Bitzec Documentation

Release 1.1.1

Nov 20, 2018

Using bitzecd

CHAPTER 1

What is bitzec?

bitzecis an implementation of the “Zerocash” protocol. Based on Bitcoin’s code, it intends to offer a far higher standard of privacy through a sophisticated zero-knowledge proving scheme that preserves confidentiality of transaction metadata. For more technical details, please check out ourProtocol Specification.

Before you get started with bitzec, please review the important items below:

Contributor Code of Conduct This project adheres to our Code of Conduct. By participating, you are expected to uphold this code.

Development Guidelines A set of guidelines and best practices to contribute to the development of bitzec.

Security Information bitzec is experimental and a work-in-progress. Use at your own risk.

Deprecation Policy A release is considered deprecated 16 weeks after the release day. There is an automatic deprecation shutdown feature which will halt the node sometime after this 16 week time period. The automatic feature is based on block height and can be explicitly disabled.

CHAPTER 2

Getting Started

For information on bitzec setup, upgrade, installation, build, configuration, and usage please see the *User Guide*.

CHAPTER 3

Need Help?

Answers to common questions from our users can be found in theFAQ.

Collaborate

bitzec development is an open collaborative process. If you'd like to contribute, join ourchat systemand check out some of these channels:

Chat Community Chat

bitzec General user chat

bitzec-Dev Software and Protocol Development

Community-Collaboration Other open source development related to bitzec

The-bitzec-Foundation A room to define and develop thebitzec FoundationAn organization to steward the community, protocol, and science around bitzec.

bitzec-Apprentices A study and peer-education room

bitzec-Wizards Mad scientist brainstorming

CHAPTER 4

License

For license information please see *License*

4.1 User Guide

4.1.1 About

The bitzec repository is a fork ofBitcoin Corewhich contains protocol changes to support theZerocash protocol. This implements the bitzec cryptocurrency, which maintains a separate ledger from the Bitcoin network, for several reasons, the most immediate of which is that the consensus protocol is different.

4.1.2 Getting Started

Welcome! This guide is intended to get you running on the official bitzec network. To ensure your bitzec client behaves gracefully throughout the setup process, please check your system meets the following requirements:

`64-bit` Linux OS

`64-bit` Processor

`5GB` of free RAM

`10GB` of free Disk (*the size of the block chain increases over time*)

Note: Currently we only officially support Linux (Debian), but we are actively investigating development for other operating systems and platforms(e.g. macOS, Ubuntu, Windows, Fedora).

Please let us know if you run into snags. We plan to make it less memory/CPU intensive and support more architectures and operating systems in the future.

If you are installing bitzec for the first time, please skip to the *Installation* section. Otherwise, the below upgrading section will provide information to update your current bitzec environment.

4.1.3 Upgrading?

If you're on a Debian-based distribution, you can follow the *Debian Binary Packages Setup* to install bitzec on your system. Otherwise, you can update your local snapshot of our code:

```
git fetch origin
```

Ensure you check the current release version fromhere.
If `v1.0.1` was current, issue the following commands:

```
git checkout v1.0.1
./zcutil/fetch-params.sh
./zcutil/build.sh -j$(nproc)
```

Note: If you don't have `nproc`, then substitute the number of cores on your system. If the build runs out of memory, try again without the `-j` argument, i.e. just `./zcutil/build.sh`. If you are upgrading from testnet, make sure that your `~/.bitzec` directory contains only `bitzec.conf` to start with, and that your `~/.bitzec/bitzec.conf` does not contain `testnet=1` or `addnode=testnet.z.cash`. If the build fails, move aside your `bitzec` directory and try again by following the instructions in the *Installation* section below.

Important: Running `make clean` before building the update can eliminate random known link errors. If you ran into any other issues upgrading to Overwinter or Sapling, please see the *Network Upgrade Developer Guide*

4.1.4 Setup

There are a couple options to setup bitzec for the first time.

1. If you would like to install binary packages for Debian-based operating systems, see *Debian Binary Packages Setup*
2. If you would like to compile bitzec from source, please continue to the *Installation* section.
3. If you would like to install via a binary tarball download, see *Binary Tarball Download & Setup*.

4.1.5 Installation

Before we begin installing bitzec, we need to get some dependencies for your system.

UBUNTU/DEBIAN

```
sudo apt-get install \
build-essential pkg-config libc6-dev m4 g++-multilib \
autoconf libtool ncurses-dev unzip git python python-zmq \
zlib1g-dev wget curl bsdmainutils automake
```

Note: If you plan to cross-compile for Windows (that is, use your Linux system to build a Windows binary), there are a few additional setup steps. As of 2018-10-16 we have tested this using Ubuntu 18.04 ("Bionic Beaver").

Install the mingw-w64 package:

```
sudo apt-get install mingw-w64
```

The following two commands will display a current selection and prompt you for a new selection. Make sure the ‘posix’ compiler variants are selected for gcc and g++.

```
sudo update-alternatives --config x86_64-w64-mingw32-gcc
sudo update-alternatives --config x86_64-w64-mingw32-g++
```

FEDORA

```
sudo dnf install \
git pkgconfig automake autoconf ncurses-devel python \
python-zmq wget curl gtest-devel gcc gcc-c++ libtool patch
```

RHEL (including Scientific Linux)

- Install devtoolset-3 and autotools-latest (if not previously installed).
- Run `scl enable devtoolset-3 'scl enable autotools-latest bash'` and do the remainder of the build in the shell that this starts.

MACOS 10.12+ (Using the Terminal application)

1. **Install macOS command line tools:**

   ```
   xcode-select --install
   ```

2. **Install Homebrew:**

   ```
   /usr/bin/ruby -e" $(curl -fsSL https://raw.githubusercontent.com/Homebrew/
   ↪install/master/install)"
   ```

3. **Install packages:**

   ```
   brew install git pkgconfig automake autoconf wget libtool coreutils
   ```

4. **Install `pip` :**

   ```
   sudo easy_install pip
   ```

5. **Install python modules for rpc-tests**

   ```
   sudo pip install pyblake2 pyzmq
   ```

Note: There is an existing bug for macOS Mojave (10.14) that causes a failure in building bitzec. A work around for this includes one more step:

```
open /Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.
↪14.pkg
```

CENTOS 7+

```
sudo yum install \
autoconf libtool unzip git python \
wget curl  automake gcc gcc-c++ patch \
glibc-static libstdc++-static
```

Please execute the below commands in order.

```
sudo yum install centos-release-scl-rh
sudo yum install devtoolset-3-gcc devtoolset-3-gcc-c++
sudo update-alternatives --install /usr/bin/gcc-4.9 gcc-4.9 /opt/rh/devtoolset-3/
↪root/usr/bin/gcc10
sudo update-alternatives --install /usr/bin/g++-4.9 g++-4.9 /opt/rh/devtoolset-3/
↪root/usr/bin/g++10
sclenabledevtoolset-3 bash
```

Note: Please see our *Supported Platform Policy* for additional details.

Dependency Version Check

Next, we need to ensure we have the correct version of `gcc` , `g++` , and `binutils`

1. **gcc/g++ 4.9 or later is required.**

   bitzec has been successfully built using gcc/g++ versions 4.9 to 7.x inclusive.

   Use `g++ --version` or `gcc --version` to check which version you have.

   On Ubuntu Trusty, if your version is too old then you can install gcc/g++ 4.9 as follows:

   ```
   $ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
   $ sudo apt-get update
   $ sudo apt-get install g++-4.9
   ```

2. **binutils 2.22 or later is required.**

   Use `as --version` to check which version you have, and upgrade if necessary.

Downloading bitzec source

Now we need to get the bitzec software from the repository:

```
git clone https://github.com/bitzec/bitzec.git
cdbitzec/
git checkout v1.0.1
./zcutil/fetch-params.sh
```

This will fetch both our Sprout proving and verifying keys (the final ones created in theParameter Generation Ceremony) in addition to the parameters generated in theSapling MPCwhich are used once Sapling activates, and place them into `~/.bitzec-params/`. These keys are just under 1.7GB in size, so it may take some time to download them. Users upgrading from earlier `1.x` releases will need to download the additional Sapling parameters which are just under 800MB in size.

The message printed by `git checkout` about a “detached head” is normal and does not indicate a problem.

4.1.6 Build

Ensure you have successfully installed all system package dependencies as described above. Then run the build, e.g.:

```
./zcutil/build.sh -j$(nproc)
```

Note: To build a Windows binary on another platform (as described in the Ubuntu/Debian section above), add a `HOST` environment variable setting with value `x86_64-w64-mingw32` to the build command, like this:

```
HOST=x86_64-w64-mingw32 ./zcutil/build.sh -j$(nproc)
```

Attention: If you recieved any errors, from the above command, please check out our *Troubleshooting Guide*

Note: This should compile our dependencies and build `bitzecd`. (Note: if you don't have `nproc`, then substitute the number of cores on your system. If the build runs out of memory, try again without the `-j` argument, i.e. just `./zcutil/build.sh`.

4.1.7 Configuration

Following the steps below will create your bitzecd configuration file which can be edited to either connect to `mainnet` or `testnet` as well as applying settings to safely access the RPC interface.

Tip: For a complete list of parameters used in `bitzec.conf`, please check out *bitzec.conf Guide*

Linux Create the data directory:

```
mkdir -p ~/.bitzec
```

macOS Your data directory is already generated at `~/Library/Application Support/bitzec`.

Mainnet

Place a configuration file inside your data directory using the following commands:

Warning: Note that this will overwrite any `bitzec.conf` settings you may have added from testnet. (If you want to run on testnet, you can retain a *bitzec.conf* from testnet.)

Linux

```
echo"addnode=mainnet.z.cash">~/.bitzec/bitzec.conf
```

macOS

```
echo"addnode=mainnet.z.cash">~/Library/Application Support/bitzec/bitzec.conf
```

Example configured for `mainnet` :

`bitzec.conf`

```
addnode=mainnet.z.cash
```

Testnet

After running the above commands to create the *bitzec.conf* file, edit the following parameters in your *bitzec.conf* file to indicate network and node discovery for *testnet*:

- add the line **testnet=1**
- **addnode=testnet.z.cash** instead of **addnode=mainnet.z.cash**

Example configured for `testnet`:

`bitzec.conf`

```
testnet=1
addnode=testnet.z.cash
```

Enabling CPU Mining

If you want to enable CPU mining, run these commands:

Linux

```
echo'gen=1'>> ~/.bitzec/bitzec.conf
echo"genproclimit=-1">>
```

macOS

```
echo'gen=1'>> ~/Library/Application Support/bitzec/bitzec.conf
echo"genproclimit=-1">> ~/Library/Application
```

Setting `genproclimit=-1` mines on the maximum number of threads possible on your CPU. If you want to mine with a lower number of threads, set `genproclimit` equal to the number of threads you would like to mine on.

The default miner is not efficient, but has been well reviewed. To use a much more efficient but unreviewed solver, you can run this command:

```
echo'equihashsolver=tromp'>> ~/.bitzec/bitzec.conf
```

Note, you probably want to read the *bitzec Mining Guide* to learn more mining details.

4.1.8 Usage

Now, run bitzecd!

```
./src/bitzecd
```

To run it in the background (without the node metrics screen that is normally displayed) use `./src/bitzecd --daemon`.

Important: If you are running bitzec for the first time you will need to allow your node to fully sync:

```
Thank you for running a bitzec node!
You're helping to strengthen the network and contributing to a social good :)

In order to ensure you are adequately protecting your privacy when using bitzec,
please see <https://z.cash/support/security/>.

        Block height | 15000
            Connections | 8
Network solution rate | 500000 Sol/s

You are currently not mining.
To enable mining, add 'gen=1' to your bitzec.conf and restart.

Since starting this node 9 minutes, 1 seconds ago:
- You have validated 7815 transactions!

[Press Ctrl+C to exit] [Set 'showmetrics=0' to hide]
```

Notice `319430`, in the above output, after the `Block height` field, this means your bitzecd is fully synced. Alter- natively, if you were *NOT* fully synced your output would look similar to below:

(continues on next page)

```
                                                                           :)
```

(continued from previous page)

```
In order to ensure you are adequately protecting your privacy when using bitzec,
please see <https://z.cash/support/security/>.

    Downloading blocks | 319610 / ~320290 (99%)
            Connections | 6
Network solution rate | 389211802 Sol/s

You are currently not mining.
To enable mining, add 'gen=1' to your bitzec.conf and restart.

Since starting this node 59 seconds ago:
- You have validated 7144 transactions!

[Press Ctrl+C to exit] [Set 'showmetrics=0' to hide]
```

Notice now how the `Block height` field has changed to `Downloading blocks` with value `319610 / ~320290 (99%)`. This indicates that your node is attempting to sync with the current block height.

You should be able to use the RPC after it finishes syncing. If you are running `bitzecd` in the background, issue the below command to test:

(If you did not run bitzecd in the background, you will need to open a new terminal)

```
./src/bitzec-cli getinfo
```

Note: If you are familiar with bitcoind's RPC interface, you can use many of those calls to send bzc between *t-addr* addresses. We do not support the 'Accounts' feature (which has also been deprecated in `bitcoind`) — only the empty string `""` can be used as an account name. The main network node at mainnet.z.cash is also accessible via Tor hidden service at zcmaintvsivr7pcn.onion.

Using bitzec

First, you want to obtain bitzec. You can purchase them from an exchange, from other users, or sell goods and services for them! Exactly how to obtain bitzec (safely) is not in scope for this document, but you should be careful. Avoid scams!

Important: Terminology

bitzec supports two different kinds of addresses, a `z-addr` (which begins with a `z`) is an address that uses zero-knowledge proofs and other cryptography to protect user privacy. There are also `t-addrs` (which begin with a `t`) that are similar to Bitcoin's addresses.

The interfaces are a commandline client (*bitzec-cli*) and a Remote Procedure Call (RPC) interface, which is documented here:

bitzec Payment API

Attention: Wallet Backup

To ensure you have properly backed up your wallet, we **strongly** encourage you to review the *Wallet Backup Instructions* .

Generating a t-addr

Let's generate a t-addr first. If you are running bitzecd for the first time, you can issue `bitzec-cli getaddressesbyaccount ""` to view existing addresses.

```
$ ./src/bitzec-cli getnewaddress
t14oHp2v54vfmdgQ3v3SNuQga8JKHTNi2a1
```

Listing t-addr

```
$ ./src/bitzec-cli getaddressesbyaccount""
```

This should show the address that was just created.

Receiving bitzec with a z-addr

Now let's generate a z-addr.

```
$ ./src/bitzec-cli z_getnewaddress
zcBqWB8VDjVER7uLKb4oHp2v54v2a1jKd9o4FY7mdgQ3gDfG8MiZLvdQga8JK3t58yjXGjQHzMzkGUxSguSs6ZzqpgTNiZG
```

This creates a private address and stores its key in your local wallet file. Give this address to the sender!

A z-addr is pretty large, so it's easy to make mistakes with them. Let's put it in an environment variable to avoid mistakes:

```
$ZADDR=
↪'zcBqWB8VDjVER7uLKb4oHp2v54v2a1jKd9o4FY7mdgQ3gDfG8MiZLvdQga8JK3t58yjXGjQHzMzkGUxSguSs6ZzqpgTNiZG
↪'
```

Listing z-addr

To get a list of all addresses in your wallet for which you have a spending key, run this command:

```
$ ./src/bitzec-cli z_listaddresses
```

You should see something like:

```
[

 ↪"zcBqWB8VDjVER7uLKb4oHp2v54v2a1jKd9o4FY7mdgQ3gDfG8MiZLvdQga8JK3t58yjXGjQHzMzkGUxSguSs6ZzqpgTNiZG
 ↪"
]
```

Sending coins with your z-addr

If someone gives you their z-addr. . .

```
$FRIEND=
↪'zcCDe8krwEt1ozWmGZhBDWrcUfmK3Ue5D5z1f6u2EZLLCjQq7mBRkaAPb45FUH4Tca91rF4R1vf983ukR71kHyXeED4quGV
↪'
```

You can send 0.8 bzc by doing. . .

```
$ ./src/bitzec-cli z_sendmany"$ZADDR""[{\"amount\": 0.8, \"address\": \"$FRIEND\"}]"
```

After waiting about a minute, you can check to see if the operation has finished and produced a result:

```
$ ./src/bitzec-cli z_getoperationresult
```

```
[
 {
     "id" :"opid-4eafcaf3-b028-40e0-9c29-137da5612f63",
     "status" :"success",
     "creation_time" :1473439760,
     "result" : {
         "txid" :"3b85cab48629713cc0caae99a49557d7b906c52a4ade97b944f57b81d9b0852d"
     },
     "execution_secs" :51.64785629
 }
]
```

Additional operations for bitzec-cli

As bitzec is an extension of bitcoin, bitzec-cli supports all commands that are part of the Bitcoin Core API (as of version 0.11.2),https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list

For a full list of new commands that are not part of bitcoin API (mostly addressing operations on z-addrs) see *bitzec Payment API*

To list all bitzec commands:

```
./src/bitzec-clihelp
```

To get help with a particular command:

```
./src/bitzec-clihelp<command>
```

> **Attention:** Known Security Issues
>
> Each release contains a *./doc/security-warnings.md* document describing security issues known to affect that release. You can find the most recent version of this document here:
>
> *Security Warnings*
>
> Please also see our security page for recent notifications and other resources:
>
> https://z.cash/support/security.html

4.2Debian Binary Packages Setup

bitzec Company operates a package repository for 64-bit Debian-based distributions. If you'd like to try out the binary packages, you can set it up on your system and install bitzec from there.

First install the following dependency so you can talk to our repository using HTTPS:

```
sudo apt-get install apt-transport-https
```

Next add the bitzec master signing key to apt's trusted keyring:

```
wget -qO - https://apt.z.cash/bitzec.asc | sudo apt-key add -
```

```
Key fingerprint = A2CB 84F2 48C7 DDC9 6C2C 55C8 AEFD 26F9 66E2 79CD
```

Add the repository to your sources:

```
echo"deb [arch=amd64] https://apt.z.cash/ jessie main"| sudo tee /etc/apt/sources.
↪list.d/bitzec.list
```

Finally, update the cache of sources and install bitzec:

```
sudo apt-get update&&sudo apt-get install bitzec
```

Lastly you can run `bitzec-fetch-params` to fetch the zero-knowledge parameters, and set up *~/.bitzec/bitzec.conf* before running bitzec as your local user, as documented in the *User Guide*.

4.2.1Troubleshooting

Note: Only x86-64 processors are supported.

If you're starting from a new virtual machine, sudo may not come installed. See this issue for instructions to get up and running:https://github.com/bitzec/bitzec/issues/1844

libgomp1 or libstdc++6 version problems

These libraries are provided with gcc/g++. If you see errors related to updating them, you may need to upgrade your gcc/g++ packages to version 4.9 or later. First check which version you have using `g++ --version`; if it is before 4.9 then you will need to upgrade.

On Ubuntu Trusty, you can install gcc/g++ 4.9 as follows:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install g++-4.9
```

Public Key Expiration error

If you see the error when updating to a new version of bitzecd:

```
The following signatures were invalid:  KEYEXPIRED 1539886450
```

Follow the instructions at *Updating Signing Keys*

Revoked Key error

If you see the error when updating to a new version of bitzecd:

```
The following signatures were invalid:  REVKEYSIG AEFD26F966E279CD
```

Remove the key marked as revoked:

```
sudo apt-key del AEFD26F966E279CD
```

Then retrieve the updated key:

```
wget -qO - https://apt.z.cash/bitzec.asc | sudo apt-key add -
```

Then update the list again:

```
sudo apt-get update
```

4.2.2 Tor

The repository is also accessible via Tor, after installing the *apt-transport-tor* package, at the address zcaptnv5ljsxpnjt.onion. Use the following pattern in your sources.list file: `deb [arch=amd64] tor+http://zcaptnv5ljsxpnjt.onion/ jessie main`

4.2.3 Updating Signing Keys

If your Debian binary package isn't updating due to an expired public key, you'll need to replace it with the updated key and apt information.

Remove the old signing key:

```
sudo apt-key del 63C4A2169C1B2FA2
```

Remove the list item from local apt:

```
sudo rm /etc/apt/sources.list.d/bitzec.list
```

Update the repository list:

```
sudo apt-get update
```

Then retrieve new key:

```
wget -qO - https://apt.z.cash/bitzec.asc | sudo apt-key add -
```

Re-get the apt info:

```
echo"deb [arch=amd64] https://apt.z.cash/ jessie main"| sudo tee /etc/apt/sources.
↪list.d/bitzec.list
```

Then update the list again:

```
sudo apt-get update
```

At this point you should be able to upgrade with the new public key.

4.3 Binary Tarball Download & Setup

The bitzec company provides a binary tarball for download.

Download Tarball for v1.0.1

After downloading but before extracting, verify that the checksum of the tarball matches the hash below:

```
sha256sum bitzec-1.0.1-linux64.tar.gz
```

Result: `1cd0f61140e2139296fda10f627133dd6d053ce1f61d34b2f50dff696d56f13f`

This checksum was generated from our gitian deterministic build process.View all gitian signatures.

Once you've verified that it matches, extract the files and move the binaries into your executables $PATH:

```
tar -xvf bitzec-1.0.1-linux64.tar.gz

mv -t /usr/local/bin/ bitzec-1.0.1/bin/*
```

Now that bitzec is installed, run this command to download the parameters used to create and verify shielded transactions:

```
bitzec-fetch-params
```

Finally, set up `~/.bitzec/bitzec.conf` before running bitzec as your local user, as documented in the *User Guide*.

4.4 Troubleshooting Guide

The General FAQ has been reorganized and relocated tohttps://z.cash/support/faq.html. Please check there for the latest updates to our frequently asked questions. The following is a list of questions for Troubleshooting bitzecd, the core bitzec client software.

4.4.1 System Requirements

64-bit Linux (easiest with a Debian-based distribution)

A compiler for C++11 if building from source. Gcc 6.x and above has full C++11 support, and gcc 4.8 and above supports some but not all features. bitzec will not compile with versions of gcc lower than 4.8.

At least 4GB of RAM to generate shielded transactions.

At least 8GB of RAM to successfully run all of the tests.

bitzec runs on port numbers that are 100 less than the corresponding Bitcoin port number. They are:

`8232` for mainnet RPC

`8233` for mainnet peer-to-peer network

`18232` for testnet RPC

`18233` for testnet peer-to-peer network

4.4.2 Building from source

If you did not build by running *build.sh*, you will encounter errors. Be sure to build with:

```
$ ./zcutil/build.sh -j$(nproc)
```

Note: If you don't have nproc, then substitute the number of your processors.

```
Error message: g++: internal compiler error: Killed(program cc1plus)
```

This means your system does not have enough memory for the building process and has failed. Please allocate at least 4GB of computer memory for this process and try again.

```
Error message:'runtime_error'(or other variable)is not a member of'std'. compilation terminated due to -Wfatal-errors.``
```

Check your compiler version and ensure that it support C++11. If you're using a version of gcc below 4.8.x, you will need to upgrade.

```
Error message: gtest failing with undefined reference``
```

If you are developing on different branches of bitzec, there may be an issue with different versions of linked libraries. Try `make clean` and build again.

4.4.3 Running bitzecd

Trying to start bitzecd for the first time, it fails with:

```
could not load param file at /home/rebroad/.bitzec-params/sprout-verifying.key
```

You didn't fetch the parameters necessary for zk-SNARK proofs. If you installed the Debian package, run *bitzec-fetch- params*. If you built from source, run *./zcutil/fetch-params.sh*.

bitzecd crashes with the message:

```
``std::bad_alloc``or``St13runtime_exception``
```

These messages indicate that your computer has run out of memory for running bitzecd. This will most likely happen with mining nodes which require more resources than a full node without running a miner. This can also happen while creating a transaction involving a z-address. You'll need to allocate at least 4GB memory for these transactions.

4.4.4 RPC Interface

To get help with the RPC interface from the command line, use *bitzec-cli help* to list all commands.

To get help with a particular command, use `bitzec-cli help $COMMAND`.

There is also additional documentation under *bitzec Payment API* .

bitzec-cli` stops responding after I use the command `z_importkey

The command has added the key, but your node is currently scanning the blockchain for any transactions related to that key, causing there to be a delay before it returns. This immediate rescan is the default setting for *z_importkey*, which you can override by adding *false* to the command if you simply want to import the key, i.e. *bitzec-cli z_importkey $KEY false*

If, when attempting to execute the `sendrawtransaction` RPC method, you receive the error:

```
AcceptToMemoryPool: absurdly high fees
```

This is most likely caused by not specifying an output address to receive the change when creating the transaction (`createrawtransaction`). This RPC call, unlike `sendmany` and `z_sendmany`, does not do this automatically.

With `createrawtransaction`, the fee is simply the sum of the inputs minus the sum of the outputs. If this difference is larger than 0.0021 bzc (210000 zatoshis), the assumption is that this is unintentional, and the transaction is not sent. If you really do wish to send a transaction with a large fee, add `true` to the end of the `sendrawtransaction` command line. This will allow an arbitrarily high fee.

What if my question isn't answered here?

First search the issues section (https://github.com/bitzec/bitzec/issues) to see if someone else has posted a similar issue and if not, feel free to report your problem there. Please provide as much information about what you've tried and what failed so others can properly assess your situation to help.

Important: If you have ran into any issues upgrading to Overwinter, please see the *Network Upgrade Developer Guide*

4.5 bitzec Payment API

4.5.1 Overview

bitzec extends the Bitcoin Core API with new RPC calls to support private bitzec payments.

Important: bitzec payments make use of **two** address formats:

`taddr` - address for transparent funds (just like a Bitcoin address, value stored in UTXOs)
`zaddr` - address for private funds (value stored in objects called notes)

When transferring funds from one taddr to another taddr, you can use either the existing Bitcoin RPC calls or the new bitzec RPC calls.

When a transfer involves zaddrs, you must use the bitzec RPC calls.

4.5.2 Compatibility with Bitcoin Core

bitzec supports all commands in the Bitcoin Core API (as of version `0.11.2`). Where applicable, bitzec will extend commands in a backwards-compatible way to enable additional functionality.

We do not recommend use of accounts which are now deprecated in Bitcoin Core. Where the account parameter exists in the API, please use `""` as its value, otherwise an error will be returned.

To support multiple users in a single node's wallet, consider using `getnewaddress` or `z_getnewaddress` to obtain a new address for each user. Also consider mapping multiple addresses to each user.

4.5.3 List of bitzec API commands

RPC Calls by Category

Accounting

z_getbalance , *z_gettotalbalance*

Addresses

z_getnewaddress , *z_listaddresses* , *z_validateaddress* , *z_exportviewingkey* , *z_importviewingkey*

Key_Management

z_exportkey , *z_importkey* , *z_exportwallet* , *z_importwallet*

Operations

z_getoperationresult , *z_getoperationstatus* , *z_listoperationids*

Payment

z_listreceivedbyaddress , *z_listunspent* , *z_sendmany* , *z_shieldcoinbase*

RPC Parameter Conventions

Parameter	Definition
taddr	*Transparent address*
zaddr	*Private address*
address	*Accepts both private and transparent addresses.*
amount	*JSON format double-precision number with 1 bzc expressed as 1.00000000.*
memo	*Metadata expressed in hexadecimal format. Limited to 512 bytes, the* *current size of the memo field of a private transaction. Zero padding* *is automatic.*

Accounting

Command: `z_getbalance`

Parameters

1. **`"address"`** ***(string)*** The selected address. It may be a transparent or private address.
2. **`minconf`** ***(numeric, optional, default=1)*** Only include transactions confirmed at least this many times.

Output

amount ***(numeric)*** The total amount in bzc received for this address.

Description

Returns the balance of a taddr or zaddr belonging to the node’s wallet. Optionally set the minimum number of confirmations a transaction must have in order to be included in the balance. Use 0 to unconfirmed transactions.

Examples

The total amount received by address “myaddress”

```
bitzec-cli z_getbalance"myaddress"

0.00000000
```

Command: `z_gettotalbalance`

Parameters

1. **`minconf`** ***(numeric, optional, default=1)*** Only include transactions confirmed at least this many times.

Output

> **“transparent”** ***(numeric)*** The total balance of transparent funds
>
> **“private”** ***(numeric)*** The total balance of private funds
>
> **“total”** ***(numeric)*** The total balance of both transparent and private funds

Description

Return the total value of funds stored in the node’s wallet. Optionally set the minimum number of confirmations a private or transparent transaction must have in order to be included in the balance. Use 0 to count unconfirmed transactions.

Examples

The total amount in the wallet

```
bitzec-cli z_gettotalbalance

{
    "transparent":"0.00",
    "private":"0.00",
    "total":"0.00"
}
```

Addresses

Command: `z_getnewaddress`

Parameters

1. **`type`** ***(string, optional, default=”sprout”)*** The type of address (e.g. “sprout”, “sapling”).

Output

> **“bitzecaddress”** ***(string)*** The new shielded address

Description

Return a new zaddr for sending and receiving payments. The spending key for this zaddr will be added to the node’s wallet.

Examples

Create a new shielded address (as of v2.0.0 and v1.0.1 Sprout is default)

```
bitzec-cli z_getnewaddress

zcU1Cd6zYyZCd2VJF8yKgmzjxdiiU1rgTTjEwoN1CGUWCziPkUTXUjXmX7TMqdMNsTfuiGN1jQoVN4kGxUR4sAPN4XZ7pxb
```

Create a new Sapling shielded address

```
bitzec-cli z_getnewaddress sapling

zs1z7rejlpsa98s2rrrfkwmaxu53e4ue0ulcrw0h4x5g8jl04tak0d3mm47vdtahatqrlkngh9sly
```

Command: `z_listaddresses`

Parameters

1. **`includeWatchonly`** ***(bool, optional,default=false)*** Also include watchonly addresses (see 'z_importviewingkey')

Output

"zaddr" *(string)* A zaddr belonging to the wallet

Description

Returns a list of all the zaddrs in this node's wallet for which you have a spending key.

Examples

List all the zaddrs in this node's wallet

```
bitzec-cli z_listaddresses

[
    "zcU1Cd6zYyZCd2VJ...",
    "zcddV3rosTRpWqNj..."
]
```

Command: `z_validateaddress`

Parameters

1. **`zaddr`** ***(string, required)*** The z address to validate

Output

"isvalid" [true|false, *(boolean)*] If the address is valid or not. If not, this is the only property returned.

"address" ["zaddr", *(string)*] The z address validated

"type" ["xxxx", *(string)*] "sprout" or "sapling"

"ismine" [true|false, *(boolean)*] If the address is yours or not

"payingkey" ["hex", *(string)*] [sprout] The hex value of the paying key, a_pk

"transmissionkey" ["hex", *(string)*] [sprout] The hex value of the transmission key, pk_enc

"diversifier" ["hex", *(string)*] [sapling] The hex value of the diversifier, d

"diversifiedtransmissionkey" ["hex", *(string)*] [sapling] The hex value of pk_d

Description

Return information about the given z address.

Examples

List all the information about a given zaddr.

```
bitzec-cli z_validateaddress
↪"zcWsmqT4X2V4jgxbgiCzyrAfRT1vi1F4sn7M5Pkh66izzw8Uk7LBGAH3DtcSMJeUb2pi3W4SQF8LMKkU2cUuVP68yAGcomL
↪"

{
  "isvalid": true,
  "address":"zcbcb6XnP8hbV5y6ZwsY...",
  "payingkey":"b4ae837...",
  "ismine": true
}
```

Key Management

Command: `z_exportkey`

Parameters

1. **`zaddr`** ***(string, required)*** The zaddr for the private key

Output

“key” ***(string)*** The private key

Description

Requires an unlocked wallet or an unencrypted wallet. Return a zkey for a given zaddr belonging to the node’s wallet. The key will be returned as a string formatted using Base58Check as described in the bitzec protocol spec.

Examples

Export a key for a given zaddr.

```
./bitzec-cli z_exportkey
↪"zcWsmqT4X2V4jgxbgiCzyrAfRT1vi1F4sn7M5Pkh66izzw8Uk7LBGAH3DtcSMJeUb2pi3W4SQF8LMKkU2cUuVP68yAGcomL
↪"

AKWUAkypwQjhZ6LLNa
```

Command: `z_importkey`

Parameters

1. **“zkey”** ***(string, required)*** The zkey (see z_exportkey)
2. **rescan** ***(string, optional, default=”whenkeyisnew”)*** Rescan the wallet for transactions - can be “yes”, “no” or “whenkeyisnew”
3. **startHeight** ***(numeric, optional, default=0)*** Block height to start rescan from

Output

NONE

Description

Wallet must be unlocked. Add a zkey as returned by z_exportkey to a node's wallet. The key should be formatted using Base58Check as described in the bitzec protocol spec. Rescan can be "yes", "no" or the default "whenkeyisnew" to rescan for transactions affecting any address or pubkey script in the wallet (including transactions affecting the newly-added address for this spending key). The startHeight parameter sets the block height to start the rescan from (default is 0).

Examples

Import the zkey with rescan

```
bitzec-cli z_importkey"mykey"
```

Import the zkey with partial rescan

```
bitzec-cli z_importkey"mykey"whenkeyisnew30000
```

Re-import the zkey with longer partial rescan

```
bitzec-cli z_importkey"mykey"whenkeyisnew30000
```

Command: `z_exportwallet`

Parameters

1. **"filename"** ***(string, required)*** The filename, saved in folder set by bitzecd -exportdir option

Output

> "path" (string) The full path of the destination file

Description

Requires an unlocked wallet or an unencrypted wallet. Creates or overwrites a file with taddr private keys and zaddr private keys in a human-readable format. Filename is the file in which the wallet dump will be placed. May be prefaced by an absolute file path. An existing file with that name will be overwritten. No value is returned but a JSON-RPC error will be reported if a failure occurred.

As of Sapling activation, the shielded private keys in this file will be separated into legacy shielded private keys under the title Zkeys and Sapling shielded private keys. The export also includes (as of Sapling activation) a comment with an HD wallet seed and associated fingerprint, both as hex strings. This seed is only for the wallet's Sapling shielded keys and addresses.

Examples

Export a wallet

```
bitzec-cli z_exportwallet"wallet_filename"

<No output will appear if successful>
```

Command: `z_importwallet`

Parameters

1. **"filename"** ***(string, required)*** The wallet file

Output

NONE

Description

Requires an unlocked wallet or an unencrypted wallet. Imports private keys from a file in wallet export file format (see z_exportwallet). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes. Filename is the file to import. The path is relative to bitzecd's working directory. No value is returned but a JSON-RPC error will be reported if a failure occurred. This command does not yet support importing HD seeds and will import Sapling addresses in a standard form (non-HD). To backup and restore the full wallet inclusive of the Sapling HD seed, use the backupwallet command.

Examples

Import a wallet

```
bitzec-cli z_importwallet"path/to/exportdir/nameofbackup"

<No output will appear if successful>
```

Command: `z_exportviewingkey`

Parameters

1. **"zaddr"** ***(string, required)*** The zaddr for the viewing key

Output

"vkey" (string) The viewing key

Description

Reveals the viewing key corresponding to 'zaddr'. Then the z_importviewingkey can be used with this output.

Examples

Export a viewing key for a given address

```
bitzec-cli z_exportviewingkey"myaddress"

ZiVtJjUXq5...
```

Command: `z_importviewingkey`

Parameters

1. **"vkey"** ***(string, required)*** The viewing key (see z_exportviewingkey)
2. **rescan** ***(string, optional, default="whenkeyisnew")*** Rescan the wallet for transactions - can be "yes", "no" or "whenkeyisnew"
3. **startHeight** ***(numeric, optional, default=0)*** Block height to start rescan from

Output

NONE

Description

Adds a viewing key (as returned by z_exportviewingkey) to your wallet.

Examples

Import a viewing key

```
bitzec-cli z_importviewingkey"vkey"
```

Import the viewing key without rescan

```
bitzec-cli z_importviewingkey"vkey", no
```

Import the viewing key with partial rescan

```
bitzec-cli z_importviewingkey"vkey"whenkeyisnew30000
```

Re-import the viewing key with longer partial rescan

```
bitzec-cli z_importviewingkey"vkey"yes20000
```

Payment

Command: `z_listreceivedbyaddress`

Parameters

1. **“address”** ***(string)*** The private address.
2. **minconf** ***(numeric, optional, default=1)*** Only include transactions confirmed at least this many times.

Output

“txid”: xxxxx, ***(string)*** The transaction id

“amount”: xxxxx, ***(numeric)*** The amount of value in the note

“memo”: xxxxx, ***(string)*** Hexademical string representation of memo field

“change”: true|false, ***(boolean)*** True if the address that received the note is also one of the sending addresses

Description

Return a list of amounts received by a zaddr belonging to the node’s wallet. Optionally set the minimum number of confirmations which a received amount must have in order to be included in the result. Use 0 to count unconfirmed transactions.

Examples

Return a list of amounts recieved by a zaddr belonging to the node’s wallet.

```
bitzec-cli z_listreceivedbyaddress
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbhtjf
↪"
```

Command: `z_listunspent`

Parameters

1. **minconf** ***(numeric, optional, default=1)*** The minimum confirmations to filter*
2. **maxconf** ***(numeric, optional, default=9999999)*** The maximum confirmations to filter

3. **"includeWatchonly"** ***(bool, optional, default=false)*** Also include watchonly addresses (see 'z_importviewingkey')
4. **"addresses"** ***(string)*** A json array of zaddrs to filter on. Duplicate addresses not allowed.

```
[
  "address"(string) zaddr
  ,...
]
```

Output

"txid" ["txid", *(string)*] The transaction id

"jsindex" [n *(numeric)*] The joinsplit index

"jsoutindex" [n *(numeric)*] [sprout] The output index of the joinsplit

"outindex" [n *(numeric)*] [sapling] The output index

"confirmations" [n *(numeric)*] The number of confirmations

"spendable" [true|false *(boolean)*] True if note can be spent by wallet, false if note has zero confirmations, false if address is watchonly

"address" ["address", *(string)*] The shielded address

"amount": xxxxx, *(numeric)* The amount of value in the note

"memo": xxxxx, *(string)* Hexademical string representation of memo field

"change": true|false, *(boolean)* True if the address that received the note is also one of the sending addresses

Description

Returns array of unspent shielded notes with between minconf and maxconf (inclusive) confirmations. Optionally filter to only include notes sent to specified addresses. When minconf is 0, unspent notes with zero confirmations are returned even though they are not immediately spendable

Examples

Return an array of unspent shielded notes

```
bitzec-cli z_listunspent
```

Returns array of unspent shielded notes with between minconf and maxconf (inclusive) confirmations. Optionally filter to only include notes sent to specified addresses.

```
bitzec-cli z_listunspent69999999  false "[\
↪"ztbx5DLDxa5ZLFTchHhoPNkKs57QzSyib6UqXpEdy76T1aUdFxJt1w9318Z8DJ73XzbnWHKEZP9Yjg712N5kMmP4QzS9iC9\
↪", \
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbhtjf\
↪"]"
```

Command: z_sendmany

Parameters

1. **"fromaddress"** ***(string, required)*** The taddr or zaddr to send the funds from.

2."amounts" (array, required) An array of json objects representing the amounts to send.

“address”:address (string, required) The address is a taddr or zaddr “amount”:amount (numeric, required) The numeric amount in bzc is the value “memo”:memo (string, optional) If the address is a zaddr, raw data represented in hexadecimal string format

3. **minconf** ***(numeric, optional, default=1)*** Only use funds confirmed at least this many times.
4. **fee** ***(numeric, optional, default=0.0001)*** The fee amount to attach to this transaction.

Output

“operationid” ***(string)*** An operationid to pass to z_getoperationstatus to get the result of the operation.

Description

This is an Asynchronous RPC call. Send funds from an address to multiple outputs. The address can be a taddr or a zaddr. Amounts is a list containing key/value pairs corresponding to the addresses and amount to pay. Each output address can be in taddr or zaddr format. When sending to a zaddr, you also have the option of of attaching a memo in hexadecimal format.

When sending coinbase funds to a zaddr, the node’s wallet does not allow any change. Put another way, spending a partial amount of a coinbase utxo is not allowed. This is not a consensus rule but a local wallet rule due to the current implementation of z_sendmany. In future, this may be removed.

Optionally set the minimum number of confirmations which a private or transparent transaction must have in order to be used as an input. When sending from a zaddr, minconf must must be greater than zero. Optionally set a transaction fee, which by default is 0.0001 bzc. Any transparent change will be sent to a new transparent address. Any private change will be sent back to the zaddr being used as the source of funds Returns an operationid. You use the operationid value with z_getoperationstatus and z_getoperationresult to obtain the result of sending funds, which if successful, will be a txid.

Examples

Send funds from a t-address to z-address output

```
bitzec-cli z_sendmany"t1M72Sfpbz1BPpXFHz9m3CdqATR44Jvaydd"'[{"address":
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbht
↪" ,"amount": 5.0}]'
```

Command: `z_shieldcoinbase`

Parameters

1. **“fromaddress”** ***(string, required)*** The address is a taddr or “*” for all taddrs belonging to the wallet.
2. **“toaddress”** ***(string, required)*** The address is a zaddr.
3. **fee** ***(numeric, optional, default=0.0001)*** The fee amount to attach to this transaction.
4. **limit** ***(numeric, optional, default=50)*** Limit on the maximum number of utxos to shield. Set to 0 to use node option -mempooltxinputlimit (before Overwinter), or as many as will fit in the transaction (after Overwinter).

Output

“remainingUTXOs”: xxx ***(numeric)*** Number of coinbase utxos still available for shielding.

“remainingValue”: xxx ***(numeric)*** Value of coinbase utxos still available for shielding.

“shieldingUTXOs”: xxx ***(numeric)*** Number of coinbase utxos being shielded.

“shieldingValue”: xxx ***(numeric)*** Value of coinbase utxos being shielded.

“opid”: xxx ***(string)*** An operationid to pass to z_getoperationstatus to get the result of the operation.

Description

This is an Asynchronous RPC call. Shield transparent coinbase funds by sending to a shielded z address. Utxos selected for shielding will be locked. If there is an error, they are unlocked. The RPC call listlockunspent can be used to return a list of locked utxos. The number of coinbase utxos selected for shielding can be set with the limit parameter, which has a default value of 50. If the parameter is set to 0, the number of utxos selected is limited by the -mempooltxinputlimit option. Any limit is constrained by a consensus rule defining a maximum transaction size of 10000 bytes. The from address is a taddr or "*" for all taddrs belonging to the wallet. The to address is a zaddr. The default fee is 0.0001. Returns an object containing an operationid which can be used with z_getoperationstatus and z_getoperationresult, along with key-value pairs regarding how many utxos are being shielded in this transaction and what remains to be shielded.

Examples

Shield transparent coinbase funds by sending to a shielded z-address.

```
bitzec-cli z_shieldcoinbase"t1M72Sfpbz1BPpXFHz9m3CdqATR44Jvaydd"
↪"ztfaW34Gj9FrnGUEf833ywDVL62NWXBM81u6EQnM6VR45eYnXhwztecW1SjxA7JrmAXKJhxhj3vDNEpVCQoSvVoSpmbht
↪"
```

Operations

Asynchronous calls return an OperationStatus object which is a JSON object with the following defined key-value pairs:

Item operationid

Description Unique identifier for the async operation. Use this value with `z_getoperationstatus` or `z_getoperationresult` to poll and query the operation and obtain its result.

Item status

Description

Current status of operation:

> **queued :** operation is pending execution **executing :** operation is currently being executed **cancelled:** operation is cancelled **failed :** operation has failed **success :** operation has succeeded

Item result

Description Result object if the status is *'success'*. The exact form of the result object is dependent on the call itself.

Item error

Description Error object if the status is *'failed'*. The error object has the following key-value pairs:

> **code :** number **message:** error message

Important: Depending on the type of asynchronous call, there may be other key-value pairs. For example, a `z_sendmany` operation will also include the following in an OperationStatus object:

> **method** : name of operation (e.g. `z_sendmany`)
> **params** : an object containing the parameters to `z_sendmany`

Currently, as soon as you retrieve the operation status for an operation which has finished, that is it has either succeeded, failed, or been cancelled, the operation and any associated information is removed.

It is currently not possible to cancel operations.

Command `z_getoperationresult`

Parameters

1. **"operationid"** ***(array, optional)*** A list of operation ids we are interested in. If not provided, examine all operations known to the node.

Output

" [object, . . .]" (array) A list of JSON objects

Description

Return OperationStatus JSON objects for all completed operations the node is currently aware of, and then remove the operation from memory. Operationids is an optional array to filter which operations you want to receive status objects for. Output is a list of operation status objects, where the status is either "failed", "cancelled" or "success".

Example

Return OperationStatus JSON objects for all completed operations the node is currently aware of

```
bitzec-cli z_getoperationresult'["operationid", ... ]'
```

Command: `z_getoperationstatus`

Parameters

1. **"operationid"** ***(array, optional)*** A list of operation ids we are interested in. If not provided, examine all operations known to the node.

Output

" [object, . . .]" ***(array)*** A list of JSON objects

Description

Return OperationStatus JSON objects for all operations the node is currently aware of. Operationids is an optional array to filter which operations you want to receive status objects for. Output is a list of operation status objects.

Example

Return OperationStatus JSON objects for all completed operations the node is currently aware of

```
bitzec-cli z_getoperationstatus'["operationid", ... ]'
```

Command: `z_listoperationids`

Parameters

1. **"status"** ***(string, optional)*** Filter result by the operation's state e.g. "success".

Output

"operationid" ***(string)*** An operation id belonging to the wallet

Description

Return a list of operationids for all operations which the node is currently aware of. State is an optional string parameter to filter the operations you want listed by their state. Acceptable parameter values values are 'queued', 'executing', 'success', 'failed',

Examples

Return a list of operationids for all operations which the node is currently aware of

```
bitzec-cli z_listoperationids
```

4.5.4 Asynchronous RPC Call Error Codes

bitzec error codes are defined inrpcprotocol.h

Table 1: z_sendmany

Value	Meaning
`-8`	*RPC_INVALID_PARAMETER*
`-5`	*RPC_INVALID_ADDRESS_OR_KEY*
`-4`	*RPC_WALLET_ERROR*
`-6`	*RPC_WALLET_INSUFFICIENT_FUNDS*
`-16`	*RPC_WALLET_ENCRYPTION_FAILED*
`-12`	*RPC_WALLET_KEYPOOL_RAN_OUT*

RPC_INVALID_PARAMETER

`RPC_INVALID_PARAMETER`	**Invalid, missing or duplicate parameter**
Minconf cannot be zero when sending from zaddr	*Cannot accept minimum confirmation value of zero when sending from zaddr*
Minconf cannot be negative	Cannot accept negative minimum confirmation number.
Minimum number of confirmations cannot be less than 0	Cannot accept negative minimum confirmation number.
From address parameter missing	Missing an address to send funds from.
No recipients	Missing recipient addresses.
Memo must be in hexadecimal format	Encrypted memo field data must be in hexadecimal format.
Memo size of_is too big, maximum allowed is _	Encrypted memo field data exceeds maximum size of 512 bytes.
From address does not belong to this node, zaddr spending key not found.	Sender address spending key not found.
Invalid parameter, expected object	Expected object.
Invalid parameter, unknown key: _	Unknown key.
Invalid parameter, expected valid size	Invalid size.
Invalid parameter, expected hex txid	Invalid txid.
Invalid parameter, vout must be positive	Invalid vout.
Invalid parameter, duplicated address	Address is duplicated.
Invalid parameter, amounts array is empty	Amounts array is empty.
Invalid parameter, unknown key	Key not found.
Invalid parameter, unknown address format	Unknown address format.
Invalid parameter, size of memo	Invalid memo field size.
Invalid parameter, amount must be positive	Invalid or negative amount.
Invalid parameter, too many zaddr outputs	z_address outputs exceed maximum allowed.
Invalid parameter, expected memo data in hexadecimal format	Encrypted memo field is not in hexadecimal format.
Invalid parameter, size of memo is larger than maximum allowed _	Encrypted memo field data exceeds maximum size of 512 bytes.

RPC_INVALID_ADDRESS_OR_KEY

`RPC_INVALID_ADDRESS_OR_KEY`	**Invalid address or key**
Invalid from address, no spending key found for zaddr	z_address spending key not found.
Invalid output address, not a valid taddr.	Transparent output address is invalid.
Invalid from address, should be a taddr or zaddr.	Sender address is invalid.
From address does not belong to this node, zaddr spending key not found.	Sender address spending key not found.

RPC_WALLET_INSUFFICIENT_FUNDS

`RPC_WALLET_INSUFFICIENT_FUNDS`	**Not enough funds in wallet or account**
Insufficient funds, no UTXOs found for taddr from address.	Insufficient funds for sending address.
Could not find any non-coinbase UTXOs to spend. Coinbase UTXOs can only be sent to a single zaddr recipient.	Must send Coinbase UTXO to a single z_address.
Could not find any non-coinbase UTXOs to spend.	No available non-coinbase UTXOs.
Insufficient funds, no unspent notes found for zaddr from address.	Insufficient funds for sending address.
Insufficient transparent funds, have_, need _ plus fee _	Insufficient funds from transparent address.
Insufficient protected funds, have_, need _ plus fee _	Insufficient funds from shielded address.

RPC_WALLET_ERROR

RPC_WALLET_ERROR	**Unspecified problem with wallet**
Could not find previous JoinSplit anchor	Try restarting node with *-reindex*.
Error decrypting output note of previous JoinSplit: _	
Could not find witness for note commitment	Try restarting node with *-rescan*.
Witness for note commitment is null	Missing witness for note commitment.
Witness for spendable note does not have same anchor as change input	Invalid anchor for spendable note witness.
Not enough funds to pay miners fee	Retry with sufficient funds.
Missing hex data for raw transaction	Raw transaction data is null.
Missing hex data for signed transaction	Hex value for signed transaction is null.
Send raw transaction did not return an error or a txid.	

RPC_WALLET_ENCRYPTION_FAILED

RPC_WALLET_ENCRYPTION_FAILED	**Failed to encrypt the wallet**
Failed to sign transaction	Transaction was not signed, sign transaction and retry.

RPC_WALLET_KEYPOOL_RAN_OUT

RPC_WALLET_KEYPOOL_RAN_OUT	**Keypool ran out, call keypoolrefill first**
Could not generate a taddr to use as a change address	Call keypoolrefill and retry.

Important: To view a community maintained list of the API, please clickhere

4.6 Wallet Backup Instructions

4.6.1 Overview

Backing up your bitzec private keys is the best way to be proactive about preventing loss of access to your bzc.

Problems resulting from bugs in the code, user error, device failure, etc. may lead to losing access to your wallet (and as a result, the private keys of addresses which are required to spend from them).

No matter what the cause of a corrupted or lost wallet could be, we highly recommend all users backup on a regular basis. Anytime a new transparent address or legacy shielded address (`zc...`) in the wallet is generated, we recommending making a new backup so all private keys for those types of addresses in your wallet are safe.

Sapling shielded addresses (`zs...`) are natively generated from an *HDseed*. Export and import of this seed is not fully supported in the RPC yet. Sapling keys can be exported and imported using the options below but will be treated as standard (non-HD) keys.

Note: A backup is a duplicate of data needed to spend bzc so where you keep your backup(s) is another important consideration. You should not store backups where they would be equally or increasingly susceptible to loss or theft.

4.6.2 Backing up your wallet and/or private keys

These instructions are specific for the officially supported bitzec Linux client. For backing up with third-party wallets, please consult with user guides or support channels provided for those services.

There are multiple ways to make sure you have at least one other copy of the private keys needed to spend your bzc and view your shielded bzc.

For all methods, you will need to include an export directory setting in your config file (*bitzec.conf* located in the data directory which is `~/.bitzec/` unless it's been overridden with `datadir=` setting):

```
exportdir=path/to/chosen/export/directory
```

You may choose any directory within the home directory as the location for export & backup files. If the directory doesn't exist, it will be created.

Note: bitzecd will need to be stopped and restarted for edits in the config file to take effect.

Using `backupwallet`

To create a backup of your wallet, use:

```
bitzec-cli backupwallet <nameofbackup>
```

The backup will be an exact copy of the current state of your wallet.dat file stored in the export directory you specified in the config file. The file path will also be returned.

If your original *wallet.dat* file becomes inaccessible for whatever reason, you can use your backup by copying it into your data directory and renaming the copy to *wallet.dat*.

If you generate new addresses in your wallet after using `backupwallet`, they will not be reflected in the backup file. Due to the deterministic property of HD wallets, if you generate new Sapling shielded addresses in your wallet after using `backupwallet`, then restoring that wallet file and calling `z_getnewaddress sapling` will regenerate keys in the same order that they were created originally. After recreating keys with this method, it is recommended to restart the client using `-rescan` to force the client to check for prior transactions and properly update the balances of those addresses.

Using `z_exportwallet` & `z_importwallet`

If you prefer to have an export of your private keys in human readable format, you can use:

```
bitzec-cli z_exportwallet <nameofbackup>
```

This will generate a file in the export directory listing all transparent and shielded private keys with their associated public addresses. The file path will be returned in the command line. As of Sapling activation, the shielded private keys in this file will be separated into legacy shielded private keys under the title *Zkeys* and Sapling shielded private keys.

As of Sapling activation, the export also includes a comment with an HD wallet seed and associated fingerprint, both as hex strings. This seed is *only* for the wallet's Sapling shielded keys and addresses. For example:

```
HDSeed=5fa7753029b99c408e...
fingerprint=5fa7753029b99c408e...
```

To import keys into a wallet which were previously exported to a file, use:

```
bitzec-cli z_importwallet <path/to/exportdir/nameofbackup>
```

Note: `z_importwallet` does not yet support importing HD seeds but will import Sapling addresses in a standard form (non-HD). To backup and restore the full wallet inclusive of the Sapling HD seed, use the instructions for `backupwallet` above..

Using `z_exportkey`, `z_importkey`, `dumpprivkey` & `importprivkey`

If you prefer to export a single private key for a shielded address, you can use:

```
bitzec-cli z_exportkey <z-address>
```

This will return the private key and will not create a new file.

For exporting a single private key for a transparent address, you can use the command inherited from Bitcoin:

```
bitzec-cli dumpprivkey <t-address>
```

This will return the private key and will not create a new file.

To import a private key for a shielded address, use:

```
bitzec-cli z_importkey <z-priv-key>
```

This will add the key to your wallet and rescan the wallet for associated transactions if it is not already part of the wallet.

The rescanning process can take a few minutes for a new private key. To skip it, instead use:

```
bitzec-cli z_importkey <z-private-key> no
```

For other instructions on fine-tuning the wallet rescan, see the command's help documentation:

```
bitzec-cli help z_importkey
```

To import a private key for a transparent address, use:

```
bitzec-cli importprivkey <t-priv-key>
```

This has the same functionality as `z_importkey` but works with transparent addresses.

See the command's help documentation for instructions on fine-tuning the wallet rescan:

```
bitzec-cli help importprivkey
```

Using `dumpwallet`

This command inherited from Bitcoin is deprecated. It will export private keys in a similar fashion as `z_exportwallet` but only for transparent addresses. This file will also include a comment with an HD wallet seed and associated fingerprint as described above.

4.7 Sapling Turnstile

4.7.1 Overview

The Sapling network upgrade requires a new type of shielded address to support the new usability and security improvements it brings to bitzec. Sapling shielded addresses start with "zs" whereas the legacy, Sprout shielded addresses start with "zc".

The Sapling turnstile is an auditing mechanism for the number of bzc in circulation. Shielded bzc cannot be accounted for in the total monetary supply because balances remain private to the owners of shielded address private keys. The Sapling turnstile provides accounting for the bzc held in Sprout shielded addresses as they are migrated to Sapling shielded addresses.

To achieve this, owners of shielded bzc will be required to send their balances to a transparent address before sending to Sapling shielded addresses. Balance transfers from shielded addresses to transparent addresses reveal the value and become associated with the transparent addresses. Transfers from transparent addresses back into shielded addresses reshield the value.

4.7.2 Migration Tool

We plan to release a Sapling migration tool to help users who have funds stored in older Sprout addresses migrate them to a newer Sapling address.

If you are a user who stores funds in older Sprout addresses, we recommend you wait for this new tool before migrating your funds. We expect to release it in 2019 Q1.

The primary goal for development of this tool is protecting the users' privacy via automation to avoid human error or misunderstanding (yes, even advanced users). With that disclaimer, we are providing privacy recommendations below for users who are not going to listen to this advice and proceed with manual migration.

4.7.3 Privacy Recommendations

There are several privacy recommendations that users should follow in order to retain privacy during a manual migration.

Use transparent addresses once for migrating shielded funds

- Do not use transparent addresses which have already received funds.
- Discard transparent addresses after migration is completed.
- Use a unique transparent address for each balance migration.

Split balances into multiple migrations

- Some users might prefer to not reveal the exact shielded balance in a single transfer.
- Remember to use a unique, unused transparent addresses for splitting a balance across transfers.
- A balance that is evenly divided across multiple migrations will have a higher chance of correlation..
- Split balances will also have a higher chance of correlation the closer they are in block time.

Note: Splitting a shielded balance into multiple transfers is a matter of user preference and might not be the best approach for everyone. Using unused transparent addresses for the Sapling migration and discarding afterwards is **highly recommended for all users**.

4.7.4 Examples

> Alice has a shielded balance of 14.6727 bzc and doesn't want to reveal that a single address holds that exact value as she migrates to a new Sapling shielded address. She decides that paying six transaction fees to migrate is worth the cost to not reveal the total balance.
>
> To do this, Alice generates 3 new transparent addresses `t1abc...`, `t1def...` and `t1xyz..` and picks 3 unequal values that add up to her total balance `10`, `4.0001` and `0.6756`.

She initiates the first transaction, to the first fresh transparent address revealing 10 bzc in address `t1abc...`. This leaves 4.6726 bzc in the legacy address accounting for the first transaction fee. Once confirmed, Alice sends 9.999 bzc (accounting for the second transaction fee) from `t1abc...` to her new shielded Sapling address.

She waits a couple of hours before initiating the second transfer, revealing 4.0001 bzc in address `t1def...`. This leaves 0.6724 bzc in the legacy address accounting for the third transaction fee. Once confirmed, she finishes the second transfer by sending 4 bzc (accounting for the fourth transaction fee) to the same Sapling address. The balance in the shielded Sapling address is now 13.999 bzc.

She waits for the next day to initiate the third and final transfer, revealing 0.6723 bzc (accounting for the fifth transaction fee) in address `t1xyz...`. Once the transaction is confirmed, the legacy Sprout address is now empty and a may be discarded. She finishes the final transfer by sending 0.6722 bzc (accounting for the sixth transaction fee) to her Sapling address which now has a balance of 14.6721 bzc (the original balance minus six transaction fees).

Note: If fees are not a concern, users are advised to split balances transactions which send values in powers of ten (.001, .01, .1, 1, 10, 100, 1000, etc.). A user with a balance of 139.34 bzc using this method would migrate 100 bzc once,10 bzc three times,1 bzc nine times,.1 bzc three times and .01 bzc four times. This adds up to 20 migrations and 40 transactions. At the default fee rate (.0001 bzc per transaction) this would cost .004 bzc.

If time is not a concern, users are advised to delay migrating their split balances over intervals of time that are sufficiently random (between 2 and 100+ hours). A user with 40 transactions to complete their migration of 20 balances using this method could pick 40 random hour intervals in which to send the transactions.

4.7.5 Additional Reading

Sapling Addresses & Turnstile Migration

Anatomy of a bitzec Transaction

Transaction Linkability

4.8 bitzec.conf Guide

Below contains information for additional configuration of the `bitzec.conf` file.

4.8.1 Network-Related Settings

Parameter	Description & Example
testnet	Run on the test network instead of the real bitzec network. `testnet=0`
regtest	Run a regression test network `regtest=0`
proxy	Connect via a SOCKS5 proxy `proxy=127.0.0.1:9050`
bind	Bind to given address and always listen on it. Use [host]:port notation for IPv6 `bind=<addr>`
whitebind	Bind to given address and whitelist peers connecting to it. Use [host]:port notation for IPv6 `whitebind=<addr>`

Quick Primer on addnode vs connect

Let's say for instance you use `addnode=4.2.2.4` addnode will connect you to and tell you about the nodes connected to 4.2.2.4. In addition it will tell the other nodes connected to it that you exist so they can connect to you. Connect will not do the above when you 'connect' to it. It will *only* connect you to 4.2.2.4 and no one else. So if you're behind a firewall, or have other problems finding nodes, add some using 'addnode'. If you want to stay private, use 'connect' to only connect to "trusted" nodes. If you run multiple nodes on a LAN, there's no need for all of them to open lots of connections. Instead 'connect' them all to one node that is port forwarded and has lots of connections.

Thanks goes to [Noodle] on Freenode.

Parameter	Description & Example

addnode	Use as many addnode= settings as you like to connect to specific peers
	`addnode=69.164.218.197`
	`addnode=10.0.0.2:8233`

connect	Alternatively use as many connect= settings as you like to connect
	ONLY to specific peers
	`connect=69.164.218.197`
	`connect=10.0.0.1:8233`

listen	Listening mode, enabled by default except when 'connect' is being used `listen=1`
maxconnections	Maximum number of inbound+outbound connections. `maxconnections=6`

4.8.2 JSON-RPC Options

Controlling a running bitzec/bitzecd process

Parameter	Description & Example

addnode	Use as many addnode= settings as you like to connect to specific peers
	`addnode=69.164.218.197`
	`addnode=10.0.0.2:8233`

server	Tells bitzecd to accept JSON-RPC commands (set as default if not specified) `server=1`
rpcbind	Bind to given address to listen for JSON-RPC connections. Use [host]:port notation for IPv6. This option can be specified multiple times (default: bind to all interfaces) `rpcbind=<addr>`
rpcuser	You must set rpcuser and rpcpassword to secure the JSON-RPC api `rpcuser=Ulysses`

rpcpassword	You must set rpcuser and rpcpassword to secure the JSON-RPC api
	`rpcpassword=YourSuperGreatPasswordNumber_`
	`↪DO_NOT_USE_THIS`

rpcclienttimeout	How many seconds bitzec will wait for a complete RPC HTTP request. after the HTTP connection is established. `rpcclienttimeout=30`
rpcallowip	By default, only RPC connections from localhost are allowed. Specify as many rpcallowip= settings as you like to allow connections from other hosts, either as a single IPv4/IPv6 or with a subnet specification. **Note:** Opening up the RPC port to hosts outside your local trusted network is NOT RECOMMENDED, because the rpcpassword is transmitted over the network unencrypted and also because anyone that can authenticate on the RPC port can steal your keys + take over the account running bitzecd For more information see

4.8.3 Transaction Fee

Parameter	Description & Example
sendfreetransactions	Send transactions as zero-fee transactions if possible (default: 0) `sendfreetransactions=0`
txconfirmtarget	Create transactions that have enough fees (or priority) so they are likely to # begin confirmation within n blocks (default: 1). This setting is overridden by the -paytxfee option. `txconfirmtarget=n`

4.8.4 Miscellaneous Options

Parameter	Description & Example
gen	Enable attempt to mine bitzec. `gen=1`
genproclimit	Set the number of threads to be used for mining bitzec (-1 = all cores). `genproclimit=1`
equihashsolver	Specify a different Equihash solver (e.g. "tromp") to try to mine bitzec faster when gen=1. `equihashsolver=default`
keypool	Pre-generate this many public/private key pairs, so wallet backups will be valid for both prior transactions and several dozen future transactions. `keypool=100`

paytxfee	Pay an optional transaction fee every time you send bitzec. Transactions with fees
	are more likely than free transactions to be included in generated blocks, so may
	be validated sooner. This setting does not affect private transactions created with
	`z_sendmany`
	`paytxfee=0.00`

4.9 bitzec Mining Guide

Welcome! This guide is intended to get you mining bitzec, a.k.a. "bzc", on the bitzec mainnet. The unit for mining is Sol/s (Solutions per second).

If you run into snags, please let us know. There's plenty of work needed to make this usable and your input will help us prioritize the worst sharpest edges earlier. For user help, we recommend using our forum:

https://forum.z.cash/

4.9.1 Setup

First, you need to set up your local bitzec node. Follow the *User Guide* up to the end of the section *Build* , then come back here.

4.9.2 Configuration

Configure your node as per *Configuration* , including the section *Enabling CPU Mining* .

4.9.3 Mining

Now, start Mining! `./src/bitzecd`

To run it in the background (without the node metrics screen that is normally displayed):

`./src/bitzecd -daemon`

You should see the following output in the debug log (*~/.bitzec/debug.log*):

```
bitzec Miner started
```

Congratulations! You are now mining on the mainnet.

To stop the bitzec daemon, enter the command:

`./src/bitzec-cli stop`

Spending Mining Rewards

Coins are mined into a t-addr (transparent address), but can only be spent to a z-addr (shielded address), and must be swept out of the t-addr in one transaction with no change. Refer to our *bitzec Payment API* for instructions on how to use the `z_sendmany` command to send coins from a **t-addr** to a **z-addr**. You will need at least 4GB of RAM for this operation.

Mining Pools

If you're mining by yourself or at home, you're most likely to succeed if you join an existing mining pool. See this community-maintainedlist of mining poolsfor further instructions.

P2PKH transactions

The internal *bitzecd* miner inherited from Bitcoin used P2PK for coinbase transactions, but bitzec 1.0.6 and lateruse P2PKH by default, following the trend for Bitcoin.

4.9.4 Configuration Options

Mine to a single address

The internal *bitzecd* miner uses a new transparent address for each mined block. If you want to instead use the same address for every mined block, use the `-mineraddress=`option available in bitzec 1.0.6 and later.

4.10 Security Warnings

4.10.1 Security Audit

bitzec has been subjected to a formal third-party security review. For security announcements, audit results and other general security information, seehttps://z.cash/support/security.html

4.10.2 x86-64 Linux Only

There areknown bugswhich make proving keys generated on 64-bit systems unusable on 32-bit and big-endian systems. It's unclear if a warning will be issued in this case, or if the proving system will be silently compromised.

4.10.3 Wallet Encryption

Wallet encryption is disabled, for several reasons:

- Encrypted wallets are unable to correctly detect shielded spends (due to the nature of unlinkability of JoinSplits) and can incorrectly show larger available shielded balances until the next time the wallet is unlocked. This problem was not limited to failing to recognize the spend; it was possible for the shown balance to increase by the amount of change from a spend, without deducting the spent amount.
- While encrypted wallets prevent spending of funds, they do not maintain the shielding properties of JoinSplits (due to the need to detect spends). That is, someone with access to an encrypted wallet.dat has full visibility of your entire transaction graph (other than newly-detected spends, which suffer from the earlier issue).
- We were concerned about the resistance of the algorithm used to derive wallet encryption keys (inherited from Bitcoin) to dictionary attacks by a powerful attacker. If and when we re-enable wallet encryption, it is likely to be with a modern passphrase-based key derivation algorithm designed for greater resistance to dictionary attack, such as Argon2i.

You should use full-disk encryption (or encryption of your home directory) to protect your wallet at rest, and should assume (even unprivileged) users who are running on your OS can read your wallet.dat file.

4.10.4 Side-Channel Attacks

This implementation of bitzec is not resistant to side-channel attacks. You should assume (even unprivileged) users who are running on the hardware, or who are physically near the hardware, that your `bitzecd` process is running on will be able to:

- Determine the values of your secret spending keys, as well as which notes you are spending, by observing cache side-channels as you perform a JoinSplit operation. This is due to probable side-channel leakage in the libsnark proving machinery.
- Determine which notes you own by observing cache side-channel information leakage from the incremental witnesses as they are updated with new notes.
- Determine which notes you own by observing the trial decryption process of each note ciphertext on the blockchain.

You should ensure no other users have the ability to execute code (even unprivileged) on the hardware your `bitzecd` process runs on until these vulnerabilities are fully analyzed and fixed.

4.10.5 REST Interface

The REST interface is a feature inherited from upstream Bitcoin. By default, it is disabled. We do not recommend you enable it until it has undergone a security review.

4.10.6 RPC Interface

Users should refrain from changing the default setting that only allows RPC connections from localhost. Allowing connections from remote hosts would enable a MITM to execute arbitrary RPC commands, which could lead to compromise of the account running bitzecd and loss of funds. For multi-user services that use one or more bitzecd instances on the backend, the parameters passed in by users should be controlled to prevent confused-deputy attacks which could spend from any keys held by that bitzecd.

4.10.7 Block Chain Reorganization: Major Differences

Users should be aware of new behavior in bitzec that differs significantly from Bitcoin: in the case of a block chain reorganization, Bitcoin's coinbase maturity rule helps to ensure that any reorganization shorter than the maturity interval will not invalidate any of the rolled-back transactions. bitzec keeps Bitcoin's 100-block maturity interval for generation transactions, but because JoinSplits must be anchored within a block, this provides more limited protection against transactions becoming invalidated. In the case of a block chain reorganization for bitzec, all JoinSplits which were anchored within the reorganization interval and any transactions that depend on them will become invalid, rolling back transactions and reverting funds to the original owner. The transaction rebroadcast mechanism inherited from Bitcoin will not successfully rebroadcast transactions depending on invalidated JoinSplits if the anchor needs to change. The creator of an invalidated JoinSplit, as well as the creators of all transactions dependent on it, must rebroadcast the transactions themselves.

Receivers of funds from a JoinSplit can mitigate the risk of relying on funds received from transactions that may be rolled back by using a higher minconf (minimum number of confirmations).

4.10.8 Logging z_* RPC calls

The option `-debug=zrpc` covers logging of the `z_*` calls. This will reveal information about private notes which you might prefer not to disclose. For example, when calling *z_sendmany* to create a shielded transaction, input notes are consumed and new output notes are created.

The option `-debug=zrpcunsafe` covers logging of sensitive information in `z_*` calls which you would only need for debugging and audit purposes. For example, if you want to examine the memo field of a note being spent.

Private spending keys for z addresses are never logged.

4.10.9 Potentially-Missing Required Modifications

In addition to potential mistakes in code we added to Bitcoin Core, and potential mistakes in our modifications to Bitcoin Core, it is also possible that there were potential changes we were supposed to make to Bitcoin Core but didn't, either because we didn't even consider making those changes, or we ran out of time. We have brainstormed and documented a variety of such possibilities inissue #826, and believe that we have changed or done everything that was necessary for the 1.0.0 launch. Users may want to review this list themselves.

4.11 Glossary

Address A bitzec address is similar to a physical address or an email address. It is the only information you need to provide for someone to send you *bzc*. There are two types of addresses in bitzec: a *shielded address* and a *transparent address*.

Block A block is a record in the bitzec blockchain that contains a set of transactions sent on the network. Pending inclusion in a block, a transaction is kept in the *mempool* in an *unconfirmed* state. Roughly every 2.5 minutes, on average, a new block is appended to the *blockchain* through *mining* and the transactions included receive their first *confirmation*.

Block reward A block reward is new *bzc* released into the network after the successful *mining* of a block. For the first four years, the block reward in bitzec is split into a *miners' reward* and a *founders' reward*. During this time, miners receive 80% (or 10 *bzc*) per block with the remaining 20% (or 2.5 bzc) split between a range of beneficiaries including a *bitzec Company* strategic reserve, the *bitzec Foundation* and many stakeholders including bitzec founders, employees, investors and advisors. After 850,000 *blocks*, the block reward halves for the first time and miners start to receive 100% of the block reward (6.25 bzc). Each subsequent 840,000 blocks triggers a new block reward halving.

Blockchain The blockchain is a public record of bitzec transactions in chronological order. The blockchain is shared between all bitzec users. It is used to verify the permanence of bitzec transactions and to prevent *double spending*.

Confirmation A transaction confirmation first occurs when that transaction has been included in a *block* and gains an additional confirmation for each subsequent block. The more confirmations a transaction has, the higher the security from a potential reversal (see: *rollback*). Some may consider a single confirmation to be secure for low value transactions, although it is generally recommended to wait for 10+ confirmations.

Cryptography Cryptography is the branch of mathematics that lets us create mathematical proofs that provide high levels of security and privacy. Services like online commerce and banking already use cryptography and in many countries, are required by law to protect customers and their data. In the case of bitzec, cryptography is used to:

1. protect user privacy (via *zk-SNARKs*)
2. make it impossible for anybody to spend funds from another user's wallet
3. prevent corruption of the blockchain database

Encrypted memo The encrypted memo is an additional field for *transactions* sent to *shielded addresses* that is visible to the recipient of a payment. The encrypted memo is visible only the to the sender and recipient, unless the *viewing key* or *payment disclosure* gets shared with a third party.

Equihash Equihash is a proof-of-work *mining* algorithm that is memory-oriented with very efficient verification.

Experimental feature An experimental feature is one that is available to users on the main *bitzec network* but should undergo further testing by users and developers. Users must explicitly opt into enabling an experimental feature until they become fully supported.

Double spend A double spend happens when a user sends the same *bzc* to two different recipients. bitzec *miners*, the bitzec *blockchain* and *zk-SNARKs* are integral for only allowing one transaction to *confirm* and be considered valid.

Memory pool The memory pool (or *mempool* for short) is a temporary staging location for *transactions* which have been verified by nodes in the *bitzec network* but have not yet been included in a *block*. Transactions in the memory pool are considered *unconfirmed*.

Mining Mining is the process where for each *block*, nodes in the bitzec network compete by doing complex mathematical calculations to find a solution based on a self-adjusting difficulty. bitzec miners are rewarded with both the *transaction fees* of the *transactions* they confirm and *block rewards*. bitzec uses a proof-of-work mining algorithm called *Equihash*.

Multi-signature A multi-signature address (also referred to as *multisig*) is a type of *address* which requires multiple *private keysignatures* in order to spend funds. This is a security mechanism to protect against theft or loss of a private key. Currently, multisig functionality is only supported by *transparent addresses*.

Network upgrade A network upgrade is a *software-updates-required* release of the bitzec software. After *activation* of a network upgrade, network nodes running older versions that are not compatible with the upgrade will be forked onto an outdated *blockchain* and will require a software upgrade to rejoin the main network. This is sometimes referred to as a *hard fork* upgrade.

Overwinter Overwinter is the first *network upgrade* for bitzec. Its purpose is strengthening the protocol for future network upgrades. It includes versioning, replay protection for network upgrades, performance improvements for transparent transactions and the *transaction expiry* feature. Overwinter *activated* at *block* height 347500.

Payment disclosure A payment disclosure is a method of proving that a payment was sent to a *shielded address* by revealing the value, receiving address and optional *encrypted memo*. The current implementation of this is as an *experimental feature*.

Private Key A private key is a secret string of data that gives access to spend the *bzc* balance of an associated *address* through a cryptographic *signature*. Your private key(s) may be stored directly in your computer or smartphone, with a custodian such as an exchange or a combination of both using *multisig*. Private keys are important to keep safe as they are the only access to spending the funds you may own. For securing your private keys with the bitzecd client, review the *Wallet Backup Instructions*.

Public parameters The bitzec public parameters are a set of global constraints required for constructing and verifying the *zk-SNARKs* used for *shielded addresses*.

Rollback A rollback is when a blockchain is rewound to a previous state and a set of the most recent *blocks* and the *transactions* they contain are discarded. bitzec has a rollback limit of 100 blocks.

Sapling Sapling is a *network upgrade* that introduces significant efficiency improvements for shielded transactions that will pave the way for broad mobile, exchange and vendor adoption of bitzec shielded addresses. Sapling is scheduled to *activate* at *block* height 419200.

Selective disclosure Selective disclosure refers to the features of *shielded addresses* where the owner may *selectively disclose* shielded transaction data. A user may share a *viewing key* or *payment disclosure* with any third party, allowing them to access shielded data while maintaining privacy from others.

Shielded address A shielded *address* (also referred to as a *zaddr*) sends or receives *transactions* such that the address, associated value and *encrypted memo* are not visible on the bitzec *blockchain*. These addresses start with the letter *z*. A shielded address uses *zk-SNARKs* to protect transaction data for value sent or received to it. A transaction consisting of only shielded addresses is called a *shielded transaction*. A transaction consisting of both shielded addresses and *transparent addresses* only protects the data associated with the shielded address. Each shielded address has a *spending key* and *viewing key*.

Shielded transaction A shielded transaction is a transaction exclusively between *shielded addresses*. The addresses, value and optional *encrypted memo* are shielded using *zk-SNARKcryptography* before the transaction is recorded in the *blockchain*.

Signature A cryptographic signature is a mathematical scheme that allows someone to authenticate digital information. When your bitzec *wallet* signs a transaction with the appropriate *private key*, the network can confirm that the signature matches the *bzc* being spent. This signing is confirmed publicly for *transparent addresses* and through the use of *zk-SNARKs* for *shielded addresses*.

Sol/s Sol/s refers to solutions per second and measures the rate at which *Equihash* solutions are found. Each one of those solutions is tested against the current target (after adding to the block header and hashing), in the same way that in Bitcoin each nonce variation is tested against the target.

Spending key A spending key is a type of *private key* that allows any user in possession of it to spend the balance of the associated *address*. For *shielded addresses*, possessing the spending key also allows the user to view the address' balance and *transaction* data.

Sprout Sprout is the first version of bitzec, launched on October 28, 2016.

TAZ TAZ is the three letter code for the valueless bitzec *testnet* currency.

Testnet The bitzec testnet is an alternative *blockchain* that attempts to mimic the main *bitzec network* for testing purposes. Testnet coins (sometimes referred to as *TAZ*) are distinct from actual *bzc* and do not have value. Developers and users can experiment with the testnet without having to use valuable currency. The testnet is also used to test *network upgrades* and their *activation* before committing to the upgrade on the main *bitzec network*.

Transaction A transaction is a payment between users. They are locally created by the user or service then submitted to the *bitzec network* for verification by nodes and eventual *confirmation* into a *block*.

Transaction expiry A transaction expires after staying *unconfirmed* in the *mempool* for too long and is discarded. Once a transaction expires, it may be resubmitted to the network or a new transaction may be submitted in its place. The default expiry in bitzec is 20 *blocks*.

Transaction fee A transaction fee is an additional value added to a *transaction* used to incentivize *miners* to include the transaction into a *block*. Transactions with low or no fee may still be mined but transactions with the default fee or higher will be preferred. If a transaction has too low of a fee, it may stay in the *mempool* until the *transaction expires*. The fee value is not protected for transactions containing *shielded addresses* and therefore it is recommended to always use the default fee of *.0001 bzc*. Unique fees may result in loss of privacy in some cases.

Transparent address A transparent *address* (also referred to as a *taddr*) sends or receives *transactions* such that the address and associated value are publicly recorded on the bitzec *blockchain*. These addresses start with the letter *t*. A transparent address does not use *zk-SNARKs* to protect transaction data for value sent or received to it. A transaction consisting of only transparent addresses reveals the entire transaction. A transaction consisting of both transparent addresses and *shielded addresses* only reveals the data associated with the transparent address.

Transparent transaction A transparent transaction is a transaction exclusively between *transparent addresses*. The addresses and value are recorded publicly on the *blockchain*.

Upgrade activation An upgrade activation is a specific *block* height that triggers a *network upgrade*.

Viewing key A viewing key is a type of *private key* that allows any user in possession of it to view the balance and transaction data of the associated *shielded address*.

Wallet A bitzec wallet contains *private key(s)* which allow the owner to spend the *bzc* balance it contains. Each bitzec wallet can show you the total balance of all *bzc* it controls and lets you pay a specific amount to a specific *address*, just like a real wallet you keep in your pocket or purse. This is different to credit cards where customers are charged by the merchant.

bitzec network The bitzec network is a *peer-to-peer* network of nodes where each node may interact directly with the others for broadcasting newly submitted *transactions*, *minedblocks* and various other messages that regulate behavior. This type of structure removes the need for a trusted regulating central party.

bitzec bitzec is an in-production cryptocurrency implementation of the Zerocash protocol, with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by Bitcoin with a *shielded* payment scheme secured by *zk-SNARKs*. It implements the *Equihash* proof-of-work *mining* algorithm. Both the network and the associated currency are referred to as *bitzec* with *bzc* referring specifically to the currency.

bitzec Foundation The bitzec Foundation is a 501(c)3 non-profit dedicated to building Internet payment and privacy infrastructure for the public good, primarily serving the users of the bitzec protocol and blockchain.

bzc bzc is the three letter currency code for the bitzec cryptocurrency. It is also used to help distinguish the *bitzec network* from the currency. Note that some exchanges use *XZC* as the bitzec currency code to conform with the ISO 4217standard for currencies and similar assets not associated with a nation.

ZECC This is the abbreviation for bitzec Electric Coin Company, the team behind the *bitzec protocol*. *bitzec Company* is a common alternative reference to this team.

Zerocash Zerocash is a cryptographic protocol invented by Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza in 2014. It improves on the earlier *Zerocoin* protocol developed by some of the same authors both in functionality and efficiency.

Zerocoin Zerocoin is a cryptographic protocol invented by Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin in 2013. It is a less efficient predecessor of *Zerocash*.

zk-SNARKs A zk-SNARK is a particular form of zero-knowledge proof used in the *bitzec protocol* which allows *shielded addresses* to prove the validity of associated *transactions* without revealing the *address* or value transacted. For Bitcoin and *transparent addresses*, *miners* can verify that a transaction has not been *double spent* because the addresses and their balances are publicly visible within transactions. zk-SNARKs allow this same double spend protection for shielded addresses. The term, which stands for *zero-knowledge Succinct Non-interactive ARguments of Knowledge*, was first used in the *Zerocash* whitepaper.

4.12 Development Guidelines

We achieve our design goals primarily through this codebase as a reference implementation. This repository is a fork ofBitcoin Coreas of upstream release 0.11.2 (many later Bitcoin PRs have also been ported to bitzec). It implements thebitzec protocoland a few other distinct features.

- Bitcoin Core:https://github.com/bitcoin/bitcoin
- bitzec Protocol:https://github.com/bitzec/zips/blob/master/protocol/protocol.pdf

4.12.1 bitzec Github Workflow

This document describes the standard workflows and terminology for developers at bitzec. It is intended to provide procedures that will allow users to contribute to the open-source code base. Below are common workflows users will encounter:

1. *Fork bitzec Repository*
2. *Create Branch*
3. *Make & Commit Changes*
4. *Create Pull Request*
5. *Discuss / Review PR*
6. *Deploy / Merge PR*

Before continuing, please ensure you have an existing Github or Gitlab account. If not, visitGithuborGitlabto create an account.

Fork bitzec Repository

This step assumes you are starting with a new Github/Gitlab environment. If you have already forked the bitzec repository, please continue to *Create Branch* section. Otherwise, open up a terminal and issue the below commands:

Note: Please replace `your_username`, with your actual Github username

```
git clone git@github.com:your_username/bitzec.git
cdbitzec
git remote set-url origin
git@github.com:your_username/bitzec.git git remote add upstream
git@github.com:bitzec/bitzec.git
git remote set-url --push upstream DISABLED
git fetch upstream
```

After issuing the above commands, your `.git/config` file should look similar to the following:

```
[core]
    repositoryformatversion=0
    filemode=true
    bare=false
    logallrefupdates=true
[remote"origin"]
    url=git@github.com:your_username/bitzec.gi
    t fetch=+refs/heads/
                        *:refs/remotes/origin/*
[branch"master"]
    remote=upstream
    merge=refs/heads/master
[remote"upstream"]
    url=git@github.com:bitzec/bitzec.git
    fetch=+refs/heads/  *:refs/remotes/upstream/*
    pushurl=DISABLED
```

This setup provides a single cloned environment to develop for bitzec. There are alternative methods using multiple clones, but this document does not cover that process.

Create Branch

While working on the bitzec project, you are going to have bugs, features, and ideas to work on. Branching exists to aid these different tasks while you write code. Below are some conventions of branching at bitzec:

1. `master` branch is **ALWAYS** deployable
2. Avoid branching directly off `master`, instead use your local fork
3. Branch names **MUST** be descriptive (e.g. `issue#_short_description`)

To create a new branch (assuming you are in `bitzec` directory):

```
git checkout -b[new_branch_name]
```

Note: Even though you have created a new branch, until you `git push` this local branch, it will not show up in your bitzec fork on Github (e.g.https://github.com/your_username/bitzec)

To checkout an existing branch (assuming you are in `bitzec` directory):

```
git checkout[existing_branch_name]
```

If you are fixing a bug or implementing a new feature, you likely will want to create a new branch. If you are reviewing code or working on existing branches, you likely will checkout an existing branch. To view the list of current bitzec Github issues, clickhere.

Make & Commit Changes

If you have created a new branch or checked out an existing one, it is time to make changes to your local source code. Below are some formalities for commits:

1. Commit messages **MUST** be clear
2. Commit messages **MUST** be descriptive
3. Commit messages **MUST** be clean (see *Squashing Commits* for details)

Commit messages should contain enough information in the first line to be able to scan a list of patches and identify which one is being searched for. Do not use "auto-close" keywords – tickets should be closed manually. The auto-close keywords are "close[ds]", "resolve[ds]", and "fix(e[ds])?"

While continuing to do development on a branch, keep in mind that other approved commits are getting merged into `master`. In order to ensure there are minimal to no merge conflicts, we need `rebase` with master.

If you are new to this process, please sanity check your remotes:

```
git remote -v
```

```
origin    git@github.com:your_username/bitzec.git(fetch)
origin
          git@github.com:your_username/bitzec.git(push
) upstream   git@github.com:bitzec/bitzec.git(fetch)
upstream     DISABLED(push)
```

This output should be consistent with your `.git/config`:

```
[branch"master"]
    remote=upstream
    merge=refs/heads/master
[remote"origin"]
    url=git@github.com:your_username/bitzec.gi
    t fetch=+refs/heads/
                        *:refs/remotes/origin/*
[remote"upstream"]
    url=git@github.com:bitzec/bitzec.git
    fetch=+refs/heads/  *:refs/remotes/upstream/*
    pushurl=DISABLED
```

Once you have confirmed your branch/remote is valid, issue the following commands (assumes you have **NO** existing uncommitted changes):

```
git fetch upstream
git rebase upstream/master
git push -f
```

If you have uncommitted changes, use `git stash` to preserve them:

```
git stash
git fetch upstream
git rebase upstream/master
git push -f
git stash pop
```

Using `git stash` allows you to temporarily store your changes while you rebase with `master`. Without this, you will rebase with master and lose your local changes.

Before committing changes, ensure your commit messages follow these guidelines:

1.Separate subject from body with a blank line

2.Limit the subject line to 50 characters

3.Capitalize the subject line

4.Do not end the subject line with a period

5.Wrap the body at 72 characters

6.Use the body to explain *what* and *why* vs. *how*

Once synced with `master`, let's commit our changes:

```
git add[files...]  # default is all files, be careful not to add unintended files
git commit -m'Message describing commit'
git push
```

Now that all the files changed have been committed, let's continue to Create Pull Request section.

Create Pull Request

On your Github page (e.g.https://github.com/your_username/bitzec), you will notice a newly created banner contain- ing your recent commit with a big green `Compare & pull request` button. Click on it.

First, write a brief summary comment for your PR – this first comment should be no more than a few lines because it ends up in the merge commit message. This comment should mention the issue number preceded by a hash symbol (e.g. #2984).

Add a second comment if more explanation is needed. It's important to explain why this pull request should be accepted. State whether the proposed change fixes part of the problem or all of it; if the change is temporary (a workaround) or permanent; if the problem also exists upstream (Bitcoin) and, if so, if and how it was fixed there.

If you click on *Commits*, you should see the diff of that commit; it's advisable to verify it's what you expect. You can also click on the small plus signs that appear when you hover over the lines on either the left or right side and add a comment specific to that part of the code. This is very helpful, as you don't have to tell the reviewers (in a general comment) that you're referring to a certain line in a certain file.

Add comments **before** adding reviewers, otherwise they will get a separate email for each comment you add. Once you're happy with the documentation you've added to your PR, select reviewers along the right side. For a trivial change (like the example here), one reviewer is enough, but generally you should have at least two reviewers, at least one of whom should be experienced. It may be good to add one less experienced engineer as a learning experience for that person.

Discuss / Review PR

In order to merge your PR with `master`, you will need to convince the reviewers of the intentions of your code.

Important: If your PR introduces code that does not have existing tests to ensure it operates gracefully, you **MUST** also create these tests to accompany your PR.

Reviewers will investigate your PR and provide feedback. Generally the comments are explicitly requesting code changes or clarifying implementations. Otherwise Reviewers will reply with PR terminology:

- **Concept ACK** - Agree with the idea and overall direction, but have neither reviewed nor tested the code changes.
- **utACK (untested ACK)** - Reviewed and agree with the code changes but haven't actually tested them.
- **Tested ACK** - Reviewed the code changes and have verified the functionality or bug fix.
- **ACK** - A loose ACK can be confusing. It's best to avoid them unless it's a documentation/comment only change in which case there is nothing to test/verify; therefore the tested/untested distinction is not there.
- **NACK** - Disagree with the code changes/concept. Should be accompanied by an explanation.

Squashing Commits

Before your PR is accepted, you might be requested to squash your commits to clean up the logs. This can be done using the following approach:

```
git checkout branch_name
git rebase -i HEAD~4
```

The integer value after ~ represents the number of commits you would like to interactively rebase. You can pick a value that makes sense for your situation. A template will pop-up in your terminal requesting you to specify what commands you would like to do with each prior commit:

```
Commands:
p, pick = use commit
r, reword = use commit, but edit the commit message
e, edit = use commit, but stop for amending
s, squash = use commit, but meld into previous commit
f, fixup = like "squash", but discard this commit's log message
x, exec = run command (the rest of the line) using shell
```

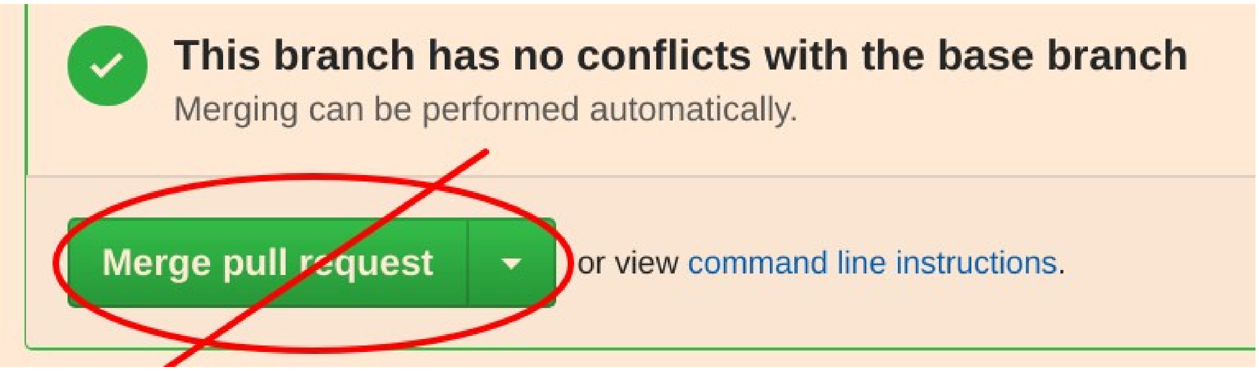
Modify each line with the according command, followed by the hash of the commit. For example, if I wanted to squash my last 4 commits into the most recent commit for this PR:

```
p 1fc6c95 Final commit message
s 6b2481b Third commit message
s dd1475d Second commit message
s c619268  First commit message
```

```
git push origin branch-name --force
```

Deploy / Merge PR

Important: DO NOT click on this button! We use a different process (`zkbot`, `Homu`) to merge code



This branch has no conflicts with the base branch

Merging can be performed automatically.

---

zkbot

We use a homu instance called `zkbot` to merge *all* PRs. (Direct pushing to the `master` branch of the repo is not allowed.) Here's just a quick overview of how it works.

If you're on our team, you can do `@zkbot <command>` to tell zkbot to do things. Here are a few examples:

- `r+ [commithash]` this will test the merge and then actually commit the merge into the repo if the tests succeed.
- `try` this will test the merge and nothing else.
- `rollup` this is like `r+` but for insignificant changes. Use this when we want to test a bunch of merges at once to save Buildbot time.

More instructions are found here:http://ci.z.cash:12477/

Once you have addressed the comments in your PR, and it has received two *ACKs* from reviewers, you can attempt to test merge the PR:

```
@zkbot try
```

Note: `@zkbot` commands are entered into Github tickets as comments

This will instruct Buildbot(aka Homu) to test merging your PR with `master` and ensure it passes the full test suite. You may or may not have permissions to run this command, but Github will reply with output indicating if you can or not.

If the `@zkbot try` fails, you will need to go back and address the issues accordingly. Otherwise, you can now attempt to merge into `master`:

```
@zkbot r+
```

Note: `@zkbot` commands are entered into Github tickets as comments

There are very few people that have `@zkbot r+` privileges, so you can request one of these people to merge the PR, or leave it for the release process to pick it up. Finally, when the PR is merged into `master` successfully, your PR will close.

There will be times when your PR is waiting for some portion of the above process. If you are requested to rebase your PR, in order to gracefully merge into `master`, please do the following:

```
git checkout branch_name
git fetch upstream
git rebase upstream/master
git push -f
```

4.12.2 bitzec Developer Workflow

Tip: The flow below assumes you have already downloaded the parameters using `./zcutil/fetch-params.sh`

Below describes a standard workflow for developing code in the bitzec repository:

1. **Clone your bitzec fork**

   ```
   git clone git@github.com:your_username/bitzec.git
   ```

2. **Create a branch for local changes**

   ```
   cdbitzec
   git checkout -b[new_branch_name]
   ```

3. **Build bitzec**

   ```
   /zcutil/build.sh -j$(nproc)
   ```

4. **Create & build changes to code**

   ```
   make
   ```

This will allow you to create/edit existing bitzec code, and build it locally. If you want to submit a PR for this newly created code, please refer back to *Make & Commit Changes* section. After completing those steps, please ensure you have also followed *Create Pull Request* and *Deploy / Merge PR* sections.

Testing

To ensure the existing bitzec code is tested, we use the following tools:

Gtest

Add unit tests for bitzec under `./src/gtest`.

To list all tests, run `./src/bitzec-gtest --gtest_list_tests`.

To run a subset of tests, use a regular expression with the flag `--gtest_filter`. Example:

```
./src/bitzec-gtest --gtest_filter=DeprecationTest.*
```

For debugging: `--gtest_break_on_failure`.

BOOST

To run a subset of BOOST tests:

```
src/test/test_bitcoin -t TESTGROUP/TESTNAME
```

RPC Tests

To run the main test suite:

```
qa/bitzec/full_test_suite.py
```

To run the RPC tests:

```
qa/pull-tester/rpc-tests.sh
```

The main test suite uses two different testing frameworks. Tests using the Boost framework are under `src/test/`; tests using the Google Test/Google Mock framework are under `src/gtest/` and `src/wallet/gtest/`. The latter framework is preferred for new bitzec unit tests.

RPC tests are implemented in Python under the qa/rpc-tests/ directory.

4.12.3 Continuous Integration

Watch theBuildbot.

Buildbot:https://ci.z.cash/

4.12.4 Releases

Starting from bitzec v1.0.0-beta1, bitzec version numbers and release tags take one of the following forms:

> v<X>.<Y>.<Z>-beta<N>
>
> v<X>.<Y>.<Z>-rc<N>
>
> v<X>.<Y>.<Z>
>
> v<X>.<Y>.<Z>-<N>

Alpha releases used a different convention: `v0.11.2.z<N>` (because bitzec was forked from Bitcoin v0.11.2).

4.13 Supported Platform Policy

Our supported platforms policy is as follows, largely inspired by theTahoe-LAFS Buildbot policy

- A supported platform must have a Buildbot builder which builds and runs all tests.
- A Buildbot builder uses the 'default' or 'standard' configuration for its platform, and if we need something substantially different, we call that something different. Examples include default file systems, default compilers, default kernels, etc…
- If build/testing for a supported platform fails, this blocks progress on all platforms.
  - If the merge-acceptance test suite fails only on macOS, but not other platforms, a merge should fail.

  - If the pre-release test suite fails only on Debian, the release is blocked on all platforms.

- Furthermore, for pre-release testing, we should run tests of candidate packages on default systems which don't even have developer tools, unless those tools come by default on that platform.

- For platforms which have frequent updates, such as *Debian testing*, we should upgrade all installed packages on the builders during each *CI deployment*.

By contrast *unsupported platforms* do not block progress. These may still have Buildbot builders, partial test suites. Unsupported platforms should still use default configurations, or be appropriately named to distinguish their uniqueness.

4.14 bitzec Improvement Proposals (ZIPs)

4.14.1 Purpose and Guidelines

ZIP stands for bitzec Improvement Proposal. A ZIP is a design document providing information to the bitzec community, or describing a new feature for bitzec or its processes or environment. The ZIP should provide a concise technical specification of the feature and a rationale for the feature.

We intend ZIPs to be the primary mechanisms for proposing new features, for collecting community input on an issue, and for documenting the design decisions that have gone into bitzec. The ZIP authors are responsible for building consensus within the community and documenting dissenting opinions.

ZIPs go through a sequence of versions as described under *Versioning*.

4.14.2 ZIP Categories

There are three kinds of ZIP:

1. A **Standards Track ZIP** describes any change that affects most or all bitzec implementations, such as a change to the network protocol, a change in block or transaction validity rules, or any change or addition that affects the interoperability of applications using bitzec. In particular, ZIPs that propose changes to consensus MUST be Standards Track.

2. An **Informational ZIP** describes a bitzec design issue, or provides general guidelines or information to the bitzec community, but does not propose a new feature. Informational ZIPs do not necessarily represent a bitzec community consensus or recommendation, so users and implementors are free to ignore Informational ZIPs or follow their advice.

3. A **Process ZIP** describes a process surrounding bitzec, or proposes a change to (or an event in) a process. Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in bitzec development.

4.14.3 ZIP Work Flow

The ZIP process begins with a new idea for bitzec. ZIPs do not replace thebitzec issue tracker; typically, an idea will first have been proposed as an issue on that tracker, and will be discussed there. Only when and if an idea has progressed to the point where it is useful to propose a more formal specification, will a ZIP bewritten.

Each potential ZIP must have one or more *authors* – people who write the ZIP using the style and format described below, shepherd the discussions in the appropriate forums, and attempt to build community consensus around the idea. See *ZIP Editors*

Vetting an idea publicly before going as far as writing a ZIP is meant to save both the potential authors and the wider community time. The bitzec issue tracker contains many ideas for changing bitzec that have been rejected for various reasons. Searching this tracker and asking the bitzec community first if an idea is original helps prevent too much time being spent on something that is guaranteed to be rejected based on prior discussions. It also helps to make sure the idea is applicable to the entire community and not just the authors. Just because an idea sounds good to the authors does not mean it will work for most people in most areas where bitzec is used.

Small enhancements or patches often don't require a ZIP. These should typically be injected into the relevant bitzec development work flow with a pull request to thebitzec issue tracker.

A ZIP should be a clear and complete description of the proposed enhancement. Technical aesthetics and security auditability are important considerations.

ZIPs need not, and generally SHOULD NOT, propose an implementation. (Note that this differs from common practice for Bitcoin Improvement Proposals.) They SHOULD, however, discuss non-trivial implementation considerations whenever appropriate.

The original form of a ZIP is written in (any regional variation of) English, but translations are encouraged and MAY be placed alongside the original by the ZIP Editor.

4.14.4 Versioning

ZIPs are strictly versioned. The versioning scheme starts with "Draft 1", "Draft 2", etc., for however many drafts are needed. When and if the document is considered by its authors and the ZIP Editor to be stable, it becomes "Version 1". Any particular ZIP might not reach this stage. Subsequent revisions, if any, are called "Version 2", etc. for however many revisions are needed.

A ZIP also has a "Change history", separate from the document itself, giving a brief summary of the changes made in each version. See Structure of the ZIPs Repository for detail on how the versions are represented.

The source files for a ZIP are maintained under revision control in theZIPs Repository, but the revision history of that repository MAY contain intermediate commits that do not correspond to document versions.

4.14.5 ZIP Editors

The ZIP Editors are tasked with managing the process of accepting ZIPs, maintaining the ZIPs Repository, assigning ZIP numbers, and performing minor editing tasks on the content and metadata of ZIPs. Any major editing SHOULD instead be performed by the author(s) of a ZIP.

There is presently a single ZIP Editor, Daira Hopwood (but this document still uses "ZIP Editors" for generality). If there is more than one ZIP Editor at a given time, they make decisions by informal consensus.

A Process ZIP describing procedures for selecting new ZIP Editors as and when that becomes necessary SHOULD be submitted before January 1st, 2017.

The ZIP Editors MAY reject a proposed ZIP or update to an existing ZIP for any of the following reasons:

Code	Reason
`0000`	it violates the *Contributor Code of Conduct*
`0001`	it appears too unfocussed or broad
`0002`	it duplicates effort in other ZIPs without sufficient technical justification (however, alternative proposals to address similar or overlapping problems are not excluded for this reason)
`0003`	it has manifest security flaws (including being unrealistically dependent on user vigilance to avoid security weaknesses)
`0004`	it disregards compatibility with the existing bitzec blockchain or ecosystem
`0005`	it is manifestly unimplementable;
`0006`	it includes buggy code, pseudocode, or algorithms
`0007`	it manifestly violates common expectations of a significant portion of the bitzec community;
`0008`	it updates a Draft ZIP to Released when there is significant community opposition to its content (however, Draft ZIPs explicitly may describe proposals to which there is, or could be expected, significant community opposition)
`0009`	in the case of a Released ZIP, the update makes a substantive change to which there is significant community opposition
`0010`	it is dependent on a patent that could potentially be an obstacle to adoption of the ZIP
`0011`	it includes commercial advertising
`0012`	it disregards formatting rules
`0013`	it makes non-editorial edits to previous entries in a ZIP's Change history
`0014`	an update to an existing ZIP extends or changes its scope to an extent that would be better handled as a separate ZIP
`0015`	a new ZIP has been proposed for a category that does not reflect its content, or an update would change a ZIP to an inappropriate category
`0016`	it updates a Released ZIP to Draft when the specification is already implemented and has been in common use
`0017`	it violates any specific "MUST" or "MUST NOT" rule in this document
`0018`	the expressed political views of an author of the document are inimical

Contributor Code of Conduct

The ZIP Editors MUST NOT unreasonably deny publication of a ZIP proposal or update that does not violate any of these criteria. If they refuse a proposal or update, they MUST give an explanation of which of the criteria were violated, with the exception that spam may be deleted without an explanation.

Note that it is not the primary responsibility of the ZIP Editors to review proposals for security, correctness, or implementability.

Please send all ZIP related communications either by email to <*zips@z.cash*> , or by opening an issue on theZIPs issue tracker. However if a communication concerns a potential security vulnerability that could affect bitzec users, the *Coordinated Security Disclosure Procedure* SHOULD be followed.

ZIPs issue tracker

Authors of proposed ZIPs MUST NOT self-assign ZIP numbers. Proposals and updates SHOULD be made as pull requests to the ZIPs Repository. A proposal for a new ZIP MUST indicate whether it is intended to be Standards Track, Informational, or Process. It is also possible to update an Informational ZIP to be Standards Track or vice-versa, with the approval of the ZIP Editors. It is not possible to change a Process ZIP to another category of ZIP, or vice versa. Each ZIP MUST be initially proposed as a Draft.

A ZIP author may at any time withdraw their authorship on any or all versions of a ZIP (even if this results in there being no authors for a given version). Withdrawal of authorship is recorded in the ZIP metadata. An author who has changed their name, formally or informally, can also ask for their name to be updated on the ZIP metadata; the result will not include their previous name unless they ask for it to. (As a technical caveat, the previous name may still be visible in previous git revisions of the *ZIPs Repository* that remain publicly accessible, although it may be possible to fix that by a force-push.)

4.14.6 Relation to the bitzec Protocol Specification

The canonical description of bitzec consensus and security requirements is the protocol specification. It is the responsibility of the ZIP Editors and the authors of the protocol specification to maintain consistency between the specification and ZIPs that overlap its scope.

The protocol specification SHOULD explicitly reference ZIPs that describe proposals that are incorporated into it. Duplication between the protocol specification and such ZIPs is inevitable and acceptable.

To minimize the risk of unintended discrepancies, a ZIP that proposes to change consensus behaviour SHOULD express its proposal in terms of specific text to be added or changed in the specification (in addition to motivation, history, alternative approaches that were not adopted, etc., which may not be appropriate for the specification).

It is highly recommended that a single ZIP contains a single key proposal or new idea. The more focused the ZIP, the more successful it is likely to be. If in doubt, split your ZIP into several well-focused ones.

Both initial proposals and updates to ZIPs SHOULD be submitted by an author of the document as a pull request to theZIPs repository.

A ZIP can also be assigned status “Deferred”. The ZIP author or editor can assign the ZIP this status when no progress is being made on the ZIP. Once a ZIP is deferred, the ZIP editor can re-assign it to draft status.

A ZIP can also be “Rejected”. Perhaps after all is said and done it was not a good idea. It is still important to have a record of this fact.

Some Informational and Process ZIPs may also have a status of “Active” if they are never meant to be completed. E.g. ZIP 1 (this ZIP).

4.14.7 What belongs in a successful ZIP?

Each ZIP should have the following parts:

- Preamble – RFC 822 style headers containing meta-data about the ZIP, including the ZIP number, a short descriptive title (limited to a maximum of 44 characters), the names, and optionally the contact info for each author, etc.
- Abstract – a short description of the issue being addressed.
- Copyright – Each ZIP MUST be licensed under the MIT License, unless the ZIP Editor makes an explicit exception to resolve a license incompatibility with a work from which the ZIP is derived. In the latter case the license MUST be explicitly stated in the ZIP metadata and MUST satisfy theOpen Source Definition
- Specification – The technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow competing, interoperable implementations in principle (whether or not multiple implementations exist).
- Motivation – The motivation is critical for ZIPs that want to change the bitzec protocol. It should clearly explain why the existing protocol specification is inadequate to address the problem that the ZIP solves. ZIP submissions without sufficient motivation may be rejected outright.
- Rationale – The rationale fleshes out the specification by describing what motivated the design and why particular design decisions were made. It should describe alternate designs that were considered and related work.
- The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.
- Backwards Compatibility – All ZIPs that introduce backwards incompatibilities MUST include a section describing these incompatibilities and their severity. The ZIP MUST explain how the author proposes to deal with these incompatibilities.

4.14.8 Formatting Rules

The metadata of a ZIP MUST be represented as a reStructuredText file. This file includes:

- a Change history . . .
- the current authors.

Each Change history entry includes:

- a description of what was changed (this can be just “initial draft” or similar in the case of the first draft).
- a link to the main reStructuredText or LaTeX source file for that version.
- a link to a rendered PDF file for that version.
- the new authors, if this is the first draft or the authors have changed.

ZIPs can be represented in either `reStructuredText` or `LaTeX` format.

Images and diagrams can be included . . . , provided that a rendering to a PNG image is included. SVG is a preferred source format. The ZIP Editor MAY accept other formats. Formats that depend on proprietary software are strongly discouraged.

4.14.9 Rules specific to reStructuredText

The source for the *rst* file MUST be readable in an editor window set to 90 columns, except possibly where prevented by reStructuredText technical limitations (such as avoiding wrapping of URLs).

The document MAY include images in .png format.

4.14.10 Rules specific to LaTeX

The ZIP directory MUST contain a `Makefile`, the default target of which produces a PDF file.

The README.rst file MUST include instructions to build the PDF (including build dependencies for at least Debian-like systems).

The typographical conventions used by a LaTeX-formatted ZIP SHOULD be consistent, as far as possible, with those used in the *bitzec protocol specification*. It is desirable, but not strictly necessary, that the macros used in the protocol specification also be used in LaTeX-formatted ZIPs. This facilitates editing accepted proposals into the main specification.

4.14.11 ZIP Header preamble

Each ZIP must begin with an RFC 822 style header preamble. The headers must appear in the following order. Headers marked with "*" are optional and are described below. All other headers are required.

```
ZIP:<ZIP number>
Title:<ZIP title>
Author:<listof authors'real names and optionally, email addrs>
Discussions-To:<email address>
Status:<Draft|Active|Accepted|Deferred|Rejected|
        Withdrawn|Final|Superseded>
Type:<Standards Track|Informational|Process>
Created:<date created on,  in ISO8601(yyyy-mm-dd)format>
Post-History:<dates of postings to bitzec mailinglist>
Replaces:<ZIP number>
Superseded-By:<ZIP number>
Resolution:<url>
```

The Author header lists the names, and optionally the email addresses of all the authors/owners of the ZIP. The format of the Author header value must be

```
Random J.User<address  @dom.ain>
```

If the email address is included, and just

```
Random J.User
```

if the address is not given.

If there are multiple authors, each should be on a separate line following RFC 2822 continuation line conventions.

Note: The Resolution header is required for Standards Track ZIPs only. It contains a URL that should point to an email message or other web resource where the pronouncement about the ZIP is made.

While a ZIP is in private discussions (usually during the initial Draft phase), a Discussions-To header will indicate the mailing list or URL where the ZIP is being discussed. No Discussions-To header is necessary if the ZIP is being discussed privately with the author, or on the bitcoin email mailing lists.

The Type header specifies the type of ZIP: Standards Track, Informational, or Process.

The Created header records the date that the ZIP was assigned a number, while Post-History is used to record the dates of when new versions of the ZIP are posted to bitzec mailing lists. Both headers should be in yyyy-mm-dd format, e.g. 2001-08-14.

ZIPs may have a Requires header, indicating the ZIP numbers that this ZIP depends on.

ZIPs may also have a Superseded-By header indicating that a ZIP has been rendered obsolete by a later document; the value is the number of the ZIP that replaces the current document. The newer ZIP must have a Replaces header containing the number of the ZIP that it rendered obsolete.

Auxiliary Files

ZIPs may include auxiliary files such as diagrams. Image files should be included in a subdirectory for that ZIP. Auxiliary files must be named ZIP-XXXX-Y.ext, where “XXXX” is the ZIP number, “Y” is a serial number (starting at 1), and “ext” is replaced by the actual file extension (e.g. “png”).

Transferring ZIP Ownership

It occasionally becomes necessary to transfer ownership of ZIPs to a new champion. In general, we’d like to retain the original author as a co-author of the transferred ZIP, but that’s really up to the original author. A good reason to transfer ownership is because the original author no longer has the time or interest in updating it or following through with the ZIP process, or has fallen off the face of the ‘net (i.e. is unreachable or not responding to email). A bad reason to transfer ownership is because you don’t agree with the direction of the ZIP. We try to build consensus around a ZIP, but if that’s not possible, you can always submit a competing ZIP.

If you are interested in assuming ownership of a ZIP, send a message asking to take over, addressed to both the original author and the ZIP editor. If the original author doesn’t respond to email in a timely manner, the ZIP editor will make a unilateral decision (it’s not like such decisions can’t be reversed :).

ZIP Editor Responsibilities & Workflow

For each new ZIP that comes in an editor does the following:

- Read the ZIP to check if it is ready: sound and complete. The ideas must make technical sense, even if they don’t seem likely to be accepted.
- The title should accurately describe the content.
- Edit the ZIP for language (spelling, grammar, sentence structure, etc.), markup, code style (examples should match ZIP 8 & 7).

If the ZIP isn’t ready, the editor will send it back to the author for revision, with specific instructions.

Once the ZIP is ready for the repository it should be submitted as a “pull request” to the [https://github.com/bitzec/ZIPs bitzec/ZIPs] repository on GitHub where it may get further feedback.

The ZIP Editors will:

- Assign a ZIP number (almost always just the next available number, but sometimes it’s a special/joke number, like 666 or 3141) in the pull request comments.
- Merge the pull request when the author is ready (allowing some time for further peer review).
- List the ZIP in [[README.mediawiki]]
- Send email back to the ZIP author with next steps (post to bitzec-dev mailing list).

The ZIP editors are intended to fulfill administrative and editorial responsibilities. The ZIP editors monitor ZIP changes, and correct any structure, grammar, spelling, or markup mistakes we see.

History

This document is derived heavily from Bitcoin's BIP 1, authored by Amir Taaki, which in turn was derived from Python's PEP-0001. In many places text was simply copied and modified. The authors of PEP-0001 (Barry Warsaw, Jeremy Hylton, and David Goodger) and BIP 1 (Amir Taaki) are not responsible for any use of their text or ideas in the bitzec Improvement Process. The *I2P Proposal Process* and the RFC Process also influenced this document.

Please direct all comments to the ZIP Editors by email to <zips@z.cash> or by filing an issue in the *ZIPs issue tracker*.

4.14.12 ZIP List

Number	Title	Authors	Category
zip0143	Transaction Signature Verification for Overwinter	Jack Grigg <jack@z.cash> Daira Hopwood <daira@z.cash>	Consensus
zip0200	Network Upgrade Mechanism	Jack Grigg <jack@z.cash>	Consensus
zip0201	Network Peer Management for Overwinter	Simon Liu <simon@z.cash>	Network
zip0202	Version 3 Transaction Format for Overwinter	Simon Liu <simon@z.cash>	Consensus
zip0203	Transaction Expiry	Jay Graber <jay@z.cash>	Consensus
zip0243	Transaction Signature Verification for Sapling	Jack Grigg <jack@z.cash> Daira Hopwood <daira@z.cash>	Consensus

Important: Please seeZIPs, for a full current list.

4.14.13 License

The contents of the ZIPs repository are released under the terms of the MIT license. See *License* for more information or seehttp://opensource.org/licenses/MIT.

4.15 Network Upgrade Developer Guide

We recommend all wallets, exchanges, and clients that accept/support bitzec to follow these guidelines to prepare for the upcoming network upgrade. Network upgrades on a bi-annual basis to maintain the bitzec network.

Below is general advice that applies to all network upgrades:

Keep your bitzecd node updated Check that you are running the latest stable version ofbitzecd

Version verifiability Clearly state the version of bitzec in a place users can find it. Somewhere inside the client's user interface, state the protocol name and version number (available from the getblockchaininfo method). This allows users to check what version of bitzec their client is running.

Pre-upgrade notification Inform users that a network upgrade is happening before it happens. 4000 blocks (approximately a week) in advance, tell users a network upgrade is happening soon, and that transactions will be unavailable for about an hour at the activation block height.

Defensive transition Disable the initiation of new transactions starting 24 blocks (approximately one hour) before the activation block-height. If a user sends a transaction right before the upgrade, it is likely to not make it onto the chain. This can cause user confusion and frustration.

Post-upgrade notification Tell users when the upgrade has finished and re-enable initiation of transactions. Notify users with a message or at their next login after the network transition.

4.15.1 Sapling

Sapling is a network upgrade that introduces significant efficiency improvements for shielded transactions that will pave the way for broad mobile, exchange and vendor adoption of bitzec shielded addresses.

Transaction formatting

> All transactions must use the new transaction format from Sapling onwards. Make sure that you can parse these *v4* transactions. Previous formats will not be valid after the Sapling upgrade, so if you create transactions, the *v4* format must be used after the upgrade has activated (but not until then). Hardware wallets and SPV clients are particularly affected here.

SeeZIP 243. Test vectors for ZIP 243 have been pushed and are being reviewed.

Shielded HD Wallets All Sapling addresses will use hierarchical deterministic key generation according toZIP 32 (`keypath m/32'/133'/k' on mainnet`). Transparent and Sprout addresses will still use traditional key generation.

SeeZIP 32.

Also seeSapling Protocol Specification.

General Guidelines

Using bitzecd

unmodified

If you use the RPC as provided in the bitzecd client, which is true for *most* exchanges and general users of bitzec, you must update your bitzecd node to at least version 1.0.1.

For an updated list of specific parameter changes for Sapling in the bitzecd wallet RPC, please see: `Sapling RPC Updates v1.0.1` (PDF).

Additionally, Sapling introduces new parameters which must be downloaded by running the `fetch-params.sh` script. These new parameters are placed in the same directory as the older Sprout parameters.

Using custom code to create/sign/send transactions

If you manually create transactions, the following changes are *critical*. Reference section 7.1 of theSapling specificationfor complete details:

- The transactions version number **MUST** be 4.
- The version group ID **MUST** be 0x892F2085.
- At least one of tx_in_count, nShieldedSpend, and nJoinSplit **MUST** be nonzero.
- If version 4 and nShieldedSpend + nShieldedOutput > 0 then:
  - Let bvk and SigHash be as defined in §4.12 **'Balance and Binding Signature (Sapling)'**;
  - bindingSig **MUST** represent a valid signature under the *transaction binding verification key* bvk of SigHash - i.e. BindingSig.Verify$_{\text{bvk}}$(SigHash, bindingSig) = 1.
- If version 4 and nShieldedSpend + nShieldedOutput = 0, then valueBalance **MUST** be 0.
- A coinbase transaction **MUST NOT** have any *JoinSplit descriptions, Spend description, or Output descriptions*.
- valueBalance **MUST** be in the range {-MAX_MONEY .. MAX_MONEY}.

In addition, consensus rules associated with each JoinSplit description (§7.2 **'Encoding of JoinSplit Descriptions'**) each Spend description (§7.3 **'Encoding of Spend Descriptions'**) and each Output description (§7.4 **'Encoding of Output Descriptions'**) **MUST** be followed.

Mining Pools

Mining pools running the Stratum protocol will have to make some changes as well.

The `hashReserved` field in the Stratum Protocol will have to be replaced by the `hashFinalSaplingRoot` field from the block header (§7.5 **'Block Header'**).

Testing

Sapling is currently activated on testnet. To test transactions you'll want to follow the testnet_guide. Alternatively, developers can use these features in regtest mode.

4.15.2 Overwinter

Overwinter is the first network upgrade for bitzec. Its purpose is strengthening the protocol for future network upgrades. It includes versioning, replay protection for network upgrades, performance improvements for transparent transactions, a new feature of transaction expiry, and more.

Overwinter activated successfully at block `347500`, mined at `June 25, 2018 20:42 UTC-04:00`

Transaction formatting All transactions must use the new transaction format from Overwinter and onwards. Make sure that you can parse these "v3" transactions (write a parser for them if you aren't using our code). Previous formats will not be valid after the Overwinter upgrade, so if you create transactions, the "v3" format must be used after the upgrade has activated (but not until then). Hardware wallets and SPV clients are particularly affected here. See ZIPs202and203.

Transaction version number The 4-byte transaction version will have its most significant bit set from Overwinter and onwards, for two-way replay protection of Overwinter and unambiguous transaction parsing of all current and future formats. For example, existing "v1" and "v2" transactions use version numbers "1" and "2", but "v3"

Overwinter transactions will use the unsigned version number "(1 << 31) | 3" in the transaction serialization format. See ZIP202.

Version group IDs A transaction version will be uniquely paired with a version group ID to ensure unambiguous transaction parsing. For example, a "v3" transaction will always have the version group ID "0x03C48270" in its serialization format, even after future network upgrades. See ZIP202.

Branch IDs Each network upgrade has an associated branch ID that identifies its consensus rules. For two-way replay protection, creating transactions will require the branch ID of the current chain tip when signing a transaction (in the BLAKE2b personalization field.) You can obtain the branch ID of any block height from the getblock API. See ZIP200.

Signature hashing There are new SegWit-like features in this upgrade, such as transaction signatures committing to values of the inputs. We suggest reusing code from SegWit (e.g. for hashing transparent outputs) when implementing the new SignatureHash function. See ZIP143.

Transaction expiry We recommend that you do use the default expiry height (20 blocks/~1 hours) and follow these UX guidelines so that bitzec users can develop a consistent expectation of when bitzec transactions expire and what happens. Zee ZIP203.

This isn't an exhaustive list of the changes. Look at the Overwinter bitzec Improvement Proposals (ZIPs) below for complete details on the changes that will be made. The five ZIPs cover network handshaking, transaction format, transaction expiry, signature hashing, and network upgrade mechanisms.

- `ZIP 143` Transaction Signature Verification for Overwinter
- `ZIP 200` Network Upgrade Mechanism
- `ZIP 201` Network Peer Management for Overwinter
- `ZIP 202` Version 3 Transaction Format for Overwinter
- `ZIP 203` Transaction Expiry

The network upgrade is coordinated via an on-chain activation mechanism.

bitzecd v1.1.0 (and future releases) running protocol version `170005` will activate Overwinter at block 347500 at which point only v3 transactions are processed. Older versions of bitzecd <= 1.0.14, running protocol versions <= 170004, will partition themselves away from the main network into a legacy chain.

Wipeout protection is provided by the new transaction format and signature hashing scheme. Blocks from the legacy chain will not be accepted by the upgraded network. That is, the upgraded network is permanent, and bitzecd v1.1.0 (and future releases) can not reorganize back to the older non-upgraded chain.

Common Issues

tx-overwinter-active This error is simply saying that Overwinter has been activated and your client must be upgraded to the latest version. Upgrade your client and try again. If the issue persists try restarting the client. If this error is appearing on a third party app like a mobile wallet, please file a support request with the developer of the product and let us know in the#user-supportchannel on the community chat -https://chat.bitzeccommunity.com/

mandatory-script-verify-flag-failed (Script evaluated without error but finished with a false/empty top stack element)

This error has been most commonly seen when using *sendrawtransaction*. This can be caused by a few things.

1. When creating raw transactions, the *signrawtransaction* step must be completed correctly. There is a field in *signrawtransaction* called *prevtxs* which can be seen here (https://bitzec-rpc.github.io/signrawtransaction.html). The *prevtxs* parameter is optional, but if it is specified, the *amount* parameter must also be specified. This amount is the total amount of the previous output. Prior to

Overwinter the *amount* parameter was not required, this is a change between Overwinter and the previous version.

2. This issue can also arise in an edge case where a user is signing the transaction from an offline node. If this is the case the offline node must be synced to above the Overwinter activation height, block `347500`.

Node sync is stuck before Overwinter activation height This bug occurs when you are starting a fresh node or restarting a node that is not synced to above the Overwinter activation height (block 347500) and causes the node to sync very slowly. The bug has to do with your node incorrectly banning peer nodes. The end result is your node will sync very slowly as it will not be able to maintain as many connections to other nodes as usual.

This issue has been fixed in 2.0.0. Pleaseupdate your clientto 2.0.0 or above.

4.16 bitzec Rust Architecture

4.16.1 Current Design

zkcrypto/pairing

- Implements BLS12-381

zkcrypto/bellman

- Implements Groth16, Circuit API

bitzec-hackworks/sapling-crypto

- Implements Sapling/Sprout circuits on top of bellman
- Implements Jubjub
- Implements some Sapling primitives necessary for testing

bitzec/librustbitzec

- "Thin" FFI surrounding our crypto, for bitzecd

4.16.2 Current Issues

- We're doing lots of refactorings and improvements to the code, but these will span many different crates until we get to a stable point. Hard to review and coordinate.
  - Example: bellman is going to be a "circuit-only" thing, agnostic to the proving system. groth16 crate will handle groth16.
  - Example: hardware wallets only want/need jubjub and sapling primitives, so we need to pull out zk-SNARK stuff (which requires an allocator, standard library, etc.)
- Code is inconsistent (with naming, as far as we know) with specification
- Nothing is labeled as constant/variable time

4.16.3 New Design

zkcrypto/jubjub

- Implements Fp, Jubjub
- No standard library requirement

zkcrypto/bls12-381 (depends on jubjub)

- Implements BLS12-381, serial FFT

zkcrypto/bellman

- Implements common circuit synthesis API, gadgets

zkcrypto/groth16

- Implements groth16

4.16.4 Strategy

- librustbitzec repository is a Rust workspace containing all of our dependencies, for the time being, via git subtrees
- We refactor code and integrate test vectors closely, following stringent code review processes and quality policies
- Later, we break the subtrees out into crates with stable APIs

4.16.5 End Goal

- Complete cleanup of code (match spec, best practices)
- More members of the team learn how all this stuff works, good documentation
- Refactor of code into modular pieces that all relate to each other nicely
- no_std support for hardware wallets and other projects
- In the meantime, everything is CI'd and developed together
- The coolest, most awesome crypto codebase written in Rust anywhere in the world

4.17 Wallet Developer UX Checklist

We have compiled a checklist of good practices that can be applied to any cryptocurrency wallet with a graphic user interface. `Download the UX Checklist PDF`.

4.17.1 bitzec Features

If you're building a wallet that supports bitzec, we encourage that you follow these guidelines.

Addresses

Use an address persistently for each use Use an address like how you would use different bank accounts (spending, saving, business spending). Transparent addresses aren't private, so we urge users to keep this in mind. We discourage using a new transparent address for each transaction; this only provides afalse sense of privacy . Shielded addresses maintain the privacy of transactions, there is no added benefit of using a new shielded address per transaction.

Indicate that transparent addresses are not encrypted! A transaction involving a transparent address (either as sender or recipient) posts the details of the transparent address and amount publicly on the blockchain.

Indicate that shielded addresses are encrypted! A shielded transaction, where funds are sent from one shielded address to another shielded address, only reveals a transaction legitimately and safely happened. The sender, receiver, and amount are not revealed on the blockchain.

If there is no zaddr support, state so clearly Most wallets throw an error stating, "invalid address" or "invalid input." We suggest instead saying, "shielded address are not supported." to indicate that shielded addresses are still valid addresses.

Provide taddr and zaddr support if possible Shielded addresses provide privacy via encryption and is bitzec's main feature. However, most mobile and desktop wallets don't support sending to shielded addresses, so some users will likely be unable to send bzc to a shielded address.

Warn users when sending from zaddrs to taddrs (deshielding transactions) Explicitly tell users that they are about to reveal transaction information. We don't think warnings other types of transactions are necessary.

Show an available balance vs owned balance Show two balances, one which includes unconfirmed finds, and another not including unconfirmed funds, i.e. "Balance: 621.14321 bzc (605.35620 bzc spendable)."

Transactions

Clearly state the fee structure The default transaction fee is set at 0.0001 bzc. We encourage that you use the default fee, so that transparent and shielded transactions have the same fee–that way, privacy doesn't costmore for users!

Disable users from setting their own transaction fees Do not allow users to customize fees. Our network is fast enough that mining incentivization is not an issue. Unique transaction fees can cause linkability within transactions, especially for zaddrs.

Do not differentiate between types of transactions We do not currently distinguish between different types of bitzec transactions: transparent (transparent to transparent), shielding (transparent to shielded), deshielding (shielded to transparent), and shielded (shielded to shielded) transactions. This is complicated by the ability to send to/from a combination of shielded and transparent addresses.

Use the default transaction expiry time Transaction expiry (see ZIPhere) is set to 20 blocks by default, which is ~1 hour. Use this default global runtime option so bitzec users can develop a consistent expectation of when bitzec transactions expire. We don't support expiry time as a per-transaction runtime option.

Visibly mark newly sent transactions in a "pending" state We suggest having a "pending transactions" or"unconfirmed transactions" section, but you can also distinguish it in the list of chronological transactions by using a color or an icon.

Tell the user the expected remaining time to expiry Users should be able to see how much time/blocks areremaining until their transaction expires. Once confirmed (10+ confirmations), unmark the sent transaction visibly in a "complete" state.

If expired, visibly mark the transaction expired and notify the user Rather than deleting the attempted transaction, keep the expired transaction in the log, but distinguished as such. We also encourage giving users suggestions ontroubleshooting their transaction.

Viewing Keys

Use viewing keys for watch only wallets Share aviewing keywith yourself to create a wallet that tracks your funds while keeping your main funds offline. Watch-only wallets are the first application of viewing keys; we exploring additional use cases as well.

Secure communication channel Encourage secure communication channels by supporting one; viewing keys should not be copy and pasted into a text or email.

Indicate that viewing keys are for all incoming transactions: At version 1.0.14, a viewing key allows the holder of the key to see all incoming transactions since the zaddr was created, but not outgoing transactions.

Memo Fields

Show the memo field in the UI Even if thememo fieldis empty, show that the field is empty rather than removing it from the UI. This is a good nudge to remind users that a memo field exists.

Liberal Use The memo field is always present and is always exactly 512 bytes long; this is necessary for privacy so that an observer can't detect the different usage patterns and memos. This means that the cost is baked in so that you don't pay a higher fee for including a memo, so encourage users to use the memo.

4.17.2 General

We encourage that you use this checklist to catch common usability problems before launch or user testing.

User Interaction

Progressive Disclosure Things should progress from simple to complex; only the necessary or requested information is displayed at a given time. This helps manage complexity without becoming confused, frustrated, or disoriented.

Feedback Every action should produce a visible, understandable, and immediate reaction. Failing to acknowledge an interaction can lead to unnecessary repetition of actions or errors (i.e. clicking "send" multiple times).

Priming Tell people what they can expect and what they should do. For example,explainingthat a camera is needed to scan QR codes before you ask for camera permissions is likely to have users who want that feature to accept it.

Communication Be context-aware of what the user is doing and the nature of the message. For instance, notify of events like transaction confirmations with push notifications, since they're probably not waiting on the app for the confirmation.

Error Handling The best way to handle errors is to prevent them. But if one occurs, put next to the relevant input field (not just at the top or bottom of the screen) to show users what they need to fix without searching for it. It should describe what happened, why it happened, suggest a fix, and not blame the user.

User interface

Hierarchy Information is presented in order of importance and the visual hierarchy of actions on a screen matches what the user expects to do first, second, third, etc.

Simplification Limit the choices that a user is presented with per screen. Provide appropriate filters if there is a large data set.

Consistency Components with a similar behavior should have a similar appearance. For example, all buttons that send a transaction should be blue, square, and labeled ‘send.’

Predictability Set good expectations. From looking at your interface, with no previous use, users should be able to answer things such as “where am I?,” “what can I do here?,” “where can I go from here?,” and “what does this button do?.”

Visibility Discoverability shouldn’t involve luck or chance. If a page requires scrolling, indicate that more content is below the screen by showing half of an image. If there are some screens you want users to find, the menu that links to those pages persists everywhere.

Content

Market Information Provide an up-to-date crypto to FIAT currency conversion, along with current exchange rates between cryptocurrencies.

Network Information Tell users if their transaction is likely to be processed quickly or not, based on mempool congestion.

Account Information Show the balance, minimum spendable, maximum spendable, and other account-specific information.

Fee Information Showhow much the fee is, what % of the transaction it is, and if it’s added on top or included.

Simplify Jargon Translate what a concept or event affects the user, rather than exposing or explaining what it is technically. For instance, say if the transaction has been confirmed or not, instead showing the number of confirmations or how many confirmations is considered safe.

Navigation

Persistence The navigation bar shouldalways be visibleon every screen. If it isn’t, users don’t know what to do next or don’t know how to do the next thing.

Uniformity Similarly styled navigational elements should behave similarly. Additionally, elements of navigation should never appear and disappear, rearrange in order, or move to a different location.

Method Choose the method that most easily lets the users find what they want. This is specific to the use case. Method include browsing via a navigation system, searching with keywords, or filtering to narrow large lists.

Sorting Alphabetical sorting is avoided unless necessitated by many navigational choices (7+). Sort by relevance, related groups, or anything else instead.

Labeling Use meaningful labels and icons for navigation menu items, links, and buttons. Don’t force people to chase information they need.

Visual Design

Alignment Every element in the UI should be aligned with one or more other elements. Alignment provides cognitive stability and creates visual relationships. In this same vein,left-align large blocks of textas users need to expend more energy to track the lines. Eyes fatigue faster, comprehension slows, but the users may not be aware why.

Proximity Group certain elements (navigation, header, articles, etc.) contextually to form a perceived whole. For the same reason, visually separate unrelated items.

Repetition Use repetition tocreate a hierarchy of visual styles. This principle applies to fonts but also colors, textures, and graphical elements. (For instance, all titles should be of one size, all buttons are square, all colors are in a color palette, etc.) Reusing elements of visual styles in visual elements creates cohesiveness.

Contrast Text iseasily readablewhen stark, complementary colors are used A lack of contrast between text and background strains the eyes because they don't know which color to focus on.

4.18 Contributor Code of Conduct

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Other unethical or unprofessional conduct

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting a project maintainer (see below). All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. Maintainers are obligated to maintain confidentiality with regard to the reporter of an incident.

You may send reports toour Conduct email.

If you wish to contact specific maintainers directly, the following have made themselves available for conduct issues:

- Daira Hopwood (daira at z.cash)
- Sean Bowe (sean at z.cash)

This Code of Conduct is adapted from theContributor Covenant, version 1.3.0, available athttps://www.contributor-covenant.org/version/1/3/0/