zcash Documentation

Release english-docs

Mar 04, 2018

User Documentation

CHAPTER 1

What is Zcash?

Zcash is 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. Technical details are available in our Protocol Specification.

This software is the Zcash client. It downloads and stores the entire history of Zcash transactions; depending on the speed of your computer and network connection, the synchronization process could take a day or more once the blockchain has reached a significant size.

CHAPTER 2

Security Warnings

Before using, be sure to read the privacy and security recommendations.

Advanced users may read the Advanced security warnings.

Zcash is unfinished and highly experimental. Use at your own risk.

CHAPTER 3

Deprecation Policy

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

CHAPTER 4

Where do I begin?

You can install the official Zcash client or find third-party wallets at https://z.cash/download.html

CHAPTER 5

Need Help

- Start with the FAQ
- Ask for help:
  - Zcash forum
  - Zcash Community chat

Participation in the Zcash project is subject to a Code of Conduct. Note that the Zcash forum and Zcash Community chat have adopted a related set of Guidelines.

CHAPTER 6

License

For license information see the file COPYING.

6.1 Zcash 1.0 "Sprout" Guide

Translations available here.

Welcome! This guide is intended to get you running on the official Zcash network. Zcash currently has some limitations: it only officially supports Linux, requires 64-bit, and in some situations requires heavy memory and CPU consumption to create transactions.

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.

6.1.1 Upgrading?

If you're on a Debian-based distribution, you can follow the Debian instructions to install zcash on your system. Otherwise, you can update your local snapshot of our code:

```
git fetch origin
git checkout v1.0.14
./zcutil/fetch-params.sh
./zcutil/build.sh --disable-rust -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 --disable-rust`.

If you are upgrading from testnet, make sure that your `~/.zcash` directory contains only `zcash.conf` to start with, and that your `~/.zcash/zcash.conf` does not contain `testnet=1` or `addnode=testnet.z.cash`.

If the build fails, move aside your `zcash` directory and try again by following the instructions in the *Compile it yourself* section below.

6.1.2 A quick note about terminology

Zcash 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.

6.1.3 Requirements

Currently, you will need:

- Linux (easiest with a Debian-based distribution)
- 64-bit processor and OS
- 3 GB of free RAM
- at least 10 GB of free disk space (the size of the block chain increases over time)

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

https://github.com/zcash/zcash/blob/v1.0.14/doc/payment-api.md

6.1.4 Security

Before installing, upgrading, or running zcash, ensure you have checked for any security issues. Please See our Security page:

https://z.cash/support/security.html

6.1.5 Get started

Binary packages for Debian-based operating systems

Follow the instructions here: https://github.com/zcash/zcash/wiki/Debian-binary-packages

Or, Compile it yourself

Install dependencies

On Ubuntu/Debian-based systems:

```
$ 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
```

On Fedora-based systems:

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

On RHEL-based systems (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.

Check GCC version

gcc/g++ 4.9 *or later* is required. Zcash has been successfully built using gcc/g++ versions 4.9 to 7.x inclusive. Use `g++ --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
```

Check binutils version

binutils 2.22 *or later* is required. Use `as --version` to check which version you have, and upgrade if necessary.

Fetch the software and parameter files

Fetch our repository with git and run `fetch-params.sh` like so:

```
$ git clone https://github.com/zcash/zcash.git
$ cd zcash/
$ git checkout v1.0.14
$ ./zcutil/fetch-params.sh
```

This will fetch our Sprout proving and verifying keys (the final ones created in the Parameter Generation Ceremony), and place them into `~/.zcash-params/`. These keys are just under 911MB in size, so it may take some time to download them.

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

Build

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

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

This should compile our dependencies and build `zcashd`. (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 --disable-rust`.)

Testing

The tests take a while to run and may require up to 8GB of RAM. If you would rather get started right away, you can skip to the next section. If you want to run the tests to make sure Zcash is working, run:

```
$ ./qa/zcash/full_test_suite.sh
```

You can also run the RPC tests, which take much longer:

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

The tests need a lot of memory to run successfully. An out-of-memory error will usually cause a FAIL or ERROR outcome with "std::bad_alloc" somewhere in the output.

6.1.6 Configuration

Create the `~/.zcash` directory and place a configuration file at `~/.zcash/zcash.conf` using the following commands:

```
mkdir -p ~/.zcash
echo "addnode=mainnet.z.cash" >~/.zcash/zcash.conf
echo "rpcuser=username" >>~/.zcash/zcash.conf
echo "rpcpassword=`head -c 32 /dev/urandom | base64`" >>~/.zcash/zcash.conf
```

Note that this will overwrite any `zcash.conf` settings you may have added from testnet. (If you want to run on testnet, you can retain a `zcash.conf` from testnet.) The commands above also assign a random password to avoid potential security issues with access to the RPC interface.

If you wish to run zcashd on testnet, change the lines in zcash.conf indicating the network and node discovery: `testnet=1` instead of `mainnet=1` and `addnode=testnet.z.cash` instead of `addnode=mainnet.z.cash`.

Enabling CPU mining:

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

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

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' >> ~/.zcash/zcash.conf
```

Note, you probably want to read the [[Mining-Guide]] to learn more mining details.

6.1.7 Running Zcash:

Now, run zcashd!

```
$ ./src/zcashd
```

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

You should be able to use the RPC after it finishes loading. Here's a quick way to test:

```
$ ./src/zcash-cli getinfo
```

NOTE: If you are familiar with bitcoind's RPC interface, you can use many of those calls to send ZEC 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.

NOTE: The main network node at mainnet.z.cash is also accessible via Tor hidden service at zcmaintvsivr7pcn.onion.

To see the peers you are connected to:

```
$ ./src/zcash-cli getpeerinfo
```

6.1.8 Using Zcash

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

Generating a t-addr

Let's generate a t-addr first.

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

Listing transparent addresses

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

This should show the address that was just created.

Receiving Zcash with a z-addr

Now let's generate a z-addr.

```
$ ./src/zcash-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
↪'
```

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

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

You should see something like:

```
[

↪"zcA6qngiR3U7HxYopyTWkaDLwYBd83D5MT7Jb9gpgTzPLMZytzRbtdPP1Syv4RvRgHeoZrJWSask3DyfwXG9DGPMWMvX7aC
↪",

↪"zcBqWB8VDjVER7uLKb4oHp2v54v2a1jKd9o4FY7mdgQ3gDfG8MiZLvdQga8JK3t58yjXGjQHzMzkGUxSguSs6ZzqpgTNiZG
↪"
]
```

Great! Now, send your z-addr to the sender. You should eventually see their transaction when checking:

```
$ ./src/zcash-cli z_listreceivedbyaddress "$ZADDR"
```

```
[
    {
        "txid" : "af1665b317abe538148114a45322f28151925501c081949cc7a5207ef21cb750",
        "amount" : 1.23,
        "memo" :
↪"48656c6c6f20ceb22100000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"
    }
]
```

Sending coins with your z-addr

If someone gives you their z-addr…

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

You can send 0.8 ZEC by doing…

```
$ ./src/zcash-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/zcash-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 zcash-cli

As Zcash is an extension of bitcoin, zcash-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 https://github.com/zcash/zcash/blob/master/doc/payment-api.md

To list all zcash commands, use `./src/zcash-cli help`.

To get help with a particular command, use `./src/zcash-cli help <command>`.

6.1.9 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:

https://github.com/zcash/zcash/blob/master/doc/security-warnings.md

Please also see our security page for recent notifications and other resources:

https://z.cash/support/security.html

6.2 Zcash Payment API

6.2.1 Overview

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

Zcash payments make use of two address formats:

- taddr - an address for transparent funds (just like a Bitcoin address, value stored in UTXOs)
- zaddr - an 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 Zcash RPC calls.

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

6.2.2 Compatibility with Bitcoin Core

Zcash supports all commands in the Bitcoin Core API (as of version 0.11.2). Where applicable, Zcash 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.

6.2.3 List of Zcash API commands

Optional parameters are denoted in [square brackets].

RPC calls by category:

- Accounting: z_getbalance, z_gettotalbalance
- Addresses : z_getnewaddress, z_listaddresses, z_validateaddress
- Keys : z_exportkey, z_importkey, z_exportwallet, z_importwallet
- Operation: z_getoperationresult, z_getoperationstatus, z_listoperationids
- Payment : z_listreceivedbyaddress, z_sendmany, z_shieldcoinbase

RPC parameter conventions:

- taddr : Transparent address
- zaddr : Private address
- address : Accepts both private and transparent addresses.
- amount : JSON format double-precision number with 1 ZC 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 | Parameters | Description — | — | — z_getbalance| address [minconf=1] | Returns the balance of a taddr or zaddr belonging to the node's wallet.Optionally set the minimum number of confirmations a private or transaction transaction must have in order to be included in the balance. Use 0 to count unconfirmed transactions. z_gettotalbalance| [minconf=1] | 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.Output:{"transparent" : 1.23,"private" : 4.56,"total" : 5.79}

Addresses

Command | Parameters | Description — | — | — z_getnewaddress | | Return a new zaddr for sending and receiving payments. The spending key for this zaddr will be added to the node's wallet.Output:zN68D8hSs3… z_listaddresses | | Returns a list of all the zaddrs in this node's wallet for which you have a spending key.Output:{ ["z123…", "z456…", "z789…"] } z_validateaddress | zaddr | Return information about a given zaddr.Output:{"isvalid" : true,"address" : "zcWsmq…","payingkey" : "f5bb3c…","transmissionkey" : "7a58c7…","ismine" : true}

Key Management

Command | Parameters | Description — | — | — z_exportkey | zaddr | *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 Zcash protocol spec.Output:AKWUAkypwQjhZ6LLNaMuuuLcmZ6gt5UFyo8m3jGutvALmwZKLdR5 z_importkey | zkey [rescan=true] | *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 Zcash protocol spec.Set rescan to true (the default) to rescan the entire local block database for transactions affecting any address or pubkey script in the wallet (including transactions affecting the newly-added address for this spending key). z_exportwallet | filename | *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. z_importwallet | filename | *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 zcashd's working directory.No value is returned but a JSON-RPC error will be reported if a failure occurred.

Payment

Command | Parameters | Description — | — | — z_listreceivedbyaddress | zaddr [minconf=1] | 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.Output:[{"txid": "4a0f…","amount": 0.54,"memo":"F0FF…",}, {…}, {…}] z_sendmany | fromaddress amounts [minconf=1] [fee=0.0001] | *This is an Asynchronous RPC call*Send funds from an address to multiple outputs. The address can be either 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 attaching a memo in hexadecimal format.**NOTE:**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 rule may be removed.Example of Outputs parameter:[{"address":"t123…", "amount":0.005},,{"address":"z010…","amount":0.03, "memo":"f508af…"}]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 be greater than zero.Optionally set a transaction fee, which by default is 0.0001 ZEC.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. z_shieldcoinbase | fromaddress toaddress [fee=0.0001] [limit=50] | *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 100000 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 trasaction and what remains to be shielded.

Operations

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

- operationid : 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.
- status : current status of operation
  - queued : operation is pending execution
  - executing : operation is currently being executed
  - cancelled
  - failed.
  - success
- result : result object if the status is 'success'. The exact form of the result object is dependent on the call itself.
- error: error object if the status is 'failed'. The error object has the following key-value pairs:
  - code : number

– message: error message

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 | Parameters | Description — | — | — z_getoperationresult | [operationids] | 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".[{"operationid": "opid-11ee…","status": "cancelled"},{"operationid": "opid-9876", "status": "failed"},{"operationid": "opid-0e0e","status":"success","execution_time":"25","result": {"txid":"af3887654…",…}},] Examples:zcash-cli z_getoperationresult '["opid-8120fa20-5ee7-4587-957b-f2579c2d882b"]' zcash-cli z_getoperationresult z_getoperationstatus | [operationids] | 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.[{"operationid": "opid-12ee…","status": "queued"},{"operationid": "opd-098a…", "status": "executing"},{"operationid": "opid-9876", "status": "failed"}]When the operation succeeds, the status object will also include the result.{"operationid": "opid-0e0e","status":"success","execution_time":"25","result": {"txid":"af3887654…",…}} z_listoperationids | [state] | 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 are 'queued', 'executing', 'success', 'failed', 'cancelled'.["opid-0e0e…", "opid-1af4…", …]

6.2.4 Asynchronous RPC call Error Codes

Zcash error codes are defined in https://github.com/zcash/zcash/blob/master/src/rpcprotocol.h

z_sendmany error codes

RPC_INVALID_PARAMETER (-8) | *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 (-5) | *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 (-6) | *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 (-4) | *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 commitement. "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 (-16) | *Failed to encrypt the wallet* ——————————————————————-| ——————————————- "Failed to sign transaction" | Transaction was not signed, sign transaction and retry.

RPC_WALLET_KEYPOOL_RAN_OUT (-12) | *Keypool ran out, call keypoolrefill first* ——————————————————————-| ———————————————— "Could not generate a taddr to use as a change address" | Call keypoolrefill and retry.

6.3 RPC Documentation

6.3.1 AddMultiSigAddress

Requires wallet support.

The `addmultisigaddress` RPC adds a P2SH multisig transparent address to the wallet. *Multisig is not supported for shielded addresses.*

Parameter #1—the number of signatures required

Name	Type	Presence	Description
Required	number (int)	Required (exactly 1)	The minimum (*m*) number of signatures required to spend this m-of-n multisig script

Parameter #2—the full public keys, or addresses for known public keys

Name	Type	Presence	Description
Keys or Addresses	array	Required (exactly 1)	An array of strings with each string being a transparent address or associated public key
→ Key Or Address	string	Required (1 or more)	A transparent address or public key against which signatures will be checked. There must be at least as many keys as specified by the Required parameter, and there may be more keys

Parameter #3—the account name

DEPRECATED. If provided, MUST be set to the empty string "" to represent the default account. Passing any other string will result in an error.

Result—a P2SH address printed and stored in the wallet

Name	Type	Presence	Description
`result`	string (base58)	Required (exactly 1)	The P2SH multisig address. The address will also be added to the wallet, and outputs paying that address will be tracked by the wallet

Example

Add a multisig address from 2 of 3 addresses:

```
zcash-cli addmultisigaddress 2 "[\"t14sSauSf5pF2UkUwvKGq4qjNRzBZYqgEL5\",\
↪"t1G1sgjn4YtPu27adkKGrdDwzRTxnRkBfKV\",\"t1PoHp2v54vfmdgQ3v3SNuQga8JKHTNi2a1\"]"
```

Result:

```
t3yVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq
```

(New P2SH multisig address also stored in wallet.)

See also

- *CreateMultiSig*: creates a P2SH multi-signature transparent address. *Multisig is not supported for shielded addresses.*
- *DecodeScript*: decodes a hex-encoded P2SH redeem script.
- **'Pay-To-Script-Hash (P2SH)'_** </en/glossary/p2sh-address>

6.3.2 AddNode

The `addnode` RPC attempts to add or remove a node from the addnode list, or to try a connection to a node once.

Parameter #1—hostname/IP address and port of node to add or remove

Name	Type	Presence	Description
Node	string	Required (exactly 1)	The node to add as a string in the form of `<IP address>:<port>`. The IP address may be a hostname resolvable through DNS, an IPv4 address, an IPv4-as-IPv6 address, or an IPv6 address

Parameter #2—whether to add or remove the node, or to try only once to connect

Name	Type	Presence	Description
Command	string	Required (exactly 1)	What to do with the IP address above. Options are: • `add` to add a node to the addnode list. Up to 8 nodes can be added additional to the default 8 nodes. Not limited by `-maxconnections` • `remove` to remove a node from the list. If currently connected, this will disconnect immediately • `onetry` to immediately attempt connection to the node even if the outgoing connection slots are full; this will only attempt the connection once

Result—'null' plus error on failed remove

Name	Type	Presence	Description
`result`	null	Required (exactly 1)	Always JSON `null` whether the node was added, removed, tried-and-connected, or tried-and-not-connected. The JSON-RPC error field will be set only if you try removing a node that is not on the addnodes list

Example

Connect to node *192.168.0.6*, trying once:

```
zcash-cli addnode "192.168.0.6:8233" "onetry"
```

Result (no output from *zcash-cli* because result is set to *null*).

See also

- **'GetAddedNodeInfo'_**: **|summary_getAddedNodeInfo|**

6.3.3 BackupWallet

Requires wallet support.

The `backupwallet` RPC safely copies *wallet.dat* to the specified alphanumeric filename only after setting `-exportdir=`.

Parameter #1—destination filename

Name	Type	Presence	Description
Filename	string (alphanumeric)	Required (exactly 1)	A filename to be created or overwritten within the directory specified in `-exportdir=`.

Result—the full path of the destination file

Name	Type	Presence	Description
`result`	string (full path of destination file)	Required (exactly 1)	Returns the full path of destination file or error message. The JSON-RPC error and message fields will be set if a failure occurred.

Example

Backup wallet after setting `-exportdir=/home/user/zcash-backup/`:

```
zcash-cli backupwallet backup
```

Result:

```
/home/user/zcash-backup/backup
```

See also

- *DumpWallet*: creates or overwrites a file with all of the wallet's transparent keys in a human-readable format to the specified alphanumeric filename only after setting `-exportdir=`.
- **'ImportWallet'_: |summary_importWallet|**

6.3.4 ClearBanned

The `clearbanned` RPC clears list of banned nodes.

Parameters: none

Result—'null' on success

Name	Type	Presence	Description
`result`	null	Required(exactly 1)	JSON *null* when the list was cleared

Example

Clears the ban list.

```
zcash-cli clearbanned
```

Result (no output from *zcash-cli* because result is set to *null*).

See also

- **'ListBanned'_: |summary_listBanned|**
- **'SetBan'_: |summary_setBan|**

6.3.5 CreateMultiSig

The `createmultisig` RPC creates a P2SH multi-signature transparent address. *Multisig is not supported for shielded addresses.*

Parameter #1—the number of signatures required

Name	Type	Presence	Description
Required	number (int)	Required (exactly 1)	The minimum (*m*) number of signatures required to spend this m-of-n multisig script

Parameter #2—the full public keys of transparent addresses, or transparent addresses for known public keys

Name	Type	Presence	Description
Keys or Addresses	array	Required (exactly 1)	An array of strings with each string being a transparent address or associated public key
→ Key Or Address	string	Required (1 or more)	A transparent address or public key against which signatures will be checked. There must be at least as many keys as specified by the Required parameter, and there may be more keys

Result—P2SH address and hex-encoded redeem script

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing the multisig address
`address`	string (base58)	Required (exactly 1)	The value of the new multisig address
`redeemScript`	string (hex)	Required (exactly 1)”	The string value of the hex-encoded redemption script.

Example

Creating a 2-of-3 P2SH multisig address by mixing two P2PKH addresses and one full public key (all transparent):

```
zcash-cli createmultisig 2 '''
[
  "t14sSauSf5pF2UkUwvKGq4qjNRzBZYqgEL5",
  "02f74d95ca23bba0180c1a38e972a3d3ef0f3045a47007f7ac39cf1c272bd4f770",
  "t1PoHp2v54vfmdgQ3v3SNuQga8JKHTNi2a1"
]
'''
```

Result:

```
{
  "address" : "t3yVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq",
  "redeemScript" :
↪"5221039cd63fb8cb183868b7cf71e0189e58786fcbf8dce452c1cf2ceb68ebf66d5e9b210231d344af857c98139f16427
↪"
}
```

See also

- *AddMultiSigAddress*: **|summary_addMultiSigAddress|**

- *DecodeScript*: decodes a hex-encoded P2SH redeem script.
- **'[Pay-To-Script-Hash (P2SH)'_** </en/glossary/p2sh-address>

6.3.6 CreateRawTransaction

The `createrawtransaction` RPC creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.

Parameter #1—references to previous outputs

Name	Type	Presence	Description
Inputs	array	Required (exactly 1)	An array of objects, each one being used as an input to the transaction
→ Input	object	Required (1 or more)	An object describing a particular input
→ → `txid`	string (hex)	Required (exactly 1)	The TXID of the outpoint encoded as hex in RPC byte order
→ → `vout`	number (int)	Required (exactly 1)	The output index number of the outpoint; the first output in a transaction is index `0`

Parameter #2—P2PKH or P2SH addresses and amounts

Name	Type	Presence	Description
Outpoints	object	Required (exactly 1)	The addresses and amounts to pay
→ Address/Amount	string: number (zcash)	Required (1 or more)	A key/value pair with the address to pay as a string (key) and the amount to pay that address (value) in zcash"

Result—the unsigned raw transaction in hex

Name	Type	Presence	Description
`result`	string	Required (exactly 1)"	The resulting unsigned raw transaction in serialized transaction format encoded as hex. If the transaction couldn't be generated, this will be set to JSON `null` and the JSON-RPC error field may contain an error message

Example

```
zcash-cli createrawtransaction '''
  [
    {
      "txid": "e9c77353da98e8897bf34ac589db0d3a407987d566cf8cbc19b6cf57e64fa1ad",
      "vout" : 0
    }
  ]''' '{ "t14sSauSf5pF2UkUwvKGq4qjNRzBZYqgEL5": 0.15 }'
```

Result (wrapped):

```
01000000011da9283b4ddf8d89eb996988b89ead56cecdc44041ab38bf787f1206cd90b51e0000000000ffffffff01405dc6
```

See also

- *DecodeRawTransaction*: decodes a serialized transaction hex string into a JSON object describing the transaction.
- **'SignRawTransaction'_**: **|summary_signRawTransaction|**
- **'SendRawTransaction'_**: **|summary_sendRawTransaction|**
- **'Serialized Transaction Format'_**

6.3.7 DecodeRawTransaction

The `decoderawtransaction` RPC decodes a serialized transaction hex string into a JSON object describing the transaction.

Parameter #1—serialized transaction in hex

Name	Type	Presence	Description
Serialized transaction	string (hex)	Required (exactly 1)	The transaction to decode in serialized transaction format

Result—the decoded transaction

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing the decoded transaction, or JSON `null` if the tranasction could not be decoded
Overwintered	boolean	Required (0 or 1)	The flag designating an Overwintered transaction
→ `txid`	string (hex)	Required (exactly 1)	The transaction's TXID encoded as hex in RPC byte order
→ `version`	number (int)', Required (exactly 1)	The transaction format version number	
→ `versiongroupid`	string (hex)	Optional (0 or 1)	The version group id for Overwintered transactions
→ `locktime`	number (int)"	Required (exactly 1)	The transaction's locktime: either a unix epoch date or block height
→ `expiryheight`	number (int)	Optional (0 or 1)	The last block height that the transaction will be mined for Overwinter compatible transactions
→ `vin`	array	Required (exactly 1)	An array of objects with each object being an input vector for this transaction. Input objects will have the same order within the arrray as they have in the transaction, so the first input listed will be input 0
→ → Input	object	Required (1 or more)	An object describing one of the transaction's inputs either regular or coinbase
→ → → `txid`	string	Optional (0 or 1)	The TXID of the outpoint being spent, encoded as hex in RPC byte order. Not present if this is a coinbase transaction
→ → → `vout`	number (int)	Optional (0 or 1)	The output index number (vout) of the outpoint being spent. The first output in a transaction has an index of *0*. Not present if this is a coinbase transaction
→ → → `scriptSig`	object	Optional (0 or 1)	An object describing the signature script of this input. Not if this is a coinbase present transaction
→ → → → `asm`	string	Required (exactly 1)	The signature script in decoded form with non-data-pushing opcodes listed
→ → → → `hex`	string (hex)	Required (exactly 1)	The signature script encoded as hex
→ → → `sequence`	number (int)	Required (exactly 1)	The input sequence number
→ `vout`	array	Required (exactly 1)	An array of objects each describing an output vector (vout) for this transaction. Output objects will have the same order within the array as they have in the transaction, so the first output listed will be output 0
→ → Output	array	Required	An object describing one of this transaction's outputs
→ → → `value`	number (zcash)	Required (exactly 1)	The number of ZEC paid to this output. May be `0`
→ → → `n`	number (int)	Required (exactly	The output index number of this output within this transaction

Example

Decode a signed one-input, three-output transaction:

```
zcash-cli decoderawtransaction␣
↪0100000001268a9ad7bfb21d3c086f0ff28f73a064964aa069ebb69a9e437da85c7e55c7d7000000006b483045022100ee6
```

Result:

```
{
    "txid" : "e9c77353da98e8897bf34ac589db0d3a407987d566cf8cbc19b6cf57e64fa1ad",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "b039c06ed7d729907f7d23f7edd017e413fe550afe78b50e3d8af241e1666d2f
↪",
            "vout" : 0,
            "scriptSig" : {
                "asm" :
↪"3045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e45a
↪03a7c1fd1fdec50e1cf3f0cc8cb4378cd8e9a2cee8ca9b3118f3db16cbbcf8f326",
                "hex" :
↪"483045022100ee69171016b7dd218491faf6e13f53d40d64f4b40123a2de52560feb95de63b902206f23a0919471eaa1e4
↪"
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.39890000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 56847befbd2360df0e35b4e3b77bae48585ae068␣
↪OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91456847befbd2360df0e35b4e3b77bae48585ae06888ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "t14sSauSf5pF2UkUwvKGq4qjNRzBZYqgEL5"
                ]
            }
        },
        {
            "value" : 0.10000000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 2b14950b8d31620c6cc923c5408a701b1ec0a020␣
↪OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a9142b14950b8d31620c6cc923c5408a701b1ec0a02088ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "t1mtya1DiYpTF166zTsBnB3nY1BP8LsSqN7"
                ]
            }
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "value" : 0.20000000,
            "n" : 2,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 0dfc8bafc8419853b34d5e072ad37d1a5159f584␣
↪OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a9140dfc8bafc8419853b34d5e072ad37d1a5159f58488ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "t1PLfoTX6Z3XuWPpS45hntVZerDKCkrsMg3"
                ]
            }
        }
    ]
}
```

See also

- *CreateRawTransaction*: creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.
- **'SignRawTransaction'_: |summary_signRawTransaction|**
- **'SendRawTransaction'_: |summary_sendRawTransaction|**

6.3.8 DecodeScript

The `decodescript` RPC decodes a hex-encoded P2SH redeem script.

Parameter #1—a hex-encoded redeem script

Result—the decoded script

Name	Type	Presence	Description
`result`	object	Required (exactly 1)	An object describing the decoded script, or JSON `null` if the script could not be decoded
→ `asm`	string	Required (exactly 1)	The redeem script in decoded form with non-data-pushing opcodes listed. May be empty
→ `type`	string	Optional (0 or 1)	The type of script. This will be one of the following: - `pubkey` for a P2PK script inside P2SH - `pubkeyhash` for a P2PKH script inside P2SH - `multisig` for a multisig script inside P2SH - `nonstandard` for unknown scripts
→ `regSigs`	number (int)	Optional (0 or 1)	The number of signatures required; this is always `1` for P2PK or P2PKH within P2SH. It may be greater than 1 for P2SH multisig. This value will not be returned for `nonstandard` script types (see the `type` key above)
→ `addresses`	array	Optional (0 or 1)	A P2PKH addresses used in this script, or the computed P2PKH addresses of any pubkeys in this script. This array will not be returned for `nonstandard` script types
→ → Address	string	Required (1 or more)	A P2PKH address
→ `p2sh`	string (hex)	Required (exactly 1)	The P2SH address of this redeem script

Example

A 2-of-3 P2SH multisig pubkey script:

```
zcash-cli decodescript␣
↪522103ede722780d27b05f0b1169efc90fa15a601a32fc6c3295114500c586831b6aaf2102ecd2d250a76d204011de6bc3
```

Result:

```
{
    "asm" : "2 03ede722780d27b05f0b1169efc90fa15a601a32fc6c3295114500c586831b6aaf␣
↪02ecd2d250a76d204011de6bc365a56033b9b3a149f679bc17205555d3c2b2854f␣
↪022d609d2f0d359e5bc0e5d0ea20ff9f5d3396cb5b1906aa9c56a0e7b5edc0c5d5 3 OP_
↪CHECKMULTISIG",
    "reqSigs" : 2,
    "type" : "multisig",
    "addresses" : [
        "t14sSauSf5pF2UkUwvKGq4qjNRzBZYqgEL5",
        "t1PoHp2v54vfmdgQ3v3SNuQga8JKHTNi2a1",
        "t1LLfoTX6Z3XuWPpS45hntVZerDKCkrsMg3"
    ],
    "p2sh" : "t3yVxxgNBk5zHRPRY2iVjGRJHYZEp1pMCSq"
}
```

See also

- *CreateMultiSig*: creates a P2SH multi-signature transparent address. *Multisig is not supported for shielded addresses.*
- **'Pay-To-Script-Hash'_** </en/glossary/p2sh-address>

6.3.9 DumpPrivKey

Requires wallet support.

The `dumpprivkey` RPC {{summary_dumpPrivKey}}

Parameter #1—the address corresponding to the private key to get

Name	Type	Presence	Description
P2PKH Address	string (base58)	Required (exactly 1)	The P2PKH address corresponding to the private key you want returned. Must be the address corresponding to a private key in this wallet

Result—the private key .. csv-table:

```
:header: |n|, |t|, |p|, |d|

"``result``", "string (base58)", "Required (exactly 1)", "The private key encoded as␣
↪base58check using wallet import format"
```

Example

```
zcash-cli dumpprivkey t14sSauSf5pF2UkUwvKGq4qjNRzBZYqgEL5
```

Result:

```
KxPQjJKyK2MT8hnbuXNutcqfh81t3XxHoTfQyF857qXuQHoTaGbX
```

See also

- **'ImportPrivKey'_**: **|summary_importPrivKey|**
- *DumpWallet*: creates or overwrites a file with all of the wallet's transparent keys in a human-readable format to the specified alphanumeric filename only after setting `-exportdir=`.

6.3.10 DumpWallet

Requires wallet support.

The `dumpwallet` RPC {{summary_dumpWallet}}

Parameter #1—a filename

Name	Type	Presence	Description
File-name	string (al-phanumeric	Required (exactly 1)	A filename to be created or overwritten within the directory speci-fied in `-exportdir=`

Result—the full path of the destination file .. csv-table:

```
:header: |n|, |t|, |p|, |d|

"``result``", "string (full path of destination file)", "Required (exactly 1)",
↪"Returns full path of destination file or error message. The JSON-RPC error and␣
↪message fields will be set if a failure occurred"
```

Example

Dump transparent keys in wallet after setting `-exportdir=/home/user/zcash-backup/`:

```
zcash-cli dumpwallet dump
```

Results:

```
/home/user/zcash-backup/dump
```

See also

- *BackupWallet*: safely copies *wallet.dat* to the specified alphanumeric filename only after setting `-exportdir=`.
- **‘ImportWallet‘_**: **|summary_importWallet|**

6.3.11 EncryptWallet

Wallet encryption is DISABLED. This call always fails.

6.3.12 EstimateFee

The `estimatefee` RPC estimates the transaction fee per kilobyte that needs to be paid for a transaction to be included within a certain number of blocks.

Parameter #1—how many blocks the transaction may wait before being included

Name	Type	Presence	Description
Blocks	number (int)	Required (exactly 1)	The maximum number of blocks a transaction should have to wait before it is predicted to be included in a block

Result—the fee the transaction needs to pay per kilobyte

Name	Type	Presence	Description
`result`	number (zcash)	Required (exactly 1)	The estimated fee the transaction should pay in order to be included within the specified number of blocks. If the node doesn’t have enough information to make an estimate, the value *-1* will be returned

Examples

```
zcash-cli estimatefee 2
```

Result:

0.00002491

Requesting data the node can't calculate yet:

```
zcash-cli estimatefee 200
```

Result:

```
-1
```

See also

- *EstimatePriority*: estimates the priority that a transaction needs in order to be included within a certain number of blocks as a free high-priority transaction.
- **'SetTxFee'_: |summary_setTxFee|**

6.3.13 EstimatePriority

The `estimatepriority` RPC {{summary_estimatePriority}}

Transaction priority is relative to a transaction's byte size.

Parameter #1—how many blocks the transaction may wait before being included as a free high-priority transaction

Name	Type	Presence	Description
Blocks	number (int)	Required (exactly 1)	The maximum number of blocks a transaction should have to wait before it is predicted to be included in a block based purely on its priority

Result—the priority a transaction needs

Name	Type	Presence	Description
`result`	number (real)	Required (exactly 1)	The estimated priority the transaction should have in order to be included within the specified number of blocks. If the node doesn't have enough information to make an estimate, the value *-1* will be returned

Examples

```
zcash-cli estimatepriority 6
```

Result:

```
718158904.10958910
```

Requesting data the node can't calculate yet:

```
zcash-cli estimatepriority 100
```

Result:

```
-1
```

See also

6.4 Wallet Backup Instructions

6.4.1 Overview

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

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 address in the wallet is generated, we recommending making a new backup so all private keys for addresses in your wallet are safe.

Note that a backup is a duplicate of data needed to spend ZEC 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.

6.4.2 Instructions for backing up your wallet and/or private keys

These instructions are specific for the officially supported Zcash 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 ZEC and view your shielded ZEC.

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

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

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

Note that zcashd 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:

`zcash-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 you generate a new Zcash address, it will not be reflected in the backup file.

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`.

Using `z_exportwallet` & `z_importwallet`

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

```
zcash-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.

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

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

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

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

```
zcash-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:

```
zcash-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:

```
zcash-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:

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

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

```
zcash-cli help z_importkey
```

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

```
zcash-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:

```
zcash-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.

6.5 Shielding Coinbase UTXOs

Summary

Use `z_shieldcoinbase` RPC call to shield coinbase UTXOs.

Who should read this document

Miners, Mining pools, Online wallets

6.5.1 Background

The current Zcash protocol includes a consensus rule that coinbase rewards must be sent to a shielded address.

6.5.2 User Experience Challenges

A user can use the z_sendmany RPC call to shield coinbase funds, but the call was not designed for sweeping up many UTXOs, and offered a suboptimal user experience.

If customers send mining pool payouts to their online wallet, the service provider must sort through UTXOs to correctly determine the non-coinbase UTXO funds that can be withdrawn or transferred by customers to another transparent address.

6.5.3 Solution

The z_shieldcoinbase call makes it easy to sweep up coinbase rewards from multiple coinbase UTXOs across multiple coinbase reward addresses.

```
z_shieldcoinbase fromaddress toaddress (fee) (limit)
```

The default fee is 0.0010000 ZEC and the default limit on the maximum number of UTXOs to shield is 50.

6.5.4 Examples

Sweep up coinbase UTXOs from a transparent address you use for mining:

```
zcash-cli z_shieldcoinbase tMyMiningAddress zMyPrivateAddress
```

Sweep up coinbase UTXOs from multiple transparent addresses to a shielded address:

```
zcash-cli z_shieldcoinbase "*" zMyPrivateAddress
```

Sweep up with a fee of 1.23 ZEC:

```
zcash-cli z_shieldcoinbase tMyMiningAddress zMyPrivateAddress 1.23
```

Sweep up with a fee of 0.1 ZEC and set limit on the maximum number of UTXOs to shield at 25:

```
zcash-cli z_shieldcoinbase "*" zMyPrivateAddress 0.1 25
```

Asynchronous Call

The `z_shieldcoinbase` RPC call is an asynchronous call, so you can queue up multiple operations.

When you invoke

```
zcash-cli z_shieldcoinbase tMyMiningAddress zMyPrivateAddress
```

JSON will be returned immediately, with the following data fields populated:

- operationid: a temporary id to use with `z_getoperationstatus` and `z_getoperationresult` to get the status and result of the operation.
- shieldedUTXOs: number of coinbase UTXOs being shielded
- shieldedValue: value of coinbase UTXOs being shielded.
- remainingUTXOs: number of coinbase UTXOs still available for shielding.
- remainingValue: value of coinbase UTXOs still available for shielding

Locking UTXOs

The `z_shieldcoinbase` call will lock any selected UTXOs. This prevents the selected UTXOs which are already queued up from being selected for any other send operation. If the `z_shieldcoinbase` call fails, any locked UTXOs are unlocked.

You can use the RPC call `lockunspent` to see which UTXOs have been locked. You can also use this call to unlock any UTXOs in the event of an unexpected system failure which leaves UTXOs in a locked state.

Limits, Performance and Transaction Confirmation

The number of coinbase UTXOs selected for shielding can be adjusted by setting the limit parameter. The default value is 50.

If the limit parameter is set to zero, the zcashd `mempooltxinputlimit` option will be used instead, where the default value for `mempooltxinputlimit` is zero, which means no limit.

Any limit is constrained by a hard limit due to the consensus rule defining a maximum transaction size of 100,000 bytes.

In general, the more UTXOs that are selected, the longer it takes for the transaction to be verified. Due to the quadratic hashing problem, some miners use the `mempooltxinputlimit` option to reject transactions with a large number of UTXO inputs.

Currently, as of November 2017, there is no commonly agreed upon limit, but as a rule of thumb (a form of emergent consensus) if a transaction has less than 100 UTXO inputs, the transaction will be mined promptly by the majority of mining pools, but if it has many more UTXO inputs, such as 500, it might take several days to be mined by a miner who has higher or no limits.

Anatomy of a z_shieldcoinbase transaction

The transaction created is a shielded transaction. It consists of a single joinsplit, which consumes coinbase UTXOs as input, and deposits value at a shielded address, minus any fee.

The number of coinbase UTXOs is determined by a user configured limit.

If no limit is set (in the case when limit parameter and `mempooltxinputlimit` options are set to zero) the behaviour of z_shieldcoinbase is to consume as many UTXOs as possible, with `z_shieldcoinbase` constructing a transaction up to the size limit of 100,000 bytes.

As a result, the maximum number of inputs that can be selected is:

- P2PKH coinbase UTXOs ~ 662
- 2-of-3 multisig P2SH coinbase UTXOs ~ 244.

Here is an example of using `z_shieldcoinbase` on testnet to shield multi-sig coinbase UTXOs.

- Block 141042 is almost ~2 MB in size (the maximum size for a block) and contains 1 coinbase reward transaction and 20 transactions, each indivually created by a call to z_shieldcoinbase.
  - https://explorer.testnet.z.cash/block/0050552a78e97c89f666713c8448d49ad1d7263274422272696187dedf6c0d03
- Drilling down into a transaction, you can see there is one joinsplit, with 244 inputs (vin) and 0 outputs (vout).
  - https://explorer.testnet.z.cash/tx/cf4f3da2e434f68b6e361303403344e22a9ff9a8fda9abc180d9520d0ca6527d

6.6 Payment Disclosure (Experimental Feature)

Summary

Use RPC calls `z_getpaymentdisclosure` and `z_validatepaymentdisclosure` to reveal details of a shielded payment.

Who should read this document

Frequent users of shielded transactions, payment processors, exchanges, block explorer

6.6.1 Experimental Feature

This is an experimental feature. Enable it by launching `zcashd` with flags:

```
zcashd -experimentalfeatures -paymentdisclosure -debug=paymentdisclosure -txindex=1
```

These flags can also be set as options in `zcash.conf`.

All nodes that generate or validate payment disclosures must run with `txindex=1` enabled.

6.6.2 Background

Payment Disclosure is an implementation of the work-in-progress Payment Disclosure ZIP [1].

The ZIP describes a method of proving that a payment was sent to a shielded address. In the typical case, this means enabling a sender to present a proof that they transferred funds to a recipient's shielded address.

[1] https://github.com/zcash/zips/pull/119

6.6.3 Example Use Case

Alice the customer sends 10 ZEC to Bob the merchant at the shielded address shown on their website. However, Bob is not sure if he received the funds.

Alice's node is running with payment disclosure enabled, so Alice generates a payment disclosure and provides it to Bob, who verifies the payment was made.

If Bob is a bad merchant, Alice can present the payment disclosure to a third party to validate that payment was indeed made.

6.6.4 Solution

A payment disclosure can be generated for any output of a JoinSplit using the RPC call:

```
z_getpaymentdisclosure txid js_index output_index (message)
```

An optional message can be supplied. This could be used for a refund address or some other reference, as currently it is not common practice to (ahead of time) include a refund address in the memo field when making a payment.

To validate a payment disclosure, the following RPC call can be used:

```
z_validatepaymentdisclosure hexdata
```

6.6.5 Example

Generate a payment disclosure for the first joinsplit, second output (index starts from zero):

```
zcash-cli z_getpaymentdisclosure␣
↪79189528d611e811a1c7bb0358dd31343033d14b4c1e998d7c4799c40f8b652b 0 1 "Hello"
```

This returns a payment disclosure in the form of a hex string:

```
706462ff000a3722aafa8190cdf9710bfad6da2af6d3a74262c1fc96ad47df814b0cd5641c2b658b0fc499477c8d991e4c4bc
```

To validate the payment disclosure:

```
zcash-cli z_validatepaymentdisclosure HEXDATA
```

This returns data related to the payment and the payment disclosure:

```
{
  "txid": "79189528d611e811a1c7bb0358dd31343033d14b4c1e998d7c4799c40f8b652b",
  "jsIndex": 0,
  "outputIndex": 1,
  "version": 0,
  "onetimePrivKey": "1c64d50c4b81df47ad96fcc16242a7d3f62adad6fa0b71f9cd9081faaa22370a
↪",
  "message": "Hello",
  "joinSplitPubKey": "d1c465d16166b602992479acfac18e87dc18065f6cefde6a002e70bc371b9faf
↪",
  "signatureVerified": true,
  "paymentAddress":
↪"ztaZJXy8iX8nrk2ytXKDBoTWqPkhQcj6E2ifARnD3wfkFwsxXs5SoX7NGmrjkzSiSKn8VtLHTJae48vX5NakvmDhtGNY5eb
↪",
  "memo":
↪"f6000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
  "value": 12.49900000,
  "commitmentMatch": true,
  "valid": true
}
```

The `signatureVerified` field confirms that the payment disclosure was generated and signed with the joinSplit-PrivKey, which should only be known by the node generating and sending the transaction 7918…652b in question.

6.6.6 Where is the data stored?

For all nodes, payment disclosure does not touch `wallet.dat` in any way.

For nodes that only validate payment disclosures, no data is stored locally.

For nodes that generate payment disclosures, a LevelDB database is created in the node's datadir. For most users, this would be in the folder:

```
$HOME/.zcash/paymentdisclosure
```

If you decide you don't want to use payment disclosure, it is safe to shut down your node and delete the database folder.

6.6.7 Security Properties

Please consult the work-in-progress ZIP for details about the protocol, security properties and caveats.

6.6.8 Reminder

Feedback is most welcome!

This is an experimental feature so there are no guarantees that the protocol, database format, RPC interface etc. will remain the same in the future.

6.6.9 Notes

Currently there is no user friendly way to help senders identify which joinsplit output index maps to a given payment they made. It is possible to construct this from `debug.log`. Ideas and feedback are most welcome on how to improve the user experience.

*** Warning: Do not assume Tor support does the correct thing in Zcash; better Tor support is a future feature goal. ***

6.7 Tor Support in Zcash

It is possible to run Zcash as a Tor hidden service, and connect to such services.

The following directions assume you have a Tor proxy running on port 9050. Many distributions default to having a SOCKS proxy listening on port 9050, but others may not. In particular, the Tor Browser Bundle defaults to listening on port 9150. See Tor Project FAQ:TBBSocksPort for how to properly configure Tor.

1. Run Zcash behind a Tor proxy

The first step is running Zcash behind a Tor proxy. This will already make all outgoing connections be anonymized, but more is possible.

```
-proxy=ip:port  Set the proxy server. If SOCKS5 is selected (default), this proxy
                server will be used to try to reach .onion addresses as well.

-onion=ip:port  Set the proxy server to use for Tor hidden services. You do not
                need to set this if it's the same as -proxy. You can use -noonion
                to explicitly disable access to hidden service.

-listen         When using -proxy, listening is disabled by default. If you want
                to run a hidden service (see next section), you'll need to enable
                it explicitly.

-connect=X      When behind a Tor proxy, you can specify .onion addresses instead
-addnode=X      of IP addresses or hostnames in these parameters. It requires
-seednode=X     SOCKS5. In Tor mode, such addresses can also be exchanged with
                other P2P nodes.
```

In a typical situation, this suffices to run behind a Tor proxy:

```
./zcashd -proxy=127.0.0.1:9050
```

1. Run a Zcash hidden server

If you configure your Tor system accordingly, it is possible to make your node also reachable from the Tor network. Add these lines to your /etc/tor/torrc (or equivalent config file):

```
HiddenServiceDir /var/lib/tor/zcash-service/
HiddenServicePort 8233 127.0.0.1:8233
HiddenServicePort 18233 127.0.0.1:18233
```

The directory can be different of course, but (both) port numbers should be equal to your zcashd's P2P listen port (8233 by default).

```
-externalip=X   You can tell Zcash about its publicly reachable address using
                this option, and this can be a .onion address. Given the above
                configuration, you can find your onion address in
                /var/lib/tor/zcash-service/hostname. Onion addresses are given
                preference for your node to advertize itself with, for connections
                coming from unroutable addresses (such as 127.0.0.1, where the
                Tor proxy typically runs).

-listen         You'll need to enable listening for incoming connections, as this
                is off by default behind a proxy.

-discover       When -externalip is specified, no attempt is made to discover local
                IPv4 or IPv6 addresses. If you want to run a dual stack, reachable
                from both Tor and IPv4 (or IPv6), you'll need to either pass your
                other addresses using -externalip, or explicitly enable -discover.
                Note that both addresses of a dual-stack system may be easily
                linkable using traffic analysis.
```

In a typical situation, where you're only reachable via Tor, this should suffice:

```
./zcashd -proxy=127.0.0.1:9050 -externalip=zctestseie6wxgio.onion -listen
```

(obviously, replace the Onion address with your own). It should be noted that you still listen on all devices and another node could establish a clearnet connection, when knowing your address. To mitigate this, additionally bind the address of your Tor proxy:

```
./zcashd ... -bind=127.0.0.1
```

If you don't care too much about hiding your node, and want to be reachable on IPv4 as well, use `discover` instead:

```
./zcashd ... -discover
```

and open port 8233 on your firewall (or use -upnp).

If you only want to use Tor to reach onion addresses, but not use it as a proxy for normal IPv4/IPv6 communication, use:

```
./zcashd -onion=127.0.0.1:9050 -externalip=zctestseie6wxgio.onion -discover
```

1. Automatically listen on Tor

Starting with Tor version 0.2.7.1 it is possible, through Tor's control socket API, to create and destroy 'ephemeral' hidden services programmatically. Zcash has been updated to make use of this.

This means that if Tor is running (and proper authentication has been configured), Zcash automatically creates a hidden service to listen on. Zcash will also use Tor automatically to connect to other .onion nodes if the control socket can be successfully opened. This will positively affect the number of available .onion nodes and their usage.

This new feature is enabled by default if Zcash is listening (`-listen`), and requires a Tor connection to work. It can be explicitly disabled with `-listenonion=0` and, if not disabled, configured using the `-torcontrol` and `-torpassword` settings. To show verbose debugging information, pass `-debug=tor`.

Connecting to Tor's control socket API requires one of two authentication methods to be configured. For cookie authentication the user running zcashd must have write access to the `CookieAuthFile` specified in Tor configuration. In some cases this is preconfigured and the creation of a hidden service is automatic. If permission problems are seen with `-debug=tor` they can be resolved by adding both the user running tor and the user running zcashd to the same group and setting permissions appropriately. On Debian-based systems the user running zcashd can be added to the debian-tor group, which has the appropriate permissions. An alternative authentication method is the use of the `-torpassword` flag and a `hash-password` which can be enabled and specified in Tor configuration.

1. Connect to a Zcash hidden server

To test your set-up, you might want to try connecting via Tor on a different computer to just a a single Zcash hidden server. Launch zcashd as follows:

```
./zcashd -onion=127.0.0.1:9050 -connect=zctestseie6wxgio.onion
```

Now use zcash-cli to verify there is only a single peer connection.

```
zcash-cli getpeerinfo

[
    {
        "id" : 1,
        "addr" : "zctestseie6wxgio.onion:18233",
        ...
        "version" : 170002,
        "subver" : "/MagicBean:1.0.0/",
        ...
    }
]
```

To connect to multiple Tor nodes, use:

```
./zcashd -onion=127.0.0.1:9050 -addnode=zctestseie6wxgio.onion -dnsseed=0 -
↪onlynet=onion
```

6.8 Security Warnings

6.8.1 Security Audit

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

6.8.2 x86-64 Linux Only

There are known bugs which 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.

6.8.3 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 runnng on your OS can read your wallet.dat file.

6.8.4 Side-Channel Attacks

This implementation of Zcash 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 `zcashd` 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 `zcashd` process runs on until these vulnerabilities are fully analyzed and fixed.

6.8.5 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.

6.8.6 RPC Interface

Users should choose a strong RPC password. If no RPC username and password are set, zcashd will not start and will print an error message with a suggestion for a strong random password. If the client knows the RPC password, they have at least full access to the node. In addition, certain RPC commands can be misused to overwrite files and/or take over the account that is running zcashd. (In the future we may restrict these commands, but full node access – including the ability to spend from and export keys held by the wallet – would still be possible unless wallet methods are disabled.)

Users should also 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 zcashd and loss of funds. For multi-user services that use one or more zcashd 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 zcashd.

6.8.7 Block Chain Reorganization: Major Differences

Users should be aware of new behavior in Zcash 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. Zcash 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 Zcash, 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).

6.8.8 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.

6.8.9 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 in issue #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.

6.9 Troubleshooting FAQ

The General FAQ has been reorganized and relocated to https://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 zcashd, the core Zcash client software.

6.9.1 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. Zcash 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 of the tests.

Zcash 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

6.9.2 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 Zcash, there may be an issue with different versions of linked libraries. Try `make clean` and build again.

6.9.3 Running Zcashd

Trying to start Zcashd for the first time, it fails with `could not load param file at /home/rebroad/.zcash-params/sprout-verifying.key`

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

Zcashd crashes with the message `std::bad_alloc` or `St13runtime_exception`.

These messages indicate that your computer has run out of memory for running zcashd. 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.

6.9.4 RPC Interface

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

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

There is also additional documentation under doc/payment-api.md.

`zcash-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. `zcash-cli z_importkey $KEY false`

6.9.5 What if my question isn't answered here?

First search the issues section (https://github.com/zcash/zcash/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.

6.10 Expectations for DNS Seed operators

Zcash attempts to minimize the level of trust in DNS seeds, but DNS seeds still pose a small amount of risk for the network. As such, DNS seeds must be run by entities which have some minimum level of trust within the Zcash community.

Other implementations of Zcash software may also use the same seeds and may be more exposed. In light of this exposure, this document establishes some basic expectations for operating DNS seeds.

1. A DNS seed operating organization or person is expected to follow good host security practices, maintain control of applicable infrastructure, and not sell or transfer control of the DNS seed. Any hosting services contracted by the operator are equally expected to uphold these expectations.
2. The DNS seed results must consist exclusively of fairly selected and functioning Zcash nodes from the public network to the best of the operator's understanding and capability.
3. For the avoidance of doubt, the results may be randomized but must not single out any group of hosts to receive different results unless due to an urgent technical necessity and disclosed.
4. The results may not be served with a DNS TTL of less than one minute.
5. Any logging of DNS queries should be only that which is necessary for the operation of the service or urgent health of the Zcash network and must not be retained longer than necessary nor disclosed to any third party.
6. Information gathered as a result of the operators node-spidering (not from DNS queries) may be freely published or retained, but only if this data was not made more complete by biasing node connectivity (a violation of expectation (1)).
7. Operators are encouraged, but not required, to publicly document the details of their operating practices.
8. A reachable email contact address must be published for inquiries related to the DNS seed operation.

If these expectations cannot be satisfied the operator should discontinue providing services and contact the active Zcash development team as well as creating an issue in the Zcash repository.

Behavior outside of these expectations may be reasonable in some situations but should be discussed in public in advance.

6.10.1 See also

- zcash-seeder is a reference implementation of a DNS seed.
- zcash.conf: contains configuration settings for zcashd
- zcashd.pid: stores the process id of zcashd while running
- blocks/blk000??.dat: block data (custom, 128 MiB per file)
- blocks/rev000??.dat; block undo data (custom)
- blocks/index/*; block index (LevelDB)
- chainstate/*; block chain state database (LevelDB)
- database/*: BDB database environment
- db.log: wallet database log file
- debug.log: contains debug information and general logging generated by zcashd
- fee_estimates.dat: stores statistics used to estimate minimum transaction fees and priorities required for confirmation
- peers.dat: peer IP address database (custom format)
- wallet.dat: personal wallet (BDB) with keys and transactions
- .cookie: session RPC authentication cookie (written at start when cookie authentication is used, deleted on shutdown): since 0.12.0
- onion_private_key: cached Tor hidden service private key for `-listenonion`: since 0.12.0

6.11 Zcash Mining Guide

Welcome! This guide is intended to get you mining Zcash, a.k.a. "ZEC", on the Zcash 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/

6.11.1 Setup

First, you need to set up your local Zcash node. Follow the 1.0 User Guide up to the end of the section "Compiling", then come back here. (You can also do the "Testing" section if you want!)

6.11.2 Configuration

Configure your node as per [[1.0-User-Guide#configuration]], including the section Enabling CPU Mining.

6.11.3 Mining

Now, start Mining!

```
$ ./src/zcashd
```

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

```
$ ./src/zcashd -daemon
```

You should see the following output in the debug log (`~/.zcash/debug.log`):

```
Zcash Miner started
```

Congratulations! You are now mining on the mainnet.

To stop the Zcash daemon, enter the command:

```
$ ./src/zcash-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 1.0 User Guide 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-maintained list of mining pools for further instructions.

6.11.4 Modifications

Mine to a single address

The internal `zcashd` miner uses a new transparent address for each mined block. If you want to instead use the same address for every mined block, find the following line in both `src/miner.cpp` (in the function `ProcessBlockFound()`) and `src/wallet/wallet.cpp` (in the function `CommitTransaction()`):

```
reservekey.KeepKey();
```

Remove or comment out that line in both places.

zcashd 1.0.6 will have out-of-the-box support for mining to a specified address using the `-mineraddress=` option.

Use P2PKH transactions

The internal `zcashd` miner inherited from Bitcoin uses P2PK for coinbase transactions. The trend in the Bitcoin blockchain has been to use P2PKH instead; we are considering changing the internal miner to use P2PKH, but not for the 1.0 release.

If you want to use P2PKH for your coinbase transactions, find the following line in `src/miner.cpp` (in the function `CreateNewBlockWithKey()`):

```
CScript scriptPubKey = CScript() << ToByteVector(pubkey) << OP_CHECKSIG;
```

Change it to:

```
CScript scriptPubKey = CScript() << OP_DUP << OP_HASH160 << ToByteVector(pubkey.
↪GetID()) << OP_EQUALVERIFY << OP_CHECKSIG;
```

6.12 Compiling/running automated tests

Automated tests will be automatically compiled if dependencies were met in configure and tests weren't explicitly disabled.

There are two scripts for running tests:

- `qa/zcash/full-test-suite.sh`, to run the main test suite
- `qa/pull-tester/rpc-tests.sh`, to run the RPC tests.

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 Zcash unit tests.

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

6.13 Development

Contents

6.13.1 Code Management

We achieve our design goals primarily through this codebase as a reference implementation. This repository is a fork of Bitcoin Core as of upstream release 0.11.2 (many later Bitcoin PRs have also been ported to Zcash). It implements the Zcash protocol and a few other distinct features.

Testing

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

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

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

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

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/

Workflow

1. Fork our repository
2. Create a new branch with your changes
3. Submit a pull request

6.13.2 Releases

Starting from Zcash v1.0.0-beta1, Zcash 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 Zcash was forked from Bitcoin v0.11.2).

6.13.3 Commit messages

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])?".

6.13.4 Continuous Integration

Watch the Buildbot. Because the Buildbot is watching you.

6.13.5 Misc Notes

These are historical, probably bit-rotted, but also probably full of important nuggets:

- Notes from the Boulder Hack Fest
- zdep Google group (requires login)

6.14 Coding

Various coding styles have been used during the history of the codebase, and the result is not very consistent. However, we're now trying to converge to a single style, so please use it in new code. Old code will be converted gradually.

- Basic rules specified in src/.clang-format. Use a recent clang-format-3.5 to format automatically.
  - Braces on new lines for namespaces, classes, functions, methods.
  - Braces on the same line for everything else.
  - 4 space indentation (no tabs) for every block except namespaces.
  - No indentation for public/protected/private or for namespaces.
  - No extra spaces inside parenthesis; don't do (this)
  - No space after function names; one space after if, for and while.

Block style example:

```
namespace foo
{
class Class
{
    bool Function(char* psz, int n)
    {
        // Comment summarising what this section of code does
        for (int i = 0; i < n; i++) {
            // When something fails, return early
            if (!Something())
                return false;
            ...
        }

        // Success return is usually at the end
        return true;
    }
}
}
```

6.14.1 Doxygen comments

To facilitate the generation of documentation, use doxygen-compatible comment blocks for functions, methods and fields.

For example, to describe a function use:

```
/**
 * ... text ...
 * @param[in] arg1    A description
 * @param[in] arg2    Another argument description
 * @pre Precondition for function...
 */
bool function(int arg1, const char *arg2)
```

A complete list of `@xxx` commands can be found at http://www.stack.nl/~dimitri/doxygen/manual/commands.html. As Doxygen recognizes the comments by the delimiters (`/**` and `*/` in this case), you don't *need* to provide any commands for a comment to be valid; just a description text is fine.

To describe a class use the same construct above the class definition:

```
/**
 * Alerts are for notifying old versions if they become too obsolete and
 * need to upgrade. The message is displayed in the status bar.
 * @see GetWarnings()
 */
class CAlert
{
```

To describe a member or variable use:

```
int var; //!< Detailed description after the member
```

Also OK:

```
///
/// ... text ...
///
bool function2(int arg1, const char *arg2)
```

Not OK (used plenty in the current source, but not picked up):

```
//
// ... text ...
//
```

A full list of comment syntaxes picked up by doxygen can be found at http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html, but if possible use one of the above styles.

6.14.2 Development tips and tricks

compiling for debugging

Run configure with the –enable-debug option, then make. Or run configure with CXXFLAGS="-g -ggdb -O0" or whatever debug flags you need.

debug.log

If the code is behaving strangely, take a look in the debug.log file in the data directory; error and debugging messages are written there.

The -debug=... command-line option controls debugging; running with just -debug or -debug=1 will turn on all categories (and give you a very large debug.log file).

testnet and regtest modes

Run with the -testnet option to run with “play zcash” on the test network, if you are testing multi-machine code that needs to operate across the internet.

If you are testing something that can run on one machine, run with the -regtest option. In regression test mode, blocks can be created on-demand; see qa/rpc-tests/ for tests that run in -regtest mode.

DEBUG_LOCKORDER

Zcash is a multithreaded application, and deadlocks or other multithreading bugs can be very difficult to track down. Compiling with -DDEBUG_LOCKORDER (configure CXXFLAGS=”-DDEBUG_LOCKORDER -g”) inserts run-time checks to keep track of which locks are held, and adds warnings to the debug.log file if inconsistencies are detected.

6.14.3 Locking/mutex usage notes

The code is multi-threaded, and uses mutexes and the LOCK/TRY_LOCK macros to protect data structures.

Deadlocks due to inconsistent lock ordering (thread 1 locks cs_main and then cs_wallet, while thread 2 locks them in the opposite order: result, deadlock as each waits for the other to release its lock) are a problem. Compile with -DDEBUG_LOCKORDER to get lock order inconsistencies reported in the debug.log file.

Re-architecting the core code so there are better-defined interfaces between the various components is a goal, with any necessary locking done by the components (e.g. see the self-contained CKeyStore class and its cs_KeyStore lock for example).

6.14.4 Threads

- ThreadScriptCheck : Verifies block scripts.
- ThreadImport : Loads blocks from blk*.dat files or bootstrap.dat.
- StartNode : Starts other threads.
- ThreadDNSAddressSeed : Loads addresses of peers from the DNS.
- ThreadMapPort : Universal plug-and-play startup/shutdown
- ThreadSocketHandler : Sends/Receives data from peers on port 8233.
- ThreadOpenAddedConnections : Opens network connections to added nodes.
- ThreadOpenConnections : Initiates new connections to peers.
- ThreadMessageHandler : Higher-level message handling (sending and receiving).
- DumpAddresses : Dumps IP addresses of nodes to peers.dat.
- ThreadFlushWalletDB : Close the wallet.dat file if it hasn’t been used in 500ms.
- ThreadRPCServer : Remote procedure call handler, listens on port 8232 for connections and services them.
- ZcashMiner : Generates zcash (if wallet is enabled).
- Shutdown : Does an orderly shutdown of everything.

6.14.5 Pull Request 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.

6.15 Block and Transaction Broadcasting With AMQP 1.0 (Experimental Feature)

AMQP is an enterprise-level message queuing protocol for the reliable passing of real-time data and business transactions between applications. AMQP supports both broker and brokerless messaging. AMQP 1.0 is an open standard and has been ratified as ISO/IEC 19464.

The Zcash daemon can be configured to act as a trusted "border router", implementing the Zcash P2P protocol and relay, making consensus decisions, maintaining the local blockchain database, broadcasting locally generated transactions into the network, and providing a queryable RPC interface to interact on a polled basis for requesting blockchain related data. However, there exists only a limited service to notify external software of events like the arrival of new blocks or transactions.

The AMQP facility implements a notification interface through a set of specific notifiers. Currently there are notifiers that publish blocks and transactions. This read-only facility requires only the connection of a corresponding AMQP subscriber port in receiving software.

Currently the facility is not authenticated nor is there any two-way protocol involvement. Therefore, subscribers should validate the received data since it may be out of date, incomplete or even invalid.

Because AMQP is message oriented, subscribers receive transactions and blocks all-at-once and do not need to implement any sort of buffering or reassembly.

6.15.1 Prerequisites

The AMQP feature in Zcash requires Qpid Proton version 0.17 or newer, which you will need to install if you are not using the depends system. Typically, it is packaged by distributions as something like *libqpid-proton*. The C++ wrapper for AMQP *is* required.

In order to run the example Python client scripts in contrib/ one must also install *python-qpid-proton*, though this is not necessary for daemon operation.

6.15.2 Enabling

By default, the AMQP feature is automatically compiled in if the necessary prerequisites are found. To disable, use –disable-proton during the *configure* step of building zcashd:

```
$ ./configure --disable-proton (other options)
```

To actually enable operation, one must set the appropriate options on the commandline or in the configuration file.

6.15.3 Usage

AMQP support is currently an experimental feature, so you must pass the option:

```
-experimentalfeatures
```

Currently, the following notifications are supported:

```
-amqppubhashtx=address
-amqppubhashblock=address
-amqppubrawblock=address
-amqppubrawtx=address
```

The address must be a valid AMQP address, where the same address can be used in more than notification. Note that SSL and SASL addresses are not currently supported.

Launch zcashd like this:

```
$ zcashd -amqppubhashtx=amqp://127.0.0.1:5672
```

Or this:

```
$ zcashd -amqppubhashtx=amqp://127.0.0.1:5672 \
    -amqppubrawtx=amqp://127.0.0.1:5672 \
    -amqppubrawblock=amqp://127.0.0.1:5672 \
    -amqppubhashblock=amqp://127.0.0.1:5672 \
    -debug=amqp
```

The debug category `amqp` enables AMQP-related logging.

Each notification has a topic and body, where the header corresponds to the notification type. For instance, for the notification `-amqpubhashtx` the topic is `hashtx` (no null terminator) and the body is the hexadecimal transaction hash (32 bytes). This transaction hash and the block hash found in `hashblock` are in RPC byte order.

These options can also be provided in zcash.conf.

Please see `contrib/amqp/amqp_sub.py` for a working example of an AMQP server listening for messages.

6.15.4 Remarks

From the perspective of zcashd, the local end of an AMQP link is write-only.

No information is broadcast that wasn't already received from the public P2P network.

No authentication or authorization is done on peers that zcashd connects to; it is assumed that the AMQP link is exposed only to trusted entities, using other means such as firewalling.

TLS support may be added once OpenSSL has been removed from the Zcash project and alternative TLS implementations have been evaluated.

SASL support may be added in a future update for secure communication.

Note that when the block chain tip changes, a reorganisation may occur and just the tip will be notified. It is up to the subscriber to retrieve the chain from the last known block to the new tip.

At present, zcashd does not try to resend a notification if there was a problem confirming receipt. Support for delivery guarantees such as *at-least-once* and *exactly-once* will be added in in a future update.

Currently, zcashd appends an up-counting sequence number to each notification which allows listeners to detect lost notifications.

6.16 Block and Transaction Broadcasting With ZeroMQ

ZeroMQ is a lightweight wrapper around TCP connections, inter-process communication, and shared-memory, providing various message-oriented semantics such as publish/subcribe, request/reply, and push/pull.

The Zcash daemon can be configured to act as a trusted "border router", implementing the zcash wire protocol and relay, making consensus decisions, maintaining the local blockchain database, broadcasting locally generated transactions into the network, and providing a queryable RPC interface to interact on a polled basis for requesting blockchain related data. However, there exists only a limited service to notify external software of events like the arrival of new blocks or transactions.

The ZeroMQ facility implements a notification interface through a set of specific notifiers. Currently there are notifiers that publish blocks and transactions. This read-only facility requires only the connection of a corresponding ZeroMQ subscriber port in receiving software; it is not authenticated nor is there any two-way protocol involvement. Therefore, subscribers should validate the received data since it may be out of date, incomplete or even invalid.

ZeroMQ sockets are self-connecting and self-healing; that is, connections made between two endpoints will be automatically restored after an outage, and either end may be freely started or stopped in any order.

Because ZeroMQ is message oriented, subscribers receive transactions and blocks all-at-once and do not need to implement any sort of buffering or reassembly.

6.16.1 Prerequisites

The ZeroMQ feature in Zcash requires ZeroMQ API version 4.x or newer, which you will need to install if you are not using the depends system. Typically, it is packaged by distributions as something like *libzmq5-dev*. The C++ wrapper for ZeroMQ is *not* needed.

In order to run the example Python client scripts in contrib/ one must also install *python-zmq*, though this is not necessary for daemon operation.

6.16.2 Enabling

By default, the ZeroMQ feature is automatically compiled in if the necessary prerequisites are found. To disable, use –disable-zmq during the *configure* step of building zcashd:

```
$ ./configure --disable-zmq (other options)
```

To actually enable operation, one must set the appropriate options on the commandline or in the configuration file.

6.16.3 Usage

Currently, the following notifications are supported:

```
-zmqpubhashtx=address
-zmqpubhashblock=address
-zmqpubrawblock=address
-zmqpubrawtx=address
```

The socket type is PUB and the address must be a valid ZeroMQ socket address. The same address can be used in more than one notification.

For instance:

```
$ zcashd -zmqpubhashtx=tcp://127.0.0.1:28332 \
           -zmqpubrawtx=ipc:///tmp/zcashd.tx.raw
```

Each PUB notification has a topic and body, where the header corresponds to the notification type. For instance, for the notification `-zmqpubhashtx` the topic is `hashtx` (no null terminator) and the body is the hexadecimal transaction hash (32 bytes).

These options can also be provided in zcash.conf.

ZeroMQ endpoint specifiers for TCP (and others) are documented in the ZeroMQ API.

Client side, then, the ZeroMQ subscriber socket must have the ZMQ_SUBSCRIBE option set to one or either of these prefixes (for instance, just `hash`); without doing so will result in no messages arriving. Please see `contrib/zmq/zmq_sub.py` for a working example.

6.16.4 Remarks

From the perspective of zcashd, the ZeroMQ socket is write-only; PUB sockets don't even have a read function. Thus, there is no state introduced into zcashd directly. Furthermore, no information is broadcast that wasn't already received from the public P2P network.

No authentication or authorization is done on connecting clients; it is assumed that the ZeroMQ port is exposed only to trusted entities, using other means such as firewalling.

Note that when the block chain tip changes, a reorganisation may occur and just the tip will be notified. It is up to the subscriber to retrieve the chain from the last known block to the new tip.

There are several possibilities that ZMQ notification can get lost during transmission depending on the communication type your are using. Zcashd appends an up-counting sequence number to each notification which allows listeners to detect lost notifications.

6.17 Release Process

Meta: There should always be a single release engineer to disambiguate responsibility.

If this is a hotfix release, please see `./hotfix-process.md` before proceeding.

6.17.1 Pre-release

Github Milestone

Ensure all goals for the github milestone are met. If not, remove tickets or PRs with a comment as to why it is not included. (Running out of time is a common reason.)

Pre-release checklist:

Check that dependencies are properly hosted by looking at the `check-depends` builder:

https://ci.z.cash/#/builders/1

Check that there are no surprising performance regressions:

https://speed.z.cash

Ensure that new performance metrics appear on that site.

Protocol Safety Checks:

If this release changes the behavior of the protocol or fixes a serious bug, verify that a pre-release PR merge updated `PROTOCOL_VERSION` in `version.h` correctly.

If this release breaks backwards compatibility or needs to prevent interaction with software forked projects, change the network magic numbers. Set the four `pchMessageStart` in `CTestNetParams` in `chainparams.cpp` to random values.

Both of these should be done in standard PRs ahead of the release process. If these were not anticipated correctly, this could block the release, so if you suspect this is necessary, double check with the whole engineering team.

6.17.2 Release dependencies

The release script has the following dependencies:

- `help2man`
- `debchange` (part of the devscripts Debian package)

You can optionally install the `progressbar2` Python module with pip to have a progress bar displayed during the build process.

6.17.3 Release process

In the commands below, and <RELEASE_PREV> are prefixed with a v, ie. v1.0.9 (not 1.0.9).

Create the release branch

Run the release script, which will verify you are on the latest clean checkout of master, create a branch, then commit standard automated changes to that branch locally:

```
$ ./zcutil/make-release.py <RELEASE> <RELEASE_PREV> <RELEASE_FROM> <APPROX_RELEASE_
↪HEIGHT>
```

Examples:

```
$ ./zcutil/make-release.py v1.0.9 v1.0.8-1 v1.0.8-1 120000
$ ./zcutil/make-release.py v1.0.13 v1.0.13-rc1 v1.0.12 222900
```

Create, Review, and Merge the release branch pull request

Review the automated changes in git:

```
$ git log master..HEAD
```

Push the resulting branch to github:

```
$ git push 'git@github.com:$YOUR_GITHUB_NAME/zcash' $(git rev-parse --abbrev-ref HEAD)
```

Then create the PR on github. Complete the standard review process, then merge, then wait for CI to complete.

6.17.4 Make tag for the newly merged result

Checkout master and pull the latest version to ensure master is up to date with the release PR which was merged in before.

```
$ git checkout master
$ git pull --ff-only
```

Check the last commit on the local and remote versions of master to make sure they are the same:

```
$ git log -1
```

The output should include something like, which is created by Homu:

```
Auto merge of #4242 - nathan-at-least:release-v1.0.9, r=nathan-at-least
```

Then create the git tag. The `-s` means the release tag will be signed. **CAUTION:** Remember the `v` at the beginning here:

```
$ git tag -s v1.0.9
$ git push origin v1.0.9
```

6.17.5 Make and deploy deterministic builds

- Run the Gitian deterministic build environment
- Compare the uploaded build manifests on gitian.sigs
- If all is well, the DevOps engineer will build the Debian packages and update the apt.z.cash package repository.

6.17.6 Add release notes to GitHub

- Go to the GitHub tags page.
- Click "Add release notes" beside the tag for this release.
- Copy the release blog post into the release description, and edit to suit publication on GitHub. See previous release notes for examples.
- Click "Publish release" if publishing the release blog post now, or "Save draft" to store the notes internally (and then return later to publish once the blog post is up).

Note that some GitHub releases are marked as "Verified", and others as "Unverified". This is related to the GPG signature on the release tag - in particular, GitHub needs the corresponding public key to be uploaded to a corresponding GitHub account. If this release is marked as "Unverified", click the marking to see what GitHub wants to be done.

6.17.7 Post Release Task List

Deploy testnet

Notify the Zcash DevOps engineer/sysadmin that the release has been tagged. They update some variables in the company's automation code and then run an Ansible playbook, which:

- builds Zcash based on the specified branch
- deploys it as a public service (e.g. betatestnet.z.cash, mainnet.z.cash)

- often the same server can be re-used, and the role idempotently handles upgrades, but if not then they also need to update DNS records
- possible manual steps: blowing away the `testnet3` dir, deleting old parameters, restarting DNS seeder

Then, verify that nodes can connect to the testnet server, and update the guide on the wiki to ensure the correct hostname is listed in the recommended zcash.conf.

Update the 1.0 User Guide

This also means updating the translations. Coordinate with the translation team for now. Suggestions for improving this part of the process should be added to #2596.

Publish the release announcement (blog, github, zcash-dev, slack)

6.17.8 Celebrate

6.18 Hotfix Release Process

Hotfix releases are versioned by incrementing the build number of the latest release. For example:

```
First hotfix:  v1.0.11   -> v1.0.11-1
Second hotfix: v1.0.11-1 -> v1.0.11-2
```

In the commands below, and <RELEASE_PREV> are prefixed with a v, ie. v1.0.11 (not 1.0.11).

6.18.1 Create a hotfix branch

Create a hotfix branch from the previous release tag, and push it to the main repository:

```
$ git branch hotfix-<RELEASE> <RELEASE_PREV>
$ git push 'git@github.com:zcash/zcash' hotfix-<RELEASE>
```

6.18.2 Implement hotfix changes

Hotfix changes are implemented the same way as regular changes (developers work in separate branches per change, and push the branches to their own repositories), except that the branches are based on the hotfix branch instead of master:

```
$ git checkout hotfix-<RELEASE>
$ git checkout -b <BRANCH_NAME>
```

6.18.3 Merge hotfix PRs

Hotfix PRs are created like regular PRs, except using the hotfix branch as the base instead of master. Each PR should be reviewed as normal, and then the following process should be used to merge:

- A CI merge build is manually run by logging into the CI server, going to the pr-merge builder, clicking the “force” button, and entering the following values:
  - Repository: https://github.com//zcash

    - must be in the set of "safe" users as-specified in the CI config.

  - Branch: name of the hotfix PR branch (not the hotfix release branch).

- A link to the build and its result is manually added to the PR as a comment.
- If the build was successful, the PR is merged via the GitHub button.

6.18.4 Release process

The majority of this process is identical to the standard release process. However, there are a few notable differences:

- When running the release script, use the `--hotfix` flag:

  $ ./zcutil/make-release.py –hotfix <RELEASE_PREV> <APPROX_RELEASE_HEIGHT>

- To review the automated changes in git:

  $ git log hotfix-..HEAD

- After the standard review process, use the hotfix merge process outlined above instead of the regular merge process.
- When making the tag, check out the hotfix branch instead of master.

6.18.5 Post-release

Once the hotfix release has been created, a new PR should be opened for merging the hotfix release branch into master. This may require fixing merge conflicts (e.g. changing the version number in the hotfix branch to match master, if master is ahead). Such conflicts **MUST** be addressed with additional commits to the hotfix branch; specifically, the branch **MUST NOT** be rebased on master.

6.19 Issue Tracking

This policy is new as of 2017-05-09.

Contents

6.19.1 Milestones

We use milestones to coordinate releases. The Project Manager creates and tracks milestones for releases to make progress on our development goals. The Release Coordinator uses a milestone to ensure a release is shipped on schedule with the right people coordinating on the right tasks to make that happen.

- They are named after the target release version, ex: `1.0.9`.
- The due date is equal to the public package release date of our Zcash releases. This implies earlier deadlines for merging, testing, preparing blog posts, etc…
- *Only* the project manager or the release coordinator (or those they delegate to) should modify milestones, including:
  - Project Manager:
    - creating milestones, selecting release dates.
    - placing tickets into milestons, and only when those tickets are the highest priority items in appropriate *projects* which fit the goals of the milestone.
  - Release coordinator:
    - Altering the order of tickets to signify *chronological priority*. For example, dependencies need to occur before dependent PRs.
    - Executing PR merges (via Homu).
    - Closing tickets.
    - Kicking tickets out of the milestone.
- If others need any of these things done, they should request that from the appropriate person.

Note: There are some other ad hoc milestones not covered by this policy, and someday those should be cleaned up.

6.19.2 Projects

We use projects to define, prioritize, and track Zcash development priorities. Projects are how the *project manager* responds to requests for prioritizing things, decides what/when to prioritize, and track that we're reaching our goals through releases or non-release work. They also allow others to know what we're prioritizing.

- The Zcash repository has public repo-scoped projects which represent different dev team priorities. This includes specific feature goals (ie "Payment Disclosure"), and ongoing processes (ie "User Support").
- Some projects' names begin with *Priority N*. Others do not.
- *Anyone* (with sufficient github ACLs) may place tickets into the *triage queue* for a project by adding that project on a ticket's page.
  - A ticket should exist in at most 1 Priority project. If someone believes it should be in a different priority they should place the ticket into the new Priority project, then notify the Project Manager.
  - In addition to at most one Priority project, a ticket may be freely added to other projects. A common case is to add tickets to a Priority project *and* the *User Support* project.
- Only the Project Manager should:
  - Create projects
  - Alter the state of tickets in projects, such as placing them into columns from the triage queue, moving them between columns, or changing their order in columns.
  - Remove tickets from projects.
  - Delete or edit projects.
- The Project Manager may delegate those responsibilities to others for specific projects. One example is the User Support project, whose column states are managed by the User Support lead.

Note: Projects are a relatively new github feature and Zcash dev is in the midst of transitioning into their use.

6.19.3 Tickets

We use tickets to document facts or register needs, *without regard* to their priority, correctness, feasibility, etc… Those determinations occur throughout a ticket's life-cycle.

We use the term *ticket* here to refer to both *Pull Requests* and non-PR tickets.

Anyone including the public may:

- create tickets,
- comment on tickets.

Anyone on the Zcash dev team (defined by having suffcient Github ACLs) may:

- edit ticket labels,
- assign people to tickets,
- request reviews,
- review tickets,
- comment on tickets,
- add tickets into projects.

Unlike our earlier process, Zcash devs should now refrain from:

- placing tickets into milestones,
- altering the project state of a ticket (other than to insert it into a project).

6.19.4 Labels

Labels are used for a variety of purposes: tracking process, facilitating search queries, associating functionally related issues, etc…

In the past we've used *some* labels for process things where new Github features or human processes have been developed. Over time we should identify and clean up those labels since they are technical debt.

- *Anyone* on the dev team may create labels, but when you do announce what purpose it serves on #zcash-dev with an `@here` notification. Also, the purpose should be evident entirely from the name. This is tricky, and it's often the case that different people will begin using labels differently to represent different things.

Deprecated labels

- `user support` -> User Support project
- `user follow-up` -> User Support project, Follow-up column

*** Warning: This document has not been updated for Zcash and may be inaccurate. ***

6.20 Sample init scripts and service configuration for bitcoind

Sample scripts and configuration files for systemd, Upstart and OpenRC can be found in the contrib/init folder.

```
contrib/init/bitcoind.service:    systemd service unit configuration
contrib/init/bitcoind.openrc:     OpenRC compatible SysV style init script
contrib/init/bitcoind.openrcconf: OpenRC conf.d file
contrib/init/bitcoind.conf:       Upstart service configuration file
contrib/init/bitcoind.init:       CentOS compatible SysV style init script
```

1. Service User

All three startup configurations assume the existence of a "bitcoin" user and group. They must be created before attempting to use these scripts.

1. Configuration

At a bare minimum, bitcoind requires that the rpcpassword setting be set when running as a daemon. If the configuration file does not exist or this setting is not set, bitcoind will shutdown promptly after startup.

This password does not have to be remembered or typed as it is mostly used as a fixed token that bitcoind and client programs read from the configuration file, however it is recommended that a strong and secure password be used as this password is security critical to securing the wallet should the wallet be enabled.

If bitcoind is run with "-daemon" flag, and no rpcpassword is set, it will print a randomly generated suitable password to stderr. You can also generate one from the shell yourself like this:

bash -c 'tr -dc a-zA-Z0-9 < /dev/urandom | head -c32 && echo'

For an example configuration file that describes the configuration settings, see contrib/debian/examples/bitcoin.conf.

1. Paths

All three configurations assume several paths that might need to be adjusted.

Binary: /usr/bin/bitcoind Configuration file: /etc/bitcoin/bitcoin.conf Data directory: /var/lib/bitcoind PID file: /var/run/bitcoind/bitcoind.pid (OpenRC and Upstart) /var/lib/bitcoind/bitcoind.pid (systemd) Lock file: /var/lock/subsys/bitcoind (CentOS)

The configuration file, PID directory (if applicable) and data directory should all be owned by the bitcoin user and group. It is advised for security reasons to make the configuration file and data directory only readable by the bitcoin user and group. Access to bitcoin-cli and other bitcoind rpc clients can then be controlled by group membership.

1. Installing Service Configuration

4a) systemd

Installing this .service file consists of just copying it to /usr/lib/systemd/system directory, followed by the command "systemctl daemon-reload" in order to update running systemd configuration.

To test, run "systemctl start bitcoind" and to enable for system startup run "systemctl enable bitcoind"

4b) OpenRC

Rename bitcoind.openrc to bitcoind and drop it in /etc/init.d. Double check ownership and permissions and make it executable. Test it with "/etc/init.d/bitcoind start" and configure it to run on startup with "rc-update add bitcoind"

4c) Upstart (for Debian/Ubuntu based distributions)

Drop bitcoind.conf in /etc/init. Test by running "service bitcoind start" it will automatically start on reboot.

NOTE: This script is incompatible with CentOS 5 and Amazon Linux 2014 as they use old versions of Upstart and do not supply the start-stop-daemon utility.

4d) CentOS

Copy bitcoind.init to /etc/init.d/bitcoind. Test by running "service bitcoind start".

Using this script, you can adjust the path and flags to the bitcoind program by setting the BITCOIND and FLAGS environment variables in the file /etc/sysconfig/bitcoind. You can also use the DAEMONOPTS environment variable here.

1. Auto-respawn

Auto respawning is currently only configured for Upstart and systemd. Reasonable defaults have been chosen but YMMV.

*** Warning: This document has not been updated for Zcash and may be inaccurate. ***

6.21 Translation Strings Policy

This document provides guidelines for internationalization of the Bitcoin Core software.

6.21.1 How to translate?

To mark a message as translatable

- In non-GUI source code (under `src`): use `_("...")`

No internationalization is used for e.g. developer scripts outside `src`.

6.21.2 Strings to be translated

On a high level, these strings are to be translated:

- GUI strings, anything that appears in a dialog or window
- Command-line option documentation

GUI strings

Anything that appears to the user in the GUI is to be translated. This includes labels, menu items, button texts, tooltips and window titles. This includes messages passed to the GUI through the UI interface through `InitMessage`, `ThreadSafeMessageBox` or `ShowProgress`.

Command-line options

Documentation for the command line options in the output of `--help` should be translated as well.

Make sure that default values do not end up in the string, but use string formatting like `strprintf(_("Threshold for disconnecting misbehaving peers (default: %u)"), 100)`. Putting default values in strings has led to accidental translations in the past, and forces the string to be retranslated every time the value changes.

Do not translate messages that are only shown to developers, such as those that only appear when `--help-debug` is used.

6.21.3 General recommendations

Avoid unnecessary translation strings

Try not to burden translators with translating messages that are e.g. slight variations of other messages. In the GUI, avoid the use of text where an icon or symbol will do. Make sure that placeholder texts in forms don't end up in the list of strings to be translated (use `<string notr="true">`).

Make translated strings understandable

Try to write translation strings in an understandable way, for both the user and the translator. Avoid overly technical or detailed messages

Do not translate internal errors

Do not translate internal errors, or log messages, or messages that appear on the RPC interface. If an error is to be shown to the user, use a generic message, then log the detailed message to the log. E.g. "Error: A fatal internal error occurred, see debug.log for details". This helps troubleshooting; if the error is the same for everyone, the likelihood is increased that it can be found using a search engine.

Avoid fragments

Avoid dividing up a message into fragments. Translators see every string separately, so may misunderstand the context if the messages are not self-contained.

Avoid HTML in translation strings

There have been difficulties with use of HTML in translation strings; translators should not be able to accidentally affect the formatting of messages. This may sometimes be at conflict with the recommendation in the previous section.

String freezes

During a string freeze (often before a major release), no translation strings are to be added, modified or removed.

This can be checked by executing `make translate` in the `src` directory, then verifying that `bitcoin_en.ts` remains unchanged.

(note: this is a temporary file, to be added-to by anybody, and moved to release-notes at release time)

6.22 Notable changes

6.23 Zcash Contributors

Jack Grigg (558) Simon Liu (286) Sean Bowe (193) Daira Hopwood (102) Wladimir J. van der Laan (71) Taylor Hornby (65) Nathan Wilcox (56) Jay Graber (53) Jonas Schnelli (49) Kevin Gallagher (38) Cory Fields (28) Pieter

Wuille (16) syd (13) nomnombtc (9) Paige Peterson (9) fanquake (8) MarcoFalke (7) Luke Dashjr (6) Johnathan Corgan (5) Gregory Maxwell (5) Ariel Gabizon (5) kozyilmaz (4) Philip Kaufmann (4) Peter Todd (4) Patrick Strateman (4) Matt Corallo (4) Karl-Johan Alm (4) Jeff Garzik (4) David Mercer (4) Daniel Cousens (4) lpescher (3) Pavel Janík (3) João Barbosa (3) Alfie John (3) str4d (2) paveljanik (2) kpcyrd (2) aniemerg (2) Scott (2) Robert C. Seacord (2) Per Grön (2) Joe Turgeon (2) Jason Davies (2) Jack Gavigan (2) ITH4Coinomia (2) Gavin Andresen (2) Bjorn Hjortsberg (2) Amgad Abdelhafez (2) zathras-crypto (1) unsystemizer (1) practicalswift (1) mruddy (1) mrbandrews (1) kazcw (1) isle2983 (1) instagibbs (1) emilrus (1) dexX7 (1) daniel (1) calebogden (1) ayleph (1) Tom Ritter (1) Stephen (1) S. Matthew English (1) Ross Nicoll (1) René Nyffenegger (1) Pavel Vasin (1) Paul Georgiou (1) Paragon Initiative Enterprises, LLC (1) Nathaniel Mahieu (1) Murilo Santana (1) Matt Quinn (1) Louis Nyffenegger (1) Leo Arias (1) Lars-Magnus Skog (1) Kevin Pan (1) Jorge Timón (1) Jonathan "Duke" Leto (1) Jeffrey Walton (1) Ian Kelling (1) Gaurav Rana (1) Forrest Voight (1) Florian Schmaus (1) Ethan Heilman (1) Eran Tromer (1) Duke Leto (1) Daniel Kraft (1) Christian von Roques (1) Chirag Davé (1) Casey Rodarmor (1) Cameron Boehmer (1) Bryan Stitt (1) Bruno Arueira (1) Boris Hajduk (1) Bob McElrath (1) Bitcoin Error Log (1) Ariel (1) Anthony Towns (1) Allan Niemerg (1) Alex van der Peet (1) Alex (1) Adam Weiss (1) Adam Brown (1) 4ZEC (1)