, you will need a copy of Google’s Protocol Buffer compiler (`protoc`), which is available in most modern Linux package managers and [directly from Google](#). Non-Google protocol buffer compilers are available for a variety of programming languages. You will also need a copy of the PaymentRequest [Protocol Buffer description](#) from the Bitcoin Core source code.

Initialization Code

With the Python code generated by `protoc`, we can start our simple CGI program.

```
#!/usr/bin/env python

## This is the code generated by protoc --python_out=./ paymentrequest.proto
from paymentrequest_pb2 import *

## Load some functions
from time import time
from sys import stdout
from OpenSSL.crypto import FILETYPE_PEM, load_privatekey, sign

## Copy three of the classes created by protoc into objects we can use
details = PaymentDetails()
request = PaymentRequest()
x509 = X509Certificates()
```

The startup code above is quite simple, requiring nothing but the epoch (Unix date) time function, the standard out file descriptor, a few functions from the OpenSSL library, and the data structures and functions created by `protoc`.

Configuration Code

Next, we’ll set configuration settings which will typically only change when the receiver wants to do something differently. The code pushes a few settings into the `request` (PaymentRequest) and `details` (PaymentDetails) objects. When we serialize them, `PaymentDetails` will be contained within the PaymentRequest.

```
## SSL Signature method
request.pki_type = "x509+sha256"  ## Default: none

## Mainnet or testnet?
details.network = "test"  ## Default: main

## Postback URL
details.payment_url = "https://example.com/pay.py"

## PaymentDetails version number
request.payment_details_version = 1  ## Default: 1

## Certificate chain
x509.certificate.append(file("/etc/apache2/example.com-cert.der", "r").read())
#x509.certificate.append(file("/some/intermediate/cert.der", "r").read())

## Load private SSL key into memory for signing later
priv_key = "/etc/apache2/example.com-key.pem"
pw = "test"  ## Key password
private_key = load_privatekey(FILETYPE_PEM, file(priv_key, "r").read(), pw)
```

Each line is described below.

```
request.pki_type = "x509+sha256"  ## Default: none
```

`pki_type`: (optional) tell the receiving wallet program what **Public-Key Infrastructure** (PKI) type you’re using to cryptographically sign your PaymentRequest so that it can’t be modified by a man-in-the-middle attack.

If you don’t want to sign the PaymentRequest, you can choose a `pki_type` of `none` (the default).

If you do choose the sign the PaymentRequest, you currently have two options defined by BIP70: `x509+sha1` and `x509+sha256`. Both options use the X.509 certificate system, the same system used for HTTP Secure (HTTPS). To use either option, you will need a certificate signed by a certificate authority or one of their intermediaries. (A self-signed certificate will not work.)

Each wallet program may choose which certificate authorities to trust, but it’s likely that they’ll trust whatever certificate authorities their operating system trusts. If the wallet program doesn’t have a full operating system, as might be the case for small hardware wallets, BIP70 suggests they use the [Mozilla Root Certificate Store](#). In general, if a certificate works in your web browser when you connect to your webserver, it will work for your PaymentRequests.

```
details.network = "test"  ## Default: main
```

`network`: (optional) tell the spender’s wallet program what Bitcoin network you’re using; BIP70 defines “main” for mainnet (actual payments) and “test” for testnet (like mainnet, but fake satoshis are used). If the wallet program doesn’t run on the network you indicate, it will reject the PaymentRequest.

```
details.payment_url = "https://example.com/pay.py"
```

`payment_url`: (required) tell the spender’s wallet program where to send the Payment message (described later). This can be a static URL, as in this example, or a variable URL such as `https://example.com/pay.py?invoice=123.` It should usually be an

HTTPS address to prevent man-in-the-middle attacks from modifying the message.

```
request.payment_details_version = 1  ## Default: 1
```

`payment_details_version`: (optional) tell the spender’s wallet program what version of the PaymentDetails you’re using. As of this writing, the only version is version 1.

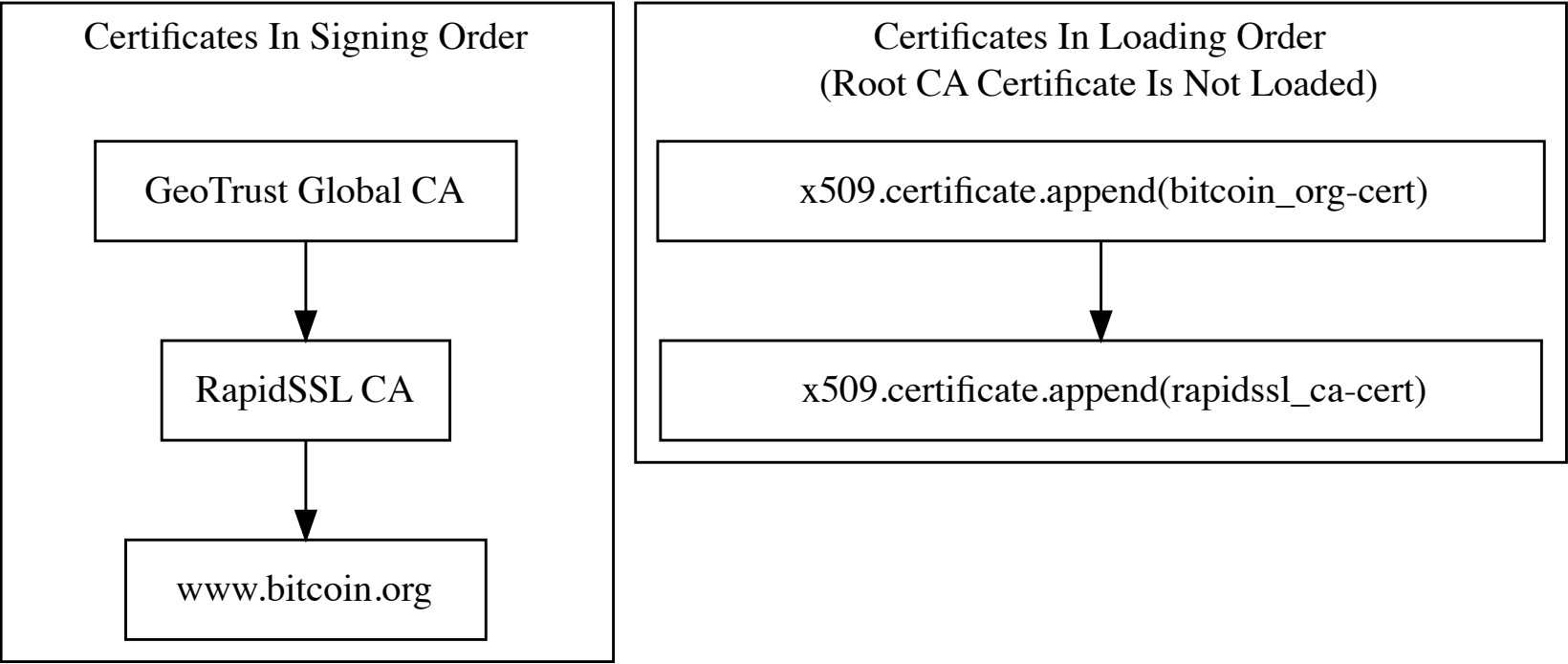
```
## This is the pubkey/certificate corresponding to the private SSL key
## that we'll use to sign:
x509.certificate.append(file("/etc/apache2/example.com-cert.der", "r").read())
```

`x509certificates`: (required for signed PaymentRequests) you must provide the public SSL key/certificate corresponding to the private SSL key you’ll use to sign the PaymentRequest. The certificate must be in ASN.1/DER format.

```
## If the pubkey/cert above didn't have the signature of a root
## certificate authority, we'd then append the intermediate certificate
## which signed it:
#x509.certificate.append(file("/some/intermediate/cert.der", "r").read())
```

You must also provide any intermediate certificates necessary to link your certificate to the root certificate of a certificate authority trusted by the spender’s software, such as a certificate from the Mozilla root store.

The certificates must be provided in a specific order—the same order used by Apache’s `SSLCertificateFile` directive and other server software. The figure below shows the **certificate chain** of the `www.bitcoin.org` X.509 certificate and how each certificate (except the root certificate) would be loaded into the `X509Certificates` protocol buffer message.



(Certificates are loaded into X509Certificates() in reverse signing order)

Example Certificate Loading Order For Payment Requests From Bitcoin.org

To be specific, the first certificate provided must be the X.509 certificate corresponding to the private SSL key which will make the signature, called the **leaf certificate**. Any **intermediate certificates** necessary to link that signed public SSL key to the **root certificate** (the certificate authority) are attached separately, with each certificate in DER format bearing the signature of the certificate that follows it all the way to (but not including) the root certificate.

```
priv_key = "/etc/apache2/example.com-key.pem"
pw = "test"  ## Key password
private_key = load_privatekey(FILETYPE_PEM, file(priv_key, "r").read(), pw)
```

(Required for signed PaymentRequests) you will need a private SSL key in a format your SSL library supports (DER format is not required). In this program, we’ll load it from a PEM file. (Embedding your passphrase in your CGI code, as done here, is obviously a bad idea in real life.)

The private SSL key will not be transmitted with your request. We’re only loading it into memory here so we can use it to sign the request later.

Code Variables

Now let’s look at the variables your CGI program will likely set for each payment.

```
## Amount of the request
amount = 10000000  ## In satoshis

## P2PKH pubkey hash
pubkey_hash = "2b14950b8d31620c6cc923c5408a701b1ec0a020"
## P2PKH pubkey script entered as hex and converted to binary
# OP_DUP OP_HASH160 <push 20 bytes> <pubKey hash> OP_EQUALVERIFY OP_CHECKSIG
#   76          a9              14          <pubKey hash>          88              ac
hex_script = "76" + "a9" + "14" + pubkey_hash + "88" + "ac"
serialized_script = hex_script.decode("hex")

## Load amount and pubkey script into PaymentDetails
details.outputs.add(amount = amount, script = serialized_script)

## Memo to display to the spender
details.memo = "Flowers & chocolates"

## Data which should be returned to you with the payment
details.merchant_data = "Invoice #123"
```

Each line is described below.

```
amount = 10000000  ## In satoshis (=100 mBTC)
```

amount: (optional) the **amount** you want the spender to pay. You’ll probably get this value from your shopping cart application or fiat-to-BTC exchange rate conversion tool. If you leave the amount blank, the wallet program will prompt the spender how much to pay (which can be useful for donations).

```
pubkey_hash = "2b14950b8d31620c6cc923c5408a701b1ec0a020"
# OP_DUP OP_HASH160 <push 20 bytes> <pubKey hash> OP_EQUALVERIFY OP_CHECKSIG
#   76          a9              14          <pubKey hash>          88              ac
hex_script = "76" + "a9" + "14" + pubkey_hash + "88" + "ac"
serialized_script = hex_script.decode("hex")
```

script: (required) You must specify the pubkey script you want the spender to pay—any valid pubkey script is acceptable. In this example, we’ll request payment to a P2PKH pubkey script.

First we get a pubkey hash. The hash above is the hash form of the address used in the URI examples throughout this section, mjSk1Ny9spzU2fouzYgLqGUD8U41iR35QN.

Next, we plug that hash into the standard P2PKH pubkey script using hex, as illustrated by the code comments.

Finally, we convert the pubkey script from hex into its serialized form.

```
details.outputs.add(amount = amount, script = serialized_script)
```

`outputs`: (required) add the pubkey script and (optional) amount to the PaymentDetails outputs array.

It’s possible to specify multiple `scripts` and `amounts` as part of a merge avoidance strategy, described later in the [Merge Avoidance subsection](#). However, effective merge avoidance is not possible under the base BIP70 rules in which the spender pays each `script` the exact amount specified by its paired `amount`. If the amounts are omitted from all `amount`/`script` pairs, the spender will be prompted to choose an amount to pay.

```
details.memo = "Flowers & chocolates"
```

`memo`: (optional) add a memo which will be displayed to the spender as plain UTF-8 text. Embedded HTML or other markup will not be processed.

```
details.merchant_data = "Invoice #123"
```

`merchant_data`: (optional) add arbitrary data which should be sent back to the receiver when the invoice is paid. You can use this to track your invoices, although you can more reliably track payments by generating a unique address for each payment and then tracking when it gets paid.

The `memo` field and the `merchant_data` field can be arbitrarily long, but if you make them too long, you’ll run into the 50,000 byte limit on the entire PaymentRequest, which includes the often several kilobytes given over to storing the certificate chain. As will be described in a later subsection, the `memo` field can be used by the spender after payment as part of a cryptographically-proven receipt.

Derivable Data

Next, let’s look at some information your CGI program can automatically derive.

```
## Request creation time
details.time = int(time()) ## Current epoch (Unix) time

## Request expiration time
details.expires = int(time()) + 60 * 10 ## 10 minutes from now

## PaymentDetails is complete; serialize it and store it in PaymentRequest
request.serialized_payment_details = details.SerializeToString()

## Serialized certificate chain
request.pki_data = x509.SerializeToString()

## Initialize signature field so we can sign the full PaymentRequest
request.signature = ""

## Sign PaymentRequest
request.signature = sign(private_key, request.SerializeToString(), "sha256")
```

Each line is described below.

--

```
details.time = int(time()) ## Current epoch (Unix) time
```

`time`: (required) PaymentRequests must indicate when they were created in number of seconds elapsed since 1970-01-01T00:00 UTC (Unix epoch time format).

```
details.expires = int(time()) + 60 * 10 ## 10 minutes from now
```

`expires`: (optional) the PaymentRequest may also set an `expires` time after which they’re no longer valid. You probably want to give receivers the ability to configure the expiration time delta; here we used the reasonable choice of 10 minutes. If this request is tied to an order total based on a fiat-to-satoshis exchange rate, you probably want to base this on a delta from the time you got the exchange rate.

```
request.serialized_payment_details = details.SerializeToString()
```

`serialized_payment_details`: (required) we’ve now set everything we need to create the PaymentDetails, so we’ll use the `SerializeToString` function from the protocol buffer code to store the PaymentDetails in the appropriate field of the PaymentRequest.

```
request.pki_data = x509.SerializeToString()
```

`pki_data`: (required for signed PaymentRequests) serialize the certificate chain **PKI data** and store it in the PaymentRequest

```
request.signature = ""
```

We’ve filled out everything in the PaymentRequest except the signature, but before we sign it, we have to initialize the signature field by setting it to a zero-byte placeholder.

```
request.signature = sign(private_key, request.SerializeToString(), "sha256")
```

`signature`: (required for signed PaymentRequests) now we make the **signature** by signing the completed and serialized PaymentRequest. We’ll use the private key we stored in memory in the configuration section and the same hashing formula we specified in `pki_type` (sha256 in this case)

Output Code

Now that we have PaymentRequest all filled out, we can serialize it and send it along with the HTTP headers, as shown in the code below.

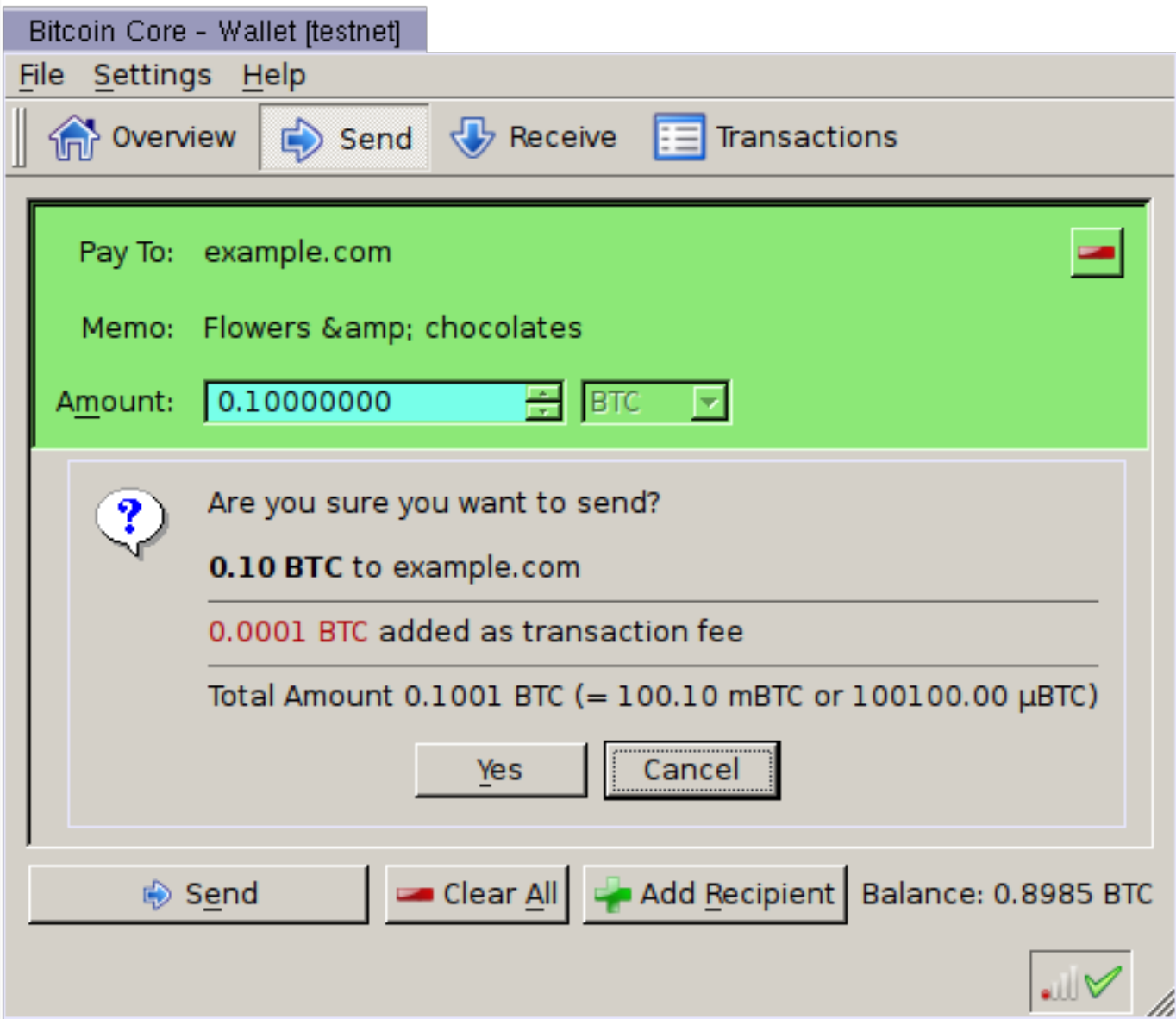
```
print "Content-Type: application/bitcoin-paymentrequest"
print "Content-Transfer-Encoding: binary"
print ""
```

(Required) BIP71 defines the content types for PaymentRequests, Payments, and PaymentACKs.


```
file.write(stdout, request.SerializeToString())
```

`request`: (required) now, to finish, we just dump out the serialized `PaymentRequest` (which contains the serialized `PaymentDetails`). The serialized data is in binary, so we can't use Python's `print()` because it would add an extraneous newline.

The following screenshot shows how the authenticated `PaymentDetails` created by the program above appears in the GUI from Bitcoin Core 0.9.



P2P Network

Creating A Bloom Filter

In this section, we'll use variable names that correspond to the field names in the `filterload` [message documentation](#). Each code block precedes the paragraph describing it.

```
#!/usr/bin/env python

BYTES_MAX = 36000
FUNCS_MAX = 50

nFlags = 0
```

We start by setting some maximum values defined in BIP37: the maximum number of bytes allowed in a filter and the maximum number of hash functions used to hash each piece of data. We also set `nFlags` to zero, indicating we don't want the remote node to update the filter for us. (We won't use `nFlags` again in the sample program, but real programs will need to use it.)

```
n = 1
```


p = 0.0001

We define the number (n) of elements we plan to insert into the filter and the false positive rate (p) we want to help protect our privacy. For this example, we will set *n* to one element and *p* to a rate of 1-in-10,000 to produce a small and precise filter for illustration purposes. In actual use, your filters will probably be much larger.

```
from math import log
nFilterBytes = int(min((-1 / log(2)**2 * n * log(p)) / 8, BYTES_MAX))
nHashFuncs = int(min(nFilterBytes * 8 / n * log(2), FUNCS_MAX))

from bitarray import bitarray # from pypi.python.org/pypi/bitarray
vData = nFilterBytes * 8 * bitarray('0', endian="little")
```

Using the formula described in BIP37, we calculate the ideal size of the filter (in bytes) and the ideal number of hash functions to use. Both are truncated down to the nearest whole number and both are also constrained to the maximum values we defined earlier. The results of this particular fixed computation are 2 filter bytes and 11 hash functions. We then use *nFilterBytes* to create a little-endian bit array of the appropriate size.

nTweak = 0

We also should choose a value for *nTweak*. In this case, we’ll simply use zero.

```
import pyhash # from https://github.com/flier/pyfasthash
murmur3 = pyhash.murmur3_32()

def bloom_hash(nHashNum, data):
    seed = (nHashNum * 0xfba4c795 + nTweak) & 0xffffffff
    return( murmur3(data, seed=seed) % (nFilterBytes * 8) )
```

We setup our hash function template using the formula and 0xfba4c795 constant set in BIP37. Note that we limit the size of the seed to four bytes and that we’re returning the result of the hash modulo the size of the filter in bits.

```
data_to_hash = "019f5b01d4195ecbc9398fbf3c3b1fa9" \
               + "bb3183301d7a1fb3bd174fcfa40a2b65"
data_to_hash = data_to_hash.decode("hex")
```

For the data to add to the filter, we’re adding a TXID. Note that the TXID is in internal byte order.

```
print "
print "nHashNum    nIndex    Filter    0123456789abcdef"
print "~~~~~      ~~~~~      ~~~~~      ~~~~~~"
for nHashNum in range(nHashFuncs):
    nIndex = bloom_hash(nHashNum, data_to_hash)

    ## Set the bit at nIndex to 1
    vData[nIndex] = True

    ## Debug: print current state
    print '          {0:2}          {1:2}          {2}          {3}'.format(
        nHashNum,
        hex(int(nIndex)),
        vData.tobytes().encode("hex"),
        vData.to01())
```

```
)

print
print "Bloom filter:", vData.tobytes().encode("hex")
```

Now we use the hash function template to run a slightly different hash function for *nHashFuncs* times. The result of each function being run on the transaction is used as an index number: the bit at that index is set to 1. We can see this in the printed debugging output:

Filter (As Bits)			
nHashNum	nIndex	Filter	0123456789abcdef
~~~~~	~~~~~	~~~~~	~~~~~
0	0x7	8000	0000000100000000
1	0x9	8002	0000000101000000
2	0xa	8006	0000000101100000
3	0x2	8406	0010000101100000
4	0xb	840e	0010000101110000
5	0x5	a40e	0010010101110000
6	0x0	a50e	1010010101110000
7	0x8	a50f	1010010111110000
8	0x5	a50f	1010010111110000
9	0x8	a50f	1010010111110000
10	0x4	b50f	1010110111110000

Bloom filter: b50f

Notice that in iterations 8 and 9, the filter did not change because the corresponding bit was already set in a previous iteration (5 and 7, respectively). This is a normal part of bloom filter operation.

We only added one element to the filter above, but we could repeat the process with additional elements and continue to add them to the same filter. (To maintain the same false-positive rate, you would need a larger filter size as computed earlier.)

Note: for a more optimized Python implementation with fewer external dependencies, see [python-bitcoinlib’s](#) bloom filter module which is based directly on Bitcoin Core’s C++ implementation.

Using the `filterload` message format, the complete filter created above would be the binary form of the annotated hexdump shown below:

```
02 ..... Filter bytes: 2
b50f ..... Filter: 1010 1101 1111 0000
0b000000 ... nHashFuncs: 11
00000000 ... nTweak: 0/none
00 ..... nFlags: BLOOM_UPDATE_NONE
```

## Evaluating A Bloom Filter

Using a bloom filter to find matching data is nearly identical to constructing a bloom filter—except that at each step we check to see if the calculated index bit is set in the existing filter.

```
vData = bytearray(endian='little')
vData.frombytes("b50f".decode("hex"))
nHashFuncs = 11
nTweak = 0
nFlags = 0
```

Using the bloom filter created above, we import its various parameters. Note, as indicated in the section above, we won’t actually use *nFlags* to update the filter.

```
def contains(nHashFuncs, data_to_hash):
    for nHashNum in range(nHashFuncs):
        ## bloom_hash as defined in previous section
        nIndex = bloom_hash(nHashNum, data_to_hash)

        if vData[nIndex] != True:
            print "MATCH FAILURE: Index {0} not set in {1}".format(
                hex(int(nIndex)),
                vData.to01()
            )
            return False
```

We define a function to check an element against the provided filter. When checking whether the filter might contain an element, we test to see whether a particular bit in the filter is already set to 1 (if it isn’t, the match fails).

```
## Test 1: Same TXID as previously added to filter
data_to_hash = "019f5b01d4195ecbc9398fbf3c3b1fa9" \
               + "bb3183301d7a1fb3bd174fcfa40a2b65"

data_to_hash = data_to_hash.decode("hex")
contains(nHashFuncs, data_to_hash)
```

Testing the filter against the data element we previously added, we get no output (indicating a possible match). Recall that bloom filters have a zero false negative rate—so they should always match the inserted elements.

```
## Test 2: Arbitrary string
data_to_hash = "1/10,000 chance this ASCII string will match"
contains(nHashFuncs, data_to_hash)
```

Testing the filter against an arbitrary element, we get the failure output below. Note: we created the filter with a 1-in-10,000 false positive rate (which was rounded up somewhat when we truncated), so it was possible this arbitrary string would’ve matched the filter anyway. It is not possible to set a bloom filter to a false positive rate of zero, so your program will always have to deal with false positives. The output below shows us that one of the hash functions returned an index number of 0x06, but that bit wasn’t set in the filter, causing the match failure:

```
MATCH FAILURE: Index 0x6 not set in 1010110111110000
```

## Retrieving A MerkleBlock

For the `merkleblock` message documentation on the reference page, an actual merkle block was retrieved from the network and manually processed. This section walks through each step of the process, demonstrating basic network communication and merkle block processing.

```
#!/usr/bin/env python

from time import sleep
from hashlib import sha256
```

```
import struct
import sys

network_string = "f9beb4d9".decode("hex") # Mainnet

def send(msg,payload):
    ## Command is ASCII text, null padded to 12 bytes
    command = msg + ( ( 12 - len(msg) ) * "\00" )

    ## Payload length is a uint32_t
    payload_raw = payload.decode("hex")
    payload_len = struct.pack("I", len(payload_raw))

    ## Checksum is first 4 bytes of SHA256(SHA256(<payload>))
    checksum = sha256(sha256(payload_raw).digest()).digest()[ :4]

    sys.stdout.write(
        network_string
        + command
        + payload_len
        + checksum
        + payload_raw
    )
    sys.stdout.flush()
```

To connect to the P2P network, the trivial Python function above was developed to compute message headers and send payloads decoded from hex.

```
## Create a version message
send("version",
    "71110100" # ..... Protocol Version: 70001
    + "0000000000000000" # ..... Services: Headers Only (SPV)
    + "c6925e5400000000" # ..... Time: 1415484102
    + "00000000000000000000000000000000"
    + "0000ffff7f000001208d" # ..... Receiver IP Address/Port
    + "00000000000000000000000000000000"
    + "0000ffff7f000001208d" # ..... Sender IP Address/Port
    + "0000000000000000" # ..... Nonce (not used here)
    + "1b" # ..... Bytes in version string
    + "2f426974636f696e2e6f726720457861"
    + "6d706c653a302e392e332f" # ..... Version string
    + "93050500" # ..... Starting block height: 329107
    + "00" # ..... Relay transactions: false
)
```

Peers on the network will not accept any requests until you send them a `version` message. The receiving node will reply with their `version` message and a `verack` message.

```
sleep(1)
send("verack", "")
```

We’re not going to validate their `version` message with this simple script, but we will sleep a short bit and send back our own `verack` message as if we had accepted their `version` message.

```
send("filterload",
    "02" ..... Filter bytes: 2
    "b50f" ..... Filter: 1010 1101 1111 0000
    "0b000000" ... nHashFuncs: 11
```

```
"0000000" ... nTweak: 0/none
"00" ..... nFlags: BLOOM_UPDATE_NONE
)
```

We set a bloom filter with the `filterload` message. This filter is described in the two preceeding sections.

```
send( "getdata",
      "01" # ..... Number of inventories: 1
+ "03000000" # ..... Inventory type: filtered block
+ "a4deb66c0d726b0aefb03ed51be407fb"
+ "ad7331c6e8f9eef231b7000000000000" # ... Block header hash
)
```

We request a merkle block for transactions matching our filter, completing our script.

To run the script, we simply pipe it to the Unix `netcat` [command](#) or one of its many clones, one of which is available for practically any platform. For example, with the original netcat and using hexdump (`hd`) to display the output:

```
## Connect to the Bitcoin Core peer running on localhost
python get-merkle.py | nc localhost 8333 | hd
```

Part of the response is shown in the section below.

## Parsing A MerkleBlock

In the section above, we retrieved a merkle block from the network; now we will parse it. Most of the block header has been omitted. For a more complete hexdump, see the example in the `merkleblock` [message section](#).

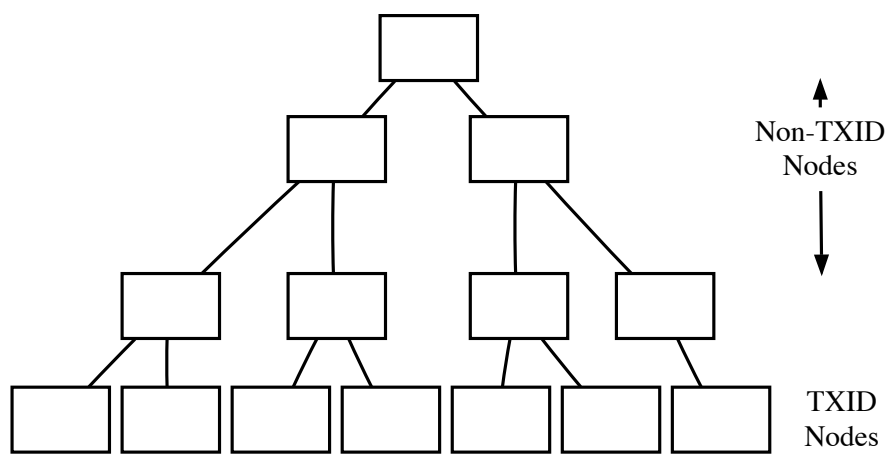
```
7f16c5962e8bd963659c793ce370d95f
093bc7e367117b3c30c1f8fdd0d97287 ... Merkle root

07000000 ..... Transaction count: 7
04 ..... Hash count: 4

3612262624047ee87660be1a707519a4
43b1c1ce3d248cbfc6c15870f6c5daa2 ... Hash #1
019f5b01d4195ecbc9398fbf3c3b1fa9
bb3183301d7a1fb3bd174fcfa40a2b65 ... Hash #2
41ed70551dd7e841883ab8f0b16bf041
76b7d1480e4f0af9f3d4c3595768d068 ... Hash #3
20d2a7bc994987302e5blac80fc425fe
25f8b63169ea78e68fbaaefa59379bbf ... Hash #4





01 ..... Flag bytes: 1
1d ..... Flags: 1 0 1 1 1 0 0 0
```

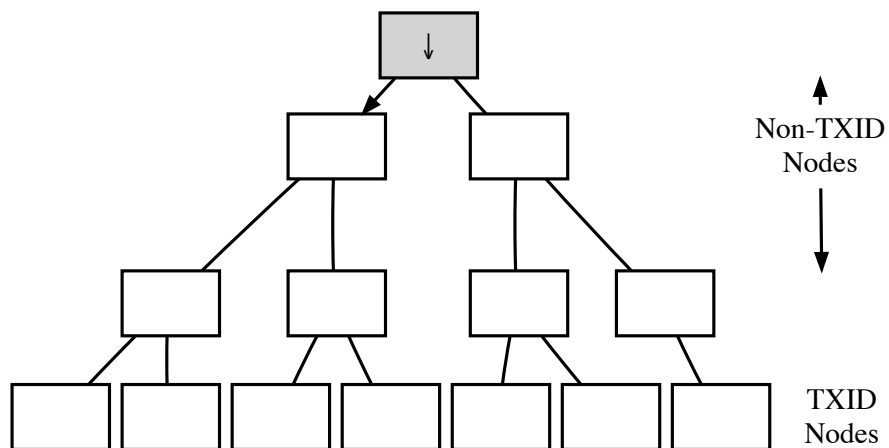
We parse the above `merkleblock` message using the following instructions. Each illustration is described in the paragraph below it.



Flags	1	0	1	1	1	0	0	0
Hashes	H1	H2	H3	H4				


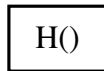


Legend

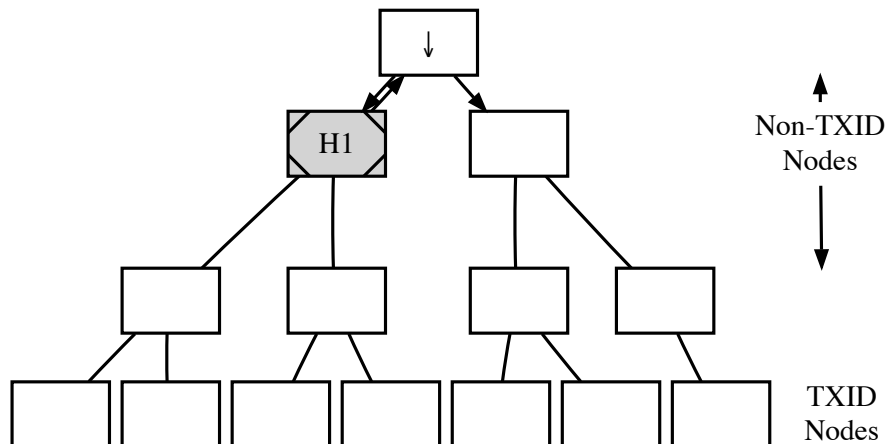
			
Wait For Child	Hash Com- puted	Hash From List	TXID Filter Match



Flags	0	1	1	1	0	0	0
Hashes	H1	H2	H3	H4			


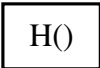


Legend

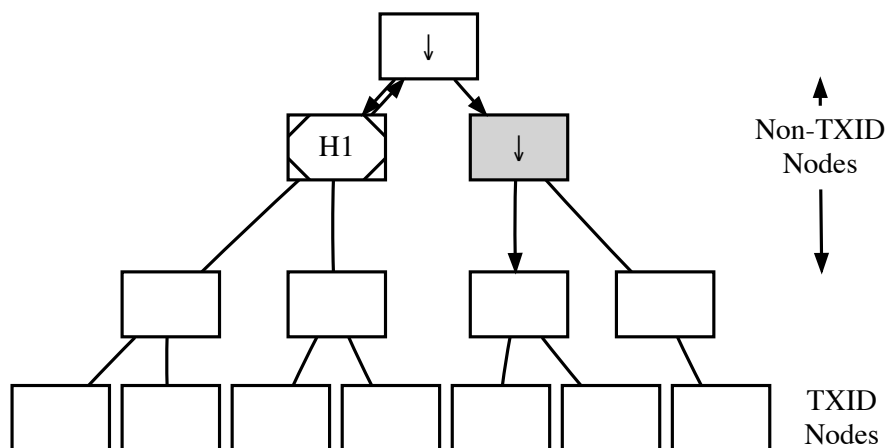
			
Wait For Child	Hash Com- puted	Hash From List	TXID Filter Match



Flags	1	1	1	0	0	0
Hashes		H2		H3		H4

Legend

			
Wait For Child	Hash Com- puted	Hash From List	TXID Filter Match



The third flag in the example is another 1 on another non-TXID node, so we descend into its left child.



