

---

# **Electrum Documentation**

***Release 2.5***

**Thomas Voegtlin**

November 12, 2016



<b>1</b>	<b>GUI and beginners</b>	<b>3</b>
1.1	Frequently Asked Questions . . . . .	3
1.2	Invoices . . . . .	8
1.3	Two Factor Authentication . . . . .	8
1.4	Multisig Wallets . . . . .	11
1.5	Cold Storage . . . . .	15
<b>2</b>	<b>Command line and merchants</b>	<b>23</b>
2.1	Command Line . . . . .	23
2.2	Using cold storage with the command line . . . . .	25
2.3	How to accept Bitcoin on a website using Electrum . . . . .	26
<b>3</b>	<b>For developers</b>	<b>33</b>
3.1	The Python Console . . . . .	33
3.2	Simple Payment Verification . . . . .	34
3.3	Electrum Seed Version System . . . . .	34
3.4	Electrum protocol specification . . . . .	36
3.5	Serialization of unsigned or partially signed transactions . . . . .	41
3.6	Build Electrum on Mac OS X . . . . .	42
<b>4</b>	<b>External Links</b>	<b>43</b>



Electrum is a lightweight Bitcoin wallet.



---

## GUI and beginners

---

### 1.1 Frequently Asked Questions

#### 1.1.1 How does Electrum work?

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

#### 1.1.2 Does Electrum trust servers?

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

#### 1.1.3 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.

#### 1.1.4 How secure is the seed?

The seed created by Electrum has 128 bits of entropy. This means that it provides the same level of security as a Bitcoin private key (of length 256 bits). Indeed, an elliptic curve key of length  $n$  provides  $n/2$  bits of security.

#### 1.1.5 How can I send the maximum available in my wallet?

Type an exclamation mark (!) in the Amount field. The fee will be automatically adjusted for that amount.

### 1.1.6 How can I send Bitcoins without paying a transaction fee?

You can create a transaction with zero fee in the GUI, by following these steps:

- Activate the option ‘set fees manually’
- Enter 0 in the fee field
- Enter the amount in the amount field

Note that transactions without fees might not be relayed by the Electrum server, or by the Bitcoin Network.

### 1.1.7 What does it mean to “Freeze” an address in Electrum?

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

### 1.1.8 What encryption is used for wallets?

Electrum uses AES-256-CBC to encrypt the seed and private keys in the wallet.

### 1.1.9 I have forgotten my password but still have my seed. Is there any way I can recover my password?

No, you cannot recover your password. However, you can still recover your money: restore your wallet from its seed, and choose a new password.

### 1.1.10 Why can I open the wallet without entering my password?

Only the seed and private keys are encrypted, and not the entire wallet file. 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.

### 1.1.11 Does Electrum support cold wallets?

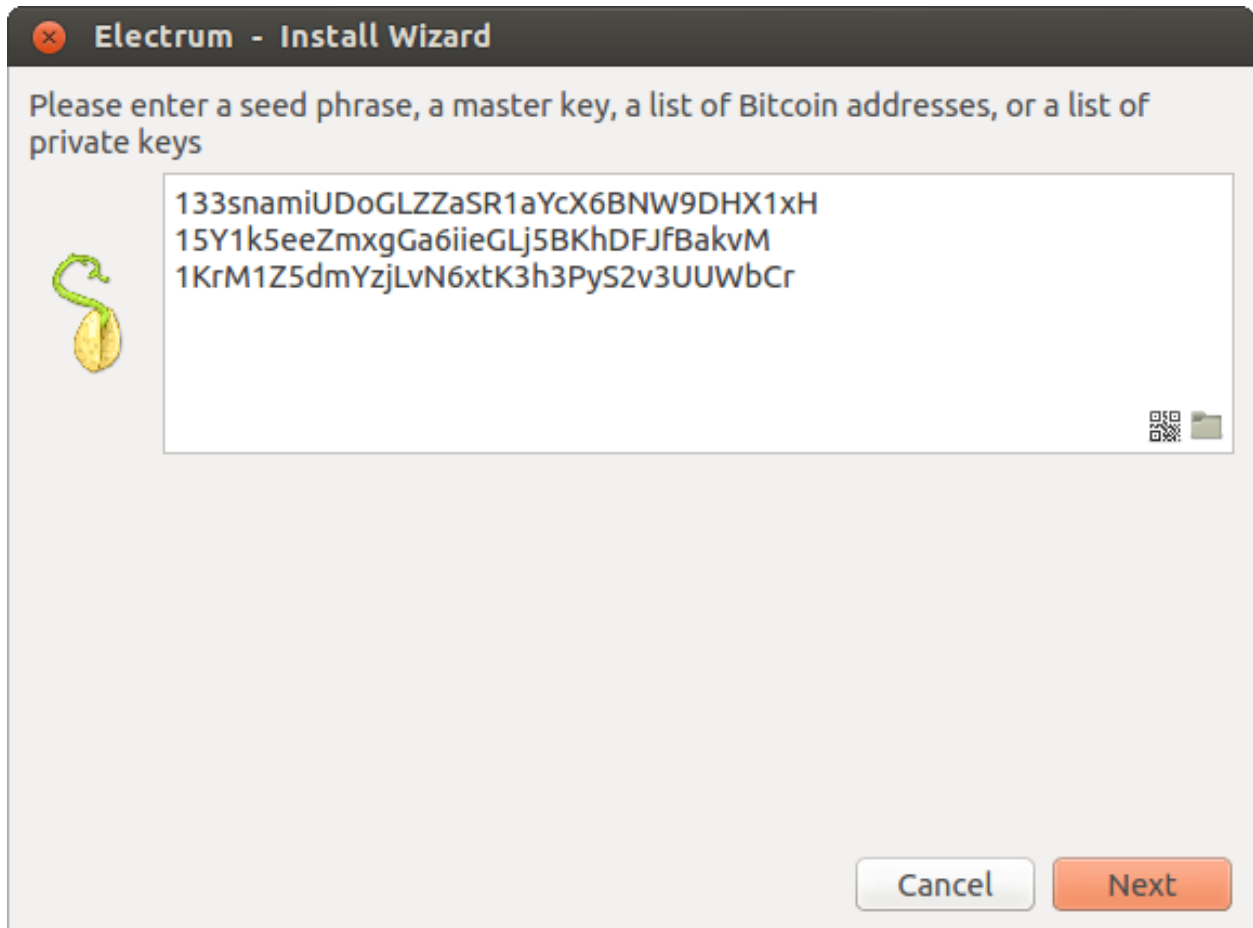
Yes. see *Cold Storage*

### 1.1.12 Can I import private keys from other Bitcoin clients?

In Electrum 2.0, 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.





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

### 1.1.13 Can I sweep private keys from other Bitcoin clients?

Sweeping private keys means to send all the bitcoins 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 bitcoins 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 electrum wallet.

### 1.1.14 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 UsersYourUserNameAppDataRoamingElectrumwallets

On Mac:

- Open Finder

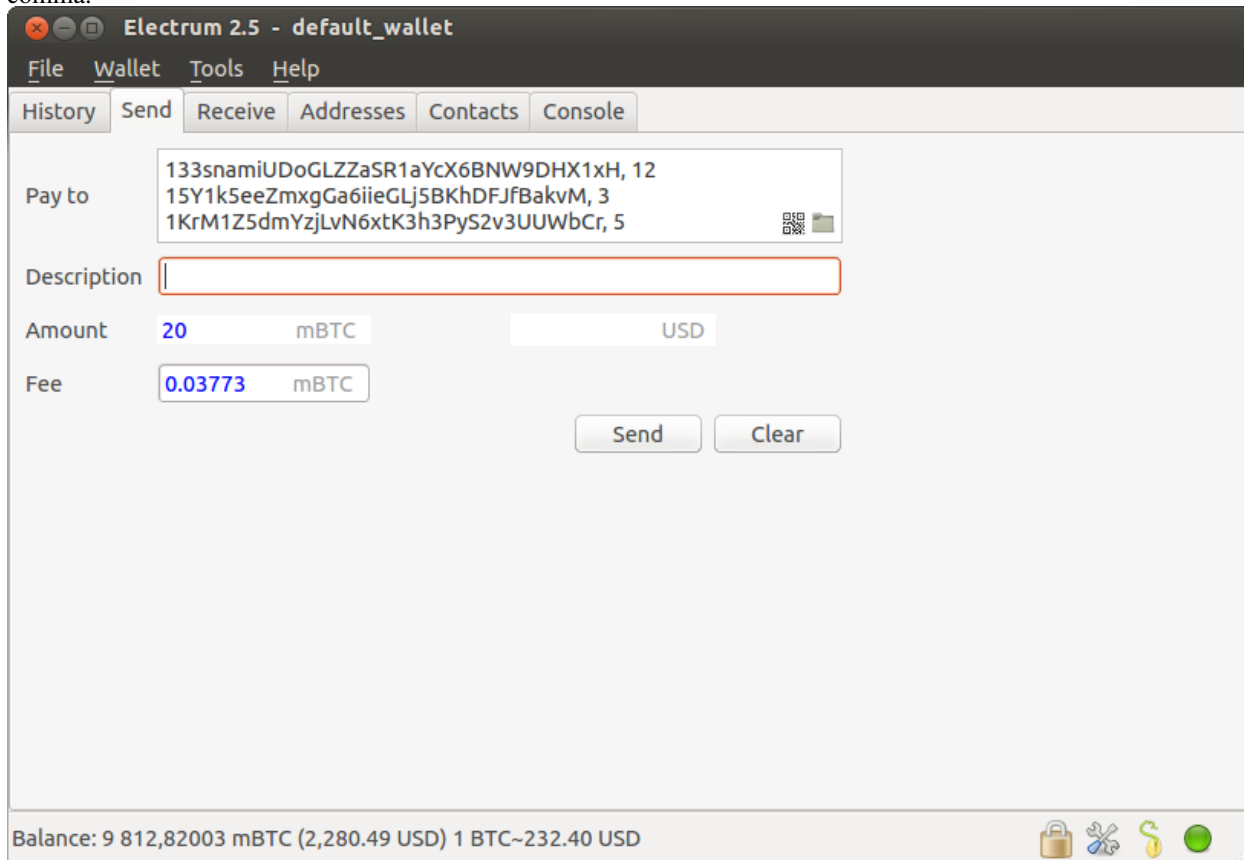
- Go to folder (shift+cmd+G) and type ~/.electrum

On Linux:

- Home Folder
- Go -> Location and type ~/.electrum

### 1.1.15 Can I do bulk payments with Electrum?

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



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.

### 1.1.16 Can Electrum create and sign raw transactions?

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

### 1.1.17 Electrum freezes when I try to send bitcoins

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 Bitcoin faucet). When you send Bitcoins, Electrum 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.

### 1.1.18 Is there a way to get a BTC/USD exchange ticker in Electrum?

Yes, go to Tools->Plugins and enable 'exchange rates'

### 1.1.19 What is the gap limit?

The gap limit is the maximum number of consecutive unused addresses in your deterministic sequence of addresses. Electrum uses it in order to stop looking for addresses. In Electrum 2.0, it is set to 20 by default, so the client will get all addresses until 20 unused addresses are found.

### 1.1.20 How can I pre-generate new addresses?

Electrum 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()` 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()
```

### 1.1.21 How to upgrade Electrum?

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

To upgrade Electrum, 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 Electrum upgrades will modify the format of your wallet files.

For this reason, it is not recommended to downgrade Electrum 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.

The following issues should be considered when upgrading Electrum 1.x wallets to Electrum 2.x:

- Electrum 2.x will need to regenerate all of your addresses during the upgrade process. Please allow it time to complete, and expect it to take a little longer than usual for Electrum to be ready.
- The contents of your wallet file will be replaced with an Electrum 2 wallet. This means Electrum 1.x will no longer be able to use your wallet once the upgrade is complete.
- The 'Addresses' tab will not show any addresses the first time you launch Electrum 2. This is expected behaviour. Restart Electrum 2 after the upgrade is complete and your addresses will be available.

- Offline copies of Electrum will not show the addresses at all because it cannot synchronize with the network. You can force an offline generation of a few addresses by typing the following into the Console: `wallet.synchronize()`. When it's complete, restart Electrum and your addresses will once again be available.

## 1.2 Invoices

Invoices are Payment Requests signed by their requestor.

When you click on a bitcoin: link, a URL is passed to electrum:

```
electrum "bitcoin:1KLxqw4MA5NkG6YP1N4S14akDFCP1vQrKu?amount=1.0&r=https%3A%2F%2Fbitpay.com%2Fi%2E
```

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 invoice 8pCMP9rAx43T7ezvepofoo for merchant BitGive Foundation

Amount 1000 mBTC 232.34 USD

Fee 0.02985 mBTC

Send Clear

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 Electrum verified the chain of signatures.

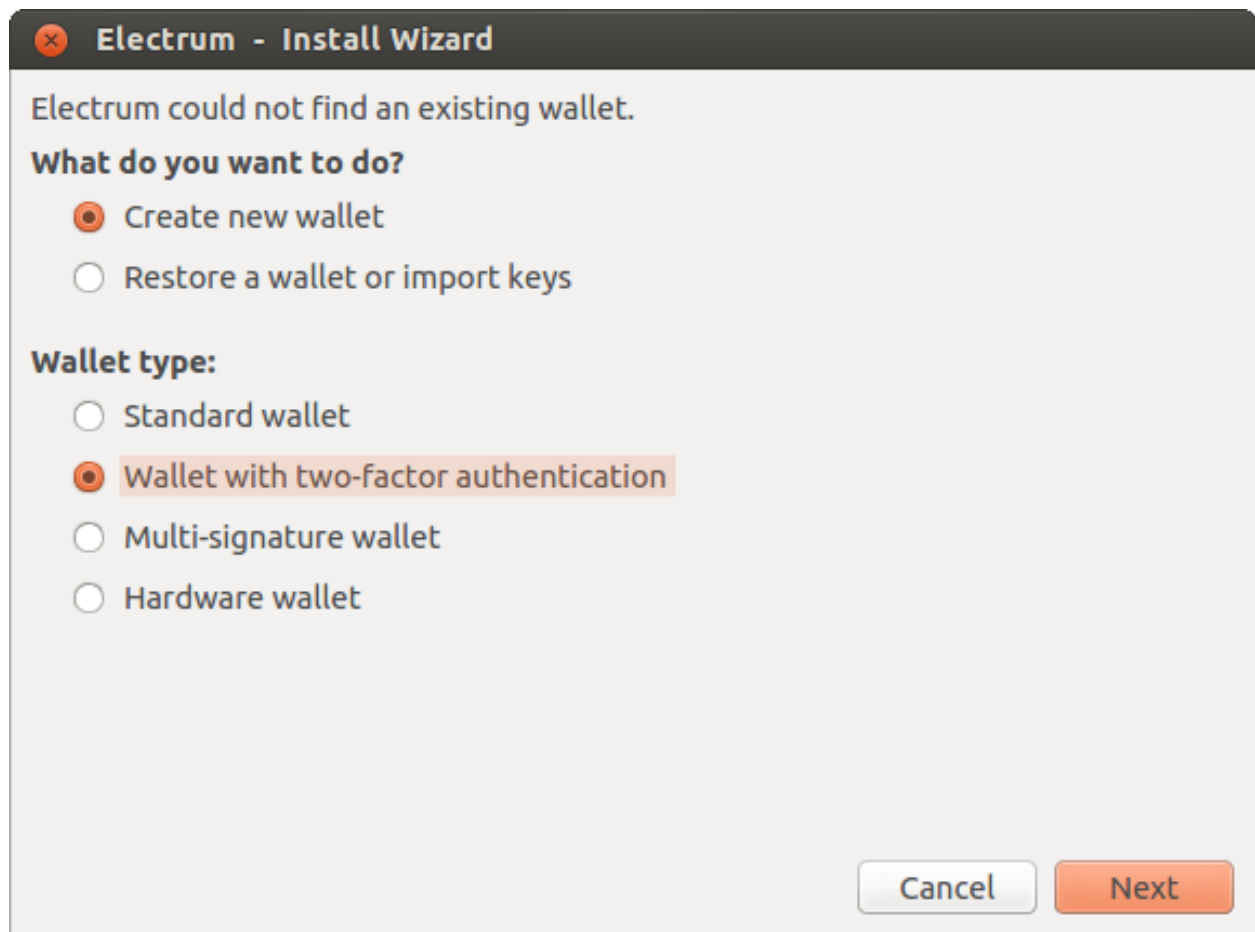
Note that the “send” tab contains a list of invoices and their status.

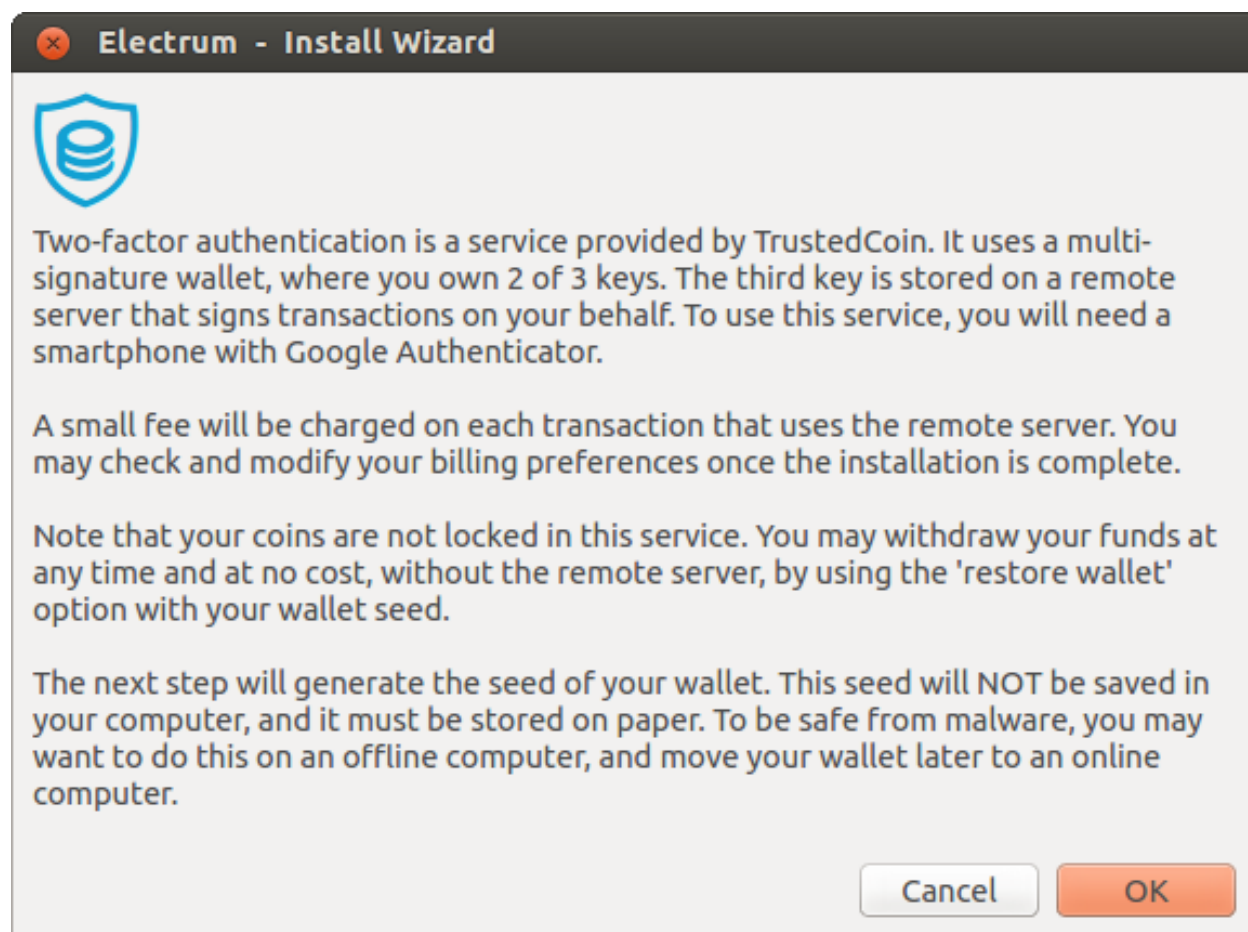
### 1.3 Two Factor Authentication

Electrum offers two-factor authenticated wallets, with a remote server acting to co-sign transactions, adding another level of security in the event of your computer being compromised.

The remote server in question is a service offered by TrustedCoin. Here is a [guide](#) on how it works.

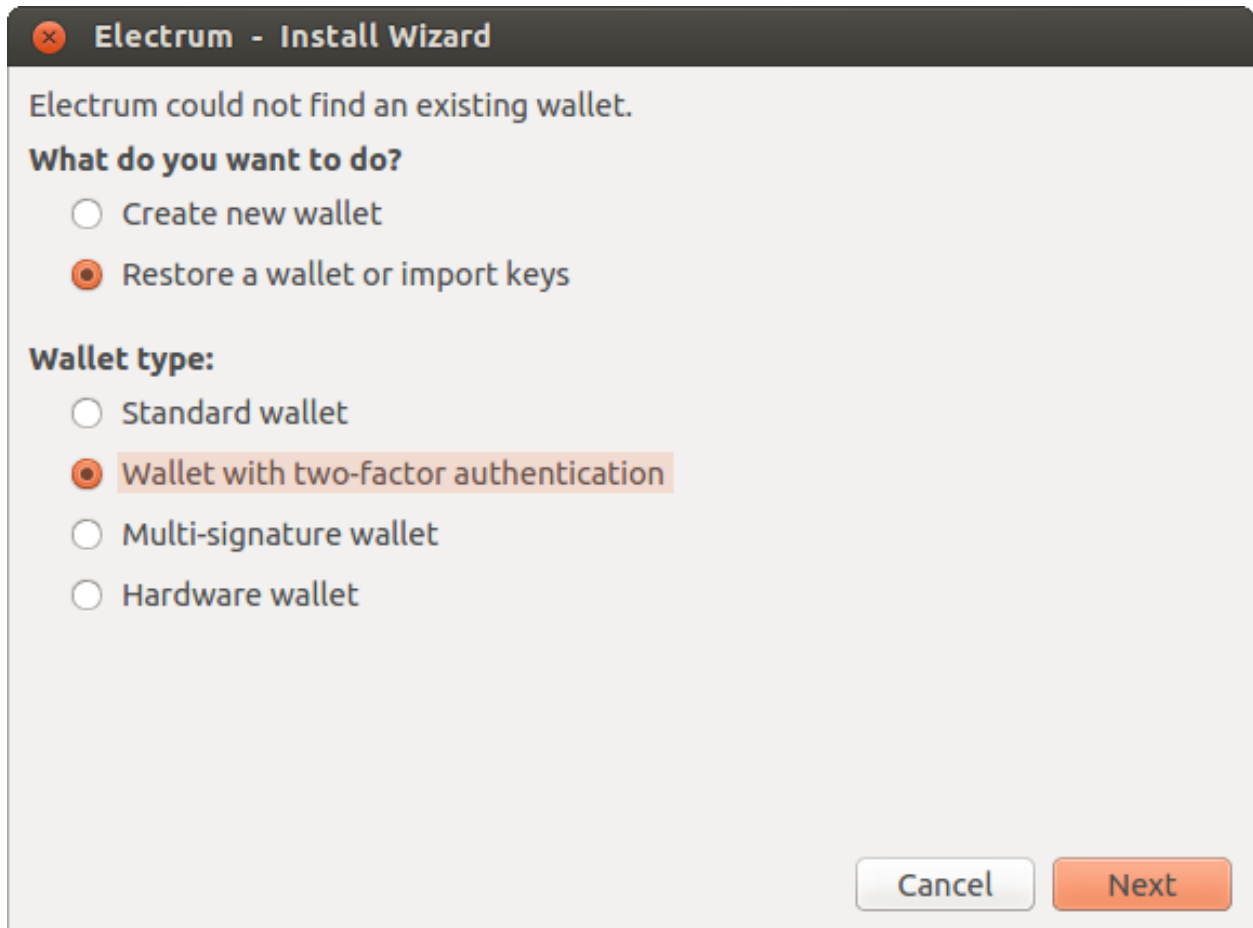
### 1.3.1 Creating a Wallet





### 1.3.2 Restoring from seed

Even if TrustedCoin is compromised or taken offline, your coins are secure as long as you still have the seed of your wallet. Your seed contains two master private keys in a 2-of-3 security scheme. In addition, the third master public key can be derived from your seed, ensuring that your wallet addresses can be restored. In order to restore your wallet from seed, select “wallet with two factor authentication”, as this tells Electrum to use this special variety of seed for restoring your wallet.



Note: The “restore” option should be used only if you no longer want to use TrustedCoin, or if there is a problem with the service. Once you have restored your wallet in this way, two of three factors are on your machine, negating the special protection of this type of wallet.

## 1.4 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.

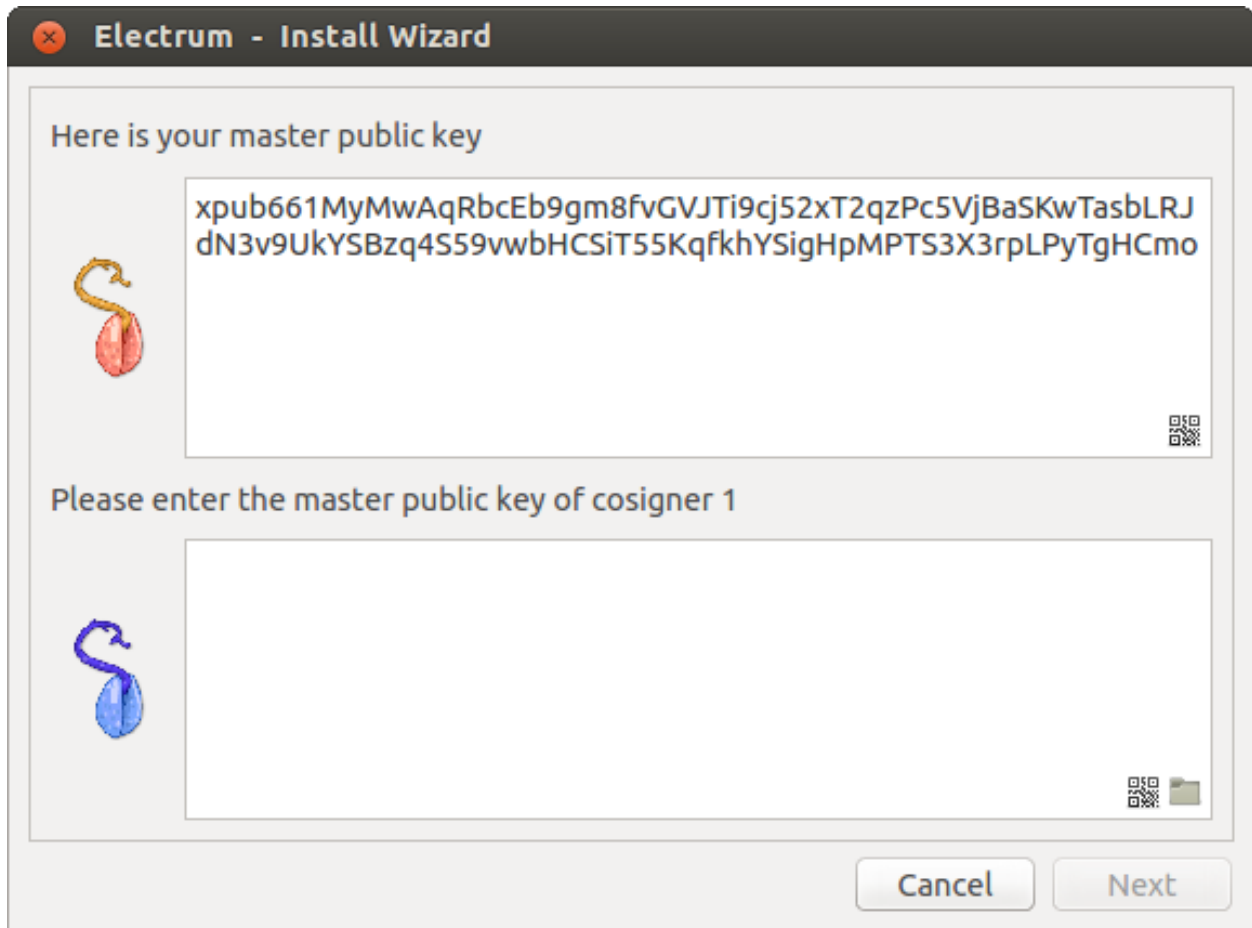
### 1.4.1 Create a pair of 2-of-2 wallets

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.



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





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.

### 1.4.2 Receiving

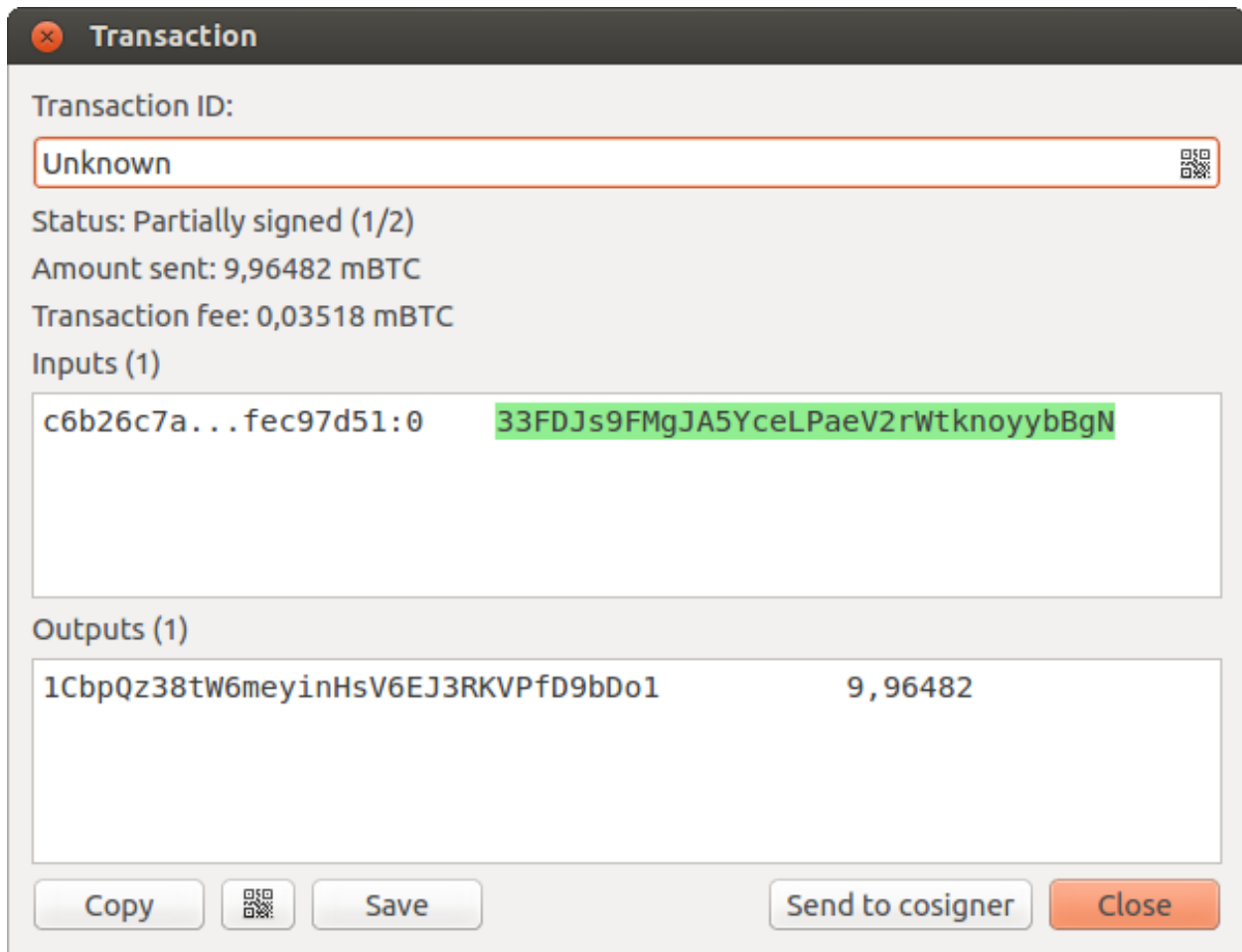
Check that both wallets generate the same set of Addresses. You can now send to these Addresses (note they start with a “3”) with any wallet that can send to P2SH Addresses.

### 1.4.3 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.



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)

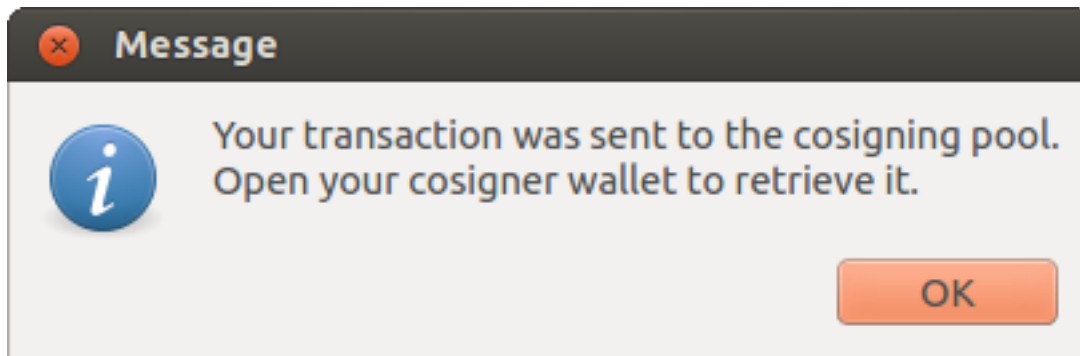
### Use QR-Code

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)

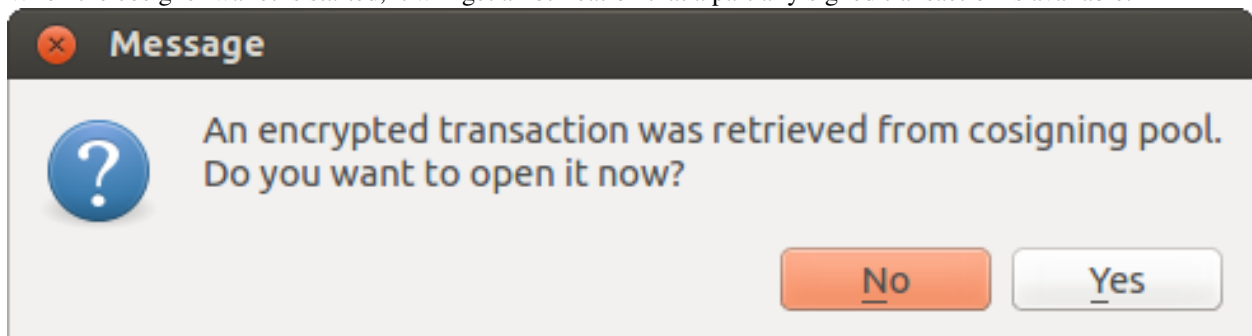
### Use the Cosigner Pool Plugin

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.



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



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 second signature to the transaction (using the “sign” button). It will then be broadcast to the network.

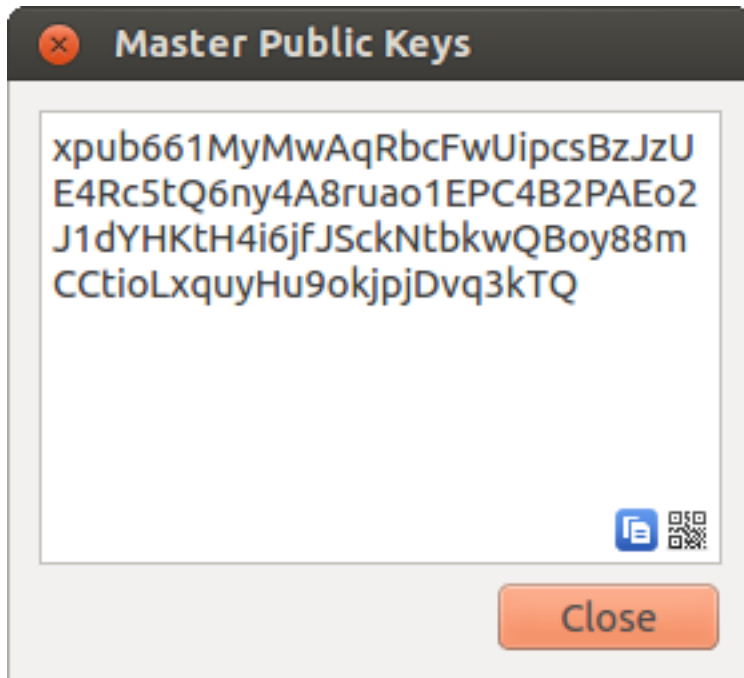
## 1.5 Cold Storage

This document shows how to create an offline wallet that holds your Bitcoins 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.

### 1.5.1 Create an offline wallet

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

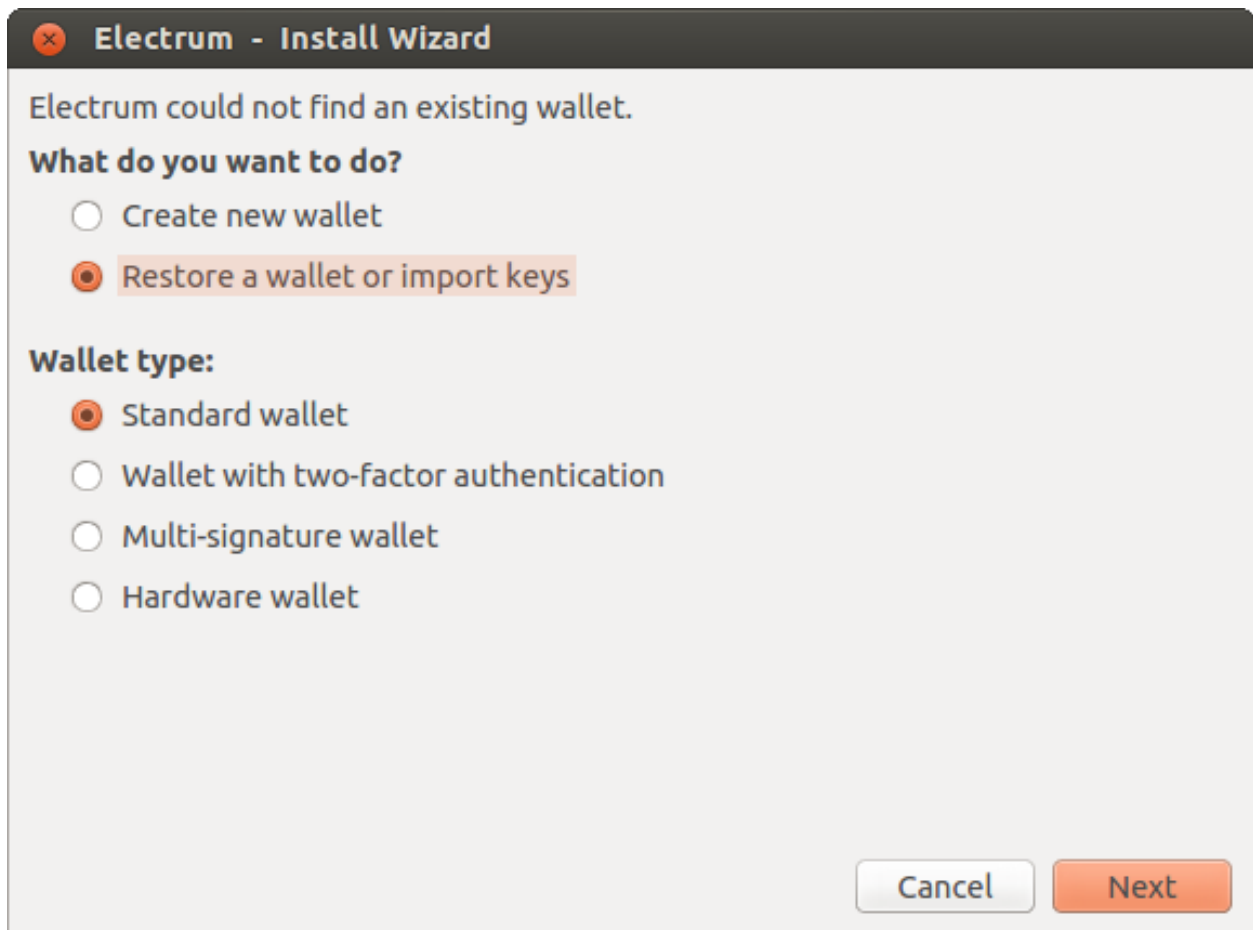
After creating the wallet, go to Wallet -> Master Public Keys.



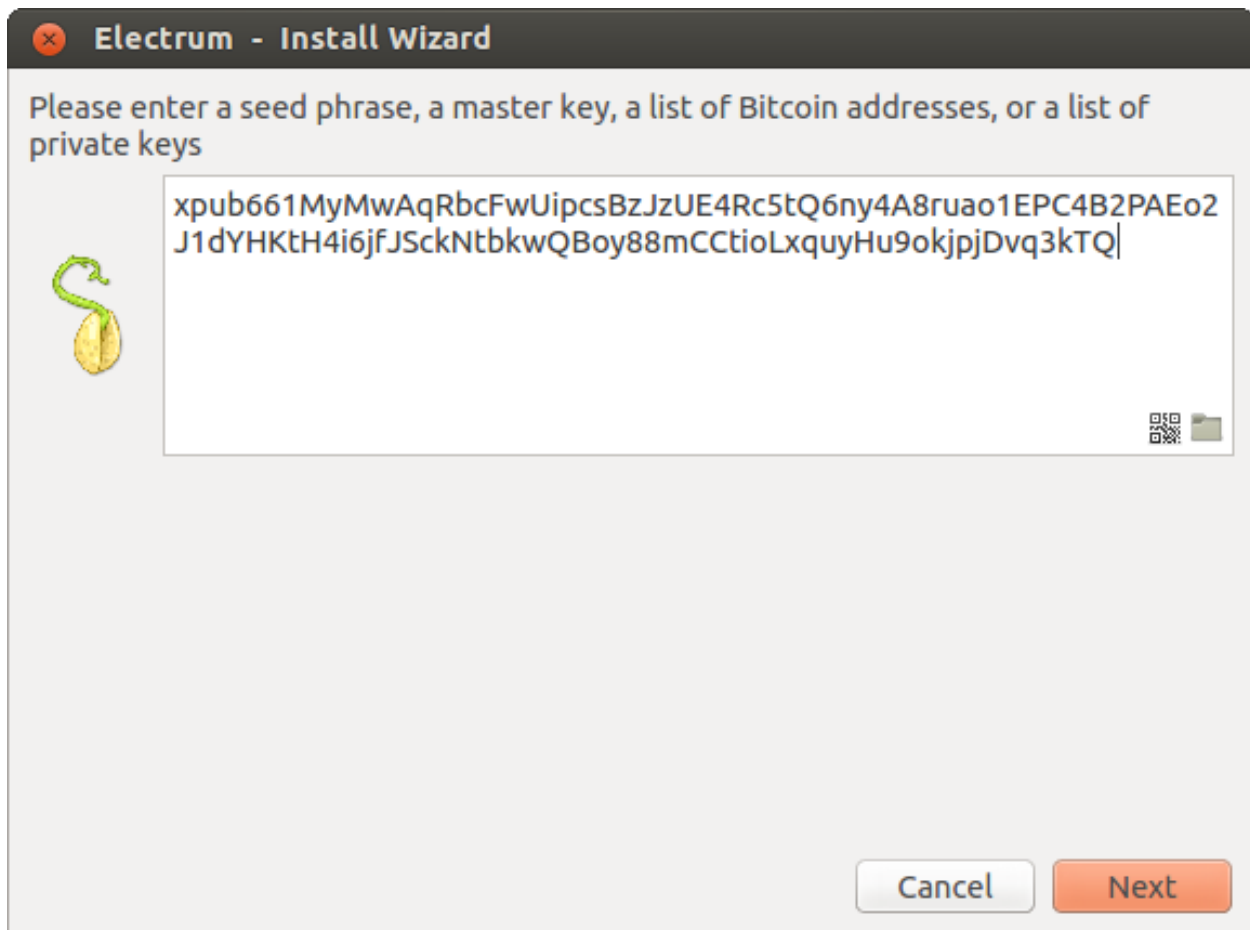
The Master Public Key of your wallet is the string shown in this popup window. Transfer that key to your online machine somehow.

### 1.5.2 Create a watching-only version of your wallet

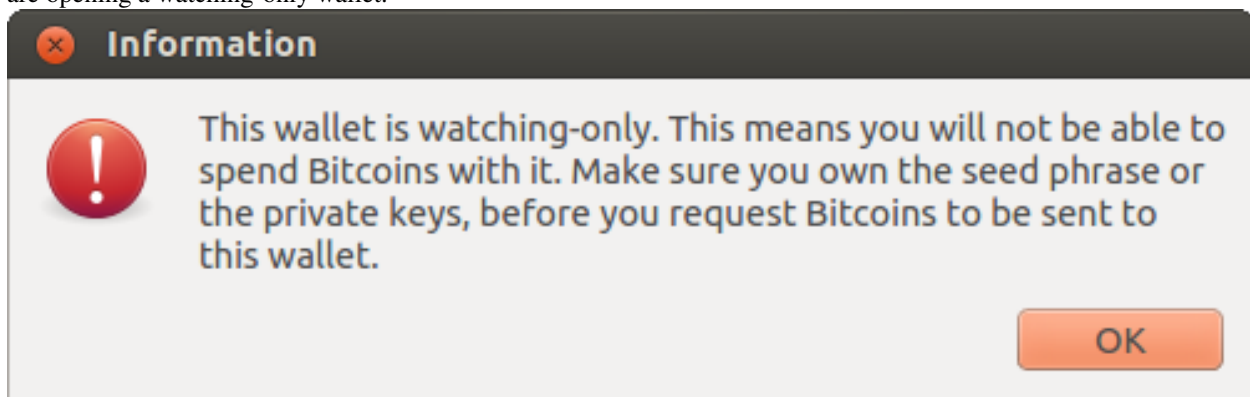
On your online machine, open up Electrum and select File -> New/Restore. Enter a name for the wallet and select “Restore a wallet or import keys”.



Paste your master public key in the box.



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.



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

### 1.5.3 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

Copy

Save

Close

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).

### 1.5.4 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...b2f77648:012nUZdGyrdQ3PSxopAebMcDqidTZGTZVZH

Outputs (2)  

3NV6KhDiqWcAk6c3e4PqCv5RAE7vqekTGC	12,
19GKSZw2sx5WXhSyoFeiAgZAisKJpaG736	16,52267

Copy

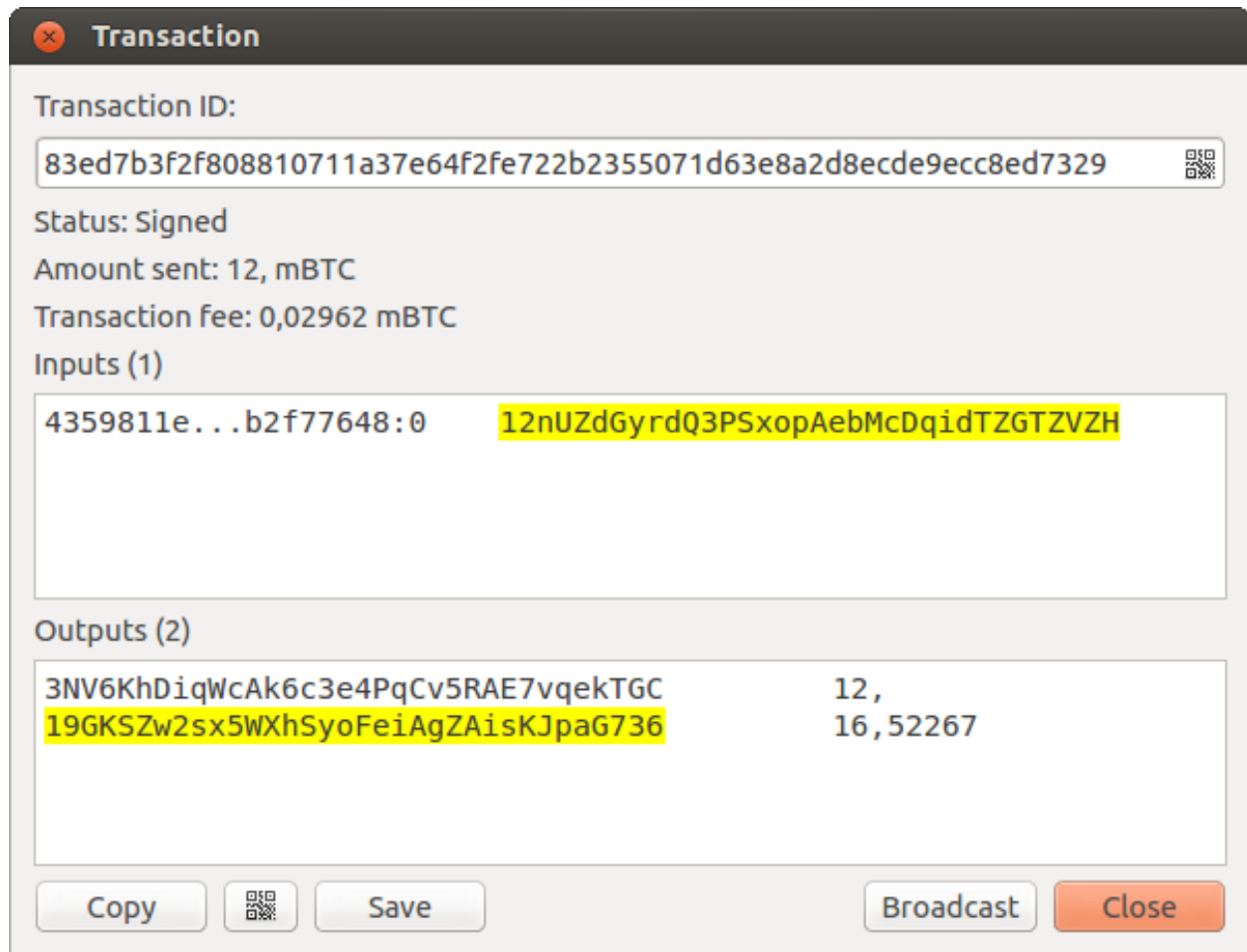
Save

Sign

Close

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





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

### 1.5.5 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 network.



---

## Command line and merchants

---

### 2.1 Command Line

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

#### 2.1.1 Using the inline help

To see the list of Electrum commands, type:

```
electrum help
```

To see the documentation for a command, type:

```
electrum help <command>
```

#### 2.1.2 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:

```
electrum 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:

```
electrum signmessage 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE ?
```

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

```
electrum 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 | electrum signmessage 1JuiT4dM65d8vBt8qUYamnDmAMJ4MjjxRE -
```

### 2.1.3 Aliases

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

```
electrum payto ecdsa.net !
```

### 2.1.4 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.

### 2.1.5 Examples

#### Sign and verify message

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

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

And:

```
cat LICENCE | electrum 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:

```
electrum listunspent | jq 'map(.value)'
```

#### Select only incoming transactions from history

Incoming transactions have a positive 'value' field

```
electrum 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")
electrum 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")
electrum history | jq --arg before $before --arg after $after '[] | select(.timestamp>$after|ton
```

## Encrypt and decrypt messages

First we need the public key of a wallet address:

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

Encrypt:

```
cat | electrum encrypt $pk -
```

Decrypt:

```
electrum 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:

```
electrum listaddresses --funded | electrum 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:

```
electrum listaddresses --funded | electrum getprivatekeys - | jq 'map(.[0])'
```

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

```
electrum listaddresses --funded | electrum getprivatekeys - | jq 'map(.[0])' | electrum sweep - 1uCM
```

## 2.2 Using cold storage with the command line

This page will show you how to sign a transaction with an offline Electrum wallet, using the Command line.

### 2.2.1 Create an unsigned transaction

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

```
electrum 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 | electrum deserialize -
```

### 2.2.2 Sign the transaction

The serialization format of Electrum 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 | electrum signtransaction - > signed.txn
```

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

### 2.2.3 Broadcast the transaction

Send your transaction to the Bitcoin network, using broadcast:

```
cat signed.txn | electrum broadcast -
```

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

## 2.3 How to accept Bitcoin on a website using Electrum

This tutorial will show you how to accept Bitcoin on a website with SSL signed payment requests. It is updated for Electrum 2.6.

### 2.3.1 Requirements

- A webserver serving static HTML
- A SSL certificate (signed by a CA)
- Electrum version  $\geq 2.6$

### 2.3.2 Create a wallet

Create a wallet on your web server:

```
electrum 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 Electrum as a daemon:

```
electrum daemon start
```

### 2.3.3 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:

```
electrum 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:

```
electrum setconfig ssl_chain /path/to/ssl.chain
```

### 2.3.4 Configure a requests directory

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

```
electrum setconfig requests_dir /var/www/r/
```

By default, electrum will display local URLs, starting with 'file:///'. In order to display public URLs, we need to set another configuration variable, url\_rewrite. For example:

```
electrum setconfig url_rewrite ["'file:///var/www/'",'https://electrum.org/']"
```

### 2.3.5 Create a signed payment request

```
electrum addrequest 3.14 -m "this is a test"
{
  "URI": "bitcoin:1MP49h5fbfLXiFpomsXeqJHGHUfNf3mCo4?amount=3.14&r=https://electrum.org/r/7c2888541a",
  "address": "1MP49h5fbfLXiFpomsXeqJHGHUfNf3mCo4",
  "amount": 314000000,
  "amount (BTC)": "3.14",
  "exp": 3600,
  "id": "7c2888541a",
  "index_url": "https://electrum.org/r/index.html?id=7c2888541a",
  "memo": "this is a test",
  "request_url": "https://electrum.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.

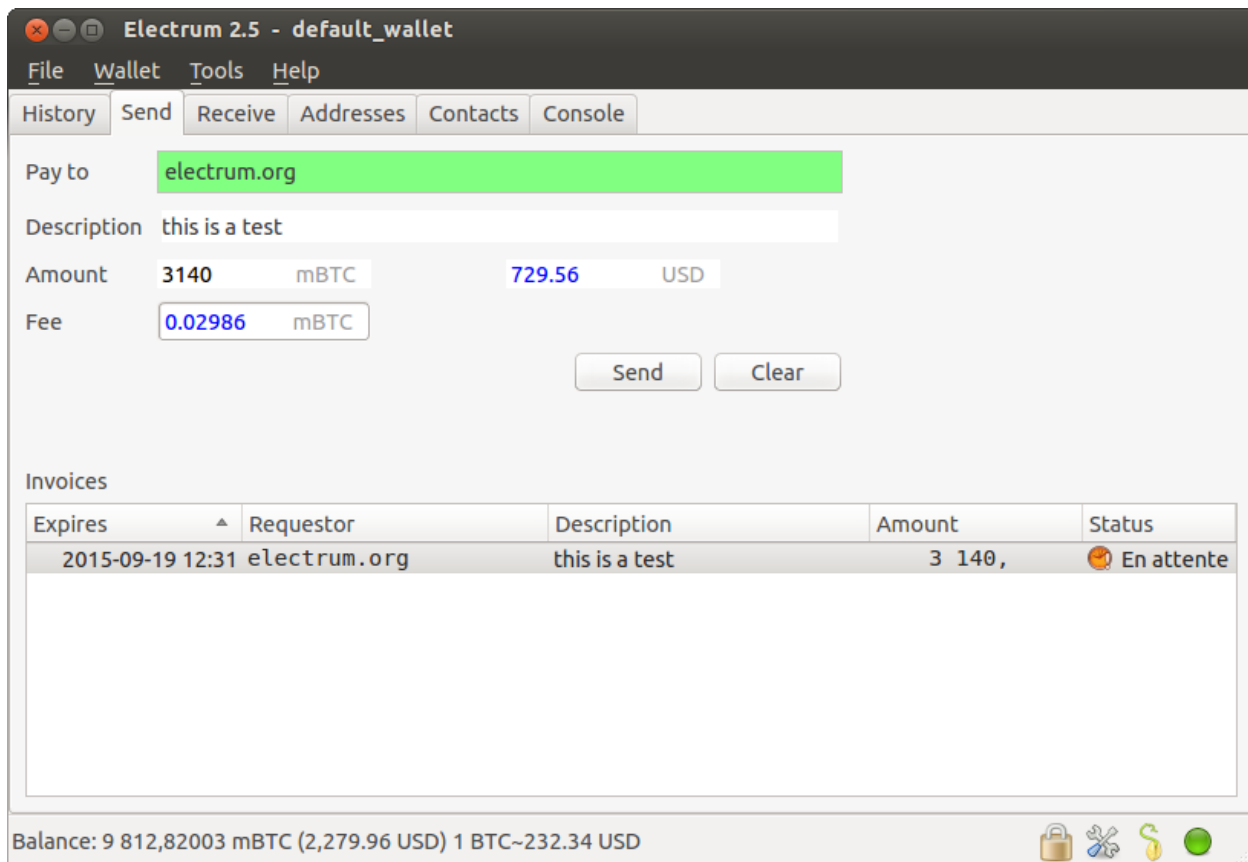
### 2.3.6 Open the payment request page in your browser

Let us open `index_url` in a web browser.





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



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

### 2.3.7 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:

```
electrum setconfig websocket_server <FQDN of your server>
electrum setconfig websocket_port 9999
```

And restart the daemon:

```
electrum daemon stop
electrum 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.

### 2.3.8 JSONRPC interface

Commands to the Electrum daemon can be sent using JSONRPC. This is useful if you want to use electrum 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):

```
electrum 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
```



---

## For developers

---

### 3.1 The Python Console

Most Electrum 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_out
```

This makes it possible to combine Electrum 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 addresses 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']))
```

## 3.2 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](#)

## 3.3 Electrum Seed Version System

This document describes the Seed Version System used in Electrum (version 2.0 and higher).

### 3.3.1 Motivation

Electrum was the first Bitcoin wallet to derive private keys from a seed phrase made of English words. Early versions of Electrum (before 2.0) used a bidirectional encoding between seed phrase and entropy. This type of encoding requires a fixed wordlist. This means that future versions of Electrum must ship with the exact same wordlist, in order to be able to read old seed phrases.

BIP39 was introduced two years after Electrum. BIP39 seeds include a checksum, in order to help users figure out typing errors. However, BIP39 suffers the same shortcomings as early Electrum seed phrases:

- A fixed wordlist is still required. Following our recommendation, BIP39 authors decided to derive keys and addresses in a way that does not depend on the wordlist. However, BIP39 still requires the wordlist in order to compute its checksum, which is plainly inconsistent, and defeats the purpose of our recommendation. This problem is exacerbated by the fact that BIP39 proposes to create one wordlist per language. This threatens the portability of BIP39 seed phrases.
- BIP39 seed phrases do not include a version number. This means that software should always know how to generate keys and addresses. BIP43 suggests that wallet software will try various existing derivation schemes within the BIP32 framework. This is extremely inefficient and rests on the assumption that future wallets will support all previously accepted derivation methods. If, in the future, a wallet developer decides not to implement a particular derivation method because it is deprecated, then the software will not be able to detect that the corresponding seed phrases are not supported, and it will return an empty wallet instead. This threatens users funds.

For these reasons, Electrum does not generate BIP39 seeds. Starting with version 2.0, Electrum uses the following Seed Version System, which addresses these issues.

### 3.3.2 Description

Electrum 2.0 derives keys and addresses from a hash of the UTF8 normalized seed phrase with no dependency on a fixed wordlist. This means that the wordlist can differ between wallets while the seed remains portable, and that future wallet implementations will not need today's wordlists in order to be able to decode the seeds created today. This reduces the cost of forward compatibility.

In addition, Electrum 2.0 seed phrases include a version number. The purpose of the version number is to indicate how addresses and keys are derived from the seed. Similar to keys derivation, the version number is also obtained by a hash of the UTF8 normalized seed phrase.

The version number is also used to check seed integrity; in order to be correct, a seed phrase must produce a registered version number.

### 3.3.3 Version number

The version number is a prefix of a hash derived from the seed phrase. The length of the prefix is a multiple of 4 bits. The prefix is computed as follows:

```
def version_number(seed_phrase):
    # normalize seed
    normalized = prepare_seed(seed_phrase)
    # compute hash
    h = hmac_sha_512("Seed version", normalized)
    # use hex encoding, because prefix length is a multiple of 4 bits
    s = h.encode('hex')
    # the length of the prefix is written on the first 4 bits
    # for example, the prefix '101' is of length 4*3 bits = 4*(1+2)
    length = int(s[0]) + 2
    # read the prefix
    prefix = s[0:length]
    # return version number
    return hex(int(prefix, 16))
```

The normalization function (`prepare_seed`) removes all but one space between words. It also removes diacritics, and it removes spaces between Asian CJK characters.

### 3.3.4 List of reserved numbers

The following version numbers are used in Electrum.

Number	Type	Description
0x01	Standard	P2PKH and Multisig P2SH wallets
0x02	Segwit	Reserved for Segwit
0x101	2FA	Two-factor authenticated wallets

### 3.3.5 Seed generation

When the seed phrase is hashed during seed generation, the resulting hash must begin with the correct version number prefix. This is achieved by enumerating a nonce and re-hashing the seed phrase until the desired version number is created. This requirement does not decrease the security of the seed (up to the cost of key stretching, that might be required to generate the private keys).

### 3.3.6 Wordlist

Electrum currently use the same wordlist as BIP39 (2048 words). A typical seed has 12 words and 132 bits of entropy.

## 3.4 Electrum protocol specification

Stratum is a universal bitcoin communication protocol used mainly by bitcoin client Electrum and miners.

### 3.4.1 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 } ]
```



### 3.4.2 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).

*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":
"0000000000000000002b3ef284c2c754ab6e6abc40a0e31a974f966d8a2b4d5206",
"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":
"0000000000000000001ba892b1717690900ae476857120a78fb50825f8b67a42d4",
"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"] }
```

### blockchain.address.get\_history

*request:*

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

*response:*

```
{ "id": 1, "result": [{ "tx_hash": "ac9cd2f02ac3423b022e86708b66aa456a7c863b9730f7ce5bc24066031fdced", "height": 340235 }, { "tx_hash": "c4a86b1324f0a1217c80829e9209900bc1862beb23e618f1be4404145baa5ef3", "height": 340237 } ] }
```

### 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":  
"0100000002f327e86da3e66bd20e1129b1fb36d07056f0b9a117199e759396526b8f3a20780000000000ffffffffff0ede03"
```

*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:*

[illegible]

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

If the node doesn't have enough information to make an estimate, the value -1 will be returned

*request:*

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

*response:*

```
{ "id": 17, "result": 0.00026809 }
```

### 3.4.3 External links

- [https://docs.google.com/a/palatinus.cz/document/d/17zHy1SUIhgtCMbypO8cHgpWH73V5iUQKk\\_0rWvMqSNs/edit?hl=en\\_US](https://docs.google.com/a/palatinus.cz/document/d/17zHy1SUIhgtCMbypO8cHgpWH73V5iUQKk_0rWvMqSNs/edit?hl=en_US) original Slush's specification of Stratum protocol
- <http://mining.bitcoin.cz/stratum-mining> specification of Stratum mining extension

## 3.5 Serialization of unsigned or partially signed transactions

Electrum 2.0 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.

### 3.5.1 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.

### 3.5.2 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)

### 3.5.3 BIP32 derivation

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

### 3.5.4 Legacy Electrum Derivation

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

### 3.5.5 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)
------	-------------------------------

## 3.6 Build Electrum on Mac OS X

---

## External Links

---

- [GOLANG](#) Stratum client library
- Tutorial: How to create a [plugin](#)
- Tutorial for setting up a [server](#)