# sphinxmix Documentation

***Release 0.0.7***

**George Danezis (UCL)**

**Jan 04, 2018**

# Contents

This documentation relates to the *sphinxmix* package version 0.0.7.

CHAPTER 1

# Installing

Install using *pip* through the command:

```
$ pip install sphinxmix
```

CHAPTER 2

# Basic usage

The sphinxmix package implements the Sphinx mix packet format core cryptographic functions.

The paper describing sphinx may be found here:

- George Danezis and Ian Goldberg. Sphinx: A Compact and Provably Secure Mix Format. IEEE Symposium on Security and Privacy 2009. [link]

All the sphinxmix cryptography is encapsulated and within a SphinxParams object that is used by all subsequent functions. To make sphinxmix use different cryptographic primitives simply extend this class, or re-implement it. The default cryptographic primitives are NIST/SEGS-p224 curves, AES and SHA256.

## 2.1 Sending Sphinx messages

To package or process sphinx messages create a new SphinxParams object:

```
>>> # Instantiate a the crypto parameters for Sphinx.
>>> from sphinxmix.SphinxParams import SphinxParams
>>> params = SphinxParams()
```

The sphinxmix package requires some rudimentary Public Key Information: mix nodes need an identifier created by Nenc and the PKI consists of a dictionary mapping node names to pki_entry records. Those include secret keys (derived using gensecret) and public keys (derived using expon).

```
>>> # The minimal PKI involves names of nodes and keys
>>> from sphinxmix.SphinxClient import pki_entry, Nenc
>>> pkiPriv = {}
>>> pkiPub = {}
>>> for i in range(10):
...     nid = i
...     x = params.group.gensecret()
...     y = params.group.expon(params.group.g, [ x ])
...     pkiPriv[nid] = pki_entry(nid, x, y)
...     pkiPub[nid] = pki_entry(nid, None, y)
```

A client may package a message using the Sphinx format to relay over a number of mix servers. The function `rand_subset` may be used to select a random number of node identifiers; the function `create_forward_message` can then be used to package the message, ready to be sent to the first mix. Note both destination and message need to be `bytes`.

```
>>> # The simplest path selection algorithm and message packaging
>>> from sphinxmix.SphinxClient import rand_subset, \
...                                    create_forward_message
>>> use_nodes = rand_subset(pkiPub.keys(), 5)
>>> nodes_routing = list(map(Nenc, use_nodes))
>>> keys_nodes = [pkiPub[n].y for n in use_nodes]
>>> dest = b"bob"
>>> message = b"this is a test"
>>> header, delta = create_forward_message(params, nodes_routing, \
...     keys_nodes, dest, message)
```

The client may specify any information in the `nodes_routing` list, that will be passed to intermediate mixes. At a minimum this should include information about the next mix.

## 2.2 Processing Sphinx messages at a mix

The heart of a Sphinx mix server is the `sphinx_process` function, that takes the server secret and decodes incoming messages. In this example the message encode above, is decoded by the sequence of mixes.

```
>>> # Process message by the sequence of mixes
>>> from sphinxmix.SphinxClient import PFdecode, Relay_flag, Dest_flag, Surb_flag,␣
↪receive_forward
>>> from sphinxmix.SphinxNode import sphinx_process
>>> x = pkiPriv[use_nodes[0]].x
>>> while True:
...     ret = sphinx_process(params, x, header, delta)
...     (tag, info, (header, delta), mac_key) = ret
...     routing = PFdecode(params, info)
...     if routing[0] == Relay_flag:
...         flag, addr = routing
...         x = pkiPriv[addr].x
...     elif routing[0] == Dest_flag:
...         assert receive_forward(params, mac_key, delta) == [dest, message]
...         break
```

It is the responsibility of a mix to record `tags` of messages to prevent replay attacks. The `PFdecode` function may be used to recover routing Information including the next mix, or any other user specified information.

## 2.3 Single use reply Blocks

A facility provided by Sphinx is the creation and use of Single Use Reply Blocks (SURB) to route messages back to an anonymous receipient. First a receiver needs to create a SURB using `create_surb` and passes on the `nymtuple` structure to the sender, and storing `surbkeytuple` keyed by the identifier `surbid`:

```
>>> from sphinxmix.SphinxClient import create_surb, package_surb
>>> surbid, surbkeytuple, nymtuple = create_surb(params, nodes_routing, keys_nodes, b
↪"myself")
```

Using the `nymtuple` a sender can package a message to be sent through the network, starting at the `nymtuple[0]` router:

```
>>> message = b"This is a reply"
>>> header, delta = package_surb(params, nymtuple, message)
```

The network processes the SURB as any other message, until it is received by the last mix in the path:

```
>>> x = pkiPriv[use_nodes[0]].x
>>> while True:
...     ret = sphinx_process(params, x, header, delta)
...     (tag, B, (header, delta), mac_key) = ret
...     routing = PFdecode(params, B)
...
...     if routing[0] == Relay_flag:
...         flag, addr = routing
...         x = pkiPriv[addr].x
...     elif routing[0] == Surb_flag:
...         flag, dest, myid = routing
...         break
```

The final mix server must sent the `myid` and `delta` to the destination `dest`, where it may be decoded using the `surbkeytuple`.

```
>>> from sphinxmix.SphinxClient import receive_surb
>>> received = receive_surb(params, surbkeytuple, delta)
>>> assert received == message
```

## 2.4 Embedding arbitrary information for mixes

A sender may embed arbitrary information to mix nodes, as demonstrated by embedding `b'info'` to each mix, and `b'final_info'` to the final mix:

```
>>> use_nodes = rand_subset(pkiPub.keys(), 5)
>>> nodes_routing = [Nenc((n, b'info')) for n in use_nodes]
>>> keys_nodes = [pkiPub[n].y for n in use_nodes]
>>> dest = (b"bob", b"final_info")
>>> message = b"this is a test"
>>> header, delta = create_forward_message(params, nodes_routing, \
...     keys_nodes, dest, message)
```

Mixes decode the arbitrary structure passed by the clients, and can interpret it to implement more complex mixing strategies:

```
>>> x = pkiPriv[use_nodes[0]].x
>>> while True:
...     ret = sphinx_process(params, x, header, delta)
...     (tag, info, (header, delta), mac_key) = ret
...     routing = PFdecode(params, info)
...     if routing[0] == Relay_flag:
...         flag, (addr, additional_info) = routing
...         assert additional_info == b'info'
...         x = pkiPriv[addr].x
...     elif routing[0] == Dest_flag:
...         [[dest, additional_info], msg] = receive_forward(params, mac_key, delta)
...         assert additional_info == b'final_info'
```

```
...         assert dest == b'bob'
...         break
```

## 2.5 Packaging mix messages to byte strings:

The *sphinxmix* package provides functions *pack_message* and *unpack_message* to serialize and deserialize mix messages using *msgpack*. Some meta-data about the parameter length are passed along the message any may be used to select an appropriate parameter environment for the decoding of the message.

```
>>> from sphinxmix.SphinxClient import pack_message, unpack_message
>>> bin_message = pack_message(params, (header, delta))
>>> param_dict = { (params.max_len, params.m):params }
>>> px, (header1, delta1) = unpack_message(param_dict, bin_message)
```

CHAPTER 3

# Development

The git repository for `sphinxmix` can be cloned from here: https://github.com/UCL-InfoSec/sphinx

The `pytest` unit tests and doctests of `sphinxmix` may be ran using `tox` simply through the command:

```
$ tox
```

To upload a new distribution of `sphinxmix` the maintainer simply uses:

```
$ python setup.py sdist upload
```

CHAPTER 4

# Core classes and functions

**class** sphinxmix.SphinxParams.**SphinxParams**(*group=None*, *header_len=192*, *body_len=1024*, *assoc_len=0*, *k=16*, *dest_len=16*)

**class** sphinxmix.SphinxParams.**Group_ECC**(*gid=713*)
Group operations in ECC

**class** sphinxmix.SphinxParamsC25519.**Group_C25519**
Group operations using Curve 25519

## 4.1 Utility functions

sphinxmix.SphinxClient.**pki_entry**(*id*, *x*, *y*)
A helper named tuple to store PKI information.

sphinxmix.SphinxClient.**Nenc**(*idnum*)
The encoding of mix names.

sphinxmix.SphinxClient.**Relay_flag = '\xf0'**
Routing flag indicating message is to be relayed.

sphinxmix.SphinxClient.**Dest_flag = '\xf1'**
Routing flag indicating message is to be delivered.

sphinxmix.SphinxClient.**Surb_flag = '\xf2'**
Routing flag indicating surb reply is to be delivered.

sphinxmix.SphinxClient.**PFdecode**(*param*, *packed*)
Decoder of prefix free encoder for commands received by mix or clients.

sphinxmix.SphinxClient.**rand_subset**(*lst*, *nu*)
Return a list of nu random elements of the given list (without replacement).

## 4.2 Client functions

sphinxmix.SphinxClient.**create_forward_message**(*params*, *nodelist*, *keys*, *dest*, *msg*, *assoc=None*)
Creates a forward Sphix message, ready to be processed by a first mix.

It takes as parameters a node list of mix information, that will be provided to each mix, forming the path of the message; a list of public keys of all intermediate mixes; a destination and a message; and optinally an array of associated data (byte arrays).

sphinxmix.SphinxClient.**receive_forward**(*params*, *mac_key*, *delta*)
Decodes the body of a forward message, and checks its MAC tag.

## 4.3 Packaging messages

sphinxmix.SphinxClient.**pack_message**(*params*, *m*)
A method to pack mix messages.

sphinxmix.SphinxClient.**unpack_message**(*params_dict*, *m*)
A method to unpack mix messages.

## 4.4 Mix functions

sphinxmix.SphinxNode.**sphinx_process**(*params*, *secret*, *header*, *delta*, *assoc=''*)
The heart of a Sphinx server, that processes incoming messages. It takes a set of parameters, the secret of the server, and an incoming message header and body. Optinally some Associated data may also be passed in to check their integrity.

## 4.5 SURB functions

sphinxmix.SphinxClient.**create_surb**(*params*, *nodelist*, *keys*, *dest*, *assoc=None*)
Creates a Sphinx single use reply block (SURB) using a set of parameters; a sequence of mix identifiers; a pki mapping names of mixes to keys; and a final destination. An array of associated data, for each mix on the path, may optionally be passed in.

**Returns:**

- A triplet (surbid, surbkeytuple, nymtuple). Where the surbid can be used as an index to store the secrets surbkeytuple; nymtuple is the actual SURB that needs to be sent to the receiver.

sphinxmix.SphinxClient.**package_surb**(*params*, *nymtuple*, *message*)
Packages a message to be sent with a SURB. The message has to be bytes, and the nymtuple is the structure returned by the create_surb call.

Returns a header and a body to pass to the first mix.

sphinxmix.SphinxClient.**receive_surb**(*params*, *keytuple*, *delta*)
Processes a SURB body to extract the reply. The keytuple was provided at the time of SURB creation, and can be indexed by the SURB id, which is also returned to the receiving user.

Returns the decoded message.

CHAPTER 5

# Ultrix Format

## 5.1 Mix functions

sphinxmix.UltrixNode.**ultrix_process**(*params*, *secret*, *header*, *delta*, *assoc=''*)

The heart of a Ultrix server, that processes incoming messages. It takes a set of parameters, the secret of the server, and an incoming message header and body. Optinally some Associated data may also be passed in to check their integrity.

## 5.2 Client functions

sphinxmix.UltrixClient.**create_forward_message**(*params*, *nodelist*, *keys*, *dest*, *msg*, *assoc=None*)

Creates a forward Sphix message, ready to be processed by a first mix.

It takes as parameters a node list of mix information, that will be provided to each mix, forming the path of the message; a list of public keys of all intermediate mixes; a destination and a message; and optinally an array of associated data (byte arrays).

sphinxmix.UltrixClient.**receive_forward**(*params*, *header*, *mac_key*, *routing*, *delta*)

Decodes the body of a forward message, and checks its MAC tag.

## 5.3 SURB functions

sphinxmix.UltrixClient.**create_surb**(*params*, *nodelist*, *keys*, *dest*, *assoc=None*)

Creates a Ultrix single use reply block (SURB) using a set of parameters; a sequence of mix identifiers; a pki mapping names of mixes to keys; and a final destination. An array of associated data, for each mix on the path, may optionally be passed in.

**Returns:**

- A triplet (surbid, surbkeytuple, nymtuple). Where the surbid can be used as an index to store the secrets surbkeytuple; nymtuple is the actual SURB that needs to be sent to the receiver.

sphinxmix.UltrixClient.**package_surb**(*params*, *nymtuple*, *message*)

Packages a message to be sent with a SURB. The message has to be bytes, and the nymtuple is the structure returned by the create_surb call.

Returns a header and a body to pass to the first mix.

sphinxmix.UltrixClient.**decode_surb**(*params*, *header*, *enc_dest*)

Decode the destination address of the SURB.

sphinxmix.UltrixClient.**receive_surb**(*params*, *keytuple*, *delta*)

Processes a SURB body to extract the reply. The keytuple was provided at the time of SURB creation, and can be indexed by the SURB id, which is also returned to the receiving user.

Returns the decoded message.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## S

# Index