zerolib Documentation

Release 0.0.0

ZeroMux

Jan 01, 2018

Contents

`zerolib` is a minimalist utility library for working with the ZeroNet protocol, which is based on MessagePack and uses deterministic elliptic curve signature. It is written for Python 3.5+

This is a highly experimental library. Before its API is stablized, developers should pay close attention to this documentation, as the public API provided by the library may be changed.

Features include:

- Digital signatures
- Hashing and data integrity checking
- TLS certificate utilities
- Packet parsing and formatting
- Connection managing
- Readers-writer locks

`zerolib` is inspired by the reference ZeroNet implementation written by shortcutme, but features more consistent API and greater flexibility.

`zerolib` is written and maintained by MuxZeroNet with help from the contributors and is licensed under the GNU General Public License version 3. If you like this project, please consider running a seed box.

Note: To avoid confusion, a **private key** is usually called a **secret key** in the documentation.

CHAPTER 1

How to install

I recommend you use `zerolib` in a virtual environment.

```
sudo apt install python3-pip python3-venv

mkdir devel
python3 -m venv devel/
cd devel
git clone https://github.com/MuxZeroNet/zerolib.git

# Super important! You must activate the virtual environment.
# The `source` command activates the virtual environment.
source ./bin/activate
which python3 && which pip

cd zerolib/
python3 -m pip install -r requirements.txt --upgrade
python3 -m pip install -r ci-requirements.txt --upgrade

python3 run_tests.py
```

CHAPTER 2

Table of contents

2.1 `zerolib.integrity` - Working with data integrity

This module provides functions and classes for signing and verifying data with appropriate abstration.

It defines the following public functions and classes.

2.1.1 Bitcoin key pair

`key_pair``()`
Generate a public key and a secret key, returning a tuple containing `(publickey, secretkey)`.

Return type (PublicKey, SecretKey)

`public_digest`(*publickey*)
Convert a public key to its `ripemd160(sha256())` digest, returning the raw 20-byte digest.

Return type bytes

2.1.2 Bitcoin address

`compute_public_address`(*publickey*)
Convert a public key to a public Bitcoin address, returning a Base58Check-encoded string.

Return type str

`compute_secret_address`(*secretkey*)
Convert a secret key to a secret Bitcoin address, returning a Base58Check-encoded string.

Return type str

`bitcoin_address``()`
Generate a public address and a secret address, returing a tuple `(public_address, secret_address)` containing two Base58Check-encoded strings.

Return type (str, str)

address_public_digest (*address*)
Convert a public Bitcoin address to its `ripemd160(sha256())` digest, returning the raw 20-byte digest.

Return type bytes

decode_secret_key (*address*)
Convert a secret Bitcoin address to a secret key, returning the secret key as a SecretKey object.

Return type SecretKey

2.1.3 Digital signature

recover_public_key (*signature*, *message*)
Recover the public key from the signature and the message, returning a PublicKey object. The recovered public key guarantees a correct signature.

Parameters

- **signature** (*bytes*) – the raw signature.
- **message** (*bytes*) – the message.

Returns a PublicKey object.

sign_data (*secretkey*, *byte_string*)
Sign the message `byte_string` with `secretkey`, returing a 65-byte serialized signature as a bytes-like string. The returned signature is compatible with ZeroNet (i.e. in the Electrum format)

Parameters

- **secretkey** (*SecretKey*) – the secret key.
- **byte_string** (*bytes*) – the message.

Returns a 65-byte binary string.

verify_data (*key_digest*, *electrum_signature*, *byte_string*)
Verify if `electrum_signature` is the signature for the message `byte_string` and is produced with the secret counterpart of `key_digest`.

Parameters

- **key_digest** (*bytes*) – the raw `ripemd160(sha256())` digest of the public key.
- **electrum_signature** (*bytes*) – the raw signature.
- **byte_string** (*bytes*) – the message.

Raises

- ***SignatureError*** – if it finds the signature forged or otherwise problematic.
- **ValueError** – if it finds the signature cannot be parsed.

2.1.4 Message digest

Note: Unless otherwise noted, `algo='sha512'` refers to the SHA-512/256 algorithm.

digest_bytes(*data*, *algo='sha512'*)
Compute the digest of `data`, a bytes-like object, returing a tuple containing `(digest, data_length)`. The first element is the raw digest. The second element is the length of the given data.

Parameters

- **data** (*bytes*) – the data to digest.
- **algo** (*str*) – the name of the digest algorithm.

Returns a two-element tuple.

Return type (bytes, int)

verify_digest_bytes(*data*, *expect_digest*, *expect_size = None*, *algo='sha512'*)
Verify if `data` have the expected digest `expect_digest` and have the expected size `expect_size`. If `expect_size` is *None*, then data size will not be checked.

Parameters

- **data** (*bytes*) – the data to digest.
- **expect_digest** (*bytes*) – the expected raw digest.
- **expect_size** (*int or None*) – the expected data size.

Raises ***DigestError*** – if the digest or size does not match.

digest_stream(*stream*, *algo='sha512'*)
Compute the digest of *stream*, a stream-like object, returning a tuple containing `(digest, stream_size)`. The first element is the raw digest. The second element is the length of the given data.

Parameters

- **stream** (*BytesIO*) – the stream to read data from and digest.
- **algo** (*str*) – the name of the digest algorithm.

Returns a two-element tuple.

Return type (bytes, int)

verify_digest_stream(*stream*, *expect_digest*, *expect_size = None*, *algo='sha512'*)
Verify if the data read from `stream` have the expected digest `expect_digest` and have the expected size `expect_size`. If `expect_size` is *None*, then stream size will not be checked.

Raises ***DigestError*** – if the digest or size does not match.

digest_file(*path*, *algo='sha512'*)
Compute the data digest of the file located at the given path. The parameter `path` should be a unicode string. Returns a tuple containing `(digest, stream_size)`. The first element is the raw digest. The second element is the length of the given data.

Parameters

- **path** (*str*) – the path to the file to read data from and digest.
- **algo** (*str*) – the name of the digest algorithm.

Returns a two-element tuple.

Return type (bytes, int)

verify_digest_file(*path*, *expect_digest*, *expect_size=None*, *algo='sha512'*)
Verify if the file at `path` has the expected digest `expect_digest` and have the expected size `expect_size`. If `expect_size` is *None*, then file size will not be checked.

Raises ***DigestError*** – if the digest or size does not match.

2.1.5 Utilities

dumps(*json_dict*, *compact=False*)

Pack the given dictionary to a JSON string, returning a unicode string. **Note that the return value is NOT a bytes-like string.**

If `compact` is *True*, the JSON string will be tightly packed. If `compact` is *False*, the keys will be sorted and the JSON object will be pretty-printed.

Parameters

- **json_dict** (*dict*) – the dictionary to stringify.
- **compact** (*bool*) – the formatting option.

Return type str

2.1.6 Exceptions

class SignatureError(*ValueError*)

class DigestError(*ValueError*)

2.2 zerolib.protocol - Helpers for common data structures

This module offers functions for parsing packets and classes representing common data structures used in the ZeroNet protocol.

It defines the following public functions and classes.

2.2.1 TLS certificate

make_cert()

Generate and return a public key PEM and a secret key PEM. The return value is a tuple `(publickey_pem, secretkey_pem)` containing the bytes of the public PEM file and the bytes of the secret PEM file.

Return type (bytes, bytes)

2.2.2 User certificate

recover_cert(*user_btc*, *portal*, *name*)

Recover the certificate from the user's Bitcoin address (string), the portal type (string) and the user's name (string). Returns the recovered certificate, as a bytes-like string.

Parameters

- **user_btc** (*str*) – the user's Bitcoin address.
- **portal** (*str*) – the portal type, usually defined by the certificate issuer.
- **name** (*str*) – the user's name.

Returns the recovered certificate.

Return type bytes

2.2.3 Routing

class Peer(*object*)
The data structure of a peer.

Variables

- **dest** (*AddrPort*) – the destination of the peer.
- **address** – property. The address of a peer, exclusing the port number. It could be an IPv4Address, an IPv6Address, an *OnionAddress*, or an *I2PAddress*.
- **port** (*int*) – property. The port number.
- **last_seen** – the time when the last request from the peer is received.
- **sites** (*set of str*) – the set of sites the peer is hosting.
- **score** (*int*) – the score rating of the peer.

__init__(*self*, *dest*, *last_seen*, *sites = None*, *dht = None*, *score = None*)

2.2.4 Packets

unpack_stream(*stream*, *sender = None*)
Unpack a stream, and indicate that it was sent from the sender. Only unpacks one packet at a time.

Parameters

- **stream** (*BytesIO*) – the stream to read data from and unpack.
- **sender** (AddrPort or None) – where the packet is from.

Raises

- **KeyError** – when a key it is looking for is missing from the packet.
- **TypeError** – when the type of a value is wrong and cannot be accepted.
- **ValueError** – when a value looks wrong.

unpack_dict(*packet*, *sender = None*)
Unpack a dictionary, and indicate that it was sent from sender.

Parameters

- **packet** (*dict*) – the dictionary to unpack.
- **sender** (AddrPort or None) – where the packet is from.

Raises

- **KeyError** – when a key it is looking for is missing from the packet.
- **TypeError** – when the type of a value is wrong and cannot be accepted.
- **ValueError** – when a value looks wrong.

unpack(*data*, *sender = None*)
Unpack a byte string, and indicate that it was sent from a network address. Only unpacks one packet at a time.

Parameters

- **data** (*bytes*) – the data to unpack.
- **sender** (AddrPort or None) – where the packet is from.

Raises

- **KeyError** – when a key it is looking for is missing from the packet.
- **TypeError** – when the type of a value is wrong and cannot be accepted.
- **ValueError** – when a value looks wrong.

class AddrPort(*object*)

A named `(address, port)` tuple.

Variables

- **address** – could be an IPv4Address, an IPv6Address, an *OnionAddress*, or an *I2PAddress*.
- **port** (*int*) – the port number.

class OnionAddress(*Address*)

A Tor Onion Service address, either v2 or v3.

packed

The packed representation of the address, either 10 bytes or 35 bytes long.

__str__(*self*)

Returns the human readable, base-32 encoded representation of the address, with the `.onion` suffix.

Return type str

class I2PAddress(*Address*)

An I2P address, the SHA-256 hash of an I2P Destination.

packed

The packed representation of the address, a SHA-256 hash.

__str__(*self*)

Returns the human readable, base-32 encoded representation of the address, with the `.b32.i2p` suffix.

Return type str

class Packet(*object*)

The base class for a packet. Every class below for parsed packets is inherited from this base class.

Variables

- **req_id** (*int*) – the request ID (sequence number) as indicated on the packet. Since the value of this attribute is taken directly from the packet, request ID is for reference purposes only.
- **sender** (*AddrPort or None*) – where the packet is from.

See also:

A full page of parsed packets

class PrefixIter(*object*)

The base class for a packet that has the `prefixes` attribute. It provides helper methods for easier iteration through the prefixes.

__iter__(*self*)

__contains__(*self*, *item*)

A packet class inherited from *PrefixIter* supports iteration.

```
>>> from protocol import unpack_dict
>>> packet = unpack_dict({b'cmd': b'response', b'to': 0,
... b'hashfield_raw': b'\x10\x11ABCDef12'})
>>> packet
<protocol.packets.RespHashSet object at 0x7fc6b1b5ad58>
>>> iter(packet)
<set_iterator object at 0x7fc6b3753990>
>>> list(iter(packet))
[b'\x10\x11', b'12', b'ef', b'AB', b'CD']
>>> b'\x10\x11' in packet
True
>>> b'\xA0\xB1' in packet
False
```

class PacketInterp(*object*)

The packet interpreter. This state machine is used to figure out the contextual meaning of each response packet and translate it. Consider the following example.

```
>>> from protocol import unpack_dict, PacketInterp
>>> request = unpack_dict({b'req_id': 0, b'cmd': b'actionCheckport',
... b'params': {b'port': 15441}})
>>> response = unpack_dict({b'cmd': b'response', b'to': 0,
... b'status': b'open', b'ip_external': b'1.2.3.4'})
>>> request
<protocol.packets.CheckPort object at 0x7f71ca453cc8>
>>> response
<protocol.packets.RespPort object at 0x7f71c9cd2948>
>>> request.port
15441
>>> response.open
True
>>> response.port
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: port
>>>
>>> state_machine = PacketInterp()
>>> state_machine.register(request)
>>> state_machine.interpret(response)
>>> response.port
15441
```

register(*self*, *packet*)

Register a request packet. If the packet is a symmetrical packet, or is not a request packet, do nothing.

interpret(*self*, *packet*)

Interpret a response packet and inject necessary atrtibutes into the packet instance. After that, the response packet and the corresponding request packet will be forgotten by the packet interpreter.

If the packet is a symmetrical packet, or is not a response packet, do nothing.

Raises

- **TypeError** – when the type of the packet is unexpected.
- **KeyError** – when it cannot find any registered request packet that has the same sequence number.

static new_id()
Returns a new usable sequence number. The sequence number is a random unsigned 32-bit integer.

See also:

What are asymmetrical packets and why?

2.3 zerolib.protocol - Full list of packets

Unlike BitTorrent, not all ZeroNet packets are symmetrical. This page lists all documented ZeroNet packets in two categories.

2.3.1 Symmetrical Packets

Every symmetrical packet contains enough information which the program can interpret without referring to the previous request packets.

class Handshake(*Packet*)
Unpacked `handshake` packet sent when the connection was initialized.

Variables

- **crypto_set** (*set of str*) – the set of supported cryptographic algorithms.
- **peer_id** (*bytes or None*) – the peer ID as a **binary string**. *(not available in Tor mode)*
- **port** (*int*) – the port number the sender is actively listening on.
- **open** (*bool*) – whether the sender believes his port is open.
- **onion** (*AddrPort or None*) – the Tor Onion Service destination of the peer. *(only used in Tor mode)*
- **onion_address** (*OnionAddress or None*) – the Tor Onion Service address of the peer, excluding the port. *(only used in Tor mode)*
- **protocol** (*str*) – a unicode string representing the protocol version.
- **version** (*str*) – the version string of the sender's software.
- **rev** (*int*) – the rev number of the sender's software.

class ACK(*Handshake*)
Response packet of *Handshake*. Marks the end of a handshake.

Variables **preferred_crypto** (*str*) – the cryptographic algorithm that the sender would like to use.

... as well as the attributes inherited from *Handshake*.

class Ping(*Packet*)

class Pong(*Packet*)

2.3.2 Asymmetrical Packets

An asymmetrical response packet itself does not contain enough information. To fully interpret an asymmetrical response, the program has to refer to its previous requests.

class PEX(*Packet*)
Unpacked `pex` packet that exchanges peers with the client. Peers are parsed at construct-time.

Variables

- **site** (*str*) – the site address, a human-readable Bitcoin address.
- **need** (*int*) – the number of peers the sender needs.
- **peers** (set of *AddrPort*) – clearnet peers.
- **onions** (set of *AddrPort*) – Tor Onion Service peers.
- **garlics** (set of *AddrPort*) – I2P Hidden Service peers.

class RespPEX(*Packet*)
Response packet of *PEX*.

Variables

- **peers** (set of *AddrPort*) – clearnet peers.
- **onions** (set of *AddrPort*) – Tor Onion Service peers.
- **garlics** (set of *AddrPort*) – I2P Hidden Service peers.

The following variables will be injected when the packet is handled by the state machine.

Variables **site** (*str*) – the site address, a human-readable Bitcoin address.

class GetFile(*Packet*)
Unpacked `getFile` packet that requests for a file.

Variables

- **site** (*str*) – the site address, a human-readable Bitcoin address.
- **inner_path** (*str*) – the relative path to the requested file.
- **offset** (*int*) – request file from this offset.
- **total_size** (*int or None*) – the total size of the requested file. *(optional)*

class RespFile(*Packet*)
Response packet of *GetFile*.

Variables

- **body** (*bytes*) – a chunk of file content.
- **last_byte** (*int*) – the absolute offset of the last byte of `body`.
- **total_size** (*int*) – the total size of the whole file.
- **offset** (*int*) – property. The absolute offset of the first byte of `body`.
- **next_offset** (*int*) – property. The start offset of the next `getFile` request.

The following variables will be injected when the packet is handled by the state machine.

Variables

- **site** (*str*) – the site address, a human-readable Bitcoin address.
- **inner_path** (*str*) – the relative path to the requested file.

class ListMod(*Packet*)
Unpacked `listModified` packet that requests for the paths of `content.json` files modified since the given time. This packet is used to heuristically list a site's new user content.

Variables

- **site** (*str*) – the site address, a human-readable Bitcoin address.
- **since** (*int*) – list modified `content.json` files since this timestamp. The timestamp is in seconds.

> **Warning:** This timestamp is defined vaguely in the spec. Is it an int or a float? Link to the spec.

class RespMod(*Packet*)
Response packet of *ListMod*.

Variables timestamps (*dict of str and int*) – the `{inner_path : mtime}` dictionary.

__iter__(*self*)

__contains__(*self*, *key*)

items(*self*)
Helper methods for iterating through the `timestamps`.

```
for (inner_path, mtime) in packet.items():
    print('New file %r, last modified %d' % (inner_path, mtime))
```

The following variables will be injected when the packet is handled by the state machine.

Variables site (*str*) – the site address, a human-readable Bitcoin address.

class GetHash(*Packet*)
Unpacked `getHashfield` packet that requests for the client's list of downloaded optional file IDs.

Variables site (*str*) – the site address, a human-readable Bitcoin address.

class RespHashSet(*Packet*, *PrefixIter*)
Response packet of *GetHash*.

Variables prefixes (*set of bytes*) – hash ID prefixes in a set.

The following variables will be injected when the packet is handled by the state machine.

Variables site (*str*) – the site address, a human-readable Bitcoin address.

class FindHash(*Packet*)
Unpacked `findHashIds` packet that asks if the client knows any peer that has the said optional file IDs.

Variables

- **site** (*str*) – the site address, a human-readable Bitcoin address.
- **prefixes** (*set of bytes*) – the set of optional file IDs. An optional file ID is the first 2 bytes of the file's hash.

class RespHashDict(*Packet*)
Response packet of *FindHash*.

class SetHash(*Packet*)
Unpacked `setHashfield` packet that announces the sender's list of optional file IDs.

Variables

- **site** (*str*) – the site address, a human-readable Bitcoin address.

- **prefixes** (*set of bytes*) – the set of optional file IDs. An optional file ID is the first 2 bytes of the file's hash.

class Predicate(*Packet*)
Status predicate. Either an `ok` packet or an `error` packet. Response packet of *Update* and *SetHash*.

Variables **ok** (*bool*) – Okay?

class Update(*Packet*)
Unpacked `update` packet that pushes a new site file.

Its response packet is a *Predicate*.

class CheckPort(*Packet*)
Unpacked `actionCheckport` packet that asks the client to check the sender's port status.

Variables **port** (*int*) – the port number which the sender would like you to check.

class RespPort(*Packet*)
Response packet of *CheckPort*.

Variables

- **status** (*str*) – port status as a human-readable string.
- **open** (*bool*) – whether the port is open.

The following variables will be injected when the packet is handled by the state machine.

Variables **port** (*int*) – the port number which the sender would like you to check.

2.4 zerolib.nettools - Higher level network I/O utilities

The `zerolib.nettools` module defines functions and classes which help in interacting with remote peers in the network.

2.4.1 Connection

class Connections(*object*)
The connection manager.

__init__(*self*, *capacity=200*, *clean_func = None*)

register(*self*, *dest*, *socket*)

__getitem__(*self*, *key*)

__delitem__(*self*, *key*)

__contains__(*self*, *key*)

__iter__(*self*)

__len__(*self*)

2.5 Discussion

2.5.1 What are asymmetrical packets and why?

When messages sent in both directions look the same, we call these messages symmetrical. BitTorrent packets are symmetrical. A response is a request. A request is a directive. The reason behind this design choice is to avoid using a sequence number, even when packets can be lost, duplicated or received not in order.

Unlike BitTorrent, not all ZeroNet packets are symmetrical. To fully interpret an asymmetrical response packet, a computer program has to refer to its previous request packets.

The reason behind this design decision is to minimize data transfer. For example, the same Bitcoin address that appears in the request is omitted in the response, so that a computer program need not receive and parse the same Bitcoin address twice. However, this design decision makes every implementation of this protocol rely on a state machine.

Index

Symbols

A

B

C

D

F

G

H

I

K

L

M

N

O

P

R

S

U

V