The Bitcoin Client

Bitcoin Core - The Reference Implementation, aka Satoshi Client

You can download the Reference Client, also known as *Bitcoin Core* from bitcoin.org. The reference client implements all aspects of the bitcoin system, including wallets, a transaction verification engine with a full copy of the entire transaction ledger (blockchain) and a full network node in the peer-to-peer bitcoin network.

Go to http://bitcoin.org/en/choose-your-wallet and select "Bitcoin Core" to download the reference client. Depending on your operating system, you will download an executable installer. For Windows, this is either a ZIP archive or an EXE executable. For Mac OS it is DMG disk image. Linux versions include a PPA package for Ubuntu or a TAR.GZ archive.

Figure 1. Bitcoin - Choose A Bitcoin Client

Bitcoin Core - Running the client for the first time

If you download an installable package, such as an EXE, DMG or PPA, you can install it the same way as any application on your operating system. For Windows, run the EXE and follow the step-by-step instructions. For Mac OS, launch the DMG and drag the Bitcoin-QT icon into your Applications folder. For Ubuntu, double-click on the PPA in your File Explorer and it will open the package manager to install the package. Once you have completed installation you should have a new application "Bitcoin-Qt" in your application list. Double-click on the icon to start the bitcoin client.

The first time you run Bitcoin Core it will start downloading the blockchain, a process that may take several days. Leave it running in the background until it displays "Synchronized" and no longer shows "Out of sync" next to the balance.

Tip Bitcoin Core keeps a full copy of the transaction ledger (blockchain), with every transaction that has ever occurred on the bitcoin network since its inception in 2009. This dataset is several gigabytes in size (approximately 16GB in late 2015) and is downloaded incrementally over several days. The client will not be able to process transactions or update account balances until the full blockchain dataset is downloaded. During that time, the client will display "Out of sync" next to the account balances and show "Synchronizing" in the footer. Make sure you have enough disk space, bandwidth and time to complete the initial synchronization.

Figure 2. Bitcoin Core - The Graphical User Interface, during the blockchain initialization

Bitcoin Core - Compiling the client from the source code

For developers, there is also the option to download the full source code as a ZIP archive or by cloning the authoritative source repository from Github. Go to https://github.com/bitcoin/bitcoin and select "Download ZIP" from the sidebar. Alternatively, use the git command line to create a local copy of the source code on your system. In the example below, we are cloning the source code from a unix-like command-line, in Linux or Mac OS:

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 31864, done.
remote: Compressing objects: 100% (12007/12007), done.
remote: Total 31864 (delta 24480), reused 26530 (delta 19621)
Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.
Resolving deltas: 100% (24480/24480), done.
$
```

Tip The instructions and resulting output may vary from version to version. Follow the documentation that comes with the code even if it differs from the instructions you see here and don't be surprised if the output displayed on your screen is slightly different from the examples here.

When the git cloning operation has completed, you will have a complete local copy of the source code repository in the directory *bitcoin*. Change to this directory by typing `cd bitcoin` at the prompt:

```
$ cd bitcoin
```

By default, the local copy will be synchronized with the most recent code which may be an unstable or "beta" version of bitcoin. Before compiling the code, we want to select a specific version by checking out a release *tag*. This will synchronize the local copy with a specific snapshot of the code repository identified by a keyword tag. Tags are used by the developers to mark specific releases of the code by version number. First, to find the available tags, we use the `git tag` command:

```
$ git tag
v0.1.5
v0.1.6test1
v0.2.0
v0.2.10
v0.2.11
v0.2.12

[... many more tags ...]

v0.8.4rc2
v0.8.5
v0.8.6
v0.8.6rc1
v0.9.0rc1
```

The list of tags shows all the released versions of bitcoin. By convention, *release candidates*, which are intended for testing, have the suffix "rc". Stable releases that can be run on production systems have no suffix. From the list above, we select the highest version release, which at this time is v0.9.0rc1. To synchronize the local code with this version, we use the `git checkout` command:

```
$ git checkout v0.9.0rc1
Note: checking out 'v0.9.0rc1'.

HEAD is now at 15ec451... Merge pull request #3605
$
```

The source code includes documentation, which can be found in a number of files. Review the main documentation located in README.md in the bitcoin directory by typing `more README.md` at the prompt and using the space bar to progress to the next page. In this chapter we will build the command-line bitcoin client, also known as `bitcoind` on Linux. Review the instructions for compiling the bitcoind command-line client on your platform by typing `more doc/build-unix.md`. Alternative instructions for Mac OSX and Windows can be found in the doc directory, as `build-os.md` or `build-msw.md` respectively.

Carefully review the build pre-requisites which are in the first part of the build documentation. These are libraries that must be present on your system before you can begin to compile bitcoin. If these pre-requisites are missing the build process will fail with an error. If this happens because you missed a pre-requisite, you can install it and then resume the build process from where you left off. Assuming the pre-requisites are installed, we start the build process by generating a set of build scripts using the `autogen.sh` script.

Tip The bitcoind build process was changed to use the autogen/configure/make system starting with version 0.9. Older versions use a simple Makefile and work slightly differently from the example below. Follow the instructions for the version you want to compile. The autogen/configure/make introduced in 0.9 is likely to be the build system used for all future versions of the code and is the system demonstrated in the examples below.

```
$ ./autogen.sh
configure.ac:12: installing `src/build-aux/config.guess'
configure.ac:12: installing `src/build-aux/config.sub'
configure.ac:37: installing `src/build-aux/install-sh'
configure.ac:37: installing `src/build-aux/missing'
src/Makefile.am: installing `src/build-aux/depcomp'
$
```

The `autogen.sh` script creates a set of automatic configuration scripts that will interrogate your system to discover the correct settings and ensure you have all the necessary libraries to compile the code. The most important of these is the `configure` script that offers a number of different options to customize the build process. Type `./configure --help` to see the various options:

```
$ ./configure --help

`configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit

[... many more options and variables are displayed below ...]

Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]

[... more options ...]

Use these variables to override the choices made by `configure' or to help
it to find libraries and programs with nonstandard names/locations.

Report bugs to <info@bitcoin.org>.

$
```

The `configure` script allows you to enable or disable certain features of bitcoind through the use of the `--enable-FEATURE` and `--disable-FEATURE` flags, where `FEATURE` is replaced by the feature name, as listed in the help output above. In this chapter, we will build the bitcoind client with all the default features. We won't be using the configuration flags, but you should review them to understand what optional features are part of the client. Next, we run the `configure` script to automatically discover all the necessary libraries and create a customized build script for our system:

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes

[... many more system features are tested ...]

configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/test/Makefile
config.status: creating src/qt/Makefile
config.status: creating src/qt/test/Makefile
config.status: creating share/setup.nsi
config.status: creating share/qt/Info.plist
config.status: creating qa/pull-tester/run-bitcoind-for-test.sh
config.status: creating qa/pull-tester/build-tests.sh
config.status: creating src/bitcoin-config.h
config.status: executing depfiles commands
$
```

If all goes well, the `configure` command will end by creating the customized build scripts that will allow us to compile bitcoind. If there are any missing libraries or errors, the `configure` command will terminate with an error instead of creating the build scripts as shown above. If an error occurs, it is most likely a missing or incompatible library. Review the build documentation again and make sure you install the missing pre-requisites. Then run `configure` again and see if that fixes the error. Next, we will compile the source code, a process that can take up to an hour to complete. During the compilation process you should see output every few seconds or every few minutes, or an error if something goes wrong. The compilation process can be resumed at any time if interrupted. Type `make` to start compiling:

```
$ make
Making all in src
make[1]: Entering directory `/home/ubuntu/bitcoin/src'
make  all-recursive
make[2]: Entering directory `/home/ubuntu/bitcoin/src'
Making all in .
make[3]: Entering directory `/home/ubuntu/bitcoin/src'
  CXX    addrman.o
  CXX    alert.o
  CXX    rpcserver.o
  CXX    bloom.o
  CXX    chainparams.o

[... many more compilation messages follow ...]
```

```
  CXX    test_bitcoin-wallet_tests.o
  CXX    test_bitcoin-rpc_wallet_tests.o
  CXXLD  test_bitcoin
make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[2]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Entering directory `/home/ubuntu/bitcoin'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/ubuntu/bitcoin'
$
```

If all goes well, bitcoind is now compiled. The final step is to install the bitcoind executable into the system path using the `make` command:

```
$ sudo make install
Making install in src
Making install in .
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'
Making install in test
make  install-am
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c test_bitcoin '/usr/local/bin'
$
```

We can confirm that bitcoin is correctly installed, as follows:

```
$ which bitcoind
/usr/local/bin/bitcoind
```

The default installation of bitcoind puts it in `/usr/local/bin`. When we first run bitcoind it will remind us to create a configuration file with a strong password for the JSON-RPC interface. We run it by typing `bitcoind` into the terminal:

```
$ bitcoind
Error: To use the "-server" option, you must set a rpcpassword in the configuration file:
/home/ubuntu/.bitcoin/bitcoin.conf
It is recommended you use the following random password:
rpcuser=bitcoinrpc
rpcpassword=2XA4DzKhCbtZXsBQRRHDEwEY2nM6M4H9Tx5dFjoAVVbK
(you do not need to remember this password)
The username and password MUST NOT be the same.
If the file does not exist, create it with owner-readable-only file permissions.
It is also recommended to set alertnotify so you are notified of problems;
for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

Edit the configuration file in your preferred editor and set the parameters, replacing the password with a strong password as recommended by bitcoind. Do **not** use the password shown below. Create a file inside the `.bitcoin` directory so that it is named `.bitcoin/bitcoin.conf` and enter a username and password:

```
rpcuser=bitcoinrpc
rpcpassword=2XA4DzKhCbtZXsBQRRHDEwEY2nM6M4H9Tx5dFjoAVVbK
```

Using bitcoind from the command line

The reference client bitcoind offers a number of commands that can be run from the command line. These are the same commands as those offered via the JSON-RPC API, so the command line allows us to experiment interactively with the capabilities that are also available programmatically. To start, we can invoke the `help` command to see a list of the available bitcoin commands:

Bitcoind command list

```
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
decodescript "hex"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
encryptwallet "passphrase"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddednodeinfo dns ( "node" )
getaddressesbyaccount "account"
getbalance ( "account" minconf )
getbestblockhash
getblock "hash" ( verbose )
getblockcount
getblockhash index
getblocktemplate ( "jsonrequestobject" )
getconnectioncount
getdifficulty
getgenerate
gethashespersec
getinfo
getmininginfo
getnettotals
getnetworkhashps ( blocks height )
getnewaddress ( "account" )
getpeerinfo
getrawchangeaddress
getrawmempool ( verbose )
getrawtransaction "txid" ( verbose )
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid"
gettxout "txid" n ( includemempool )
gettxoutsetinfo
getunconfirmedbalance
getwork ( "data" )
help ( "command" )
importprivkey "bitcoinprivkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf )
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty )
listreceivedbyaddress ( minconf includeempty )
listsinceblock ( "blockhash" target-confirmations )
listtransactions ( "account" count from )
listunspent ( minconf maxconf  ["address",...] )
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
ping
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "comment-to" )
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )
sendrawtransaction "hexstring" ( allowhighfees )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )
setaccount "bitcoinaddress" "account"
setgenerate generate ( genproclimit )
settxfee amount
signmessage "bitcoinaddress" "message"
signrawtransaction "hexstring" ( [{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...] ["privatekey1",...] sighashtype )
stop
submitblock "hexdata" ( "jsonparameterobject" )
validateaddress "bitcoinaddress"
verifychain ( checklevel numblocks )
verifymessage "bitcoinaddress" "signature" "message"
```

Running bitcoind

Commands: -daemon, getinfo

Now, run the bitcoin client. The first time you run it the bitcoin blockchain will be rebuilt. This is a multi-gigabyte file and will take on average 2 days to download in full. You can shorten the blockchain initialization time by downloading a partial copy of the blockchain using bittorrent from `http://sourceforge.net/projects/bitcoin/files/Bitcoin/blockchain/`.

Run bitcoind in the background with the option `-daemon`:

```
$ bitcoind -daemon
$
Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
Using OpenSSL version OpenSSL 1.0.1c 10 May 2012
Default data directory /home/bitcoin/.bitcoin
Using data directory /bitcoin/
Using at most 4 connections (1024 file descriptors available)
init message: Verifying wallet...
dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log
Bound to [::]:8333
Bound to 0.0.0.0:8333
init message: Loading block index...
Opening LevelDB in /bitcoin/blocks/index
Opened LevelDB successfully
Opening LevelDB in /bitcoin/chainstate
Opened LevelDB successfully

[... more startup messages ...]
```

Bitcoin's `getinfo` command shows us basic information about the status of the bitcoin network node, the wallet and the blockchain database:

```
$ bitcoind getinfo
{
    "version" : 90000,
    "protocolversion" : 70002,
    "walletversion" : 60000,
    "balance" : 0.00000000,
    "blocks" : 286216,
    "timeoffset" : -72,
    "connections" : 4,
    "proxy" : "",
    "difficulty" : 2621404453.06461525,
    "testnet" : false,
    "keypoololdest" : 1374553827,
    "keypoolsize" : 101,
    "paytxfee" : 0.00000000,
    "errors" : ""
}
```

The data is returned in JavaScript Object Notation (JSON), a format which can easily be "consumed" by all programming languages but is also quite human-readable. Among this data we see the version of the bitcoin software client (90000), protocol (70002) and wallet version (60000). We see the current balance contained in the wallet, which is zero. We see the current block height, showing us how many blocks are known to this client, 286216. We also see various statistics about the bitcoin network and the settings related to this client. We will explore these settings in more detail in the rest of this chapter.

Tip It will take some time, perhaps more than a day, for the bitcoind client to "catch up" to the current blockchain height as it downloads blocks from other bitcoin clients. You can check its current progress using getinfo to see the number of known blocks.

Wallet setup and encryption

Commands: bitcoind encryptwallet, walletpassphrase

Before we proceed with creating keys and other commands, we will first encrypt the wallet with a password. For this example, we use the `encryptwallet` command with the password "foo". Obviously, replace "foo" with a strong and complex password!

```
$ bitcoind encryptwallet foo
wallet encrypted; Bitcoin server stopping, restart to run with encrypted wallet. The keypool has been flushed, you need to make a new backup.
$
```

We can verify the wallet has been encrypted by running `getinfo` again. This time you will notice a new entry `unlocked_until` which is a counter showing how long the wallet decryption password will be stored in memory, keeping the wallet unlocked. At first this will be set to zero, meaning the wallet is locked:

```
$ bitcoind getinfo
{
    "version" : 90000,

[... other information...]

    "unlocked_until" : 0,
    "errors" : ""
}
$
```

To unlock the wallet, we issue the `walletpassphrase` command that takes two parameters, the password and a number of seconds until the wallet is locked again automatically (a time counter):

```
$ bitcoind walletpassphrase foo 360
$
```

Confirm the wallet is unlocked and see the timeout by running `getinfo` again:

```
$ bitcoind getinfo
{
    "version" : 90000,

[... other information ...]

    "unlocked_until" : 1392580909,
    "errors" : ""
}
```

Wallet backup, plain-text dump and restore

Commands: backupwallet, importwallet, dumpwallet

Next, we will practice creating a wallet backup file and then restoring the wallet from the backup file. Use the `backupwallet` command to backup, providing the file name as the parameter. Here we backup the wallet to the file `wallet.backup`:

```
$ bitcoind backupwallet wallet.backup
$
```

Now, to restore the backup file, use the `importwallet` command. If your wallet is locked, you will need to unlock it first (see `walletpassphrase` above) in order to import the backup file:

```
$ bitcoind importwallet wallet.backup
$
```

The `dumpwallet` command can be used to dump the wallet into a text file that is human-readable:

```
$ bitcoind dumpwallet wallet.txt
$ more wallet.txt
# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
# * Created on 2014-02- 8dT20:34:55Z
# * Best block at time of backup was 286234 (0000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),
#   mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstHRsqP26QKJCzLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z change=1 # addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F
Kz3dVz7R6mUpXzdZy4gJEVZxXJwA15f198eVui4CUivXotzLBDKY 2013-07- 4dT04:30:27Z change=1 # addr=17oJds8kaN8LP8kuAkWTco6ZM7BGXFC3gk
[... many more keys ...]

$
```

Wallet addresses and receiving transactions

Commands: getnewaddress, getreceivedbyaddress, listtransactions, getaddressesbyaccount, getbalance

The bitcoin reference client maintains a pool of addresses, the size of which is displayed by `keypoolsize` when you use the command `getinfo`. These addresses are generated automatically and can then be used as public receiving addresses or change addresses. To get one of these addresses, you can use the `getnewaddress` command:

```
$ bitcoind getnewaddress
1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

Now, we can use this address to send a small amount of bitcoin to our bitcoind wallet from an external wallet (assuming you have some bitcoin in an exchange, web wallet or other bitcoind wallet held elsewhere). For this example, we will send 50 milliBits (0.050 bitcoin) to the address returned above.

We can now query the bitcoind client for the amount received by this address, and specify how many confirmations are required before an amount is counted in that balance. For this example, we will specify zero confirmations. A few seconds after sending the bitcoin from another wallet, we will see it reflected in the wallet. We use `getreceivedbyaddress` with the address and the number of confirmations set to zero (0):

```
$ bitcoind getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0
0.05000000
```

If we omit the zero from the end of this command, we will only see the amounts that have at least `minconf` confirmations, where `minconf` is the setting for the minimum number of confirmations before a transaction is listed in the balance. The `minconf` setting is specified in the bitcoind configuration file. Since the transaction sending this bitcoin was only sent in the last few seconds, it has still not confirmed and therefore we will see it list a zero balance:

```
$ bitcoind getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
0.00000000
```

The transactions received by the entire wallet can also be displayed using the `listtransactions` command:

```
$  bitcoind listtransactions
[
    {
        "account" : "",
        "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
        "category" : "receive",
        "amount" : 0.05000000,
        "confirmations" : 0,
        "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
        "time" : 1392660908,
        "timereceived" : 1392660908
    }
]
```

We can list all addresses in the entire wallet using the `getaddressesbyaccount` command:

```
$ bitcoind getaddressesbyaccount ""
[
    "1LQoTPYy1TyERbNV4zZbhEmgyfAipC6eqL",
    "17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",
    "1FvRHWhHBBZA8cGRRsGiAeqEzUmjJkJQWR",
    "1NVJK3JsL41BF1KyxrUyJW5XHjunjfp2jz",
    "14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",
    "1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",
    "15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",
    "1Q3q6taTsUiv3mMemEuQQJ9sGLEGaSjo81",
    "1HoSiTg8sb16oE6SrmazQEwcGEv8obv9ns",
    "13fE8BGhBvnoy68yZKuWJ2hheYKovSDjqM",
    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
    "1KHUmVfCJteJ21LmRXHSpPoe23rXKifAb2",
    "1LqJZz1D9yHxG4cLkdujnqG5jNNGmPeAMD"
]
```

Finally, the command `getbalance` will show the total balance of the wallet, adding up all transactions confirmed with at least `minconf` confirmations:

```
$ bitcoind getbalance
0.05000000
```

Tip If the transaction has not yet confirmed, the balance returned by getbalance will be zero. The configuration option "minconf" determines the minimum number of confirmations that are required before a transaction shows in the balance.

Exploring and decoding transactions

Commands: gettransaction, getrawtransaction, decoderawtransaction

We'll now explore the incoming transaction that was listed above using the `gettransaction` command. We can retrieve a transaction by its transaction hash, shown at `txid`, above with the `gettransaction` command:

```
$ bitcoind gettransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3
{
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}
```

Tip Transaction IDs are not authoritative until a transaction has been confirmed. Absence of a transaction hash in the blockchain does not mean the transaction was not processed. This is known as "transaction malleability", as transaction hashes can be modified prior to confirmation in a block. After confirmation, the txid is immutable and authoritative.

The transaction form shown above with the command `gettransaction` is the simplified form. To retrieve the full transaction code and decode it we will use two commands, `getrawtransaction` and `decoderawtransaction`. First, `getrawtransaction` takes the transaction hash (txid) as a parameter and returns the full transaction as a "raw" hex string, exactly as it exists on the bitcoin network:

```
$ bitcoind getrawtransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3
0100000001d717279515f88e2f56ce4e8a31e2ae3e9f00ba1d0add648e80c480ea22e0c7d3000000008b483045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9b2ac1bd193dfba2014104793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e4c4b884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5ceaffffffff02404b4c0000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac1f312906000000001976a914107b7086b31518935c8d28703d66d09b3623134388ac00000000
```

To decode this hex string, we can use the `decoderawtransaction` command. Copy and paste the hex as the first parameter of `decoderawtransaction` to get the full contents interpreted as a JSON data structure (for formatting reasons the hex string is shortened in the example below):

```
$ bitcoind decoderawtransaction 0100000001d717...388ac00000000
{
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "d3c7e022ea80c4808e64dd0a1dba009f3eaee2318a4ece562f8ef815952717d7",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "3045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9b2ac1bd193dfba201
04793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e4c4b884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5cea",
```

```
                "hex" :
"483045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9b2ac1bd193dfba2014104793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e4c4b884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5cea"

            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.05000000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
                ]
            }
        },
        {
            "value" : 1.03362847,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 107b7086b31518935c8d28703d66d09b36231343 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914107b7086b31518935c8d28703d66d09b3623134388ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "12W9goQ3P7Waw5JH8fRVs1e2rVAKoGnvoy"
                ]
            }
        }
    ]
}
```

The transaction decode shows all the components of this transaction, including the transaction inputs and outputs. In this case we see that the transaction that credited our new address with 50 milliBits used one input and generated two outputs. The input to this transaction was the output from a previously confirmed transaction (shown as the vin txid starting with `d3c7` above). The two outputs correspond to the 50 millibit credit and an output with change back to the sender.

We can further explore the blockchain by examining the previous transaction referenced by its txid in this transaction using the same commands (eg. `gettransaction`). Jumping from transaction to transaction we can follow a chain of transactions back as the coins are transmitted from owner address to owner address.

Once the transaction we received has been confirmed by inclusion in a block, the `gettransaction` command will return additional information, showing the block hash (identifier) in which the transaction was included:

```
$ bitcoind gettransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3
{
    "amount" : 0.05000000,
    "confirmations" : 1,
    "blockhash" : "000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c2a717b",
    "blockindex" : 18,
    "blocktime" : 1392660808,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}
```

Above, we see the new information in the entries `blockhash`, the hash of the block in which the transaction was included, and `blockindex` with value 18, indicating that our transaction was the 18th transaction in that block.

Exploring blocks

Commands: getblock, getblockhash

Now that we know which block our transaction was included in, we can query that block. We use the `getblock` command with the block hash as the parameter:

```
$ bitcoind getblock 000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c2a717b true
{
    "hash" : "000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c2a717b",
    "confirmations" : 2,
    "size" : 248758,
    "height" : 286384,
    "version" : 2,
    "merkleroot" : "9891747e37903016c3b77c7a0ef10acf467c530de52d84735bd55538719f9916",
    "tx" : [
        "46e130ab3c67d31d2b2c7f8fbc1ca71604a72e6bc504c8a35f777286c6d89bf0",
        "2d5625725b66d6c1da88b80b41e8c07dc5179ae2553361c96b14bcf1ce2c3868",
        "923392fc41904894f32d7c127059bed27dbb3cfd550d87b9a2dc03824f249c80",
        "f983739510a0f75837a82bfd9c96cd72090b15fa3928efb9cce95f6884203214",
        "190e1b010d5a53161aa0733b953eb29ef1074070658aaa656f933ded1a177952",
        "ee791ec8161440262f6e9144d5702f0057cef7e5767bc043879b7c2ff3ff5277",
        "4c45449ff56582664abfadeb1907756d9bc90601d32387d9cfd4f1ef813b46be",
        "3b031ed886c6d5220b3e3a28e3261727f3b4f0b29de5f93bc2de3e97938a8a53",
        "14b533283751e34a8065952fd1cd2c954e3d37aaa69d4b183ac6483481e5497d",
        "57b28365adaff61aaf60462e917a7cc9931904258127685c18f136eeaebd5d35",
        "8c0cc19fff6b66980f90af39bee20294bc745baf32cd83199aa83a1f0cd6ca51",
        "1b408640d54a1409d66ddaf3915a9dc2e8a6227439e8d91d2f74e704ba1cdae2",
        "0568f4fad1fdeff4dc70b106b0f0ec7827642c05fe5d2295b9deba4f5c5f5168",
        "9194bfe5756c7ec04743341a3605da285752685b9c7eebb594c6ed9ec9145f86",
        "765038fc1d444c5d5db9163ba1cc74bba2b4f87dd87985342813bd24021b6faf",
        "bff1caa9c20fa4eef33877765ee0a7d599fd1962417871ca63a2486476637136",
        "d76aa89083f56fcce4d5bf7fcf20c0406abdac0375a2d3c62007f64aa80bed74",
        "e57a4c70f91c8d9ba0ff0a55987ea578affb92daaa59c76820125f31a9584dfc",
        "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

[... many more transactions ...]

    ],
    "time" : 1392660808,
    "nonce" : 3888130470,
    "bits" : "19015f53",
    "difficulty" : 3129573174.52228737,
    "chainwork" : "000000000000000000000000000000000000000000001931d1658fc04879e466",
    "previousblockhash" : "0000000000000000177e61d5f6ba6b9450e0dade9f39c257b4d48b4941ac77e7",
    "nextblockhash" : "0000000000000001239d2c3bf7f4c68a4ca673e434702a57da8fe0d829a92eb6"
}
```

The block contains 367 transactions and as you see above, the 18th transaction listed (`9ca8f9…`) is the txid of the one crediting 50 milliBits to our address. The `height` entry tells us this is the 286384th block in the blockchain.

We can also retrieve a block by its block height using the `getblockhash` command, which takes the block height as the parameter and returns the block hash for that block:

```
$ bitcoind getblockhash 0
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

Above, we retrieve the block hash of the "genesis block", the first block mined by Satoshi Nakamoto, at height zero. Retrieving this block shows:

```
$ bitcoind getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "confirmations" : 286388,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
    "tx" : [
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
    ],
    "time" : 1231006505,
    "nonce" : 2083236893,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "chainwork" : "0000000000000000000000000000000000000000000000000000000100010001",
    "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

The `getblock`, `getblockhash` and `gettransaction` commands can be used to explore the blockchain database, programmatically.

Creating, signing and submitting transactions based on unspent outputs

Commands: listunspent, gettxout, createrawtransaction, decoderawtransaction, signrawtransaction, sendrawtransaction

Bitcoin's transactions are based on the concept of spending "outputs", which are the result of previous transactions, to create a transaction chain that transfers ownership from address to address. Our wallet has now received a transaction that assigned one such output to our address. Once this is confirmed, we can now spend that output.

First, we use the `listunspent` command to show all the unspent **confirmed** outputs in our wallet:

```
$ bitcoind listunspent
[
    {
        "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
        "vout" : 0,
        "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
        "account" : "",
        "scriptPubKey" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
        "amount" : 0.05000000,
        "confirmations" : 7
    }
]
```

We see that the transaction `9ca8f9…` created an output (with vout index 0) assigned to the address `1hvzSo…` for the amount of 50 millibits, which at this point has received 7 confirmations. Transactions use previously created outputs as their inputs by referring to them by the previous txid and vout index. We will now create a transaction that will spend the 0th vout of the txid `9ca8f9…` as its input and assign it to a new output that sends value to a new address.

First, let's look at the specific output in more detail. We use the `gettxout` to get the details of this unspent output above. Transaction outputs are always referenced by txid and vout and these are the parameters we pass to `gettxout`:

```
$ bitcoind gettxout 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3 0
{
    "bestblock" : "0000000000000001405ce69bd4ceebcdfdb537749cebe89d371eb37e13899fd9",
    "confirmations" : 7,
    "value" : 0.05000000,
    "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
            "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
        ]
    },
    "version" : 1,
    "coinbase" : false
}
```

What we see above is the output that assigned 50 millibits to our address `1hvz…`. To spend this output we will create a new transaction. First, let's make an address to send the money to:

```
$ bitcoind getnewaddress
1LnfTndy3qxXGN19Jwscj1T8LR3WVe3JDb
```

We will send 25 millibits to the new address `1LnfTn...` we just created in our wallet. In our new transaction, we will spend the 50 millibit output and send 25 millibits to this new address. Because we have to spend the **whole** output from the previous transaction, we must also generate some change. We will generate change back to the `1hvz...` address, sending the change back to the address from which the value originated. Finally, we will also have to pay a fee for this transaction. To pay the fee, we will reduce the change output by 0.5 millibits, and return 24.5 millibits in change. The difference between the sum of the new outputs (25mBTC + 24.5mBTC = 49.5mBTC) and the input (50mBTC) will be collected as a transaction fee by the miners.

We use the `createrawtransaction` to create the transaction described above. As parameters to `createrawtransaction` we provide the transaction input (the 50 millibit unspent output from our confirmed transaction) and the two transaction outputs (money sent to the new address and change sent back to the previous address):

```
bitcoind createrawtransaction
 '[{"txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3", "vout" : 0}]'
'{"1LnfTndy3qxXGN19Jwscj1T8LR3WVe3JDb": 0.025,
"1hvzSofGwT8cjb8JU7nRaCSfEVQX5u9CL": 0.0245}'

0100000001e34acle2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000000ffffffff02a025260000000000001976a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac506225000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000
```

The `createrawtransaction` command produces a raw hex string that encodes the transaction details we supplied. Let's confirm everything is correct by decoding this raw string using the `decoderawtransaction` command:

```
$ bitcoind decoderawtransaction
0100000001e34acle2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000000ffffffff02a025260000000000001976a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac506225000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000

{
    "txid" : "0793299cb26246a8d24e468ec285a9520a1c30fcb5b6125a102e3fc05d4f3cba",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "",
                "hex" : ""
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.02500000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 d90d36e98ff62968d2bc9bbd68107564a156a9bcf OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1LnfTndy3qxXGN19Jwscj1T8LR3WVe3JDb"
                ]
            }
        },
        {
            "value" : 0.02450000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1hvzSofGwT8cjb8JU7nRaCSfEVQX5u9CL"
                ]
            }
        }
    ]
}
```

That looks correct! Our new transaction "consumes" the unspent output from our confirmed transaction and then spends it in two outputs, one for 25 millibits to our new address and one for 24.5 millibits as change back to the original address. The difference of 0.5 millibits represents the transaction fee and will be credited to the miner who finds the block that includes our transaction.

As you may notice, the transaction contains an empty `scriptSig` because we haven't signed it yet. Without a signature, this transaction is meaningless, we haven't yet proven that we **own** the address from which the unspent output is sourced. By signing, we remove the encumberance on the output and prove that we own this output and can spend it. We use the `signrawtransaction` command to sign the transaction. It takes the raw transaction hex string as the parameter.

Tip An encrypted wallet must be unlocked before a transaction is signed because signing requires access to the secret keys in the wallet.

```
$ bitcoind walletpassphrase foo 360
$ bitcoind signrawtransaction
0100000001e34acle2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000000ffffffff02a025260000000000001976a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac506225000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000

{
    "hex" :
"0100000001e34acle2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127ffffffff02a025260000000000001976a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac506225000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000",
    "complete" : true
}
```

The `signrawtransaction` command returns another hex encoded raw transaction. We decode it to see what changed, with `decoderawtransaction`:

```
$ bitcoind decoderawtransaction
0100000001e34acle2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127ffffffff02a025260000000000001976a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac506225000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000

{
    "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "3044022203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a0601
03c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127",
                "hex" :
"473044022203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127"
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.02500000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 d90d36e98ff62968d2bc9bbd68107564a156a9bcf OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1LnfTndy3qxXGN19Jwscj1T8LR3WVe3JDb"
                ]
            }
        },
        {
            "value" : 0.02450000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1hvzSofGwT8cjb8JU7nRaCSfEVQX5u9CL"
                ]
            }
        }
    ]
}
```

Now, the inputs used in the transaction contain a `scriptSig`, which is a digital signature proving ownership of address `1hvz...` and removing the encumberance on the output so that it can be spent. The signature makes this transaction verifiable by any node in the bitcoin network.

Now it's time to submit the newly created transaction to the network. We do that with the command `sendrawtransaction` which takes the raw hex string produced by `signrawtransaction`. This is the same string we just decoded above:

```
$ bitcoind sendrawtransaction
0100000001e34acle2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5127ffffffff02a025260000000000001976a914d90d36e98ff62968d2bc9bbd68107564a156a9bcf88ac506225000000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000


ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346
```

The command `sendrawtransaction` returns a transaction hash (txid) as it submits the transaction on the network. We can now query that transaction id with `gettransaction`:

```
$ bitcoind gettransaction ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346
{
    "amount" : 0.00000000,
    "fee" : -0.00050000,
    "confirmations" : 0,
    "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
    "time" : 1392666702,
    "timereceived" : 1392666702,
    "details" : [
        {
            "account" : "",
            "address" : "1LnfTndy3qxXGN19Jwscj1T8LR3WVe3JDb",
            "category" : "send",
            "amount" : -0.02500000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nRaCSfEVQX5u9CL",
            "category" : "send",
            "amount" : -0.02450000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1LnfTndy3qxXGN19Jwscj1T8LR3WVe3JDb",
            "category" : "receive",
            "amount" : 0.02500000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nRaCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.02450000
        }
    ]
}
```

As before, we can also examine this in more detail using the `getrawtransaction` and `decodetransaction` commands. These commands will return the exact same hex string that we produced and decoded previously just before we sent it on the network.

Alternative clients, libraries and toolkits

Beyond the reference client, bitcoind, there are other clients and libraries that can be used to interact with the bitcoin network and data structures. These are implemented in a variety of programming languages, offering programmers native interfaces in their own language.

Alternative implementations include:

- libbitcoin and sx tools, a C++ multi-threaded full node client and library with command-line tools (https://libbitcoin.dyne.org/)
- bitcoinj, a Java full node client library (https://code.google.com/p/bitcoinj/)
- btcd, a Go language full node bitcoin client (https://opensource.conformal.com/wiki/btcd)
- Bits of Proof (BOP), a Java enterprise-class implementation of bitcoin (https://bitsofproof.com)
- picocoin, a C implementation of a light-weight client library for bitcoin (https://github.com/jgarzik/picocoin)

Many more libraries exist in a variety of other programming languages and more are created all the time.

Libbitcoin and sx tools

The libbitcoin library is a C++ scalable multi-threaded and modular implementation that supports a full-node client and a command-line toolset named "sx", which offers many of the same capabilities as the bitcoind client commands we illustrated in this chapter. The sx tools also offer some key management and manipulation tools that are not offered by bitcoind, including type-2 deterministic keys and key mnemonics.

Installing sx

To install sx and the supporting library libbitcoin, download and run the online installer on a Linux system:

```
$ wget http://sx.dyne.org/install-sx.sh
$ sudo bash ./install-sx.sh
```

You should now have the sx tools installed. Type `sx` with no parameters to display the help text, which lists all the available commands:

```
Usage: sx COMMAND [ARGS]...

  -c, --config              Specify a config file

The sx commands are:

DETERMINISTIC KEYS AND ADDRESSES
   genaddr                   Generate a Bitcoin address deterministically from a wallet
                             seed or master public key.
   genpriv                   Generate a private key deterministically from a seed.
   genpub                    Generate a public key deterministically from a wallet
                             seed or master public key.
   mpk                       Extract a master public key from a deterministic wallet seed.
   newseed                   Create a new deterministic wallet seed.

TRANSACTION PARSING
   showscript                Show the details of a raw script.
   showtx                    Show the details of a transaction.

BLOCKCHAIN QUERIES (blockexplorer.com)
   blke-fetch-transaction    Fetches a transaction from blockexplorer.com

FORMAT
   base58-decode             Convert from base58 to hex
   base58-encode             Convert from hex to base58
   base58check-decode        Convert from base58check to hex
   base58check-encode        Convert from hex to base58check
   decode-addr               Decode an address to its internal RIPEMD representation.
   embed-addr                Generate an address used for embedding record of data into the blockchain.
   encode-addr               Encode an address to base58check form.
   ripemd-hash               RIPEMD hash data from STDIN.
   unwrap                    Validates checksum and recovers version byte and original data from hexstring.
   validaddr                 Validate an address.
   wrap                      Adds version byte and checksum to hexstring.

BRAINWALLET
   brainwallet               Make a private key from a brainwallet
   mnemonic                  Work with Electrum compatible mnemonics (12 words wallet seed).

BLOCKCHAIN WATCHING
   monitor                   Monitor an address.
   watchtx                   Watch transactions from the network searching for a certain hash.

BLOCKCHAIN QUERIES (blockchain.info)
   bci-fetch-last-height     Fetch the last block height using blockchain.info.
   bci-history               Get list of output points, values, and their spends
                             from blockchain.info

MISC
   btc                       Convert Satoshis into Bitcoins.
   initchain                 Initialize a new blockchain.
   qrcode                    Generate Bitcoin QR codes offline.
   satoshi                   Convert Bitcoins into Satoshis.
   showblkhead               Show the details of a block header.
   wallet                    Experimental command line wallet.

MULTISIG ADDRESSES
   scripthash                Create BIP 16 script hash address from raw script hex.

LOOSE KEYS AND ADDRESSES
   addr                      See Bitcoin address of a public or private key.
   get-pubkey                Get the pubkey of an address if available
   newkey                    Create a new private key.
   pubkey                    See the public part of a private key.

STEALTH
   secret-to-wif             Convert a secret exponent value to Wallet. Import. Format.
   stealth-new               Generate a new master stealth secret.
   stealth-recv              Regenerate the secret from your master secret and provided nonce.
   stealth-send              Generate a new sending address and a stealth nonce.

CREATE TRANSACTIONS
   mktx                      Create an unsigned tx.
   rawscript                 Create the raw hex representation from a script.
   set-input                 Set a transaction input.
   sign-input                Sign a transaction input.

VALIDATE
   validsig                  Validate a transaction input's signature.

BLOCKCHAIN QUERIES
   balance                   Show balance of a Bitcoin address in satoshis.
   fetch-block-header        Fetch raw block header.
   fetch-last-height         Fetch the last block height.
   fetch-transaction         Fetch a raw transaction using a network connection to make requests against the obelisk load balancer backend.
   fetch-transaction-index   Fetch block height and index in block of transaction.
   get-utxo                  Get enough unspent transaction outputs from a given set of
                             addresses to pay a given number of satoshis
   history                   Get list of output points, values, and their spends for an
                             address. grep can filter for just unspent outputs which can
                             be fed into mktx.
   validtx                   Validate a transaction.

BLOCKCHAIN UPDATES
   sendtx-bci                Send tx to blockchain.info/pushtx.
   sendtx-node               Send transaction to a single node.
   sendtx-obelisk            Send tx to obelisk server.
   sendtx-p2p                Send tx to bitcoin network.

See 'sx help COMMAND' for more information on a specific command.

SpesmiloXchange home page: <http://sx.dyne.org/>
```

Generating and manipulating keys with sxBitcoin Core

Generate a new private key with the operating system's random number generator by using the `newkey` command. We save the standard output into the file `private_key`:

```
$ sx newkey > private_key
$ cat private_key
5Jgx3UAaXw8AcCQCi1j7uaTaqpz2fqNR9K3r4apxdYn6rTzR1PL
```

Now, generate the public key from that private key using the `pubkey` command. Pass the `private_key` file into the standard input and save the standard output of the command into a new file `public_key`:

```
$ sx pubkey < private_key > public_key
$ cat public_key
02fca46a6006a62dfdd2dbb2149359d0d97a04f430f12a7626dd409256c12be500
```

We can re-format the public_key as an address using the `addr` command. We pass the `public_key` into standard input:

```
$ sx addr < public_key
17re1S4Q8ZHyCP8Kw7xQad1Lr6XUzWUnkG
```

Deterministic keys with sx

The keys generated above are so called type-0 non-deterministic keys. That means that each one is generated from a random number generator. The sx tools also support type-2 deterministic keys, where a "master" key is created and then extended to produce a chain or tree of subkeys.

First, we generate a "seed" that will be used as the basis to derive a chain of keys, compatible with the Electrum wallet and other similar implementations. We use the `newseed` command to produce a seed value:

```
$ sx newseed > seed
$ cat seed
eb68ee9f3df6bd4441a9feadec179ff1
```

The seed value can also be exported as a word mnemonic that is human readable and easier to store and type than a hexadecimal string using the `mnemonic` command:

```
$ sx mnemonic < seed > words
$ cat words
adore repeat vision worst especially veil inch woman cast recall dwell appreciate
```

The mnemonic words can be used to reproduce the seed using the `mnemonic` command again:

```
$ sx mnemonic < words
eb68ee9f3df6bd4441a9feadec179ff1
```

With the seed, we can now generate a sequence of private and public keys, a key chain. We use the `genpriv` command to generate a sequence of private keys from a seed and the `addr` command to generate the corresponding public key.

```
$ sx genpriv 0 < seed
5JzY2cPZGViPGgXZ4Syb9Y4eUGjJpVt6sR8noxrpEcqgyj7LK7i
$ sx genpriv 0 < seed | sx addr
1esVQV2vR9JZPhFeRaeWkAhzmWq7Fi7t7

$ sx genpriv 1 < seed
5JdtL7ckAn3iFBFyVG1Bs3A5TqziFTaB9f8NeyNo8crnE2Sw5Mz
$ sx genpriv 1 < seed | sx addr
1G1oTeXitk76c2fvQWny4pryTdH1RTqSPW
```

With deterministic keys we can generate and re-generate thousands of keys, all derived from a single seed in a deterministic chain. This technique is used in many wallet applications to generate keys that can be backed up and restored with a simple multi-word mnemonic. This is easier than having to back up the wallet with all its randomly generated keys every time a new key is created.

Tip | The sx toolkit offers many useful commands for encoding and decoding addresses, converting to and from different formats and representations. Use them to explore the various formats such as base58, base58check, hex etc.

Last updated 2014-06-25 19:31:46 CDT