Electron Cash Documentation

Release 3.1.2

Daniel Connolly

Jun 11, 2019

Contents

SECURITY ALERT: Electron Cash version 3.1.2 has been released. A JSONRPC security hole was discovered that affects all Electrum forks. This has been fully patched in Electron Cash 3.1.2. If you were running a prior version as an online wallet without a strong password, you should consider the wallet compromised and create a fresh wallet with version 3.1.2 or better.

Electron Cash is a lightweight Bitcoin Cash wallet.

Current versions are:

- Windows, Mac, and Linux - 4.0.6
- Android - 3.3.5

Electron Cash can be downloaded from https://electroncash.org/ or the PlayStore. These are the *only* authentic sources.

CHAPTER 1

Getting Started

1.1 Downloading and Verifying Electron Cash

IMPORTANT: it is vital that you only use information and downloads from https://electroncash.org

1.1.1 Android

Install the app from the PlayStore.

The PlayStore has verification of downloads built into the system, no verification is needed.

1.1.2 Linux

- retrieve Jonald's public key from his repo. You only need to do this once but it is recommended to insure that you are installing an original version of electron cash.
  - `wget "https://github.com/fyookball/keys-n-hashes/raw/master/pubkeys/jonaldkey2.txt"`
  - `gpg --import jonaldkey2.txt`
- Download the latest version from https://electroncash.org. You need the `tar.gz` file, such as `ElectronCash-3.1.2.tar.gz`
- Download the correct signature and checksum files from https://github.com/fyookball/keys-n-hashes/tree/master/sigs-and-sums
- Verify the download
  - `sha256sum -c SHA256.ElectronCash-3.1.2.tar.gz.txt`
  - `gpg --verify ElectronCash-3.1.2.tar.gz.sig`

1.2 Installing Electron Cash

1.2.1 Installing on Linux

Linux users will need to run Electron Cash directly from the source code.

1. You will need Python version 3.4.0 or higher. Check your python version with `python3 --version`.
2. Install the dependencies:

- `sudo apt-get install python3-setuptools`
- `sudo pip3 install PyQt5`

3. You should have already *downloaded and verified* a gzipped tar of the source. The file will be named something like `ElectronCash-3.1.2.tar.gz`.
4. Extract the files using `tar -xvzf ElectronCash-3.1.2.tar.gz` and `cd` into the directory.
5. Install using `sudo python3 setup.py install`
6. You can now run Electron Cash using `./electron-cash`

Some common issues are:

- `No local packages or working download links found for pyqt5` - you will need to install PyQt5 before installing Electron Cash. See step 2 above.

CHAPTER 2

Contributing

To contribute to Electron Cash, check the GitHub repo and contact Jonald Fyookball.

To contribute to this documentation, submit a pull request to the GitHub documentation repo.

CHAPTER 3

Old Documentation

The documentation is currently being re-written from scratch. The old documentation is being temporarily maintained here until it is completely replaced.

3.1 Frequently Asked Questions

3.1.1 How does Electron Cash work?

Electron Cash's focus is speed, with low resource usage and simplifying Bitcoin Cash. Startup times are instant because it operates in conjunction with high-performance servers that handle the most complicated parts of the Bitcoin Cash system.

3.1.2 Does Electron Cash trust servers?

Not really; the Electron Cash client never sends private keys to the servers. In addition, it verifies the information reported by servers, using a technique called *Simple Payment Verification*

3.1.3 What is the Seed?

The seed is a random phrase that is used to generate your private keys.

Example:

```
constant forest adore false green weave stop guy fur freeze giggle clock
```

Your wallet can be entirely recovered from its seed. For this, select the "restore wallet" option in the startup.

3.1.4 How secure is the seed?

The seed phrase created by Electron Cash has 132 bits of entropy. This means that it provides the same level of security as a Bitcoin Cash private key (of length 256 bits). Indeed, an elliptic curve key of length n provides n/2 bits of security.

3.1.5 I have forgotten my password. What can I do?

It is not possible to recover your password. However, you can restore your wallet from its seed phrase, and choose a new password. If you lose both your password and your seed, there is no way to recover your money. This is why we ask you to save your seed phrase on paper.

3.1.6 My transaction has been unconfirmed for a long time. What can I do?

Bitcoin Cash transactions become 'confirmed' when miners accept to write them in the Bitcoin Cash blockchain. In general, the speed of confirmation depends on the fee you attach to your transaction; miners prioritize transaction that pay the highest fees.

Recent versions of Electron Cash use 'dynamic fees', in order to make sure that the fee you pay with your transaction is adequate. This feature is enabled by default in recent versions of Electron Cash.

If you have made a transaction that is unconfirmed, you can:

- Wait for a long time. Eventually, your transaction will either be confirmed or cancelled. This might take several days.
- Increase the transaction fee. This is only possible for 'replaceable' transactions. To create this type of transaction, you must have enabled 'Replace by Fee' in your preferences, before sending the transaction.
- Create a 'Child Pays For Parent' transaction. A CPFP is a new transaction, that pays a high fee in order to compensate for the small fee of its parent transaction. It can be done by the recipient of the funds, or by the sender, if the transaction has a change output.

3.1.7 What does it mean to "Freeze" an address in Electron Cash?

When you freeze an address, the funds in that address will not be used for sending Bitcoin Cash. You can not send Bitcoin Cash if you don't have enough funds in the non-frozen addresses.

3.1.8 How is the wallet encrypted?

Electron Cash uses two separate levels of encryption:

- Your seed and private keys are encrypted using AES-256-CBC. The private keys are decrypted only briefly, when you need to sign a transaction; for this you need to enter your password. This is done in order to minimize the amount of time during which sensitive information is unencrypted in your computer's memory.
- In addition, your wallet file may be encrypted on disk. Note that the wallet information will remain unencrypted in the memory of your computer for the duration of your session. If a wallet is encrypted, then its password will be required in order to open it. Note that the password will not be kept in memory; Electron Cash does not need it in order to save the wallet on disk, because it uses asymmetric encryption (ECIES).

Wallet file encryption is activated by default since version 2.8.

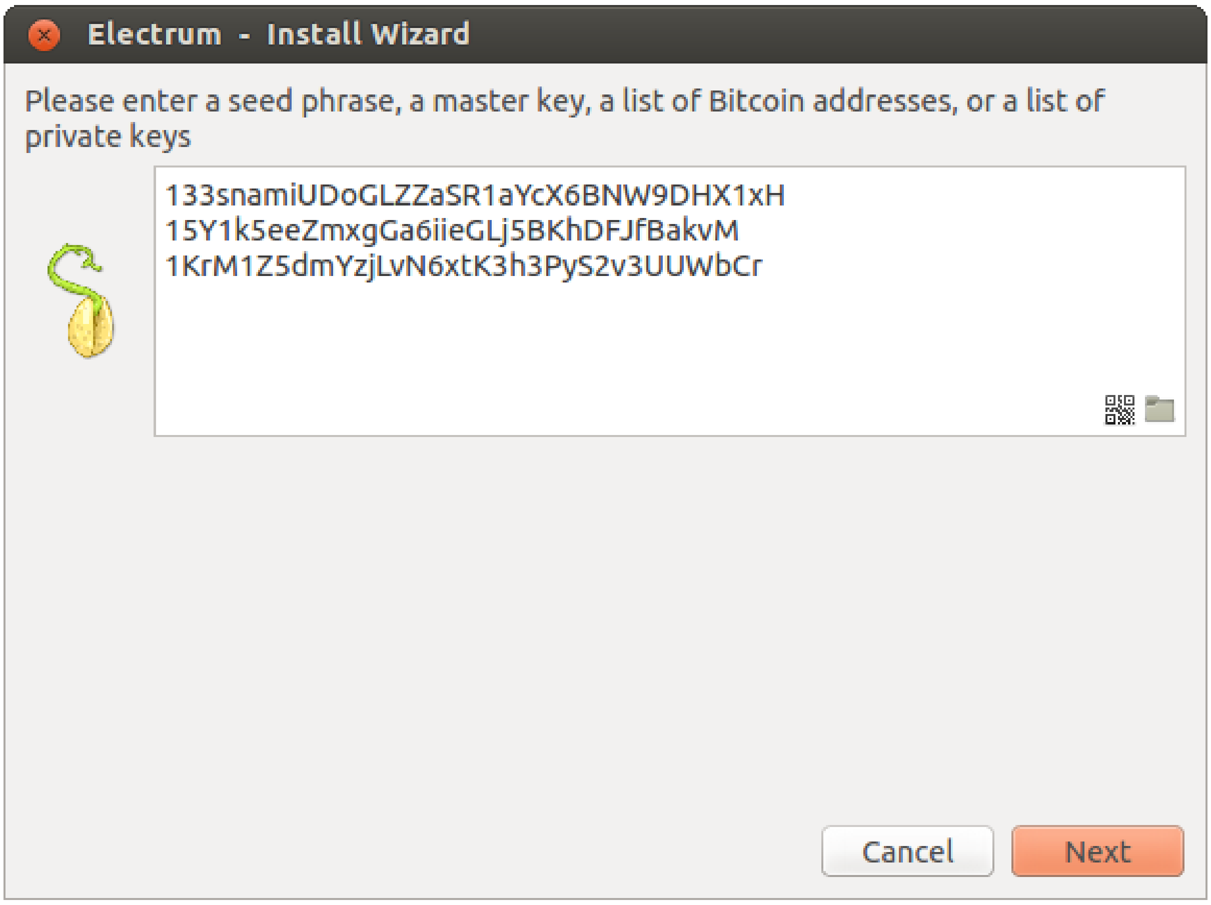
3.1.9 Does Electron Cash support cold wallets?

Yes. see *Cold Storage*

3.1.10 Can I import private keys from other Bitcoin Cash clients?

In Electron Cash, you cannot import private keys in a wallet that has a seed. You should sweep them instead.

If you want to import private keys and not sweep them you need to create a special wallet that does not have a seed. For this, create a new wallet, select "restore", and instead of typing your seed, type a list of private keys, or a list of addresses if you want to create a watching-only wallet.



Please enter a seed phrase, a master key, a list of Bitcoin addresses, or a list of private keys

```
133snamiUDoGLZZaSR1aYcX6BNW9DHX1xH
15Y1k5eeZmxgGa6iieGLj5BKhDFJfBakvM
1KrM1Z5dmYzjLvN6xtK3h3PyS2v3UUWbCr
```

You will need to back up this wallet, because it cannot be recovered from seed.

3.1.11 Can I sweep private keys from other Bitcoin Cash clients?

Sweeping private keys means to send all the Bitcoin Cash they control to an existing address in your wallet. The private keys you sweep do not become a part of your wallet. Instead, all the Bitcoin Cash they control are sent to an address that has been deterministically generated from your wallet seed.

To sweep private keys go to Wallet menu -> Private Keys -> Sweep. Enter the private keys in the appropriate field. Leave the 'Address' field unchanged. That is the destination address and it'll be from your existing Electron Cash wallet.

3.1.12 Where is my wallet file located?

The default wallet file is called default_wallet which is created when you first run the application and located under the /wallets folder.

On Windows:

- Show hidden files
- Go to \Users\YourUserName\AppData\Roaming\Electron Cash\wallets (or %APPDATA%\Electrum\wallets)

On Mac:

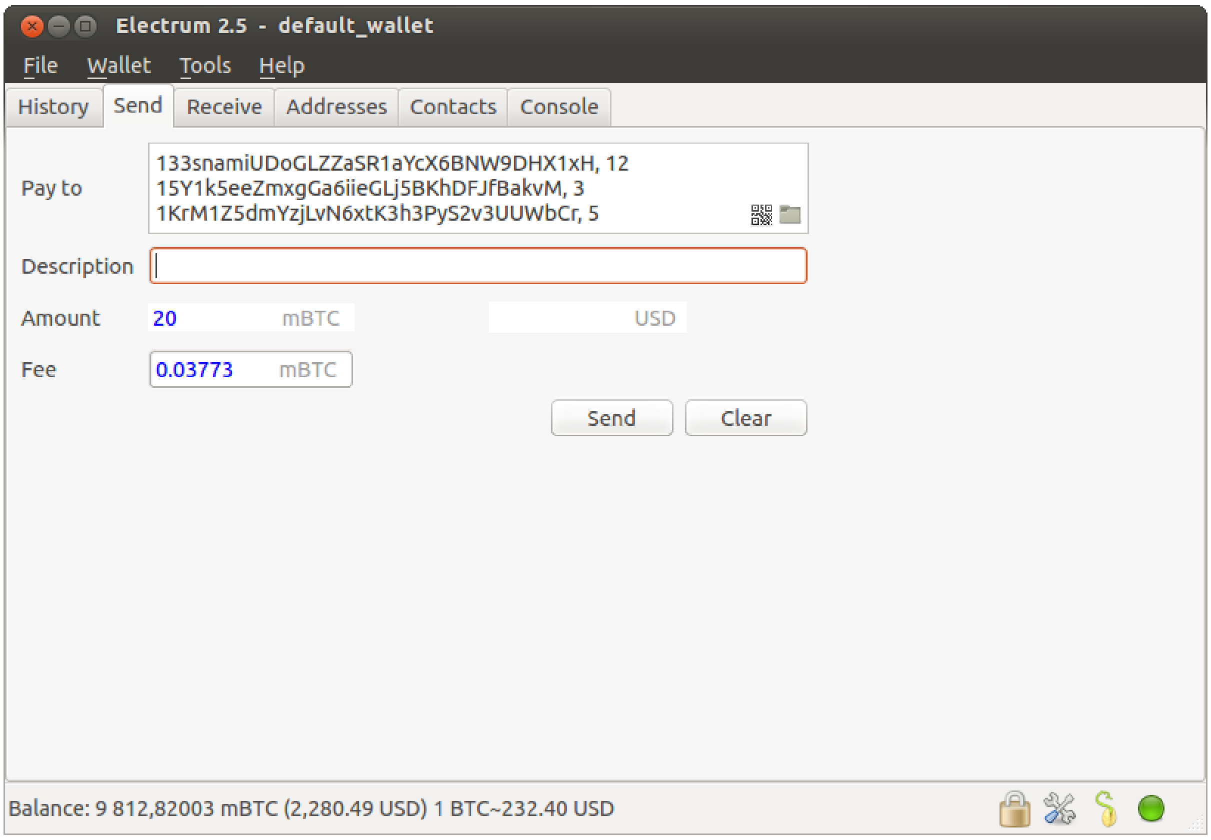
- Open Finder
- Go to folder (shift+cmd+G) and type ~/.electron-cash

On Linux:

- Home Folder
- Go -> Location and type ~/.electron-cash

3.1.13 Can I do bulk payments with Electron Cash?

You can create a transaction with several outputs. In the GUI, type each address and amount on a line, separated by a comma.


Electrum 2.5 - default_wallet
File Wallet Tools Help
History Send Receive Addresses Contacts Console
Send Clear

Pay to
```
133snamiUDoGLZZaSR1aYcX6BNW9DHX1xH, 12
15Y1k5eeZmxgGa6iieGLj5BKhDFJfBakvM, 3
1KrM1Z5dmYzjLvN6xtK3h3PyS2v3UUWbCr, 5
```
Description
Amount 20 mBTC USD
Fee 0.03773 mBTC

Balance: 9 812,82003 mBTC (2,280.49 USD) 1 BTC~232.40 USD

Amounts are in the current unit set in the client. The total is shown in the GUI.

You can also import a CSV file in the 'Pay to' field, by clicking on the folder icon.

3.1.14 Can Electron Cash create and sign raw transactions?

Electron Cash lets you create and sign raw transactions right from the user interface using a form.

3.1.15 Electron Cash freezes when I try to send Bitcoin Cash

This might happen if you are trying to spend a large number of transactions outputs (for example, if you have collected hundreds of donations from a faucet). When you send Bitcoin Cash, Electron Cash looks for unspent coins that are in your wallet, in order to create a new transaction. Unspent coins can have different values, much like physical coins and bills.

If this happens, you should consolidate your transaction inputs, by sending smaller amounts of bitcoins to one of your wallet addresses; this would be the equivalent of exchanging a stack of nickels for a dollar bill.

3.1.16 What is the gap limit?

The gap limit is the maximum number of consecutive unused addresses in your deterministic sequence of addresses. Electron Cash uses it in order to stop looking for addresses. In Electron Cash, it is set to 20 by default, so the client will get all addresses until 20 unused addresses are found.

3.1.17 How can I pre-generate new addresses?

Electron Cash will generate new addresses as you use them, until it hits the *gap limit*

If you need to pre-generate more addresses, you can do so by typing wallet.create_new_address(False) in the console. This command will generate one new address. Note that the address will be shown with a red background in the address tab, to indicate that it is beyond the gap limit. The red color will remain until the gap is filled.

WARNING: Addresses beyond the gap limit will not automatically be recovered from seed. To recover them will require either increasing the client's gap limit or generating new addresses until the used addresses are found.

If you wish to generate more than one address, you may use a 'for' loop. For example, if you wanted to generate 50 addresses, you could do this:

```
for x in range(0, 50):
    print wallet.create_new_address(False)
```

3.1.18 How to upgrade Electron Cash?

Warning: always save your wallet seed on paper before doing an upgrade.

To upgrade Electron Cash, just install the most recent version. The way to do this will depend on your OS.

Note that your wallet files are stored separately from the software, so you can safely remove the old version of the software if your OS does not do it for you.

Some Electron Cash upgrades will modify the format of your wallet files.

For this reason, it is not recommended to downgrade Electron Cash to an older version, once you have opened your wallet file with the new version. The older version will not always be able to read the new wallet file.

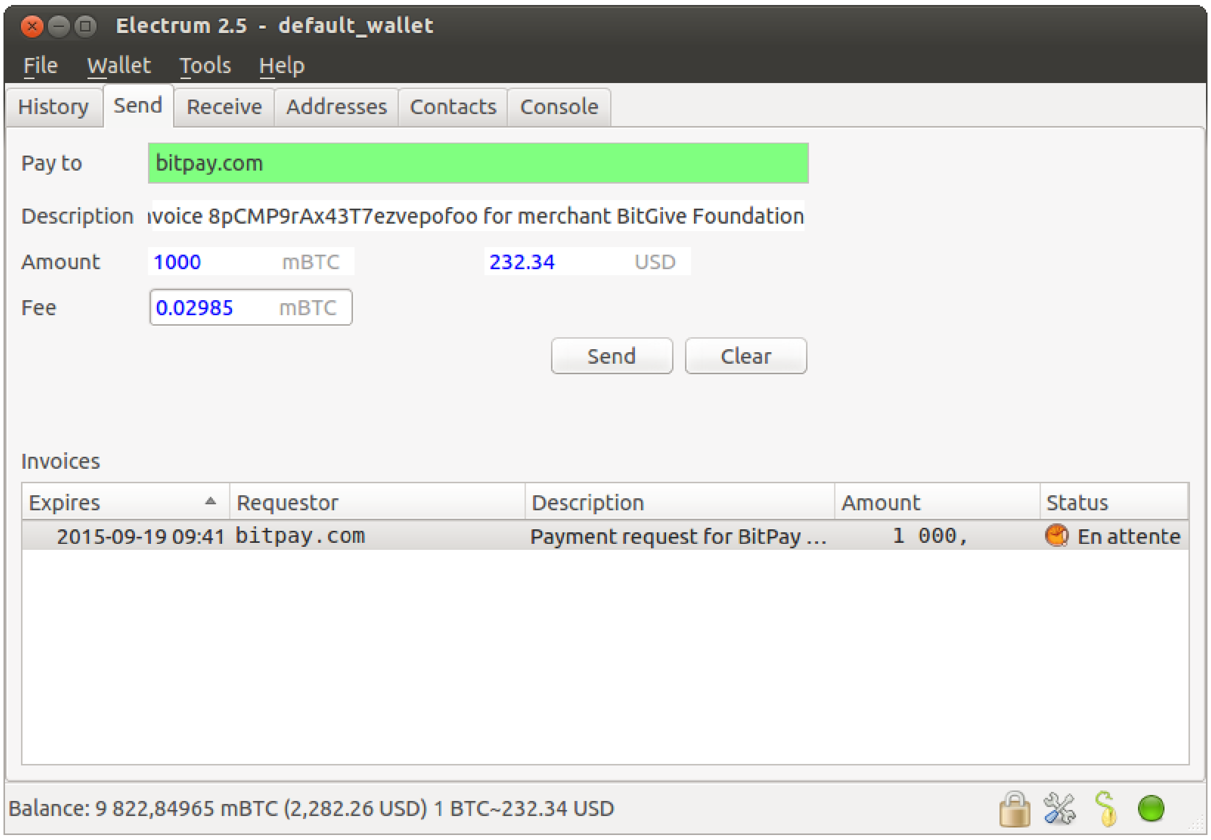
3.2 Invoices

Invoices are Payment Requests signed by their requestor.

When you click on a bitcoincash: link, a URL is passed to Electron Cash:

```
electron-cash "bitcoincash:1KLxqw4MA5NkG6YP1N4S14akDFCP1vQrKu?amount=1.0&amp;
↪r=https%3A%2F%2Fbitpay.com%2Fi%2FXxaGtEpRSqckRnhsjZwtrA"
```

This opens the send tab with the payment request:


Electrum 2.5 - default_wallet
File Wallet Tools Help
History Send Receive Addresses Contacts Console

Pay to: bitpay.com
Description: ıvoice 8pCMP9rAx43T7ezvepofoo for merchant BitGive Foundation
Amount: 1000 mBTC 232.34 USD
Fee: 0.02985 mBTC

Invoices

Expires	Requestor	Description	Amount	Status
2015-09-19 09:41	bitpay.com	Payment request for BitPay ...	1 000,	En attente

Balance: 9 822,84965 mBTC (2,282.26 USD) 1 BTC~232.34 USD

The green color in the “Pay To” field means that the payment request was signed by bitpay.com’s certificate, and that Electron Cash verified the chain of signatures.

Note that the “send” tab contains a list of invoices and their status.

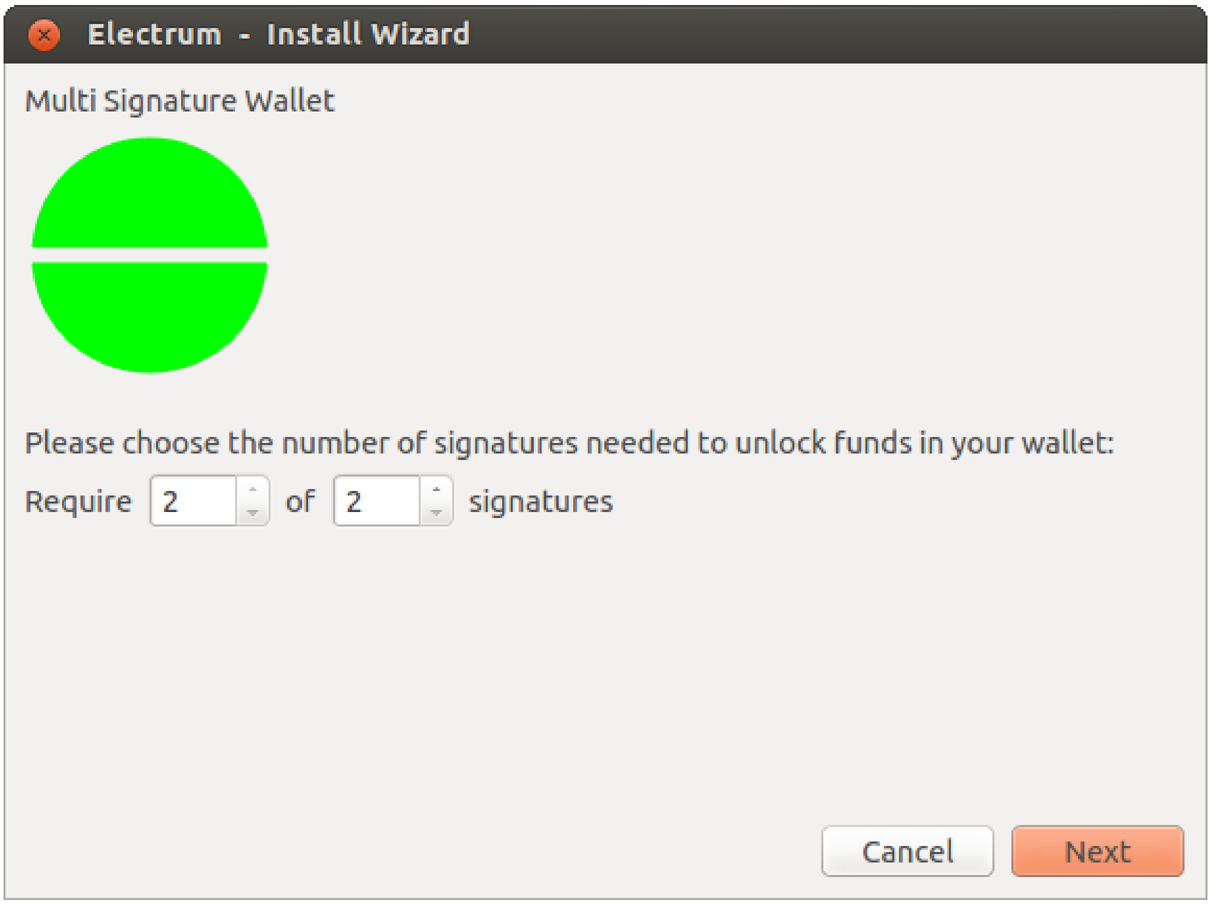
3.3 Multisig Wallets

This tutorial shows how to create a 2 of 2 multisig wallet. A 2 of 2 multisig consists of 2 separate wallets (usually on separate machines and potentially controlled by separate people) that have to be used in conjunction in order to access the funds. Both wallets have the same set of Addresses.

- A common use-case for this is if you want to collaboratively control funds: maybe you and your friend run a company together and certain funds should only be spendable if you both agree.
- Another one is security: One of the wallets can be on your main machine, while the other one is on a offline machine. That way you make it very hard for an attacker or malware to steal your coins.

3.3.1 Create a pair of 2-of-2 wallets

Each cosigner needs to do this: In the menu select File->New, then select “Multi-signature wallet”. On the next screen, select 2 of 2.

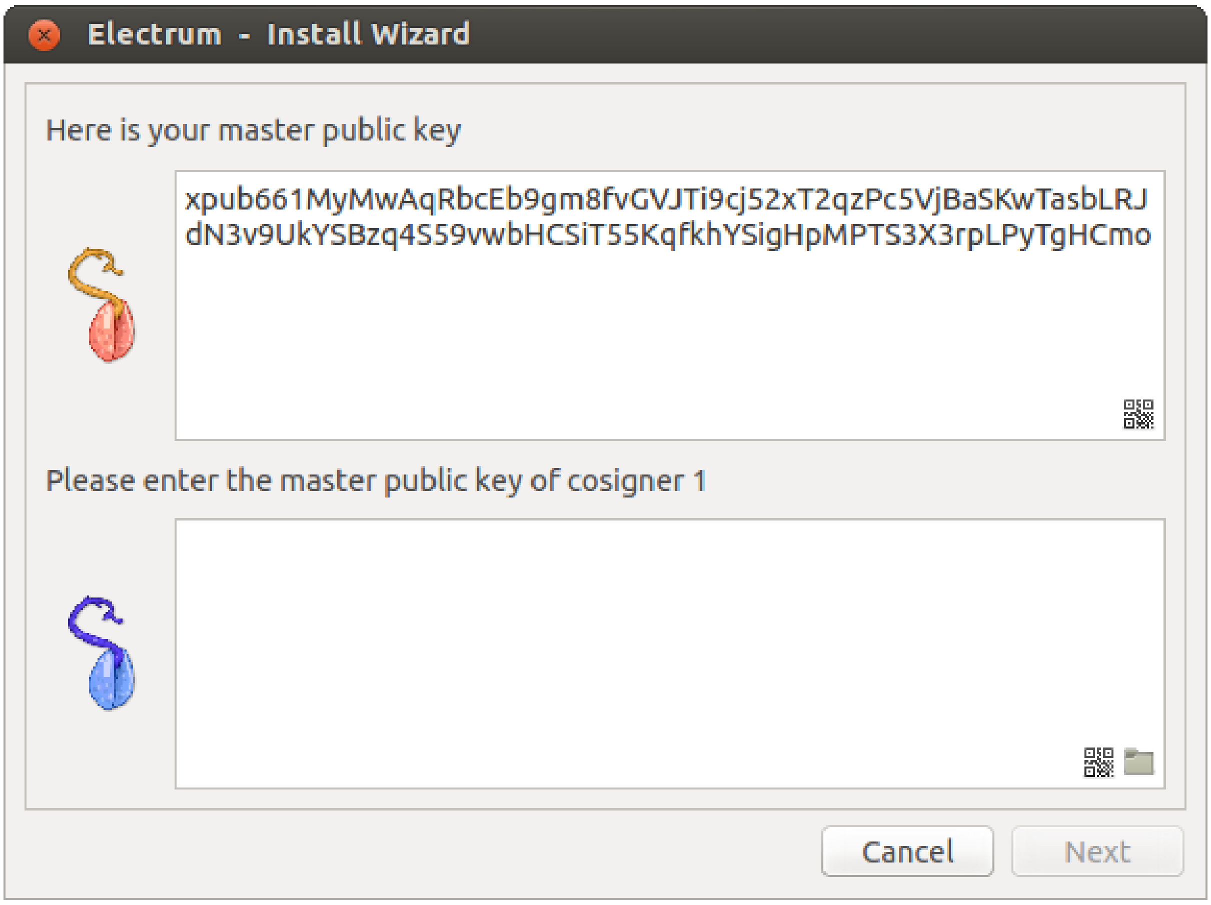


Multi Signature Wallet

Please choose the number of signatures needed to unlock funds in your wallet:

Require 2 of 2 signatures

After generating a seed (keep it safely!) you will need to provide the master public key of the other wallet.



Here is your master public key

xpub661MyMwAqRbcEb9gm8fvGVJTi9cj52xT2qzPc5VjBaSKwTasbLRJdN3v9UkYSBzq4S59vwbHCSiT55KqfkhYSigHpMPTS3X3rpLPyTgHCmo

Please enter the master public key of cosigner 1

Put the master public key of the other wallet into the lower box. Of course when you create the other wallet, you put the master public key of this one.

You will need to do this in parallel for the two wallets. Note that you can press cancel during this step, and reopen the file later.

3.3.2 Receiving

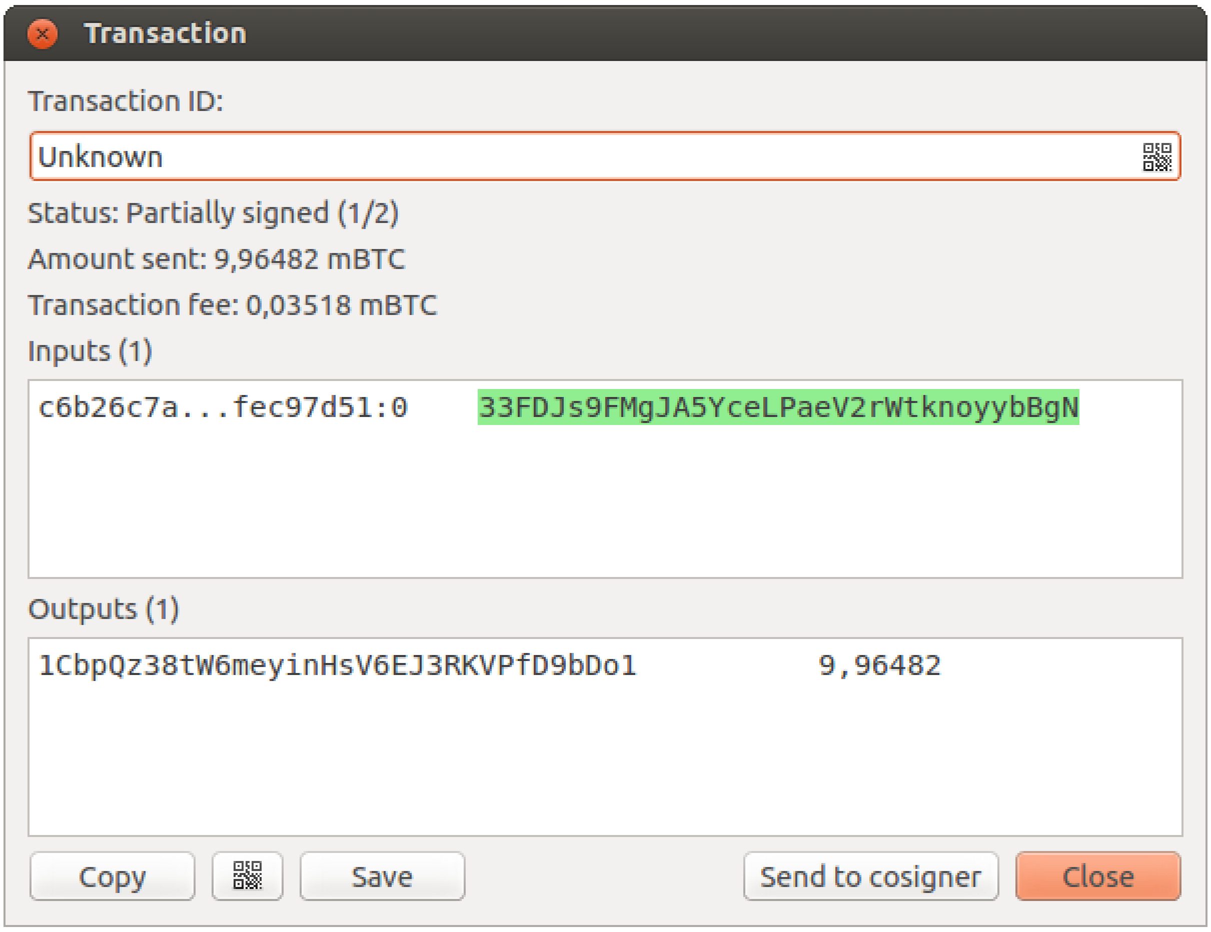
Check that both wallets generate the same set of Addresses. You can now send to these Addresses (note they start with a “3”) with any wallet that can send to P2SH Addresses.

3.3.3 Spending

To spend coins from a 2-of-2 wallet, two cosigners need to sign a transaction collaboratively.

To accomplish this, create a transaction using one of the wallets (by filling out the form on the “send” tab)

After signing, a window is shown with the transaction details.


Transaction

Transaction ID:

Unknown

Status: Partially signed (1/2)

Amount sent: 9,96482 mBTC

Transaction fee: 0,03518 mBTC

Inputs (1)

```
c6b26c7a...fec97d51:0     33FDJs9FMgJA5YceLPaeV2rWtknoyybBgN
```

Outputs (1)

```
1CbpQz38tW6meyinHsV6EJ3RKVPfD9bDo1          9,96482
```

The transaction has to be sent to the second wallet.

For this you have multiple options:

- you can transfer the file on a usb stick
- you can use QR codes
- you can use a remote server, with the CosignerPool plugin.

Transfer a file

You can save the partially signed transaction to a file (using the "save" button), transfer that to the machine where the second wallet is running (via usb stick, for example) and load it there (using Tools -> Load transaction -> from file)

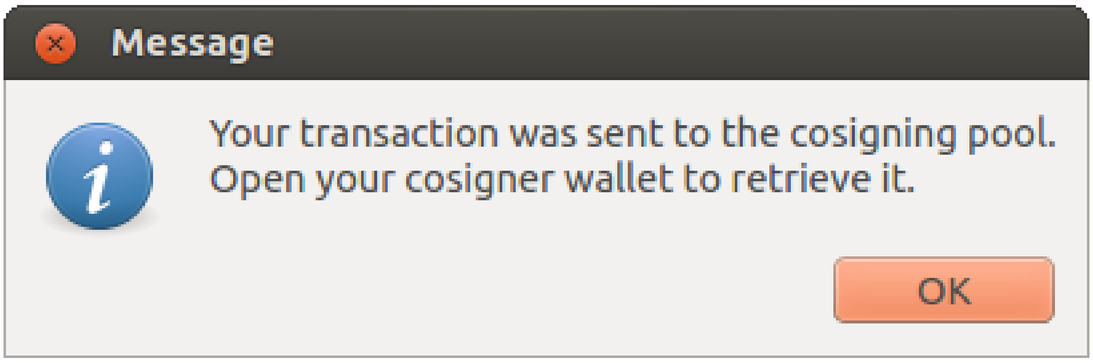
Use QR-Code

There's also a button showing a qr-code icon. Clicking that will display a qr-code containing the transaction that can be scanned into the second wallet (Tools -> Load Transaction -> From QR Code)

Use the Cosigner Pool Plugin

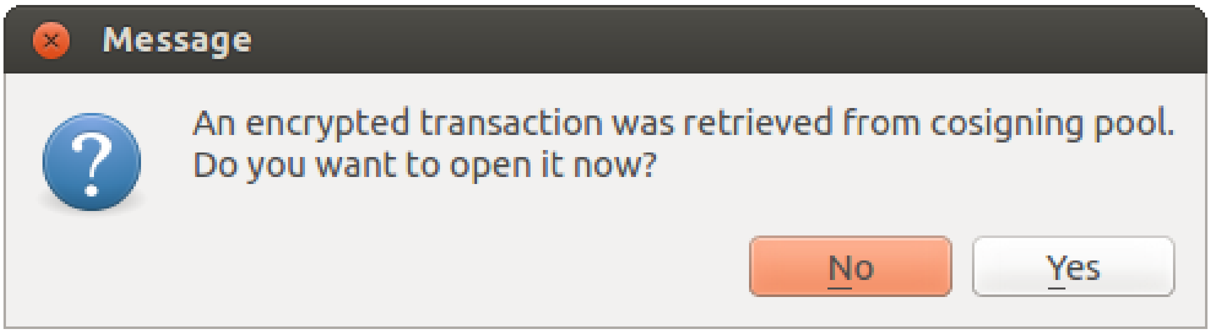
For this to work the Plugin "Cosigner Pool" needs to be enabled (Tools -> Plugins) with both wallets.

Once the plugin is enabled, you will see a button labeled "Send to cosigner". Clicking it sends the partially signed transaction to a central server. Note that the transaction is encrypted with your cosigner's master public key.


Message

Your transaction was sent to the cosigning pool.
Open your cosigner wallet to retrieve it.

When the cosigner wallet is started, it will get a notification that a partially signed transaction is available:


Message

An encrypted transaction was retrieved from cosigning pool.
Do you want to open it now?

The transaction is encrypted with the cosigner's master public key; the password is needed to decrypt it.

With all of the above methods, you can now add the seconds signature the the transaction (using the "sign" button). It will then be broadcast to the network.

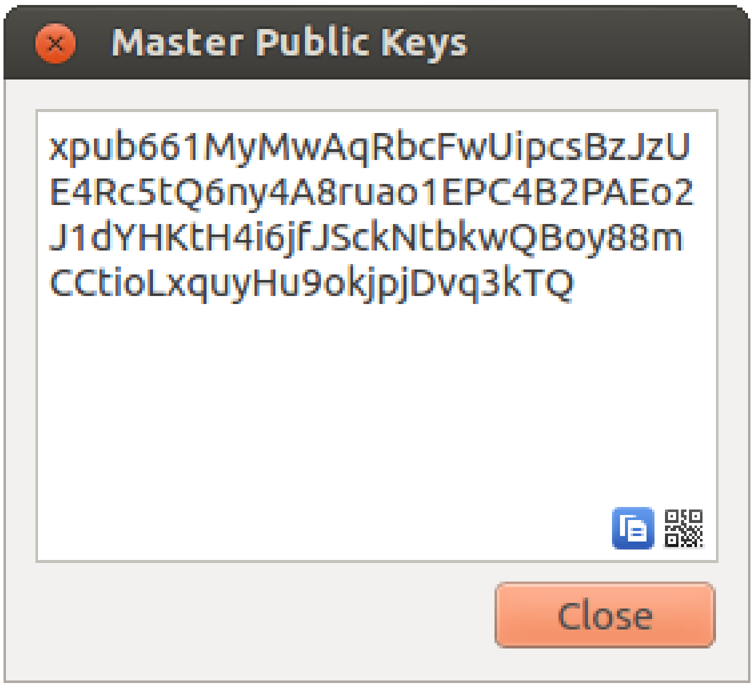
3.4 Cold Storage

This document shows how to create an offline wallet that holds your Bitcoin Cash and a watching-only online wallet that is used to view its history and to create transactions that have to be signed with the offline wallet before being broadcast on the online one.

3.4.1 Create an offline wallet

Create a wallet on an offline machine, as per the usual process (file -> new) etc.

After creating the wallet, go to Wallet -> Master Public Keys.

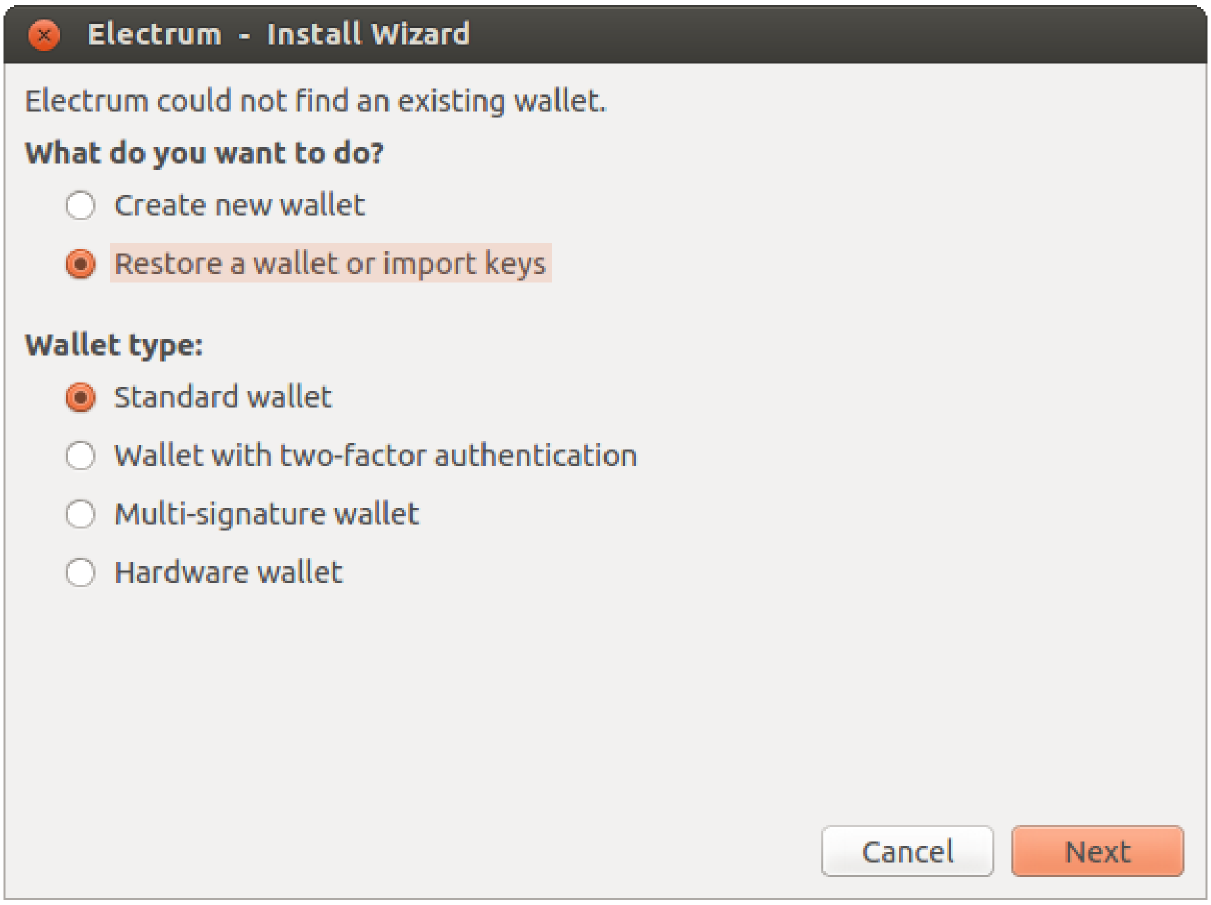

Master Public Keys

xpub661MyMwAqRbcFwUipcsBzJzUE4Rc5tQ6ny4A8ruao1EPC4B2PAEo2J1dYHKtH4i6jfJSckNtbkwQBoy88mCCtioLxquyHu9okjpjDvq3kTQ

The Master Public Key of your wallet is the string shown in this popup window. Transfer that key to your online machine somehow.

3.4.2 Create a watching-only version of your wallet

On your online machine, open up Electron Cash and select File -> New/Restore. Enter a name for the wallet and select "Restore a wallet or import keys".


Electrum - Install Wizard

Electrum could not find an existing wallet.

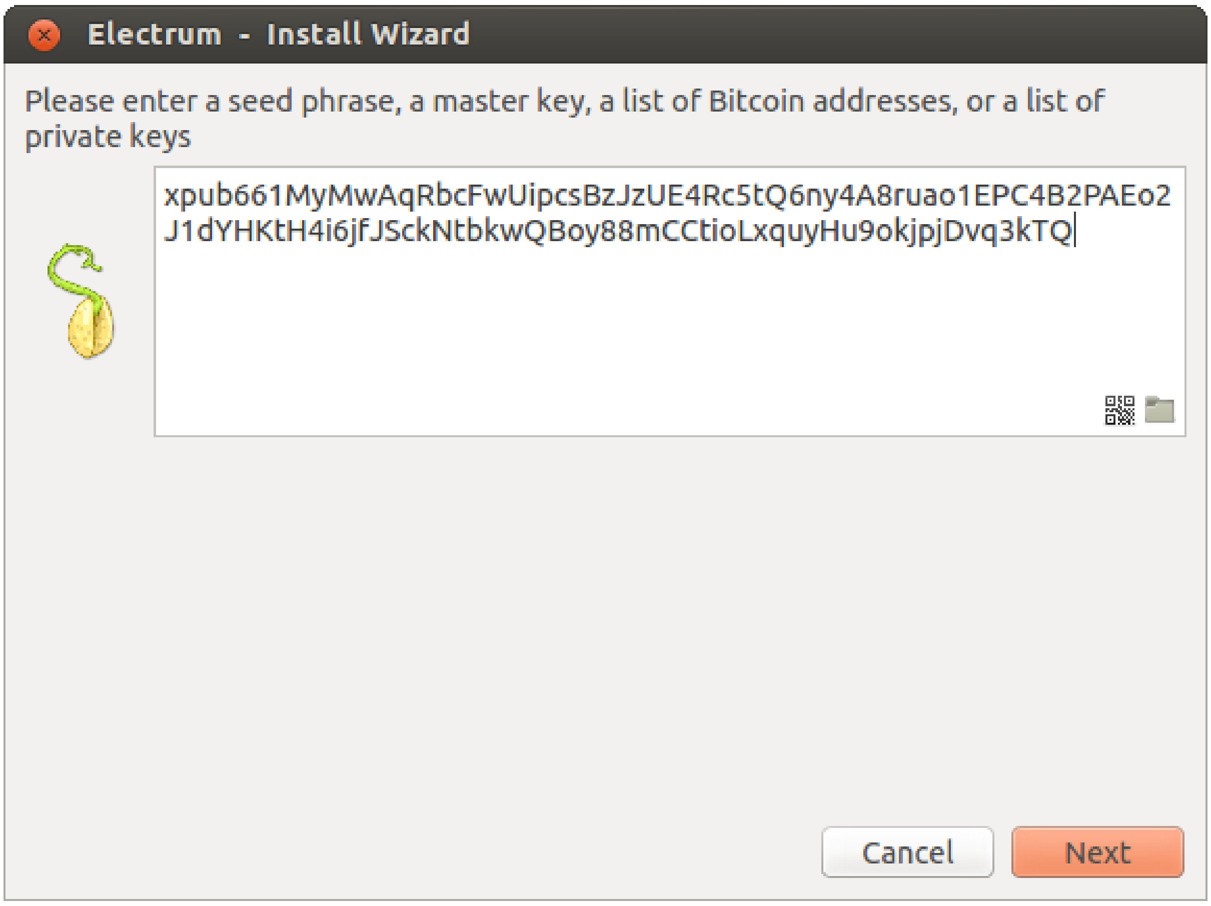
What do you want to do?

- ○ Create new wallet
- ● Restore a wallet or import keys

Wallet type:

- ● Standard wallet
- ○ Wallet with two-factor authentication
- ○ Multi-signature wallet
- ○ Hardware wallet

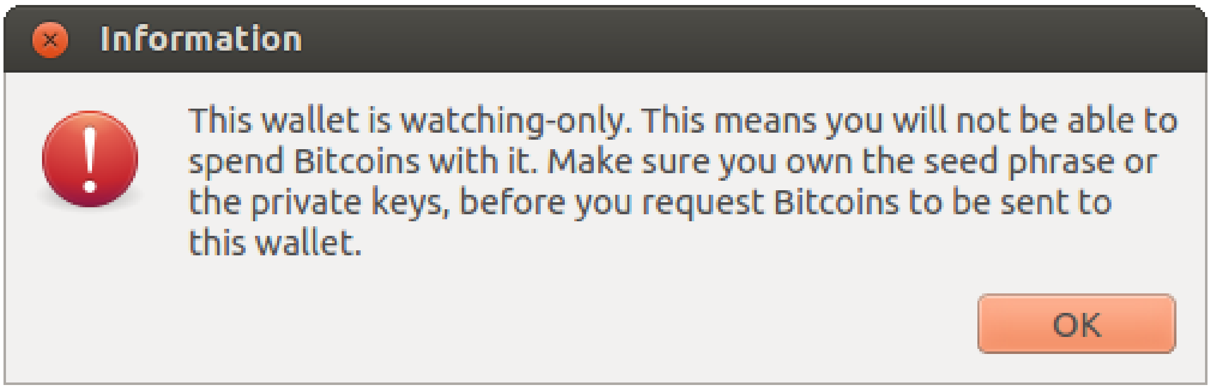
Paste your master public key in the box.


Electrum - Install Wizard

Please enter a seed phrase, a master key, a list of Bitcoin addresses, or a list of private keys

xpub661MyMwAqRbcFwUipcsBzJzUE4Rc5tQ6ny4A8ruao1EPC4B2PAEo2J1dYHKtH4i6jfJSckNtbkwQBoy88mCCtioLxquyHu9okjpjDvq3kTQ

Click Next to complete the creation of your wallet. When you're done, you should see a popup informing you that you are opening a watching-only wallet.

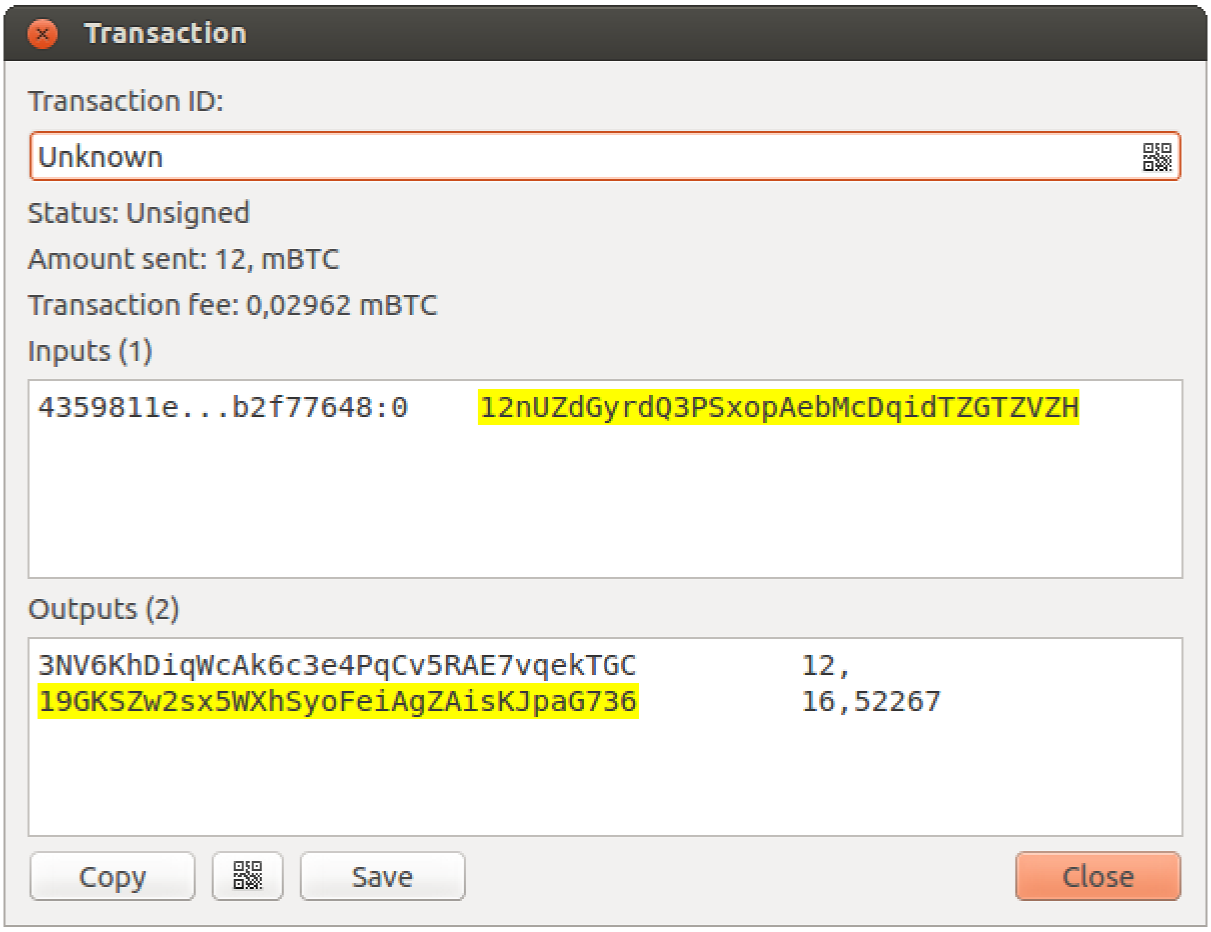

Information

This wallet is watching-only. This means you will not be able to spend Bitcoins with it. Make sure you own the seed phrase or the private keys, before you request Bitcoins to be sent to this wallet.

Then you should see the transaction history of your cold wallet.

3.4.3 Create an unsigned transaction

Go to the "send" tab on your online watching-only wallet, input the transaction data and press "Send…". A window opens up, informing you that a transaction fee will be added. Continue.


Transaction

Transaction ID:

Unknown

Status: Unsigned

Amount sent: 12, mBTC

Transaction fee: 0,02962 mBTC

Inputs (1)

```
4359811e...b2f77648:0     12nUZdGyrdQ3PSxopAebMcDqidTZGTZVZH
```

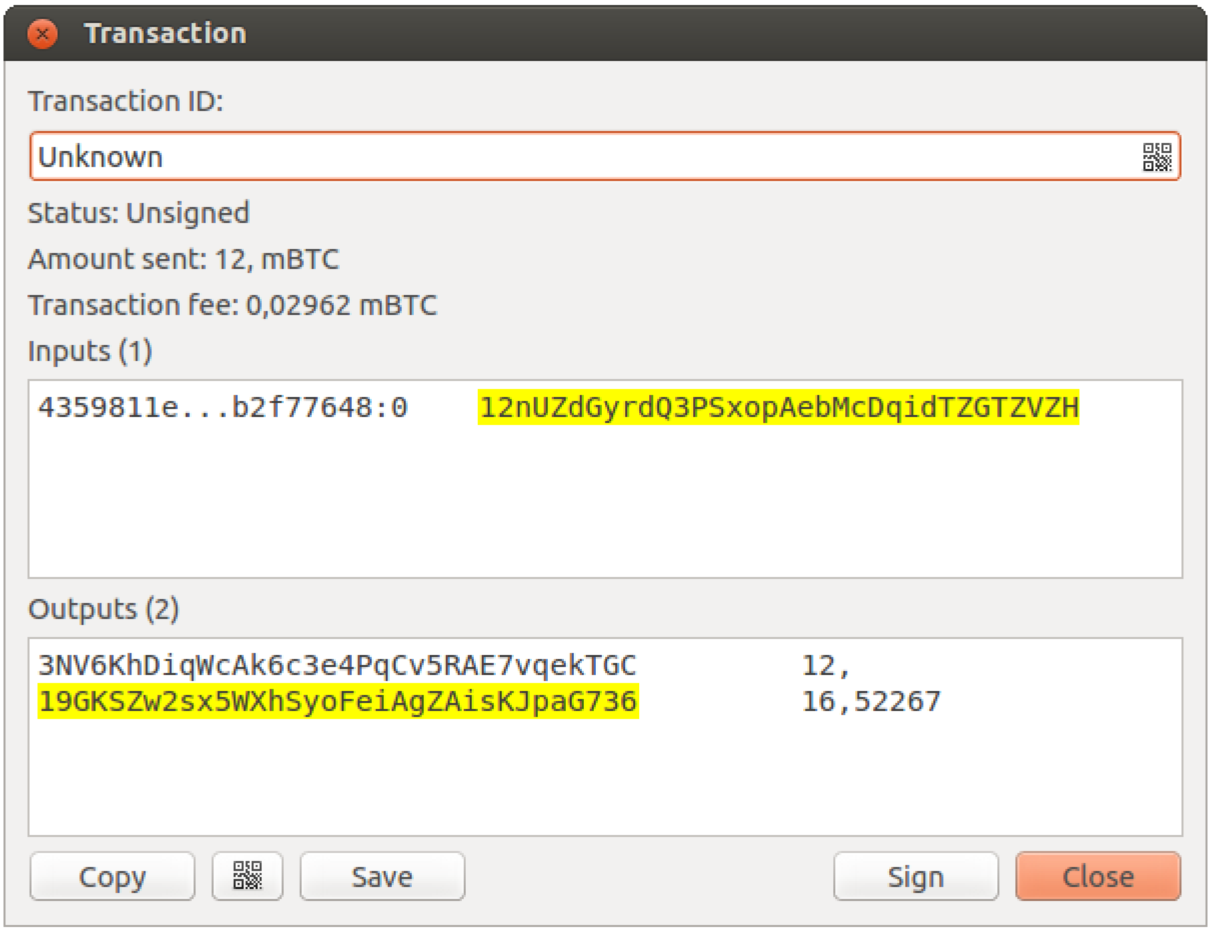
Outputs (2)

```
3NV6KhDiqWcAk6c3e4PqCv5RAE7vqekTGC          12,
19GKSZw2sx5WXhSyoFeiAgZAisKJpaG736          16,52267
```

In the window that opens up, press "save" and save the transaction file somewhere on your computer. Close the window and transfer the transaction file to your offline machine (e.g. with a usb stick).

3.4.4 Get your transaction signed

On your offline wallet, select Tools -> Load transaction -> From file in the menu and select the transaction file created in the previous step.


Transaction

Transaction ID:

Unknown

Status: Unsigned

Amount sent: 12, mBTC

Transaction fee: 0,02962 mBTC

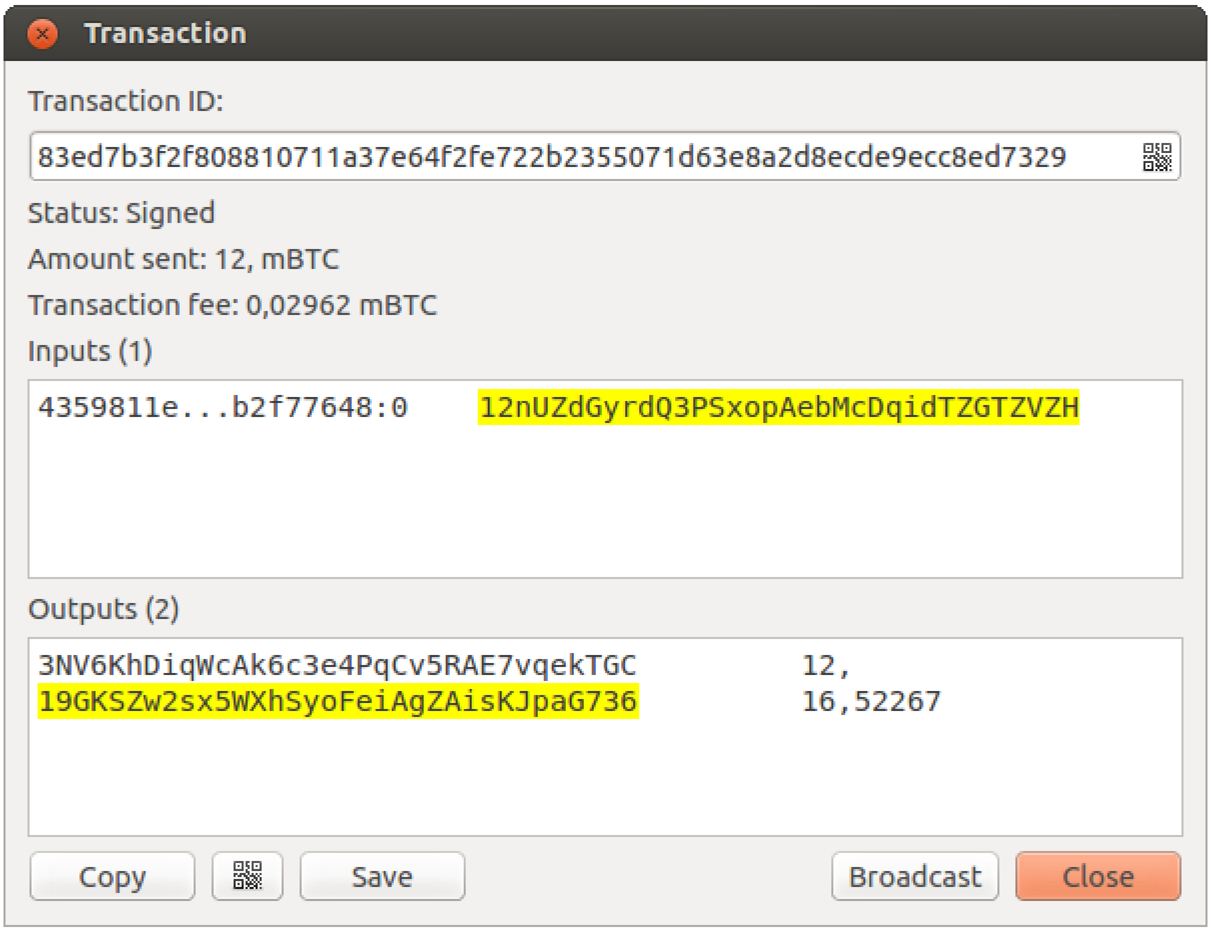
Inputs (1)

```
4359811e...b2f77648:0     12nUZdGyrdQ3PSxopAebMcDqidTZGTZVZH
```

Outputs (2)

```
3NV6KhDiqWcAk6c3e4PqCv5RAE7vqekTGC          12,
19GKSZw2sx5WXhSyoFeiAgZAisKJpaG736          16,52267
```

Press "sign". Once the transaction is signed, the Transaction ID appears in its designated field.



Transaction ID:

83ed7b3f2f808810711a37e64f2fe722b2355071d63e8a2d8ecde9ecc8ed7329

Status: Signed

Amount sent: 12, mBTC

Transaction fee: 0,02962 mBTC

Inputs (1)

```
4359811e...b2f77648:0     12nUZdGyrdQ3PSxopAebMcDqidTZGTZVZH
```

Outputs (2)

```
3NV6KhDiqWcAk6c3e4PqCv5RAE7vqekTGC          12,
19GKSZw2sx5WXhSyoFeiAgZAisKJpaG736          16,52267
```

Press save, store the file somewhere on your computer, and transfer it back to your online machine.

3.4.5 Broadcast your transaction

On your online machine, select Tools -> Load transaction -> From File from the menu. Select the signed transaction file. In the window that opens up, press "broadcast". The transaction will be broadcasted over the Bitcoin Cash network.

3.5 Command Line

Electron Cash has a powerful command line. This page will show you a few basic principles.

3.5.1 Using the inline help

To see the list of Electron Cash commands, type:

```
electron-cash help
```

To see the documentation for a command, type:

```
electron-cash help <command>
```

3.5.2 Magic words

The arguments passed to commands may be one of the following magic words: ! ? : and -.

- The exclamation mark ! is a shortcut that means 'the maximum amount available'.

  Example:

  ```
  electron-cash payto 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE !
  ```

  Note that the transaction fee will be computed and deducted from the amount.

- A question mark ? means that you want the parameter to be prompted.

  Example:

  ```
  electron-cash signmessage 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE ?
  ```

- Use a colon : if you want the prompted parameter to be hidden (not echoed in your terminal).

  ```
  electron-cash importprivkey :
  ```

  Note that you will be prompted twice in this example, first for the private key, then for your wallet password.

- A parameter replaced by a dash - will be read from standard input (in a pipe)

  ```
  cat LICENCE | electron-cash signmessage 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE -
  ```

3.5.3 Aliases

You can use DNS aliases in place of bitcoin addresses, in most commands.

```
electron-cash payto ecdsa.net !
```

3.5.4 Formatting outputs using jq

Command outputs are either simple strings or json structured data. A very useful utility is the 'jq' program. Install it with:

```
sudo apt-get install jq
```

The following examples use it.

3.5.5 Examples

Sign and verify message

We may use a variable to store the signature, and verify it:

```
sig=$(cat LICENCE| electron-cash signmessage 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE -)
```

And:

```
cat LICENCE | electron-cash verifymessage 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE $sig -
```

Show the values of your unspents

The 'listunspent' command returns a list of dict objects, with various fields. Suppose we want to extract the 'value' field of each record. This can be achieved with the jq command:

```
electron-cash listunspent | jq 'map(.value)'
```

Select only incoming transactions from history

Incoming transactions have a positive 'value' field

```
electron-cash history | jq '.[] | select(.value>0)'
```

Filter transactions by date

The following command selects transactions that were timestamped after a given date:

```
after=$(date -d '07/01/2015' +"%s")

electron-cash history | jq --arg after $after '.[] | select(.timestamp>(
↪$after|tonumber))'
```

Similarly, we may export transactions for a given time period:

```
before=$(date -d '08/01/2015' +"%s")

after=$(date -d '07/01/2015' +"%s")

electron-cash history | jq --arg before $before --arg after $after '.[] | select(.
↪timestamp&gt;($after|tonumber) and .timestamp&lt;($before|tonumber))'
```

Encrypt and decrypt messages

First we need the public key of a wallet address:

```
pk=$(electron-cash getpubkeys 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE| jq -r '.[0]')
```

Encrypt:

```
cat | electron-cash encrypt $pk -
```

Decrypt:

```
electron-cash decrypt $pk ?
```

Note: this command will prompt for the encrypted message, then for the wallet password

Export private keys and sweep coins

The following command will export the private keys of all wallet addresses that hold some bitcoins:

```
electron-cash listaddresses --funded | electron-cash getprivatekeys -
```

This will return a list of lists of private keys. In most cases, you want to get a simple list. This can be done by adding a jq filer, as follows:

```
electron-cash listaddresses --funded | electron-cash getprivatekeys - | jq 'map(.
↪[0])'
```

Finally, let us use this list of private keys as input to the sweep command:

```
electron-cash listaddresses --funded | electron-cash getprivatekeys - | jq 'map(.
↪[0])' | electron-cash sweep - 1uCMeviLYzwWh1P2gEh3R4X34ArzVUR1R
```

3.6 Using cold storage with the command line

This page will show you how to sign a transaction with an offline Electron Cash wallet, using the command line.

3.6.1 Create an unsigned transaction

With your online (watching-only) wallet, create an unsigned transaction:

```
electron-cash payto 1Cpf9zb5Rm5Z5qmmGezn6ERxFWvwuZ6UCx 0.1 --unsigned > unsigned.
↪txn
```

The unsigned transaction is stored in a file named 'unsigned.txn'. Note that the –unsigned option is not needed if you use a watching-only wallet.

You may view it using:

```
cat unsigned.txn | electron-cash deserialize -
```

3.6.2 Sign the transaction

The serialization format of Electron Cash contains the master public key needed and key derivation, used by the offline wallet to sign the transaction.

Thus we only need to pass the serialized transaction to the offline wallet:

```
cat unsigned.txn | electron-cash signtransaction - > signed.txn
```

The command will ask for your password, and save the signed transaction in 'signed.txn'

3.6.3 Broadcast the transaction

Send your transaction to the Bitcoin Cash network, using broadcast:

```
cat signed.txn | electron-cash broadcast -
```

If successful, the command will return the ID of the transaction.

3.7 How to accept Bitcoin Cash on a website using Electron Cash

This tutorial will show you how to accept Bitcoin Cash on a website with SSL signed payment requests.

3.7.1 Requirements

- A webserver serving static HTML
- A SSL certificate (signed by a CA)
- Electron Cash

3.7.2 Create a wallet

Create a wallet on your web server:

```
electron-cash create
```

You can also use a watching only wallet (restored from xpub), if you want to keep private keys off the server.

Once your wallet is created, start Electron Cash as a daemon:

```
electron-cash daemon start
```

3.7.3 Add your SSL certificate to your configuration

You should have a private key and a public certificate for your domain.

Create a file that contains only the private key:

```
-----BEGIN PRIVATE KEY-----
your private key
-----BEGIN END KEY-----
```

Set the path to your the private key file with setconfig:

```
electron-cash setconfig ssl_privkey /path/to/ssl.key
```

Create another file, file that contains your certificate, and the list of certificates it depends on, up to the root CA. Your certificate must be at the top of the list, and the root CA at the end.

```
-----BEGIN CERTIFICATE-----
your cert
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
intermediate cert
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
root cert
-----END CERTIFICATE-----
```

Set the ssl_chain path with setconfig:

```
electron-cash setconfig ssl_chain /path/to/ssl.chain
```

3.7.4 Configure a requests directory

This directory must be served by your webserver (eg Apache)

```
electron-cash setconfig requests_dir /var/www/r/
```

By default, Electron Cash will display local URLs, starting with 'file://' In order to display public URLs, we need to set another configuration variable, url_rewrite. For example:

```
electron-cash setconfig url_rewrite "['file:///var/www/','https://electroncash.org/
↪']"
```

3.7.5 Create a signed payment request

```
electron-cash addrequest 3.14 -m "this is a test"
{
   "URI": "bitcoincash:1MP49h5fbfLXiFpomsXeqJHGHUfNf3mCo4?amount=3.14&r=https://
↪electroncash.org/r/7c2888541a",
   "address": "1MP49h5fbfLXiFpomsXeqJHGHUfNf3mCo4",
   "amount": 314000000,
   "amount (BCC)": "3.14",
   "exp": 3600,
   "id": "7c2888541a",
   "index_url": "https://electroncash.org/r/index.html?id=7c2888541a",
   "memo": "this is a test",
   "request_url": "https://electroncash.org/r/7c2888541a",
   "status": "Pending",
   "time": 1450175741
}
```

This command returns a json object with two URLs:

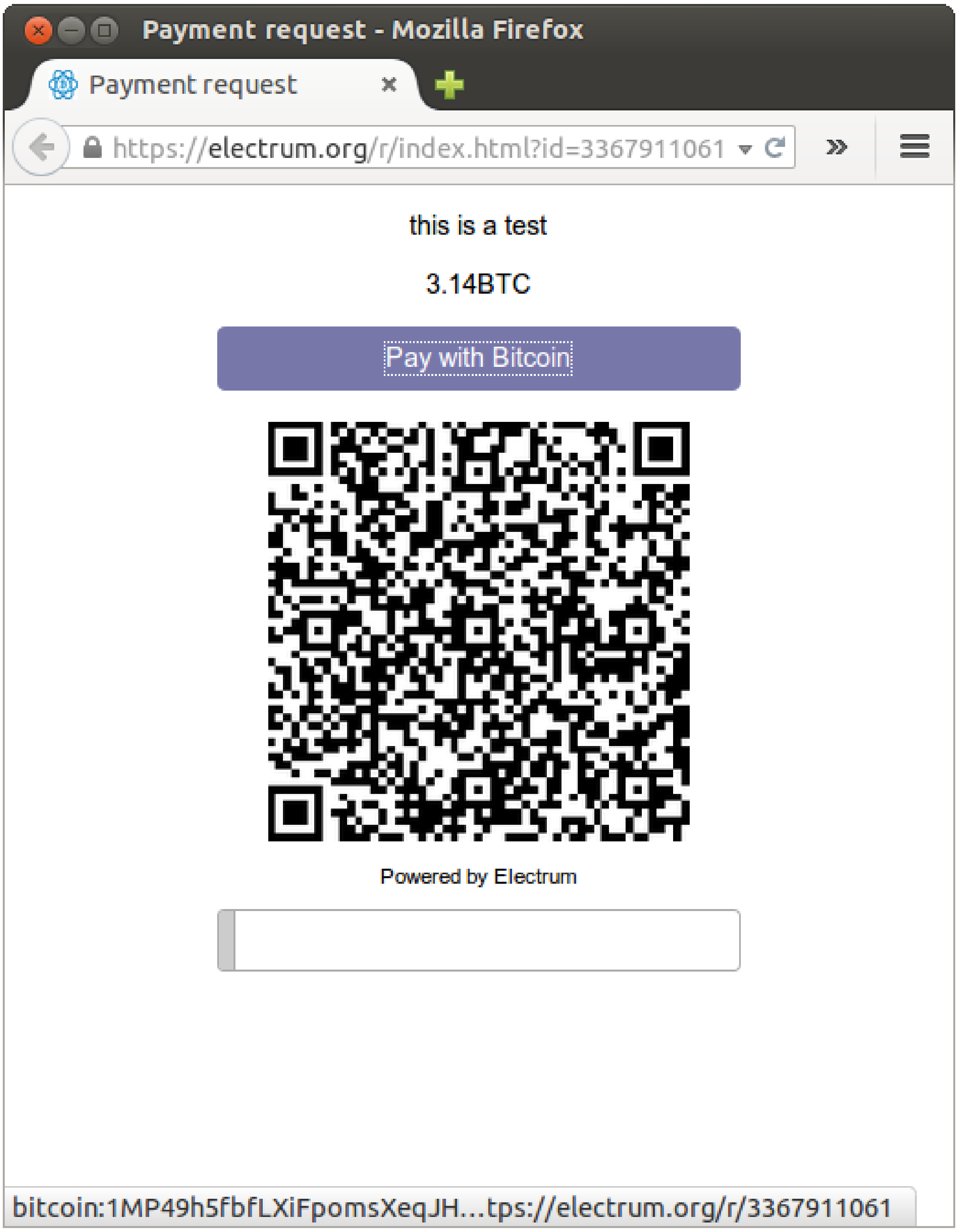
- request_url is the URL of the signed BIP70 request.
- index_url is the URL of a webpage displaying the request.

Note that request_url and index_url use the domain name we defined in url_rewrite.

You can view the current list of requests using the ‘listrequests’ command.

3.7.6 Open the payment request page in your browser

Let us open index_url in a web browser.


Payment request - Mozilla Firefox
Payment request
https://electrum.org/r/index.html?id=3367911061

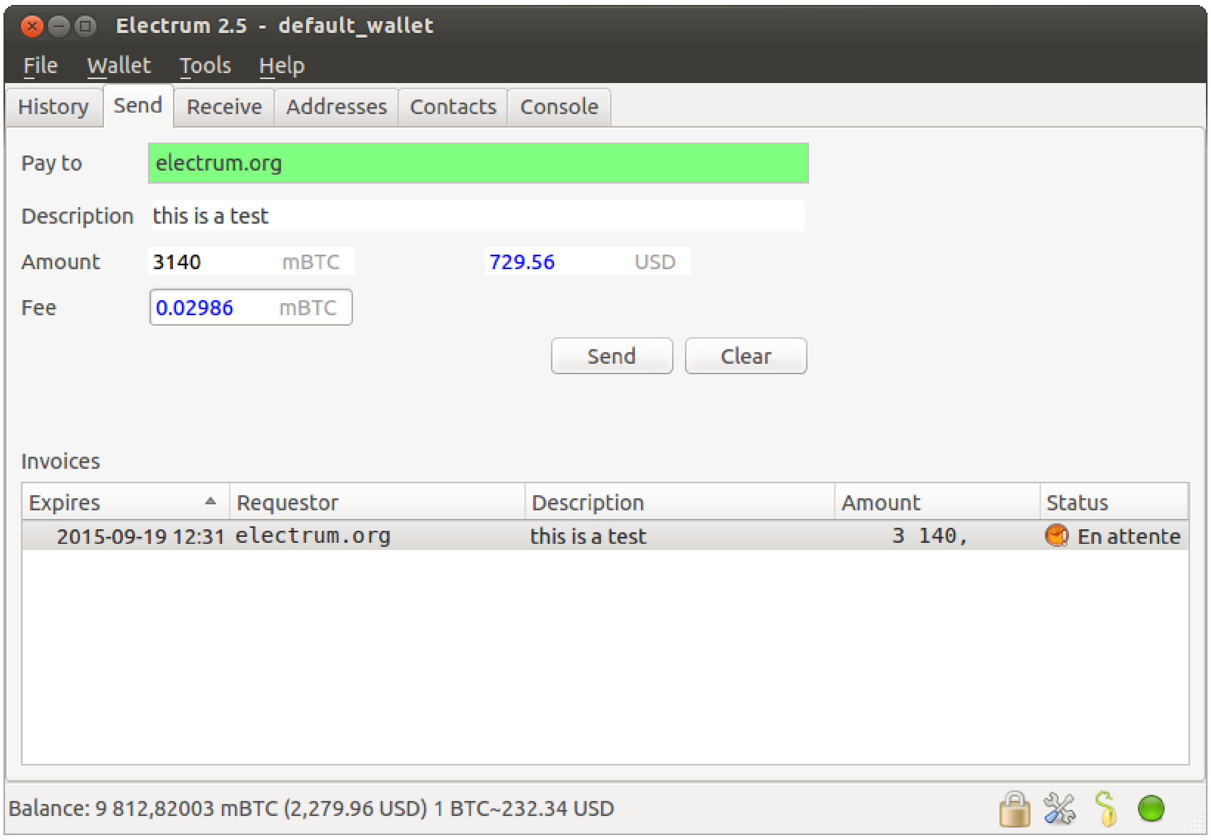
this is a test

3.14BTC

Pay with Bitcoin
Powered by Electrum

bitcoin:1MP49h5fbfLXiFpomsXeqJH...tps://electrum.org/r/3367911061

The page shows the payment request. You can open the bitcoincash: URI with a wallet, or scan the QR code. The bottom line displays the time remaining until the request expires.



Pay to: electrum.org
Description: this is a test
Amount: 3140 mBTC 729.56 USD
Fee: 0.02986 mBTC

Invoices

Expires	Requestor	Description	Amount	Status
2015-09-19 12:31	electrum.org	this is a test	3 140,	En attente

Balance: 9 812,82003 mBTC (2,279.96 USD) 1 BTC~232.34 USD

This page can already used to receive payments. However, it will not detect that a request has been paid; for that we need to configure websockets

3.7.7 Add web sockets support

Get SimpleWebSocketServer from here:

```
git clone https://github.com/ecdsa/simple-websocket-server.git
```

Set `websocket_server` and `websocket_port` in your config:

```
electron-cash setconfig websocket_server <FQDN of your server>

electron-cash setconfig websocket_port 9999
```

And restart the daemon:

```
electron-cash daemon stop

electron-cash daemon start
```

Now, the page is fully interactive: it will update itself when the payment is received. Please notice that higher ports might be blocked on some client's firewalls, so it is more safe for example to reverse proxy websockets transmission using standard `443` port on an additional subdomain.

3.7.8 JSONRPC interface

Commands to the Electron Cash daemon can be sent using JSONRPC. This is useful if you want to use Electron Cash in a PHP script.

Note that the daemon uses a random port number by default. In order to use a stable port number, you need to set the 'rpcport' configuration variable (and to restart the daemon):

```
electron-cash setconfig rpcport 7777
```

With this setting, we can perform queries using curl or PHP. Example:

```
curl --data-binary '{"id":"curltext","method":"getbalance","params":[]}' http://
↪127.0.0.1:7777
```

Query with named parameters:

```
curl --data-binary '{"id":"curltext","method":"listaddresses","params":{"funded
↪":true}}' http://127.0.0.1:7777
```

Create a payment request:

```
curl --data-binary '{"id":"curltext","method":"addrequest","params":{"amount":"3.14
↪","memo":"test"}}' http://127.0.0.1:7777
```

3.8 The Python Console

Most Electron Cash commands are available not only using the command-line, but also in the GUI Python console.

The results are Python objects, even though they are sometimes rendered as JSON for clarity.

Let us call listunspent(), to see the list of unspent outputs in the wallet:

```
>> listunspent()
[
 {
     "address": "12cmY5RHRgx8KkUKASDcDYRotget9FNso3",
     "index": 0,
     "raw_output_script": "76a91411bbdc6e3a27c44644d83f783ca7df3bdc2778e688ac",
     "tx_hash": "e7029df9ac8735b04e8e957d0ce73987b5c9c5e920ec4a445130cdeca654f096",
     "value": 0.01
 },
 {
     "address": "1GavSCND6TB7HuCnJSTEbHEmCctNGeJwXF",
     "index": 0,
     "raw_output_script": "76a914aaf437e25805f288141bfcdc27887ee5492bd13188ac",
     "tx_hash": "b30edf57ca2a31560b5b6e8dfe567734eb9f7d3259bb334653276efe520735df",
     "value": 9.04735316
 }
]
```

Note that the result is rendered as JSON. However, if we save it to a Python variable, it is rendered as a Python object:

```
>> u = listunspent()
>> u
[{'tx_hash': u'e7029df9ac8735b04e8e957d0ce73987b5c9c5e920ec4a445130cdeca654f096',
↪'index': 0, 'raw_output_script':
↪'76a91411bbdc6e3a27c44644d83f783ca7df3bdc2778e688ac', 'value': 0.01, 'address':
↪'12cmY5RHRgx8KkUKASDcDYRotget9FNso3'}, {'tx_hash': u
↪'b30edf57ca2a31560b5b6e8dfe567734eb9f7d3259bb334653276efe520735df', 'index': 0,
↪'raw_output_script': '76a914aaf437e25805f288141bfcdc27887ee5492bd13188ac', 'value
↪': 9.04735316, 'address': '1GavSCND6TB7HuCnJSTEbHEmCctNGeJwXF'}]
```

This makes it possible to combine Electron Cash commands with Python. For example, let us pick only the addresses in the previous result:

```
>> map(lambda x:x.get('address'), listunspent())
[
 "12cmY5RHRgx8KkUKASDcDYRotget9FNso3",
 "1GavSCND6TB7HuCnJSTEbHEmCctNGeJwXF"
]
```

Here we combine two commands, listunspent and dumpprivkeys, in order to dump the private keys of all adresses that have unspent outputs:

```
>> dumpprivkeys( map(lambda x:x.get('address'), listunspent()) )
{
 "12cmY5RHRgx8KkUKASDcDYRotget9FNso3":
↪"***************************************************",
 "1GavSCND6TB7HuCnJSTEbHEmCctNGeJwXF":
↪"***************************************************"
}
```

Note that dumpprivkey will ask for your password if your wallet is encrypted. The GUI methods can be accessed through the gui variable. For example, you can display a QR code from a string using gui.show_qrcode. Example:

```
gui.show_qrcode(dumpprivkey(listunspent()[0]['address']))
```

3.9 Simple Payment Verification

Simple Payment Verification (SPV) is a technique described in Satoshi Nakamoto's paper. SPV allows a lightweight client to verify that a transaction is included in the Bitcoin blockchain, without downloading the entire blockchain. The SPV client only needs download the block headers, which are much smaller than the full blocks. To verify that a transaction is in a block, a SPV client requests a proof of inclusion, in the form of a Merkle branch.

SPV clients offer more security than web wallets, because they do not need to trust the servers with the information they send.

Reference: Bitcoin: A peer-to-peer Electronic Cash System

3.10 Electrum protocol specification

Stratum is a universal bitcoin communication protocol used mainly by the bitcoin client Electrum, the Bitcoin Cash client Electron Cash, and miners.

3.10.1 Format

Stratum protocol is based on JSON-RPC 2.0 (although it doesn't include "jsonrpc" information in every message). Each message has to end with a line end character (n).

Request

Typical request looks like this:

```
{ "id": 0, "method":"some.stratum.method", "params": [] }
```

- id begins at 0 and every message has its unique id number
- list and description of possible methods is below
- params is an array, e.g.: ["1myfirstaddress", "1mysecondaddress", "1andonemoreaddress"]

Response

Responses are similar:

```
{ "id": 0, "result":
↪"616be06545e5dd7daec52338858b6674d29ee6234ff1d50120f060f79630543c"}
```

- id is copied from the request message (this way client can pair each response to one of his requests)
- result can be:
  - null
  - a string (as shown above)
  - a hash, e.g.:

    ```
    { "nonce": 1122273605, "timestamp": 1407651121, "version": 2, "bits":␣
    ↪406305378 }
    ```

  - an array of hashes, e.g.:

    ```
    [ { "tx_hash:
    "b87bc42725143f37558a0b41a664786d9e991ba89d43a53844ed7b3752545d4f",
    "height": 314847 }, { "tx_hash":
    "616be06545e5dd7daec52338858b6674d29ee6234ff1d50120f060f79630543c",
    "height": 314853 } ]
    ```

3.10.2 Methods

server.version

This is usually the first client's message, plus it's sent every minute as a keep-alive message. Client sends its own version and version of the protocol it supports. Server responds with its supported version of the protocol (higher number at server-side is usually compatible).

The version of the protocol being explained in this documentation is: 0.10.

request:

```
{ "id": 0, "method": "server.version", "params": [ "1.9.5", "0.6" ] }
```

response:

```
{ "id": 0, "result": "0.8" }
```

server.banner

request:

```
{ "id": 1, "method": "server.banner", "params": [] }
```

server.donation_address

server.peers.subscribe

Client can this way ask for a list of other active servers. Servers are connected to an IRC channel (#electrum at freenode.net) where they can see each other. Each server announces its version, history pruning limit of every address ("p100", "p10000" etc.–the number means how many transactions the server may keep for every single address) and supported protocols (“t” = tcp@50001, “h” = http@8081, “s” = tcp/tls@50002, “g” = https@8082; non-standard port would be announced this way: “t3300” for tcp on port 3300).

Note: At the time of writing there isn’t a true subscription implementation of this method, but servers only send one-time response. They don’t send notifications yet.

request:

```
{ "id": 3, "method":
"server.peers.subscribe", "params": [] }<br/>
```

response:

```
{ "id": 3, "result": [ [ "83.212.111.114",
"electrum.stepkrav.pw", [ "v0.9", "p100", "t", "h", "s",
"g" ] ], [ "23.94.27.149", "ultra-feather.net", [ "v0.9",
"p10000", "t", "h", "s", "g" ] ], [ "88.198.241.196",
"electrum.be", [ "v0.9", "p10000", "t", "h", "s", "g" ] ] ]
}
```

blockchain.numblocks.subscribe

A request to send to the client notifications about new blocks height. Responds with the current block height.

request:

```
{ "id": 5, "method":
"blockchain.numblocks.subscribe", "params": [] }
```

response:

```
{ "id": 5, "result": 316024 }
```

message:

```
{ "id": null, "method":
"blockchain.numblocks.subscribe", "params": 316024 }
```

blockchain.headers.subscribe

A request to send to the client notifications about new blocks in form of parsed blockheaders.

request:

```
{ "id": 5, "method":
"blockchain.headers.subscribe", "params": [] }
```

response:

```
{ "id": 5, "result": { "nonce":
3355909169, "prev_block_hash":
"00000000000000002b3ef284c2c754ab6e6abc40a0e31a974f966d8a2b4d5206",
"timestamp": 1408252887, "merkle_root":
"6d979a3d8d0f8757ed96adcd4781b9707cc192824e398679833abcb2afdf8d73",
"block_height": 316023, "utxo_root":
"4220a1a3ed99d2621c397c742e81c95be054c81078d7eeb34736e2cdd7506a03",
"version": 2, "bits": 406305378 } }
```

message:

```
{ "id": null, "method":
"blockchain.headers.subscribe", "params": [ { "nonce":
881881510, "prev_block_hash":
"00000000000000001ba892b1717690900ae476857120a78fb50825f8b67a42d4",
"timestamp": 1408255430, "merkle_root":
"8e92bdbf1c5c581b5942fc290c6c52c586f091b279ea79d4e21460e138023839",
"block_height": 316024, "utxo_root":
"060f780c0dd07c4289aaaa2ef24723f73380095b31d60795e1308170ec742ffb",
"version": 2, "bits": 406305378 } ] }
```

blockchain.address.subscribe

A request to send to the client notifications when status (i.e., transaction history) of the given address changes. Status is a hash of the transaction history. If there isn't any transaction for the address yet, the status is null.

request:

```
{ "id": 6, "method":"blockchain.address.subscribe", "params": [
↪"1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L"] }
```

response:

```
{ "id": 6, "result":
↪"b87bc42725143f37558a0b41a664786d9e991ba89d43a53844ed7b3752545d4f" }
```

message:

```
{ "id": null, "method":"blockchain.address.subscribe", "params": [
↪"1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L",
↪"690ce08a148447f482eb3a74d714f30a6d4fe06a918a0893d823fd4aca4df580"]}
```

blockchain.address.get_history

For a given address a list of transactions and their heights (and fees in newer versions) is returned.

request:

```
{"id": 1, "method": "blockchain.address.get_history", "params": [
↪"1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L"] }
```

response:

```
{"id": 1, "result": [{"tx_hash":
↪"ac9cd2f02ac3423b022e86708b66aa456a7c863b9730f7ce5bc24066031fdced", "height":␣
↪340235}, {"tx_hash":
↪"c4a86b1324f0a1217c80829e9209900bc1862beb23e618f1be4404145baa5ef3", "height":␣
↪340237}]}
{"jsonrpc": "2.0", "id": 1, "result": [{"tx_hash":
↪"16c2976eccd2b6fc937d24a3a9f3477b88a18b2c0cdbe58c40ee774b5291a0fe", "height": 0,
↪"fee": 225}]}
```

blockchain.address.get_mempool

blockchain.address.get_balance

request:

```
{ "id": 1, "method":"blockchain.address.get_balance", "params":[
↪"1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L"] }
```

response:

```
{"id": 1, "result": {"confirmed": 533506535, "unconfirmed": 27060000}}
```

blockchain.address.get_proof

blockchain.address.listunspent

request:

```
{ "id": 1, "method":
"blockchain.address.listunspent", "params":
["1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L"] }<br/>
```

response:

```
{"id": 1, "result": [{"tx_hash":
"561534ec392fa8eebf5779b233232f7f7df5fd5179c3c640d84378ee6274686b",
"tx_pos": 0, "value": 24990000, "height": 340242},
{"tx_hash":"620238ab90af02713f3aef314f68c1d695bbc2e9652b38c31c025d58ec3ba968",
"tx_pos": 1, "value": 19890000, "height": 340242}]}
```

blockchain.utxo.get_address

blockchain.block.get_header

blockchain.block.get_chunk

blockchain.transaction.broadcast

Submits raw transaction (serialized, hex-encoded) to the network. Returns transaction id, or an error if the transaction is invalid for any reason.

request:

```
{ "id": 1, "method":
"blockchain.transaction.broadcast", "params":

↪"0100000002f327e86da3e66bd20e1129b1fb36d07056f0b9a117199e759396526b8f3a20780000000000ffffffff0
↪" }<br/>
```

response:

```
{"id": 1, "result":
↪"561534ec392fa8eebf5779b233232f7f7df5fd5179c3c640d84378ee6274686b"}
```

blockchain.transaction.get_merkle

blockchain.transaction.get_merkle [$txid, $txHeight]

blockchain.transaction.get

Method for obtaining raw transaction (hex-encoded) for given txid. If the transaction doesn't exist, an error is returned.

request:

```
{ "id": 17, "method":"blockchain.transaction.get", "params": [
"0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098"
] }
```

response:

```
{ "id": 17, "result":
↪"01000000010000000000000000000000000000000000000000000000000000000000000000ffffffff0704ffff001d
↪"}
```

error:

```
{ "id": 17, "error": "{ u'message': u'No information available about transaction',␣
↪u'code': -5 }" }
```

blockchain.estimatefee

Estimates the transaction fee per kilobyte that needs to be paid for a transaction to be included within a certain number of blocks. If the node doesn't have enough information to make an estimate, the value -1 will be returned.

Parameter: How many blocks the transaction may wait before being included.

request:

```
{ "id": 17, "method": "blockchain.estimatefee", "params": [ 6 ] }
```

response:

```
{ "id": 17, "result": 0.00026809 }
{ "id": 17, "result": 1.169e-05 }
```

error:

```
{ "id": 17, "result": -1 }
```

3.10.3 External links

- https://docs.google.com/a/palatinus.cz/document/d/17zHy1SUlhgtCMbypO8cHgpWH73V5iUQKk_0rWvMqSNs/edit?hl=en_US" original Slush's specification of Stratum protocol
- http://mining.bitcoin.cz/stratum-mining specification of Stratum mining extension

3.11 Serialization of unsigned or partially signed transactions

Electron Cash uses an extended serialization format for transactions. The purpose of this format is to send unsigned and partially signed transactions to cosigners or to cold storage.

This is achieved by extending the 'pubkey' field of a transaction input.

3.11.1 Extended public keys

The first byte of the pubkey indicates if it is an extended pubkey:

- 0x02, 0x03, 0x04: legal Bitcoin public key (compressed or not).
- 0xFF, 0xFE, 0xFD: extended public key.

Extended public keys are of 3 types:

- 0xFF: bip32 xpub and derivation
- 0xFE: legacy electrum derivation: master public key + derivation
- 0xFD: unknown pubkey, but we know the Bitcoin address.

3.11.2 Public key

This is the legit Bitcoin serialization of public keys.

0x02 or 0x03	compressed public key (32 bytes)
0x04	uncompressed public key (64 bytes)

3.11.3 BIP32 derivation

0xFF	xpub (78 bytes)	bip32 derivation (2*k bytes)

3.11.4 Legacy Electrum Derivation

0xFE	mpk (64 bytes)	derivation (4 bytes)

3.11.5 Bitcoin address

Used if we don't know the public key, but we know the address (or the hash 160 of the output script). The cosigner should know the public key.

0xFD	hash_160_of_script (20 bytes)

CHAPTER 4

Indices and tables

- genindex
- search