Securing Digital Assets - Bitcoin Full Node API for kdb+

Jeremy Lucid

September 11, 2018

Table of Contents

1 Introduction

Bitcoin is a peer-to-peer, globally distributed and decentralised payments network where each participant computer in the network runs a software which implements the Bitcoin protocol. This protocol is used to control the issuance and secure transfer of bitcoin (BTC) tokens, the native currency of the network. In the original Satoshi whitepaper [1], Bitcoin is presented as an ”electronic cash” which can be used in a permission-less way and transacted with across the internet without requiring a trusted third party. As a new form of money, not issued by a government or central authority, Bitcoin has often been accessed by its ability to perform the three traditional functions of money: store of value, medium of exchange and unit of account.

Of these functions, the store of value proposition is one of the most interesting. Historically, stores of value have been associated with scarce tangible commodities like gold, however, while intangible, bitcoin shares with gold the property of scarcity, albeit digital. Specifically, the rules of the Bitcoin software ensure that bitcoin tokens exhibit a strict digital scarcity, with a maximum total cap and disinflationary supply schedule which is mathematically regulated, predictable and unchangable. Regardless of demand from investors and exchanges, new bitcoin supply remains inelastic, in that the number of bitcoins mined into existence, roughly ever ten minutes, is independent of demand. Even the ever increasing growth of the bitcoin mining industry, together with technological advancements in bitcoin mining, cannot increase supply beyond what the protocol rules permit. This strict adherence to a dis-inflationary monetary policy has likened Bitcoin to a 'digital-gold', or 'hard money' in monetary economics. In addition, the property of being purely digital, and software driven, in turn makes Bitcoin highly extensible, meaning it can evolve and be improved upon over time. Today, this extensibility can be clearly seen in the continued development taking place on both the base protocol and layer-two protocols, such as the lightening network [2].

While exposure and usage of Bitcoin is increasing for both retail and institutional investors, still many have expressed concern about how best to store this digital asset in a way which is secure and legally compliant. Although many secure and user-friendly options such as hardware-wallets are available for retail investors, enabling self custody, suitable third party Custodian services for institutions are less further developed. In response to demand, specialised 'Crypto-Custodian' services are currently being developed by Nomura and Ledger, Xapo, Gemini and Coinbase. For such Custodians, some of the major challenges include

- How best to safeguard such assets from a cyber security aspect. For example, making use of cold storage solutions and multi-signature wallets.
- The technological infastructure required for maximum privacy and trust, notably full nodes and wallet software.
- The end-to-end process of receiving funds, implementing access-control, managing funds and providing auditable proof of ownership

One of the most essential and core tools for addressing these challenges are Bitcoin full nodes, the topic of this paper. In particular, the following chapters will explore some of the general key concepts and best practice methodologies with regards to Bitcoin security and full nodes. This paper will demonstrate usage of the qbitcoind library, a q library designed to interact with the Bitcoin Core full node and wallet implementations. While this demonstration focuses on Bitcoin, the concepts can largely be applied across a wide range of cryptocurrency assets.

2 Why run a Full Node

Computers which participate in the Bitcoin network are called nodes, and those which fully verify all of the rules of Bitcoin are called full nodes. These nodes are an integral software component of the Bitcoin network and along with validating all transactions and blocks, also help relay them to other nodes. The full node software is essential for users who wish to use Bitcoin in a trustless way to maximise security, privacy and avail of the full Bitcoin functionality. Therefore, this software is often run by individual Bitcoin users, miners, cryptocurrency exchanges, wallet providers, payment processors and blockchain analytics providers.

Without running a full node, a user would be dependant on third party services, such as blockchain explorer services, for all transactional information, block information and address balance information. This reliance on a third party could result in receiving misleading information, or open the possibility for man-in-the-middle attacks. Even the act of requesting transactional information from a third party introduces a risk of revealing sensitive transactional and account information. By running a full node, users can create their own transactions, monitor all incoming and outgoing funds, confirm that all blocks and transactions follow the consensus rules, that there have been no double spends, that the correct signatures are present on transactions, and that transactions and blocks are in the correct data format. While maintaining a full node has associated hardware requirements in terms of memory and bandwidth, see Full Node Setup, these costs can be viewed as the cost of certainty, privacy and security when using Bitcoin.

In the following sections, some of the most recommended security practices for running a full node securely and managing the core wallet will be described, together with examples on how to perform particular actions using the qbitcoind library. This library can be used to communicate with a node running on the same server. While qbitcoind is designed to be a convenient option for q enthusiasts, dealing primariliy with q dictionary, list and table structures, the bitcoin-cli software (a command line based RPC tool) should still be considered the most secure and functional.

3 Running and Connecting to a Full Node

The most popular and trusted implementation of full nodes is called Bitcoin Core, and its latest release can be found on Github. To install and run a Bitcoin Core full node (bitcoind) follow the instructions as described on the following webpage, Install. Upon startup the full node will connect to multiple peers and begin downloading and validating the Bitcoin blockchain. In this paper, the node will be used to perform the following tasks.

- Creating watch-only wallets, allowing address balances to be monitored without compromising security
- Confirming the settlement of incoming funds and viewing transactional information on the Bitcoin blockchain
- Creating and broadcasting transactions which were signed off-line

3.1 Configuring

The Bitcoin Core implementation, bitcoind, is a headless daemon which syncs the Bitcoin blockchain on startup and provides a JSON-RPC interface to enable easy integration with other software or payment systems. It can be interacted with by using the Bitcoin command line utility, bitcoin-cli, or via HTTP JSON-RPC commands. To enable communication via JSON-RPC set the server=1 value in the bitcoin.conf file prior to startup, as shown below. By default, bitcoind will only accept connections from other processes running on the same machine, and requires basic access authentication when being communicated with. To enable username and password authentication, set the rpc username and password values in the bitcoin.conf file before running the daemon.

```
# Instructions to be added to the bitcoin.conf file
# [rpc]
# Accept command line and JSON-RPC commands.
server=1
rpcuser=<username>
rpcpassword=<password>

# Starting bitcoind on the command line
$bitcoind -daemon
```

3.2 Installing the q library

To load the qbitcoind library follow the instructions given on Github. The library can be loaded using two methods, either by using the standard library loading package qutil, or by simply loading the load.q script in the lib folder, \l *load.q*. Once loaded into a q session, this library can be used to communicate directly with the node running on the localhost. Connections will only be accepted after the rpc username and password values have been set. To set these credentials use the .bitcoind.initPass function, as shown below, and ensure the values match those in the bitcoin.conf file. To confirm authentication with the node is working correctly, run a simple command like .bitcoind.getblockcount to retrieve the current block height.

```
q).utl.require "qbitcoind"  // or  q)\l /path/to/bitcoind/lib/load.q
q).bitcoind.initPass["MyUserName";"MyPassword"]
q).bitcoind.getblockcount[]
  result| 522825
  error |
  id    | 0
```

To confirm the node has indeed made connections to other nodes on the network, call the .bitcoind.getnetworkinfo function with no arguments. Below is shown a typical output of this function, highlightng that 80 connections have been created with other network peers.

```
q).bitcoind.getnetworkinfo[][`result]
  version        | 160000f
  subversion     | "/Satoshi:0.16.0/"
  protocolversion| 70015f
  localservices  | "000000000000040d"
  localrelay     | 1b
  timeoffset     | 0f
  networkactive  | 1b
  connections    | 80f
```

3.3 Watch only addresses

When the Bitcoin Core full node software is running online it is continuously validating transactions and blocks, but it also has fully in-built wallet functionality allowing the user to create receive addresses and manage funds. There are, however, security concerns to consider when using the wallet functionality while the node is online. In particular, it is not recommended to generate or store bitcoin private keys on an online computer as this greatly increases the risk of cyber theft. A much better solution is to either generate the addresses on an offline hardware wallet device, or run another instance of the bitcoind software on a completely offline computer, so as to securely generate valid bitcoin addresses and sign transactions using the wallet functionality. To see how to do this, go to Section - Offline wallet.

The online node can then be used, in conjunction, to monitor those addresses for incoming and outgoing transactions, and to broadcast transactions which were signed offline. While this practice will be explained in more detail in coming sections, for now we will focus on using the online full node for creating watch-only wallets. Wallet addresses generated offline can be imported into the online bitcoin node while keeping the private keeps on the offline wallet. Below the .bitcoind.importaddress function is used to load an address into an account called "Watch Only Addr One".

```
//  Arguments:
//  1. "script" (string,required) The hex-encoded script (or address)
//  2. "label"  (string,optional,default="") An optional label
//  3. rescan   (boolean,optional,default=true) Rescan the wallet
//  4. p2sh     (boolean,optional,default=false) Add the P2SH version

q)Address:"37cCd24chWjXSi7xmQahTzyGjbJauTp9xq"
q).bitcoind.importaddress[Address;"Watch Only Addr One"]
```

3.4 Address Balances

Once the previous watch-only address has been loaded, the wallet will continue to track all ingoing and outgoing transactions to this address and keep track of the total balance. The balance can be viewed using the .bitcoind.getbalance function, which takes three optional argument: the account name, the minimum number of confirmations a transaction should have to be included, and whether to include watch only addresses or not. In this case, we can specify the same wallet account name as above, a minimum number of confirmations of 6 and a boolean value of 1b to include watch only addresses. The balance on the address is reported in the results field. By using the node to determine address balances, multiple accounts (whose private keys are kept in cold storage) can be kept track of securely and without requiring a third party.

```
//  Arguments:
//  1. account (string,optional,default="") The acount name
//  2. minconf (int,optional,default="") Min number of confirmations
//  3. watchonly (boolean,optional,default=0b) Include watchonly

q).bitcoind.getbalance["Watch Only Addr One";6;1b]
  result| 0.1245
  error |
  id    | 0
```

3.5 Transaction IDs

Transaction IDs (txid) are identification numbers associated with transactions on the Bitcoin blockchain. They are always 32 bytes long and encoded in hexidecimal format. In Section - Broadcast Transaction, a sample txid is returned when a new transaction is broadcast to the Bitcoin network from the full node using the .bitcoind.sendrawtransaction function. To perform a lookup against the local copy of the blockchain for this particular txid, we can use the .bitcoind.getrawtransaction or .bitcoind.gettransaction functions, as shown below.

```
// Arguments:
// 1. "txid" (string,required) The transaction id
// 2. verbose (bool,optional,default=false)
//            If false, return string, else json object
// 3. "blockhash" (string,optional) The block in which
//                 to look for the transaction

t:"78889f97c5d105b5b5a13807d10438bab700db08cb88581a1f8bb3c10ad368e0"
q).bitcoind.getrawtransaction[t;1][`result]
txid         | "78889f97c5d105b5b5a13807d10438bab700db08cb885..."
hash         | "8208dc54d4a44d2d972a51246fcc61d38912f81015215..."
version      | 2f
size         | 192f
vsize        | 110f
locktime     | 0f

// Arguments:
// 1. "txid" (string,required) The transaction id
// 2. "include_watchonly" (bool,optional,default=false) Whether to
//   include watch-only addresses in balance calculation and details[]

q).bitcoind.gettransaction[t;1b][`result]
amount            | -0.01071
fee               | -2.73e-06
confirmations     | 3540f
blockhash         | "0000000000000000002732151191c6f43487f591..."
blockindex        | 2825f
blocktime         | 1.532991e+09
```

```
txid                |  "78889f97c5d105b5b5a13807d10438bab700db08..."
walletconflicts     |  ()
time                |  1.532988e+09
timereceived        |  1.532988e+09
bip125-replaceable|  "no"
```

4 Running Bitcoin Core Wallet in Offline Mode

The Bitcoin Core software contains a secure digital wallet which can be used to generate wallet addresses, store private keys, and create valid transactions. Running the bitcoin core wallet in an offline mode is a highly recommended and necessary step to help mitigate the risk of cyber attacks, malicous software, and help keep funds secure. This approach, also known as cold storage, involves running the wallet software and storing the recovery wallet file (wallet.dat) in a secured place that is not connected to the internet. This security step is followed by Custodians such as the Swiss Crypto Vault, Xapo and multiple cryptocurrency exchanges. To run a bitcoin core wallet in offline mode, instruct it to connect to the localhost, as shown below.

```
$bitcoind -connect=0
```

As in the previous case, when the node was run in an on-line mode, the same queries can be used to confirm the blockchain had not been downloaded, nor any connections made with other nodes.

```
q).utl.require "qbitcoind"
q).bitcoind.initPass["MyUserName";"MyPassword"]
q).bitcoind.getblockcount[]
  result| 0
  error |
  id    | 0
q).bitcoind.getnetworkinfo[][`result]
  version         | 160000f
  subversion      | "/Satoshi:0.16.0/"
  protocolversion| 70015f
  localservices   | "000000000000040d"
  localrelay      | 1b
  timeoffset      | 0f
  networkactive   | 1b
  connections     | 0f
```

4.1 Address generation

In order for a user to receive and store bitcoin payments, a valid bitcoin address needs to be created by the wallet and associated with a given account. Multiple addresses can be used to segregate the funds of a given user, and separate users. Below the .bitcoind.getnewaddress function is used to generate a new valid address and associate it with the account "Account One".

```
// Arguments:
// 1. "account"  (string, optional) DEPRECATED. The account
//    name for the address to be linked to. If not provided,
//    the default account "" is used. It can also be set to
//    the empty string "" to represent the default account.
//    The account does not need to exist, it will be created if
//    there is no account by the given name.
// 2. "address_type"   (string, optional) The address type to use.
//  Options are "legacy", "p2sh-segwit", and "bech32". Default is
//    set by -addresstype.

q).bitcoind.getnewaddress["Account One";"bech32"]
result| "bc1qxuhlv9kpq9zz07xm0at2w44ene6la7qm2ew6un"
error | 0n
id    | 0f
```

4.2 Encrypted wallet

In addition to cold storage, one of the main security features of the Bitcoin core wallet is the ability to encrypt the wallet.dat file which contains the key pairs for each address, account information, transaction information and more. When the wallet is encrypted then the private keys cannot be viewed or used to create a transaction, even by someone with physical access to the machine. Only when the wallet is decrypted can private keys be viewed and transactions generated. The wallet.dat file can be encrypted using a pass phrase, as shown below. Note this will not protect against keylogging hardware or software.

```
// Arguments:
// 1. "passphrase"  (string) The pass phrase to encrypt wallet with.
// It must be at least 1 character, but should be long.

q).bitcoind.encryptwallet["My Encrypt Pass Phrase"]
```

Once the wallet is encrypted, a user with physical access is still unable to view the private key of any address without entering the correct decrypt pass phrase. To illustrate, below an attempt is made to display the private key associated with a particular wallet address which is unsuccessful.

```
// Arguments:
// 1. "address" (string, required)
//    The bitcoin address for the private key

q)R:.bitcoind.dumpprivkey["3JtpCVqn8cXseSCXLEYdYwvxziSYQSabGd"]
q)R[`error][`message]
"Error:Please enter the wallet passphrase with walletpassphrase first"
```

To minimise risk while the wallet is decrypted, a timer can be set at the time of decryption allowing the wallet to be accessed for a brief period of time. Below, the wallet is decrypted for a period of 30 seconds using the .bitcoind.walletpassphase function.

```
// Arguments:
// 1. "passphrase" (string, required) The wallet passphrase
// 2. timeout     (numeric, required) The time to keep the
//    decryption key in seconds; capped at 100000000 (~3 years).

q).bitcoind.walletpassphrase["My Encrypt Pass Phrase";30]
```

If after entering the wallet pass phrase the transaction task is completed prior to the timeout time being reached, the wallet can be locked again immediately and the encryption key removed from memory using the .bitcoind.walletlock function.

```
// Removes the wallet encryption key from memory, locking the wallet.
// After calling this method, you will need to call walletpassphrase
// again before being able to call any methods which require the
// wallet to be unlocked.

q).bitcoind.walletlock[]
```

4.3 Creating a wallet backup

To mitigate the risk of data loss due to either computer failures or human mistakes, additional recovery wallet files can be created at any time and stored offline across multiple physical locations so there is no single point of failure to prevent recovery. Below, the .bitcoind.backupwallet function is used to write a new wallet.dat file to a backup directory.

```
// Arguments:
// 1. "destination" (string) The destination directory or file
```

```
q).bitcoind.backupwallet["/home/btc/mybackup/backupDir/"]
```

4.4 Multi-Signature Wallet

Multi-Signature wallet addresses are one of the best known ways to secure crypto-assets. These are addresses which require a number of private keys to be used together to move funds. The multiple private keys can be stored across different physical locations and by multiple individuals, minimising the risk of funds being stolen should a single private key be comprimised. Below, the .bitcoind.addmultisigaddress function is used to generate a multi-sig address requiring two private keys to perform a transaction.

```
// Arguments:
// 1. nrequired (numeric, required)
// The number of required signatures out of the n keys or addresses.
// 2. "keys" (string, required) A json array of bitcoin addresses
//           or hex-encoded public keys
//   [
//    "address" (string) bitcoin address or hex-encoded public key
//     ...,
//   ]
// 3. "account" (string, optional) DEPRECATED.
//    An account to assign the addresses to.
// 4. "address_type" (string, optional) The address type to use.
//     Options are "legacy", "p2sh-segwit", and "bech32".

q).bitcoind.getnewaddress["MSig Acc";"bech32"]
result| "bc1q8j05t0m04lv7gkn2crkeq7g5gq9xn0alk6p5lu"
q).bitcoind.getnewaddress["MSig Acc";"bech32"]
result| "bc1qmaadgvjrqj45u758rc8grfppdaedwshz38l46a"
q).bitcoind.getnewaddress["MSig Acc";"bech32"]
result| "bc1qgkvxclhgf2yw2snh0pw0sf9yutug9pxhjl304z"

q)addressOne:"bc1q8j05t0m04lv7gkn2crkeq7g5gq9xn0alk6p5lu"
q)addressTwo:"bc1qmaadgvjrqj45u758rc8grfppdaedwshz38l46a"
q)addressThree:"bc1qgkvxclhgf2yw2snh0pw0sf9yutug9pxhjl304z"
q)Keys:(addressOne;addressTwo;addressThree)

q).bitcoind.addmultisigaddress[2;Keys;"multiSig";"bech32"][`result]
address |
 "bc1qvadv2tzue3nvxg7u2lmfg9mrevrd2jjq7pa7afkscuvjl2j2xnhqcqq4qn"
redeemScript|
  "5221030030f8b96f11da696e5d1cbfb13b7bf1055dee22e7eee523179a192446
   f6d77121022fd014adef186d4e1a3df09146d9fc0da653fba5d42ae31233aa38
   8be9e53fac21027c0a88252ac7b4345b7351e3daccf1965da3ea214fda6fa6ec
```

```
    ceb5c45a5986e453ae"
```

4.5 Proof of Ownership

One requirement for any custodian provider is to be able to prove that they are in full control over their clients funds. For Bitcoin, this requires the custodian to have full cryptographic control over all assets, meaning they are in possession of the private keys associated with the addesses which hold funds. The core wallet provides a simple and effecive way to provide this proof-of-ownership through the use of digital signatures. In this case, a challenge message is signed using the private key associated with a given address which holds funds. The challenge message, signed message and public address can then be used to validate that the private key used in signing is valid. If the signature is valid then the signer has control over the funds held on the adddress. In the example below, a short challenge message containing the text "Proof of Ownership", is signed using the .bitcoind.signmessage function, and the signed message subsequently validated using the .bitcoind.validatemessage function. Validation can be performed independently.

```
// Arguments:
// 1. "address" (string,required)
//     The bitcoin address to use for the private key.
// 2. "message" (string,required)
//     The message to create a signature of.

// Sign Challenge Message with Private Key
q)walletAddr:"1CzqfiNq6u3aF2CPq9LRKQ6pFJFV5tDVrG"
q)challengeMsg:"Proof of Ownership"
q).bitcoind.signmessage[walletAddr;challengeMsg]
result| "H0Pz8ndn+HFZNNHUHahXZhfBRPuNi29tJWfb4XEoos9TE3riI5K..."
error | 0n
id    | 0f


// Arguments:
// 1. "address"  (string,required)
//       The bitcoin address to use for the signature.
// 2. "signature"(string,required) The signature provided
//     by the signer in base 64 encoding (see signmessage).
// 3. "message"  (string, required) The message that was signed.

// Verify Signed Message
q)signedMsg:.bitcoind.signmessage[walletAddr;challengeMsg][`result]
q).bitcoind.verifymessage[walletAddr;signedMsg;challengeMsg]
result| 1b
error | 0n
id    | 0f
```

5 Receiving and Sending funds securely

5.1 Create a receive address

The first step in receiving funds securely is to generate a new valid bitcoin address using either an offline hardware wallet, or by running the bitcoin core wallet completely offline and using the command below, as previously described in Section - Address Generation.

```
// Offline wallet
q).bitcoind.getnewaddress["TestAddr";"bech32"]
result | bc1qkmp5q4v8vzkcke8dxryv2qa2uarytx4t4srrgh
```

Next, import this generated address into the online full node so that ingoing and outgoing funds can be monitored, see Section - Watch Only Wallet

```
// Online Full node
q)receiveAddress:"bc1qkmp5q4v8vzkcke8dxryv2qa2uarytx4t4srrgh"
q).bitcoind.importaddress[receiveAddress;"Watch Only Address";0b;0b]
result|
error |
id    | 0
```

Confirm that the address has been assigned to the correct account using the .bitcoind.getaccount function.

```
// Arguments:
// 1. "address" (string, required)
//  The bitcoin address for account lookup.

q).bitcoind.getaccount["bc1qkmp5q4v8vzkcke8dxryv2qa2uarytx4t4srrgh"]
result| "Watch Only Address"
error | 0n
id    | 0f
```

5.2 Confirming Received Funds

Once the watch-only address has been added to the "Watch Only Address" account then the .bitcoind.listaccounts function can be used to confirm the total balance of bitcoin on that account. The function takes two optional arguments, the first being the minimum number of confirmations (minconf) a transaction must have in order for its balance to be counted, and the second being a boolean flag indicating whether to include watch-only addresses in the result. The number of confirmations is a measure of how many blocks deep the transaction is on the blockchain, the higher the number the more confident you can be in the transactions finalisation. Most merchants opt for 6 confirmations to be received before deeming funds confirmed. In the example below, we can see that for an incoming transaction with just one confirmation in real-time the listaccounts function shows a positive balance for a minconf of 1, but a zero balance for a minconf value of 6.

```
// Arguments:
// 1. minconf  (numeric, optional, default=1) Only include
//      transactions with at least this many confirmations
// 2. include_watchonly   (bool, optional, default=false)
// Include balances in watch-only addresses (see 'importaddress')

q).bitcoind.listaccounts[1;1b][`result]
Watch Only Address | 0.01071273

q).bitcoind.listaccounts[6;1b][`result]
Watch Only Address | 0f
```

Once funds have been confirmed the .bitcoind.listunspent function can be used to display the amount of unspent funds held on a given address together with some important transactional information such as the transaction identifier, txid. The function takes 3 arguments, the minimum and maximum number of confirmations and the list of addresses to filter. Below the spendable flag returns a value of 0b, which indicates the wallet does not have the private key to spend the output, because it is stored safely on the offline computer.

```
// Arguments:
// 1. minconf (numeric, required) The minimum confirmations to filter
// 2. maxconf (numeric, required) The maximum confirmations to filter
// 3. addresses (string) A json array of bitcoin addresses to filter
//   [
//     "address"     (string) bitcoin address
//     ,...
//   ]

q)flip .bitcoind.listunspent[1;100000;()][`result]
txid          | "ee9abf639bf459aea85fdf353a362b8f99
                db1d6c048fe55b69935b89b4c490fa"
vout          | 0
address       | "bc1qkmp5q4v8vzkcke8dxryv2qa2uarytx4t4srrgh"
account       | "Watch Only Address"
scriptPubKey  | "0014b6c340558760ad8b64ed30c8c503aae746459aab"
amount        | 0.01071273
confirmations| 1
spendable     | 0
```

```
solvable      | 0
safe          | 1
```

5.3 Create Transaction

Bitcoin transactions involve inputs and outputs, inputs are bitcoins you can combine together to use in a transaction (like the coins in your pocket), and outputs are the amounts of bitcoin sent to new addresses as part of the transaction. In the example below, a transaction is created to spend the funds associated with the watch-only address in the previous section. To create a new transaction, we use the .bitcoind.createrawtransaction function which takes as a first argument a dictionary containing the transaction ID (txid) and vout of the input to the transaction. This information was already presented in the previous section, and is contained in the output of the .bitcoind.listunspent function. The second argument, the outputs variable, is again a dictionary containing as key the addresses to send bitcoin to and the associated amount. In this case, we have one input and one output, so we are moving bitcoin from one address to another. The final two arguments are optional and correspond to the locktime and replacable values which are out of the scope of this document.

```
// Arguments:
// 1. "inputs"   (array, required) A json array of json objects
//     [
//       {
//         "txid":"id",    (string, required) The transaction id
//         "vout":n,         (numeric, required) The output number
//         "sequence":n      (numeric, optional) The sequence number
//       }
//       ,...
//     ]
// 2. "outputs"   (object, required) a json object with outputs
//    {
//      "address": x.xxx, (numeric or string, required)
//                        The key is the bitcoin address,
//                        the numeric value is the BTC amount
//      "data": "hex"      (string, required) The key is "data",
//                        the value is hex encoded data
//      ,...
//    }
// 3. locktime (numeric, optional, default=0) Raw locktime.
//                 Non-0 value also locktime-activates inputs
// 4. replaceable (boolean, optional, default=false)
//               Marks this transaction as BIP125 replaceable.
//   Allows this transaction to be replaced by a transaction
//   with higher fees. If provided, it is an error if explicit
//   sequence numbers are incompatible.
```

```
q)t:"ee9abf639bf459aea85fdf353a362b8f99db1d6c048fe55b69935b89b4c490fa"
q)sendToAddr:`bc1qzsg3qkd9r3986q96h7kgqhhfrps9hpa5a2yn8j
q)inputs:enlist `txid`vout!(t;0)
q)outputs:(enlist sendToAddr)!enlist 0.01071
q).bitcoind.createrawtransaction[inputs;outputs;0;1b]
result|"0200000001fa90c4b4895b93695be58f046c1ddb998f2b363a35df5fa8ae
        59f49b63bf9aee0000000000fdffffff0198571000000000001600141411
        1059a51c4a7d00babfac805ee918605b87b400000000"
error | 0n
id    | 0f
```

The above step of creating a transaction can be performed on either and online or offline computer. The output result, however, needs to trnsferred to the offline wallet for private key signing, described in detail below.

5.4 Sign Transaction

Signing a transaction is the next important step in which the private keys associated with the spending addresses are required. The function which can be used to perform the signing action is called .bitcoind.signrawtransaction, and its argument list is given below. Note, this step should be performed on an offline computer for increased security. Below, the scriptPubKey value is again taken from the output of the function .bitcoind.listunspent.

```
// Arguments:
// 1. "hexstring"     (string, required) The transaction hex string
// 2. "prevtxs"       (string, optional) An json array of previous
//                     dependent transaction outputs
//     [               (json array of json objects, or 'null' )
//       {
//         "txid":"id",     (string, required) The transaction id
//         "vout":n,        (numeric, required) The output number
//         "scriptPubKey": "hex", (string, required) script key
//         "redeemScript": "hex", (string, required) redeem script
//         "amount": value  (numeric, required) The amount spent
//       }
//       ,...
//     ]
// 3. "privkeys"    (string, optional) A json array of base58-encoded
//                  private keys for signing
//     [           (json array of strings, or 'null' if none provided)
//       "privatekey"   (string) private key in base58-encoding
//       ,...
//     ]
// 4. "sighashtype"     (string, optional, default=ALL)
//                      The signature hash type. Must be one of
//         "ALL"
//         "NONE"
//        "SINGLE"
//         "ALL|ANYONECANPAY"
//         "NONE|ANYONECANPAY"
//         "SINGLE|ANYONECANPAY"


q)hexstring:"0200000001fa90c4b4895b93695be58f046c1ddb998f2b363a35df
             5fa8ae59f49b63bf9aee0000000000fdffffff0198571000000000
             001600141411059a51c4a7d00babfac805ee918605b87b400000000"
q)t:"ee9abf639bf459aea85fdf353a362b8f99db1d6c048fe55b69935b89b4c490fa"
q)scriptPubKey:"0014b6c340558760ad8b64ed30c8c503aae746459aab"
q)prevtxsValues:(t;0f;scriptPubKey;0.01071273)
q)prevtxs:enlist `txid`vout`scriptPubKey`amount!prevtxsValues
q)privkeys:enlist "Private Key Goes Here"
q).bitcoind.signrawtransaction[hexstring;prevtxs;privkeys]
result| `hex`complete!("02000000000101fa9...";1b)
error | 0n
id    | 0f
```

Note that the previous two steps of creating a transaction and signing a transactions can also be performed on the Trezor hardware wallet using the ”Sign without Sending” option, and will produce a similar hex value as shown in the example above.

5.5 Broadcast Transaction

The output hex string from the signing action can now be broadcast to the bitcoin network from the online full node by using the .bitcoind.sendrawtransaction function. If successful, the result value contains a transaction ID which can be used to track the state of the transaction on the blockchain.

```
// Arguments:
// 1. "hexstring"    (string, required)
//    The hex string of the raw transaction)
// 2. allowhighfees    (boolean, optional, default=false)
//                      Allow high fees

q).bitcoind.sendrawtransaction["02000000000101fa9...";1b][`result]
"78889f97c5d105b5b5a13807d10438bab700db08cb88581a1f8bb3c10ad368e0"
```

5.6 Sending from a hot wallet

In the case where the private keys associated with funds are already present in the wallet of the online running node, then it is possible to perform a more straight forward transaction using the .bitcoind.sendtoadddress function. This performs the create transaction, sign transaction and broadcast transaction in one convenient step. At a minimum, this function needs only the receving address and the amount to send. Below are shown the additional optional arguments.

```
// Arguments:
// 1. "address" (string, required) The bitcoin address to send to.
// 2. "amount"  (numeric or string, required) The amount in BTC to
//     send. eg 0.1
// 3. "comment" (string, optional) A comment used to store what the
//     transaction is for.
// This is not part of the transaction, just kept in your wallet.
// 4. "comment_to" (string, optional) A comment to store the name of
// the person or organization to which you're sending the transaction.
// This is not part of the transaction, just kept in your wallet.
// 5. subtractfeefromamount  (boolean, optional, default=false) The
// fee will be deducted from the amount being sent. The recipient
// will receive less bitcoins than you enter in the amount field.
// 6. replaceable (boolean, optional) Allow this transaction to be
//  replaced by a transaction with higher fees via BIP 125
// 7. conf_target  (numeric, optional) Confirmation target(blocks)
// 8. "estimate_mode" (string, optional, default=UNSET) The fee
// estimate mode, must be one of:
//       "UNSET","ECONOMICAL","CONSERVATIVE"

q)ToAddr:bc1q9sldykvtnt2grrq44qyfaxe8p6lwmrewjk7yqj
q).bitcoind.sendtoaddress[ToAddr;0.01070][`result]
"67e5a14dcac0d7243a6c21b25b15f9520d28864dbca337d3e97138d3df8e393e"
```

References

[1] Nakamoto, Satoshi, *Bitcoin: A Peer-to-Peer Electronic Cash System*
https://bitcoin.org/bitcoin.pdf

[2] Poon, Joseph and Dryja, Thaddeus *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*
https://lightning.network/lightning-network-paper.pdf