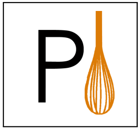
Ref. Ares(2018)595056 - 31/01/2018



David Stainton (CCT)
Vincent Breitmoser (CCT)
Harry Halpin (Greenhost/LEAP))
Kali Kaneko (Greenhost/LEAP)
Ruben Pollan (Greenhost/LEAP)
Claudia Diaz (KUL)
Tariq Elahi (KUL)
George Danezis (UCL)
Ania Piotrowska (UCL)

Open-source code of integrated system for desktops

Deliverable D7.2

January 31, 2018
PANORAMIX Project, # 653497, Horizon 2020
http://www.panoramix-project.eu

Horizon 2020
European Union funding
for Research & Innovation

Revision History

Revision	Date	Author(s)	Description
0.1	2018-1-24	HH (GH)	First Draft for review
0.9	2018-1-31	HH (GH))	Edited due to internal review
1.0	2018-1-31	MW (UEDIN)	Final review and submission to the EC

Executive Summary

This deliverable overviews the state of development of the Panoramix mix-net as deployed for messaging. We overview how, using the command-line, a new mix-net node can be set-up as well as the LEAP provider (server) to send and receive messages. The LEAP providers can act as entry and exit nodes to the mix net. We also provide instructions for installing a desktop client, *Bitmask* (using the Pixelated mail user agent), and a mobile client for Android, *K-9 mail*. Lastly, we include specifications of the core mix-net functions needed for messaging that have been implemented in Go. This includes the overall architecture, the PKI infrastructure, the Sphinx message format, end-to-end protocol between client and provider, and wire protocol. Therefore, this deliverable presents the first working version of the Panoramix mix-net for messaging, and further development based on user-feedback and new research will continue to the end of the project.

Contents

Executive Summary **5**

1 Introduction **11**

2 Server-side Instructions for Sysadmins **13**
2.1 How to Set Up Your Own Katzenpost Mixnet . . . 13
2.2 Setting up a Mix node . . . 13
2.2.1 Set up the PKI . . . 13
2.2.2 Set up the Mix . . . 14
2.2.3 Add Users to the Provider . . . 14
2.2.4 Run the Authority . . . 14
2.3 Installing a LEAP-enabled Server . . . 14
2.3.1 Introduction . . . 14
2.3.2 Prepare your workstation . . . 15
2.3.3 Create a provider instance . . . 16
2.3.4 Add a node to the provider . . . 17
2.3.5 Deploy your provider . . . 18
2.3.6 Test that things worked correctly . . . 19
2.3.7 Create an administrator . . . 19
2.3.8 Add an end-user service . . . 20
2.4 Setting up a Mixnet-enabled LEAP mail server . . . 20
2.4.1 Add email services to the node . . . 20
2.4.2 Deploy to the node . . . 20
2.4.3 Setup DNS . . . 21
2.4.4 Test it out . . . 21

3 Open-Source Desktop Client **23**
3.1 Installing Bitmask . . . 23
3.2 Installing Pixelated with Mix-net . . . 24
3.3 Turning On Mix-net . . . 25
3.4 Mix-net integration description . . . 26

4 Open-Source Mobile Client **27**
4.1 Katzenpost/Mail Android . . . 27
4.2 Build Instructions . . . 28
4.2.1 Prepare Toolchain . . . 28
4.2.2 Prepare Gomobile . . . 28
4.2.3 Prepare Repository and Compile Native Bindings . . . 29
4.2.4 Compile application . . . 29
4.3 Mix-net integration description . . . 29

5 Mix Network for E-mail Architecture Specification **31**
5.1 Introduction . . . 31
5.1.1 Terminology . . . 31
5.2 System Overview . . . 32
5.2.1 Threat Model . . . 32
5.2.2 Network Topology . . . 33
5.3 Packet Format Overview . . . 33
5.3.1 Sphinx Cryptographic Primitives . . . 33
5.3.2 Sphinx Packet Parameters . . . 34
5.3.3 Sphinx Per-hop Routing Information Extensions . . . 34
5.4 Mix Node Operation . . . 35
5.4.1 Link Layer Connection Management . . . 35
5.4.2 Sphinx Mix and Provider Key Rotation . . . 35
5.4.3 Sphinx Packet Processing . . . 36
5.5 Anonymity Considerations . . . 37
5.5.1 Topology . . . 37
5.5.2 Mixing strategy . . . 37
5.6 Security Considerations . . . 37

6 Public Key Infrastructure Specification **39**
6.1 Introduction . . . 39
6.1.1 Terminology . . . 39
6.1.2 Security Properties Overview . . . 40
6.1.3 Differences from Tor's and Mixminion's Directory Authority systems . . . 40
6.2 Overview of Mix PKI Interaction . . . 41
6.2.1 Directory Authority Server Schedule . . . 41
6.2.2 Scheduling Mix Downtime . . . 42
6.3 Voting for Consensus Protocol . . . 42
6.3.1 Protocol Messages . . . 42
6.3.2 Vote Exchange . . . 43
6.3.3 Vote Tabulation for Consensus Computation . . . 43
6.3.4 Signature Collection . . . 43
6.3.5 Publication . . . 44
6.4 PKI Protocol Data Structures . . . 44
6.4.1 Mix Descriptor Format . . . 44
6.4.2 Directory Format . . . 44
6.5 PKI Wire Protocol . . . 45
6.5.1. Retrieving a directory . . . 45
6.5.2. Publishing a mix descriptor . . . 45

7 Sphinx Message Format Specification **47**
7.1 Introduction . . . 47
7.1.1 Terminology . . . 47
7.2 Cryptographic Primitives . . . 48
7.2.1 Sphinx Key Derivation Function . . . 48
7.3 Sphinx Packet Parameters . . . 49
7.3.1 Sphinx Parameter Constants . . . 49
7.3.2 Sphinx Packet Geometry . . . 49
7.4 The Sphinx Cryptographic Packet Structure . . . 50
7.4.1 Sphinx Packet Header . . . 50
7.4.2 Sphinx Packet Payload . . . 52
7.5 Sphinx Packet Creation . . . 52

7.5.1 Create a Sphinx Packet Header 52
7.5.2 Create a Sphinx Packet 55
7.6 Sphinx Packet Processing 56
7.6.1 Sphinx_Unwrap Operation 56
7.6.2 Post Sphinx_Unwrap Processing 58
7.7 Single Use Reply Block (SURB) Creation 59
7.7.1 Create a Sphinx SURB and Decryption Token 60
7.7.2 Decrypt a Sphinx Reply Originating from a SURB 60
7.8 Single Use Reply Block Replies 61
7.9 Anonymity Considerations 62
7.9.1 Optional Non-constant Length Sphinx Packet Header Padding 62
7.9.2 Additional Data Field Considerations 62
7.9.3 Forward Secrecy Considerations 62
7.9.4 Compulsion Threat Considerations 62
7.9.5 SURB Usage Considerations for Volunteer Operated Mix Networks 63
7.10 Security Considerations 63
7.10.1 Sphinx Payload Encryption Considerations 63

8 Mix Network End-to-end Protocol Specification 65
8.1 Introduction 65
8.1.1. Terminology 65
8.1.2 Constants 66
8.2 Mix Network Packet Format Considerations 66
8.3 Client and Provider Core Protocol 66
8.3.1 Handshake and Authentication 66
8.3.2 Client Retrieval of Queued Messages 66
8.4 Client and Provider processing of received packets 68
8.4.1 Provider Behavior for Receiving Messages from the Mix Network 71
8.4.2 Client Receive Message Behavior 71
8.5 Sphinx Packet Composition Considerations 72
8.5.1 Choosing Delays: for single Block messages and for multi Block messages . 72
8.5.2 Path selection algorithm 73
8.6 E-mail Client Integration Considerations 73
8.6.1 Message Retrieval 73
8.6.2 Message Sending 74
8.7 Client Integration Considerations 74
8.7.1 Message Retrieval 74
8.7.2 Information available to clients 74
8.8 Anonymity Considerations 74
8.9 Security Considerations 75

9 Mix Network Wire Protocol Specification 77
9.1 Introduction 77
9.1.1 NewHope-Simple Key Encapsulation Mechanism 77
9.2 Core Protocol 77
9.2.1 Handshake Phase 78
9.2.2 Data Transfer Phase 79
9.3 Predefined Commands 80
9.3.1 The no_op Command 80
9.3.2 The disconnect Command 80
9.3.3 The send_packet Command 80
9.4 Anonymity Considerations 81

9.5 Security Considerations 81

10 Bibliography **83**

1. Introduction

This deliverable is a guide to the code available on GitHub, which is composed of a number of specifications that describe the code. This guide allows system administrators to deploy the mix-networking infrastructure for email, and instructions for users describe clients for desktop and mobile (Android).

This deliverable completes *Task 7.2 Integration with LEAP software infrastructure* and *Task 7.3 Integration of LEAP software with Mobile infrastructure.* This does not mean that the software development is complete, as the software will continue to evolve until the end of the *Testing, Validation and Deployment to End-Users* in response to user feedback and bug reports. Greenhost and LEAP developers have been using weekly scrums in both voice and IRC (Internet Relay Chat) to co-ordinate its spatially disparate programming team. At each meeting the developers determine the tasks needed and assign programming resources as appropriately as possible. This process will continue into Task 7.3. Furthermore, in order to speed up the programming process, there have also been weekly meetings between CCT, GH, KUL, and UCL to write the specifications. Each step in the integration, the necessary steps have been documented in Github, rather than wiki, as Github provides superior integration into code and task management. Lastly, in order to increase the productivity of the programming and co-ordination, there have been joint coding meetings in Athens (Dec 10-17 2017) and Brussels (Jan 28-Feb 3 2018) between CCT and GH, and GRNET attended the Athens meeting to focus on the integration of the generic mix-net API from WP4.

All produced code is open source, and available for re-use under the GNU General Public License (GPL). The code specifications are attached in all the relevant chapters. Chapter 5 describes the overall mix-net design for messaging. Chapter 6 describes the public-key infrastructure. Chapter 7 describes the message format used by the mix network. Chapter 8 explains the integration of the mix-net library into the client and server libraries for LEAP and K-9 mobile. Chapter 9 specifies the protocol used on the wire protocol.

The code for messaging with Panoramix has multiple components. The mobile-client under development is a fork of the Android *K-9* mail. The desktop version can be ran as a SMTP local proxy via *mailproxy* for simple use-cases. For higher security, and secure e-mail delivery and multi-device synchronization, the *Bitmask* client. Different user-agents can be used with *Bitmask*, and our current integrated client is *Pixelated*, although we hope to branch out after initial user-testing into other clients like *Thunderbird.* The core code libraries that implement the messaging-specific code and complement the core Panoramix APIs are called *Katzenpost* to distinguish them from the EC project and the core Panoramix APIs given in D4.3.

In Chapter 2, this report integrates into the server-side “puppet” scripts that allow easy deployment of the infrastructure on Linux-based architectures, and these scripts will have to enable the padding and mixing operations whose parameters are determined due to data gathered in D7.1 and subsequent work to be reported in D7.3.

Although not strictly required, as the software - by virtue of running a local SMTP proxy -

automatically integrates into existing email clients and the operation of the mix networking infrastructure should be invisible to users (except for those using advanced options), the integration of various processing states and associated error codes of the generic mix networking into the larger LEAP infrastructure is explained. In Chapter 3, we demonstrate how we integrate the infrastructure into the Python-based client software, which takes advantage of LEAP's unique key exchange and user management software, Soledad, that allows privacy preferences and identity information to be synchronized across multiple devices and operating systems.

In Chapter 4, an enhanced privacy mode is being built as an integrated feature branch to the K-9 Mail app. K-9 is a mature open source e-mail client for the Android platform, with a reported user base of a million active users, and support for end-to-end encryption via OpenPGP. It is suitable as a base for privacy-enhanced communication via e-mail, and self-supporting in the long term due to its established developer community. Support for the mix-net will be developed as a routing option that is automatically used if support is indicated by the user's MTA (Message Transfer Agent), adding only minimal settings and visual indicators of the employed routing strategy to the user interface. This optimizes for a seamless user experience, which is unintrusive while still providing enhanced privacy properties through the Panoramix mix-net. In preparation for Task 7.4, the K-9 user interface has been evaluated with a focus on usability, and subsequently improved in regard to the identified pain points in order to create a mix-net enabled e-mail client with a modern, well-rounded user experience.

To reiterate:

1. **Introduction**: Outline of the mix networking architecture
2. **Deployment Guide for Sysadmins on Server**: Guide to setting up a mix-net enabled node and LEAP server-side code.
3. **Open-Source Desktop Client**: Guide to LEAP desktop client.
4. **Open-Source Mobile Client**: Guide to K-9 client.
5. **Mix Network for E-mail Architecture Specification**: Specification for the overall mix-net design
6. **Public Key Infrastructure Specification**: Specification for PKI for mix-net
7. **Sphinx Message Format Specification**: Message format used by the mix-net.
8. **Mix Network End-to-end Protocol Specification**: Specification of end-to-end integration needed by LEAP and K-9.
9. **Mix Network Wire Protocol Specification**: Wire format used by the mix-net.

The goal is to prepare software that is capable of being used and will be available on Google Play either as part of the core K-9 mail, or possibly (to be determined by end of the project) a PANORAMIX branded-app that allows re-branding by partners and other interested providers. This will also investigate if a non-Google version can be available via F-Droid.

2. Server-side Instructions for Sysadmins

Note that the instructions will first cover how to set up a mix node. Then, it will cover how to set-up a generic LEAP provider node, and finally, enabling e-mail.

2.1 How to Set Up Your Own Katzenpost Mixnet

2.2 Setting up a Mix node

Caution: Mix networks are meant to be decentralized and therefore should be operated by multiple entities. You can of course be the only operator of a mix network for testing purposes. However, if you are serious about running a mix-node, please contact the PANORAMIX project in order to join the Panoramix mix network.

A Katzenpost mix network has two binary programs, a PKI and a Mix/Provider.

Katzenpost server side requires golang 1.8 or later. See golang install instructions: *https://golang.org/doc/install*

The Katzenpost minclient library requires golang 1.9 or later.

You can build this software on your computer and copy the binaries to the server from `$GOPATH/bin`:

```
go get -u -v github.com/katzenpost/daemons/
authority/nonvoting
go get -u -v github.com/katzenpost/daemons/
server
```

The produced binaries are statically linked, so you can build the authority and the server code on one machine, and then distribute them to any Linux based machines to run.

Each network component, the PKI and mixes/providers MUST have the correct time. We recommend installing chrony for the purpose of time synchronization.

```
apt install chrony
```

2.2.1 Set up the PKI

Configure the PKI (or Network Authority) based on an example configuration file:

```
wget -O authority.toml https://raw.githubusercontent.com/katzenpost/
```

`daemons/master/authority/nonvoting/authority.toml.sample`

You need to edit at least the [Authority] section, and configure the address(es) the authority should bind to (Addresses) as well as its data directory (DataDir).

Now, you can generate the authority identity key:

```
$GOPATH/bin/nonvoting -f authority.toml -g
```

This -g option causes the Authority server to generate an authority identity key which will get written to the specified data directory and printed in the log. This Authority Identity key is used in the mix configuration file and allows mixes to interact with the PKI.

In the next step, we will set up at least one Provider and some Mix nodes, and add their public identity keys to the authority before we run it.

2.2.2 Set up the Mix

Configure the mix node: *https://raw.githubusercontent.com/Katzenpost/daemons/master/server/katzenpost.toml.sample*

Generate the key:

```
$GOPATH/bin/server -f katzenpost.toml -g
```

The generated Mix Identity key MUST be entered into the PKI configuration file. Once the PKI is configured with all of the mix identity keys you can start the PKI server and then start all the mixes.

2.2.3 Add Users to the Provider

Add Users to the Provider using the management interface:

```
socat unix:/<path-to-data-dir>/management_sock STDOUT
ADD_USER alice X25519_public_key_in_hex_or_base64
```

2.2.4 Run the Authority

```
$GOPATH/bin/nonvoting -f authority.toml
```

2.3 Installing a LEAP-enabled Server

2.3.1 Introduction

We are going to create a minimal LEAP provider, but one that does not offer any actual services. The LEAP provider provides secure email services and is integrated against the mix-net.

Our goal is something like this:

```
$ leap list
        NODES   SERVICES          TAGS
   wildebeest   couchdb, webapp
```

NOTE: You won't be able to run that `leap list` command yet, not until we actually create the node configurations.

Requirements

1. A workstation: This is your local machine on which you will run commands.
2. A server: This is the machine that you will deploy to. The server can be either:
   (a) A local Vagrant virtual machine: a Vagrant machine can only be useful for testing.
   (b) A real or paravirtualized server: The server must have Debian Jessie installed, and you must be able to SSH into the machine as root. Paravirtualization includes KVM, Xen, OpenStack, Amazon, but not VirtualBox or OpenVZ. Proxmox has an known issue when changing the resolver.

Other things to keep in mind:

- The ability to create/modify DNS entries for your domain is preferable, but not needed. If you don't have access to DNS, you can workaround this by modifying your local resolver, i.e. editing `/etc/hosts`.
- You need to be aware that this process will make changes to your servers, so please be sure that these machines are a basic install with nothing configured or running for other purposes.
- Your servers will need to be connected to the internet, and not behind a restrictive firewall.

2.3.2 Prepare your workstation

In order to be able to manage your servers, you need to setup the LEAP Platform on your desktop. This consists of three parts: the platform recipes, the `leap` command, and your provider instance. We will go over these step-by-step below, you can find more details in the platform introduction.

Install pre-requisites

Install core prerequisites on your workstation.

Debian Unstable (sid)

```
workstation$ sudo apt-get install git rsync openssh-client openssl zlib1g-dev
```

Other Debian & Ubuntu

```
workstation$ sudo apt-get install git ruby ruby-dev rsync openssh-client
openssl rake make bzip2 zlib1g-dev
```

Mac OS

```
workstation$ brew install ruby-install
workstation$ ruby-install ruby
```

The platform recipes

The LEAP platform recipes are a set of modules designed to work together to provide you everything you need to manage your provider. You typically do not need to modify these, but do need them available for deploying your provider.

To obtain the platform recipes, simply clone the git repository, and then check out the most recent stable release branch:

```
workstation$ git clone -b version/0.9.x https://leap.se/git/leap_platform
```

If you want to get the latest development branch (Beware: it could be unstable!) you could simply use the master branch instead by:

```
workstation$ git clone https://leap.se/git/leap_platform
```

Install the LEAP command-line utility

The `leap` command line tool is what you use to manage everything about your provider.

Keep these rules in mind:

- `leap` is run on your workstation: The `leap` command is always run locally on your workstation, never on a server you are deploying to.
- `leap` is run from within a provider instance: The `leap` command requires that the current working directory is a valid provider instance, except when running `leap new` to create a new provider instance.

If on Debian Unstable (sid), simply do this:

```
workstation$ sudo apt install leap-cli
```

Otherwise, you will need to do this:

```
workstation$ sudo gem install leap_cli
```

Alternately, you can install `leap` locally without root privileges:

```
workstation$ gem install --user-install leap_cli
workstation$ PATH="$PATH:$(ruby -e 'puts Gem.user_dir')/bin"
```

If you choose a local install, you probably want to permanently add the –user-install directory to your PATH by adding this to your `~/.profile` file (requires logout):

```
[ $(which ruby) ] && PATH="$PATH:$(ruby -e 'puts Gem.user_dir')/bin"
```

To confirm that you installed `leap` correctly, try running `leap --version`.

2.3.3 Create a provider instance

A provider instance is a directory tree, residing on your workstation, that contains everything you need to manage an infrastructure for a service provider.

In this case, we create one for example.org and call the instance directory 'example'.

```
workstation$ leap new ~/example
```

The `leap new` command will ask you for several required values:

- domain: The primary domain name of your service provider. In this tutorial, we will be using "example.org".
- name: The name of your service provider (we use "Example").
- contact emails: A comma separated list of email addresses that should be used for important service provider contacts (for things like postmaster aliases, Tor contact emails, etc).
- platform: The directory where you have a copy of the `leap_platform` git repository

checked out. If the platform directory does not yet exist, the `leap_platform` will be downloaded and placed in that directory.

You could also have passed these configuration options on the command-line, like so:

```
workstation$ leap new --contacts your@email.here --domain example.org
 --name Example --platform=~/leap/leap_platform .
```

You should now have the following files:

```
workstation$ tree example
example
- common.json
- Leapfile
- nodes/
- provider.json
- services/
- tags/
```

Now add yourself as a privileged sysadmin who will have access to deploy to servers:

```
workstation$ cd example
workstation$ leap add-user louise --self
```

Replace “louise” with whatever you want your sysadmin username to be.

NOTE: Make sure you change directories so that the `leap` command is run from within the provider instance directory. Most `leap` commands only work when run from a provider instance.

Now create the necessary keys and certificates:

```
workstation$ leap cert ca
workstation$ leap cert csr
```

What do these commands do? The first command will create two Certificate Authorities (CA), one that clients will use to authenticate with the servers and one for backend servers to authenticate with each other. The second command creates a Certificate Signing Request (CSR) suitable for submission to a commercial CA. It also creates two “dummy” files for you to use temporarily:

- `files/cert/example.org.crt` – This is a “dummy” certificate for your domain that can be used temporarily for testing. Once you get a real certificate from a CA, you should replace this file.
- `files/cert/commercial_ca.crt` – This is “dummy” CA cert the corresponds to the dummy domain certificate. Once you replace the domain certificate, also replace this file with the CA cert from the real Certificate Authority.

If you plan to run a real service provider, see important information on managing keys and certificates.

2.3.4 Add a node to the provider

A “node” is a server that is part of your infrastructure. Every node can have one or more services associated with it. We will now add a single node with two services, “webapp” and “couchdb”.

You have two choices for node type: a real node or a local node.

- Real Node: A real node is any physical or paravirtualized server, including KVM, Xen,

OpenStack Compute, Amazon EC2, but not VirtualBox or OpenVZ (VirtualBox and OpenVZ use a more limited form of virtualization). The server must be running Debian Jessie.
- Local Node: A local node is a virtual machine created by Vagrant, useful for local testing on your workstation.

Getting Vagrant working can be a pain and is covered in other tutorials. If you have a real server available, we suggest you try this tutorial with a real node first.

Option A: Add a real node

Note: Installing LEAP Platform on this server will potentially destroy anything you have previously installed on this machine.

Create a node, with the services "webapp" and "couchdb":

```
workstation$ leap node add wildebeest ip_address:x.x.x.w services:webapp,couchdb
```

NOTE: replace x.x.x.x with the actual IP address of this server.

Option B: Add a local node

Create a node, with the services "webapp", "soledad" and "couchdb", and then start the local virtual machine:

```
workstation$ leap node add --local wildebeest services:webapp,couchdb,soledad
workstation$ leap local start wildebeest
```

It will take a while to download the Virtualbox base box and create the virtual machine.

Option C: Add a virtual machine in the cloud

In order to create a provider using the cloud, please follow these instructions.

2.3.5 Deploy your provider

Initialize the node

Node initialization only needs to be done once, but there is no harm in doing it multiple times:

```
workstation$ leap node init wildebeest
```

This will initialize the node `wildebeest`.

For non-local nodes, when `leap node init` is run, you will be prompted to verify the fingerprint of the SSH host key and to provide the root password of the server(s). You should only need to do this once.

Deploy to the node

The next step is to deploy the LEAP platform to your node. Deployment can take a while to run, especially on the first run, as it needs to update the packages on the new machine.

```
workstation$ leap deploy wildebeest
```

Watch the output for any errors (in red), if everything worked fine, you should now have your first running node. If you do have errors, try doing the deploy again.

Setup DNS

The next step is to configure the DNS for your provider. For testing purposes, you can just modify your `/etc/hosts` file. Please don't forget about these entries, they will override DNS queries if you setup your DNS later. For a list of what entries to add to `/etc/hosts`, run this command:

```
workstation$ leap compile hosts
```

Alternately, if you have access to modify the DNS zone entries for your domain:

```
workstation$ leap compile zone
```

NOTE: The resulting zone file is incomplete because it is missing a serial number. Use the output of `leap compile zone` as a guide, but do not just copy and paste the output. Also, the `compile zone` output will always exclude mention of local nodes.

The DNS method will not work for local nodes created with Vagrant.

2.3.6 Test that things worked correctly

To run troubleshooting tests:

```
workstation$ leap test
```

Alternately, you can run these same tests from the server itself:

```
workstation$ leap ssh wildebeest
wildebeest# run_tests
```

2.3.7 Create an administrator

The registration of new users is (since platform 0.10) restricted by default. You will need to generate an `invite code.` A simple `leap run invite` will give you an code like `ptxy-rsy1` which you can use to register a user. Assuming that you set up your DNS or `/etc/hosts` file, you should be able to load `https://example.org` in your web browser (where example.org is whatever domain name you actually used).

Your browser will complain about an untrusted cert, but for now just bypass this. From there, you should be able to register a new user and login.

Once you have created a user, you can now make this user an administrator. For example, if you created a user `kangaroo`, you would create the file `services/webapp.json` with the following content:

```
{
    "webapp": {
        "admins": ["kangaroo"]
    }
}
```

Save that file and run `leap deploy` again. When you next log on to the web application, the user kangaroo will now be an admin.

If you want to open up registration of new users, see webapp for configuration options.

2.3.8 Add an end-user service

You should now have a minimal service provider with a single node. This service provider is pointless at the moment, because it does not include any end-user services like mix-enabled e-mail. In the next section, we set up an email interface.

2.4 Setting up a Mixnet-enabled LEAP mail server

This tutorial walks you through the initial process of creating and deploying a minimal LEAP-enabled email service provider.

We are going to create a minimal LEAP provider offering email service.

Our goal is something like this:

```
$ leap list
    NODES        SERVICES                        TAGS
    wildebeest   couchdb, mx, soledad, webapp
```

Where 'wildebeest' is whatever name you chose for your node in the Quick Start Tutorial.

2.4.1 Add email services to the node

In order to add services to a node, edit the node's JSON configuration file.

In our example, we would edit `nodes/wildebeest.json`:

```
{
  "ip_address": "XXX.XXX.XXX.XXX",
  "services": ["couchdb", "webapp", "mx", "soledad"]
}
```

Where "XXX.XXX.XXX.XXX" should be replaced by your IP provider.

Here, we added `mx` and `soledad` to the node's `services` list. Briefly:

- **mx**: nodes with the **mx** service will run postfix mail transfer agent, and are able to receive and relay email on behalf of your domain. You can have as many as you want, spread out over as many nodes as you want.
- **soledad**: nodes with **soledad** service run the server-side daemon that allows the client to synchronize a user's personal data store among their devices. Currently, **soledad** only runs on nodes that are also **couchdb** nodes.

For more details, see the Services overview, or the individual pages for the mx and soledad services.

2.4.2 Deploy to the node

Now you should deploy to your node.

```
workstation$ leap deploy
```

2.4.3 Setup DNS

There are several important DNS entries that all email providers should have:

- SPF (Sender Policy Framework): With SPF, an email provider advertises in their DNS which servers should be allowed to relay email on behalf of your domain.
- DKIM (DomainKey Identified Mail): With DKIM, an email provider is able to vouch for the validity of certain headers in outgoing mail, allowing the receiving provider to have more confidence in these values when processing the message for spam or abuse.

In order to take advantage of SPF and DKIM, run this command:

```
workstation$ leap compile zone
```

Then take the output of that command and merge it with the DNS zone file for your domain.

CAUTION: the output of `leap compile zone` is not a complete zone file since it is missing a serial number. You will need to manually merge it with your existing zone file.

2.4.4 Test it out

First, run:

```
workstation# leap test
```

Then fire up the bitmask client, register a new user with your provider, and try sending and receiving email.

3. Open-Source Desktop Client

code: *https://github.com/leapcode/pixelated-user-agent*
install binaries: *https://pypi.python.org/pypi/leap.pixelated/*
install (via Python's pypi): *https://pypi.python.org/pypi/leap.pixelated/*

The open-source desktop client is based on *Bitmask*, a cross-platform desktop secure email platform that can be hooked into multiple front ends, as Bitmask runs as a local SMTP proxy that securely delivers and retrieves mail. The front-end currently being used is *Pixelated*, an e-mail client that fits in via *Bitmask* the larger server-side LEAP platform and is compatible with sending and receiving standardized SMTP and GPG via the client-side. Note that LEAP goes above and beyond GPG-encrypted email and resistance to metadata collection via mix networking, as it gives users more protection by encrypting emails on arrival and multi-device synchronization (via *Soledad*, as described in D7.1). Figure 3.1 illustrates the Panoramix-enabled Pixelated client.

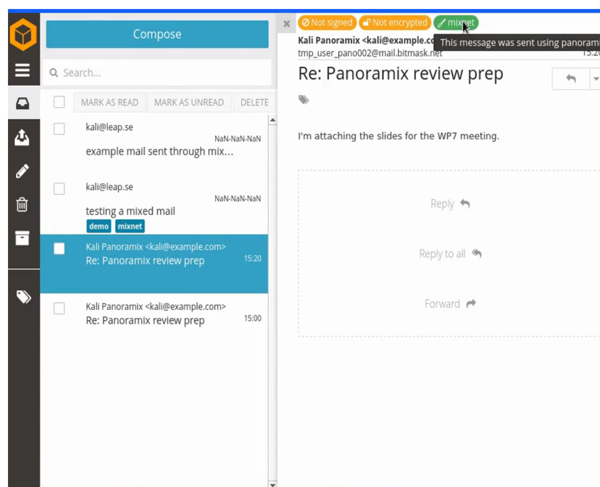


Figure 3.1: Panoramix-enabled Pixelated client.

The Panoramix-enabled client uses simple icons to communicate whether or not a message has been received via the Panoramix mix-net, where the ability to support the mix-net is kept in a whitelist of providers and via an HTTP HEADER @@. When an e-mail is sent, the Panoramix icon is activated as in Figure 3.2. Likewise, when an email is received the Panoramix icon is activated as in Figure 3.3.

3.1 Installing Bitmask

Bitmask, with the Pixelated mail client@@?, can be downloaded and automatically installed via *https://bitmask.net/en/install*. It currently works on Linux and MacOS, with Windows still

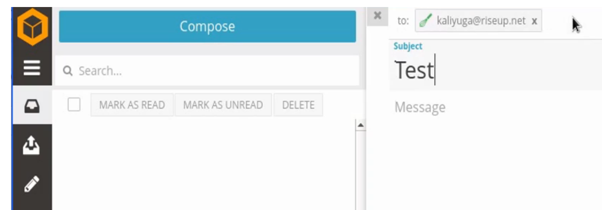


Figure 3.2: Bitmask sending a mix-net enabled message

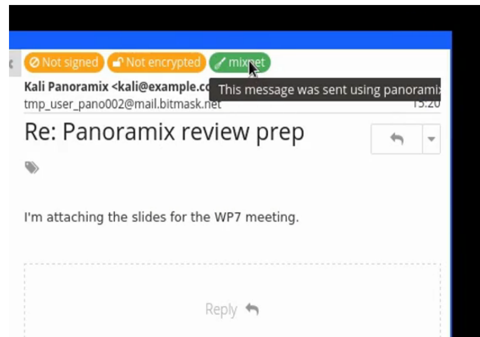


Figure 3.3: Bitmask receiving a mix-net enabled message

being under development. See Figure 3.4.

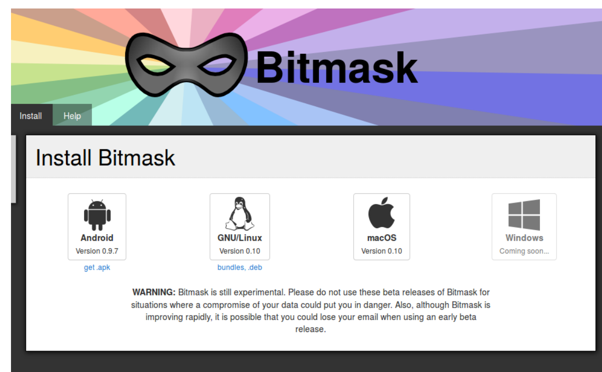


Figure 3.4: Bitmask Installation

3.2 Installing Pixelated with Mix-net

The Pixelated User Agent is the email client of the Pixelated ecosystem. It is composed of two parts, a web interface written in JavaScript (FlightJS and React) and a Python API that interacts with a LEAP Provider, the email platform that Pixelated is built on.

If you want to run and test it locally, then before you have to install the following dependencies:

- Vagrant, a tool that automates the setup of a virtual machine with the development environment;
- A vagrant compatible provider, e.g. Virtual Box;
- If you don't want to use vagrant, check the Developer-Setup-on-native-OS page.

Option 1: Run Pixelated User Agent against an existing LEAP provider

1. If you don't have access to an existing LEAP provider, you can create an account at Bitmask mail demo provider.
2. Clone the Pixelated User Agent repo and start the virtual machine (downloads 600MB, you may want get a coffee or tea in the meantime):

   ```
   $ git clone https://github.com/pixelated/pixelated-user-agent.git
   $ cd pixelated-user-agent
   $ vagrant up
   ```

3. Log into the VM:
   - You can access the guest OS shell via the command `vagrant ssh` run within the `pixelated-user-agent` folder in the host OS.
   - `/vagrant` in the guest OS is mapped to the `pixelated-user-agent` folder in the host OS. File changes on either side will reflect in the other.

   ```
   $ vagrant ssh
   $ cd /vagrant
   ```

4. Start the Pixelateduser agent:

   ```
   $ pixelated-user-agent --host 0.0.0.0 --multi-user --provider=mail.bitmask.net
   ```

   You also have other ways to start the user agent. Check the "Single User Mode vs Multi User Mode" page.

5. Go to localhost:3333 on your browser. You should see the login screen, where you can put your username and password created on step 1. Once you login, you'll see your inbox.

   First time email sync could be slow, please be patient. This could be the case if you have a lot of emails and it is the first time you setup the user agent on your machine.

Option 2: Run Pixelated User Agent against a local LEAP provider

We suggest you use the following instructions:

- Install Pixelated User Agent using Developer-Setup-on-native-OS page.
- Install a local LEAP provider using the LEAP Platform installation with vagrant instructions.
- Run the user agent against a local LEAP provider.

3.3 Turning On Mix-net

The Bitmask client needs to enable the Panoramix mix-net service by enabling it in the config file. In Linux this is located at */.config/leap/bitmaskd.cfg* and the Panoramix mix-net can by turning on the following switch:

```
[services]
mixnet = True
```

3.4 Mix-net integration description

A number of development goals have been reached in order to integrate Panoramix into Bitmask. There are currently Panoramix bindings for Python, so that the client can be configured, started, and controlled from a Python client such as Pixelated. The compilation of the Panoramix static client within the current scripts for packaging and distributing Panoramix together with the LEAP packages and bundles. There is an implementation for the server-side mix-net interface that is responsible for authentication, so that the mix-net servers operation can delegate authentication of incoming messages to the framework for client authentication currently used by the LEAP services. Thereby enabling the LEAP server to authenticate incoming email. We have also integrated LEAP's automatic and transparent key management solution in a way that the Panoramix client can send the identity keys to either LEAP's nickserver or a transitional mix-net-mx service, so that in the end both authentication of the client against the server using ECDH keys and key discovery can work out of the box. The mix-net API has been integrated within the Bitmask local daemon so that it can inject new incoming mail into the soledad-based IMAP storage. For error protection, the Bitmask client can discover whether the mix-net infrastructure is up (PKI server up and running), and whether the provider's entrypoint is ready to accept incoming messages, so that the user can be notified in case mix-net routing is not possible due to limited availability of the mix-net infrastructure. Most importantly for the user, the needed UI changes so that the end user is aware of the fact that incoming and outgoing messages are being delivered via the mix-net when possible as given in Figure 3.2 and Figure 3.3.

4. Open-Source Mobile Client

Code and Build Instructions: *https://github.com/katzenpost/katzenpost-android-mail*

The open-source desktop client used is *K-9 mail*, an e-mail system that works on Android and is compatible with sending and receiving standardized SMTP, as well as GPG via the *OpenKeyChain* app. Figures 4.1 and 4.2 illustrate the Panoramix-enabled Pixelated client. Note that K-9 is the only open-source client for mail on Android, but as it is unfeasible to run a local mix-nets SMTP proxy on Android due to device constraints, the client code-base must be done separately and developed in paralled to both the local proxie and Bitmask (the desktop client).

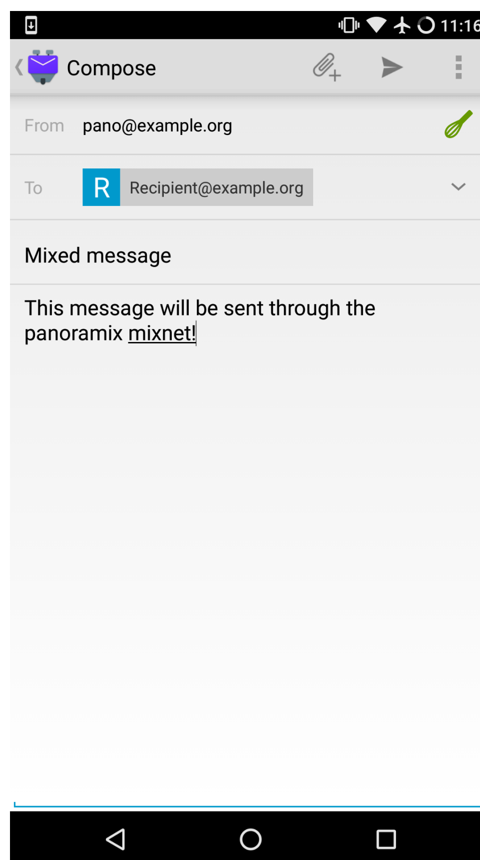


Figure 4.1: K9 Mail sending a mix-net enabled message

4.1 Katzenpost/Mail Android

This is a Katzenpost client, based on K-9 Mail.

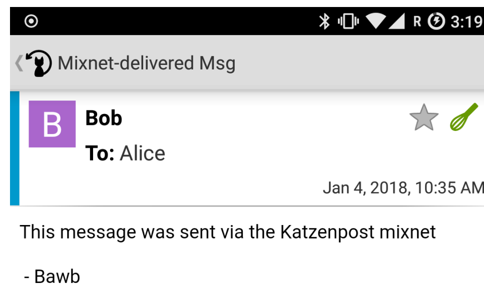


Figure 4.2: K9 Mail receiving a mix-net enabled message

4.2 Build Instructions

Building the project depends on the Android SDK, Android NDK, Go 1.7+, and gomobile.

4.2.1 Prepare Toolchain

1. Download Android SDK https://developer.android.com/sdk/download.html
2. Extract SDK to $HOME/android_sdk
3. Download Android NDK 16.1 (or newer) https://developer.android.com/ndk/downloads/index.html
4. Extract NDK to $HOME/android_sdk/ndk-bundle
5. Install Go 1.7 https://golang.org/doc/install
6. Add to PATH: $HOME/android_sdk/platform_tools and $HOME/android_sdk/tools
7. `git clone github.com/katzenpost/katzenpost-android-mail`

4.2.2 Prepare Gomobile

1. Set a GOPATH environment variable
2. `go get golang.org/x/mobile/cmd/gomobile`
3. `gomobile init -ndk $HOME/android_sdk/ndk-bundle`

4.2.3 Prepare Repository and Compile Native Bindings

1. `cd katzenpost-android-mail`
2. `mkdir bindings; cd bindings`
3. `go get -v github.com/katzenpost/bindings`
4. `gomobile bind -v -target android github.com/katzenpost/bindings`

4.2.4 Compile application

1. `./gradlew`
2. `./gradlew k9mail:assembleDebug`
3. `adb install -r app/build/outputs/apk/fcm/debug/app-fcm-debug.apk`

4.3 Mix-net integration description

We produced language bindings that bridge the Katzenpost Go-code with Java. This is necessary to allow use of the library in a JVM environment, and especially on the Android platform. Next, we adapted the build chain to include the bindings into the K-9 codebase, to make the required methods available for integration and distribution. This needs to take into account the target build architectures. Also, we supplanted the SMTP transport with a Katzenpost-enabled variant, that can be used for sending messages to available recipients in the mix-net.

We started work on a Katzenpost-enabled retrieval method for incoming messages, based on the POP3 implementation. For a first iteration, this was limited to a polling mechanism. Implementation of continuous retrieval or push semantics is a separate problem, that improves usability but is not required for initial deployment.

There is ongoing work on adding a UI workflow for registering users with the mix-net, and creating an account on the mix-net. This involves creation or input of a link-layer key for authentication, and upload of an identity key to use for end-to-end communication within the mix-net.

This included adding UI components to indicate the Katzenpost message status, both for incoming and outgoing messages. This serves to reassure the user of the provided security properties and provide them with details on click. It can also be used to display warnings in cases where there is an indication that either anonymity or confidentiality might have been compromised for a given message or recipient.

There is still much work to be done on creating suitable reporting facilities to handle error cases. These error cases include outages of the mix-net, undeliverable messages due to route errors, or rejected messages e.g. due to unknown recipients.

Initial user feedback indicated that users are more comfortable changing between different security contexts if they switch between apps, rather than different accounts within the same app. This hypothesis will have to be validated with A/B testing.

5. Mix Network for E-mail Architecture Specification

Code:*https://github.com/katzenpost/*
Docs:*https://katzenpost.mixnetworks.org/docs/specs.html*

Note that the code link to Github includes all the open source code produced. The docs include the latest version of this specification and all related specifications in future chapters.

5.1 Introduction

This specification provides the design of a mix network meant to provide an anonymous messaging service. Various system components such as client software, end to end reliability protocol, Sphinx cryptographic packet format and wire protocol are described in their own specification documents.

5.1.1 Terminology

- A `KiB` is defined as 1024 8 bit octets.
- `Mix` - A server that provides anonymity to clients. This is accomplished by accepting layer-encrypted packets from a Provider or another Mix, decrypting a layer of the encryption, delaying the packet, and transmitting the packet to another Mix or Provider.
- `Mix-net` - A network of mixes.
- `Provider` - A service operated by a third party that Clients communicate directly with to communicate with the Mix-net. It is responsible for Client authentication, forwarding outgoing messages to the Mix-net, and storing incoming messages for the Client. The Provider MUST have the ability to perform cryptographic operations on the relayed packets.
- `Node` - A Mix or Provider instance.
- `User` - An agent using the Katzenpost system.
- `Client` - Software run by the User on its local device to participate in the Mix-net.
- `Katzenpost` - A project to design an improved mix service as described in this specification. Also, the name of the reference software to implement this service, currently under development.
- Classes of traffic - We distinguish the following classes of traffic:

  - ACKs
  - Small messages
  - Large messages (big attachments)

- `Packet` - A string transmitted anonymously thought the Katzenpost network. The length of the packet is fixed for every class of traffic.
- `Payload` - The portion of a Packet containing a message, or part of a message, to be delivered anonymously.
- `Message` - A variable-length sequence of octets sent anonymously through the network. Short messages are sent in a single packet; long messages are fragmented across multiple packets (see the Katzenpost Mix Network End-to-end Protocol Specification for more information about encoding messages into payloads).
- `MSL` - Maximum Segment Lifetime, 120 seconds.

5.2 System Overview

The presented system design is based on [LOOPIX]. The detailed End-to-end specification, describing the operations performed By the sender and recipient, as well sender's provider and Recipient's provider, are presented in "Katzenpost Mix Network End-to-end Protocol Specification". Below, we present the system overview.

The Provider ran by each service provider is responsible for accepting packets from the client, and forwarding them to the mix network, which then relays packets to the recipient's Provider. Upon receiving a packet from the mix network, the Provider is responsible for signaling that the packet was received by sending an acknowledgment, as well as storing the packet until it is retrieved by the recipient.

The Provider and Client behavior is specified in Chapter 8 though certain aspects of the Provider behavior are also specified here, as Providers are Nodes. The mix network provides neither reliable nor in-order delivery semantics. It is up to the applications that make use of the mix network to implement additional mechanism if either property is desired.

5.2.1 Threat Model

We assume that the sender and recipient do know each other's addresses. This system guarantees third-party anonymity, meaning that no parties other than sender and recipient are able to learn that the sender and recipient are communicating. Note that this is in contrast with other designs, such as Mixminion, which provide sender anonymity towards recipients as well as anonymous replies.

Additionally as all of a given client's messages go through a single provider instance, it is assumed that in the absence of any specific additional defenses, that the Provider can determine the approximate mail volume originating from and destined to a given client. We consider the provider follows the protocol and might be an honest-but-curious adversary.

External local network observers can determine the number of Packets traversing their region of the network because at this time no decoy traffic has been specified. Global observers will not be able to de-anonymize packet paths if there are enough packets traversing the mix network.

A malicious mix only has the ability to remember which input packets correspond to the output packets. To discover the entire path all of the mixes in the path would have to be malicious.

Moreover, the malicious mixes can drop, inject, modify or delay the packets for more or less time than specified.

5.2.2 Network Topology

The Katzenpost Mix Network uses a layered topology consisting of a fixed number of layers, each containing a set of mixes. At any given time each Mix MUST only be assigned to one specific layer. Each Mix in a given layer N is connected to every other Mix in the previous and next layer, and or every participating Provider in the case of the mixes in layer 0 or layer N (first and last layer).

The network topology MUST also maximize the number of security domains traversed by the packets. This can be achieved by not allowing mixes from the same security domain to be in different layers.

Requirements for the topology:

- Should allow for non-uniform throughput of each mix.
- Should maximize distribution among security domains, in this case the mix descriptor specified family field would indicate the security domain or entity operating the mix.
- Other legal jurisdictional region awareness for increasing the cost of compulsion attacks.

5.3 Packet Format Overview

For the packet format of the transported messages we use the Sphinx cryptographic packet format. The detailed description of the packet format, construction and processing is presented in Chapter 7.

As the Sphinx packet format is generic, the Katzenpost Mix Network must provide a concrete instantiation of the format, as well as additional Sphinx per-hop routing information commands.

5.3.1 Sphinx Cryptographic Primitives

For the current version of the Katzenpost Mix Network, let the following cryptographic primitives be used as described in the Sphinx specification.

- `H(M)` - As the output of this primitive is only used locally to a Mix, any suitable primitive may be used.
- `MAC(K, M)` - HMAC-SHA256-128 [RFC6234], M_KEY_LENGTH of 32 bytes (256 bits), and MAC_LENGTH of 16 bytes (128 bits).
- `KDF(SALT, IKM)` - HKDF-SHA256, HKDF-Expand only, with SALT used as the info parameter.
- `S(K, IV)` - CTR-AES128 [SP80038A], S_KEY_LENGTH of 16 bytes (128 bits), and S_IV_LENGTH of 12 bytes (96 bits), using a 32 bit counter.
- `SPRP_Encrypt(K, M)/SPRP_Decrypt(K, M)` - AEZv5 [AEZV5], SPRP_KEY_LENGTH of 48 bytes (384 bits). As there is a disconnect between AEZv5 as specified and the Sphinx usage, let the following be the AEZv5 parameters:
  - nonce - 16 bytes, reusing the per-hop Sphinx header IV.
  - additional_data - Unused.

  – tau - 0 bytes.

- `EXP(X, Y)` - X25519 [RFC7748] scalar multiply,
  GROUP_ELEMENT_LENGTH of 32 bytes (256 bits), G is the X25519 base point.

5.3.2 Sphinx Packet Parameters

The following parameters are used as for the Katzenpost Mix Network instantiation of the Sphinx Packet Format:

- `AD_SIZE` - 2 bytes.
- `SECURITY_PARAMETER` - 16 bytes.
- `PER_HOP_RI_SIZE` - 42
- `NODE_ID_SIZE` - 32 bytes, the size of the Ed25519 public key, used as Node identifiers.
- `RECIPIENT_ID_SIZE` - 64 bytes, the maximum size of local-part component in an e-mail address.
- `SURB_ID_SIZE` - Single Use Reply Block ID size, 16 bytes.
- `MAX_HOPS` - 5, the ingress provider, a set of three mixes, and the egress provider.
- `PAYLOAD_SIZE` - 42
- `KDF_INFO` - The byte string 'Katzenpost-kdf-v0-hkdf-sha256'.

The Sphinx Packet Header `additional_data` field is specified as follows:

```
struct {
    uint8_t version;  /* 0x00 */
    uint8_t reserved; /* 0x00 */
} KatzenpostAdditionalData;
```

All nodes MUST reject Sphinx Packets that have `additional_data` that is not as specified in the header.

5.3.3 Sphinx Per-hop Routing Information Extensions

The following extensions are added to the Sphinx Per-Hop Routing Information commands.

Let the following additional routing commands be defined in the extension RoutingCommandType range (0x80 - 0xff):

```
enum {
    mix_delay(0x80),
} KatzenpostCommandType;
```

The mix_delay command structure is as follows:

```
struct {
    uint32_t delay_ms;
} NodeDelayCommand;
```

5.4 Mix Node Operation

All Mixes behave in the following manner:

- Accept incoming connections from peers, and open persistent connections to peers as needed (Section 5.4.1).
- Periodically interact with the PKI to publish Identity and Sphinx packet public keys, and to obtain information about the peers it should be communicating with, along with periodically rotating the Sphinx packet keys for forward secrecy (Section 5.4.2).
- Process inbound Sphinx Packets, delay them for the specified time and forward them to the appropriate Mix and or Provider (Section 5.4.3).

All Nodes are identified by their link protocol signing key, for the purpose of the Sphinx packet source routing hop identifier.

All Nodes participating in the Mix Network MUST share a common view of time, via NTP or similar time synchronization mechanism.

5.4.1 Link Layer Connection Management

All communication to and from participants in the Katzenpost Mix Network is done via the Katzenpost Mix Network Wire Protocol [KATZMIXWIRE].

Nodes are responsible for establishing the connection to the next hop, for example, a mix in layer 0 will accept inbound connections from all Providers listed in the PKI, and will proactively establish connections to each mix in layer 1.

Nodes MAY accept inbound connections from unknown Nodes, but MUST not relay any traffic until they became known via listing in the PKI document, and MUST terminate the connection immediately if authentication fails for any other reason.

Nodes MUST impose an exponential backoff when reconnecting if a link layer connection gets terminated, and the minimum retry interval MUST be no shorter than 5 seconds.

Nodes MAY rate limit inbound connections as required to keep load and or resource use at a manageable level, but MUST be prepared to handle at least one persistent long lived connection per potentially eligible peer at all times.

5.4.2 Sphinx Mix and Provider Key Rotation

Each Node MUST rotate the key pair used for Sphinx packet processing periodically for forward secrecy reasons and to keep the list of seen packet tags short. The Katzenpost Mix Network uses a fixed interval (`epoch`), so that key rotations happen simultaneously throughout the network, at predictable times.

Let each epoch be exactly `10800 seconds (3 hours)` in duration, and the 0th Epoch begin at `2017-06-01 00:00 UTC`.

To facilitate smooth operation of the network and to allow for delays that span across epoch boundaries, Nodes MUST publish keys to the PKI for at least 3 epochs in advance, unless the node will be otherwise unavailable in the near future due to planned downtime.

Thus, at any time, keys for all Nodes for the Nth through N + 2nd epoch will be available, allowing for a maximum round trip (forward message + SURB) delay + transit time of 6 hours.

Node PKI interactions are conducted according to the following schedule, where `T` is the next epoch transition.

`T - 3600 sec` - Deadline for publication of all mixes documents for the next epoch.

`T - 2700 sec` - Start attempting to fetch PKI documents.

`T - 1800 sec` - Start establishing connections to the new set of relevant nodes in advance of the next epoch.

`T - 1MSL` - Start accepting new Sphinx packets encrypted to the next epoch's keys.

`T + 1MSL` - Stop accepting new Sphinx packets encrypted to the previous epoch's keys, close connections to peers no longer listed in the PKI documents and erase the list of seen packet tags.

5.4.3 Sphinx Packet Processing

The detailed processing of the Sphinx packet is described in the Sphinx specification in Chapter 7. Below, we present an overview of the steps which the node is performing upon receiving the packet:

1. Records the time of reception.
2. Perform a `Sphinx_Unwrap` operation to authenticate and decrypt a packet, discarding it immediately if the operation fails.
3. Apply replay detection to the packet, discarding replayed packets immediately.
4. Act on the routing commands.

   All packets processed by Mixes MUST contain the following commands.

   - `NextNodeHopCommand`, specifying the next Mix or Provider that the packet will be forwarded to.
   - `NodeDelayCommand`, specifying the delay in milliseconds to be applied to the packet, prior to forwarding it to the Node specified by the NextNodeHopCommand, as measured from the time of reception.

   Mixes MUST discard packets that have any commands other than a `NextNodeHopCommand` or a `NodeDelayCommand`. Note that this does not apply to Providers or Clients, which have additional commands related to recipient and SURB processing.

Nodes MUST continue to accept the previous epoch's key for up to 1MSL past the epoch transition, to tolerate latency and clock skew, and MUST start accepting the next epoch's key 1*MSL prior to the epoch transition where it becomes the current active key.

Upon the final expiration of a key (1MSL past the epoch transition), Nodes MUST securely destroy the private component of the expired Sphinx packet processing key along with the backing store used to maintain replay information associated with the expired key.

Nodes MAY discard packets at any time, for example to keep congestion and or load at a manageable level, however assuming the `Sphinx_Unwrap` operation was successful, the packet MUST be fed into the replay detection mechanism.

Nodes MUST discard packets that have been delayed for more time than specified by the `NodeDelayCommand`.

5.5 Anonymity Considerations

5.5.1 Topology

Layered topology is used because it offers the best level of anonymity and ease of analysis, while being flexible enough to scale up traffic. Whereas most mix-net papers discuss their security properties in the context of a cascade topology, which does not scale well, or a free-route network, which quickly becomes intractable to analyze when the network grows, while providing slightly worse anonymity than a layered topology. [MIXTOPO10]

Important considerations when assigning mixes to layers, in order of decreasing importance, are:

1. Security: do not allow mixes from one security domain to be in different layers to maximise the number of security domains traversed by a packet
2. Performance: arrange mixes in layers to maximise the capacity of the layer with the lowest capacity (the bottleneck layer)
3. Security: arrange mixes in layers to maximise the number of jurisdictions traversed by a packet (this is harder to do really well than it seems, requires understanding of legal agreements such as MLATs).

5.5.2 Mixing strategy

As a mixing technique is used the Poisson mix strategy [LOOPIX] [KESDOGAN98], which requires that a packet at each hop in the route is delayed be some amount of time, randomly selected by the sender from an exponential distribution. This strategy allows to prevent the timing correlation if the incoming and outgoing traffic from each node. Additionally, the parameters of the distribution used for generating the delay can be tuned up and down depending on the amount of the traffic in the network and the application for which the system is deployed.

5.6 Security Considerations

The source of all authority in the mix-net system comes from the Directory Authority system which is also known as the mix-net PKI. This system gives the mixes and clients a consistent view of the network while allowing human intervention when needed. All public mix key material and network connection information is distributed by this Directory Authority system.

6. Public Key Infrastructure Specification

6.1 Introduction

Mix-nets are designed with the assumption that a PKI exists and it gives each client the same view of the network. This specification is inspired by the Tor and Mixminion Directory Authority systems [MIXMINIONDIRAUTH] [TORDIRAUTH] whose main features are precisely what we need in our PKI. These are decentralized systems meant to be collectively operated by multiple entities.

The mix network directory authority system (PKI) is essentially a cooperative decentralized database and voting system that is used to produce network consensus documents which mix clients periodically retrieve and use for their path selection algorithm when creating Sphinx packets. These network consensus documents are derived from a voting process between the Directory Authority servers.

This design prevents mix clients from using only a partial view of the network for their path selection so as to avoid fingerprinting and bridging attacks [FINGERPRINTING] [BRIDGING] [LOCALVIEW].

The PKI is also used by Authority operators to specify network-wide parameters, for example in the Katzenpost Decryption Mix Network [KATZMIXNET] the Poisson mix strategy is used and therefore all the clients must use the same lambda parameter for their exponential distribution function when choosing hop delays in the path selection. The Mix Network PKI SHALL be used to distribute such network-wide parameters in the network consensus document that have an impact on security and performance.

6.1.1 Terminology

- `PKI` - public key infrastructure
- `Directory Authority system` - refers to specific PKI schemes used by Mixminion and Tor
- `MSL` - maximum segment lifetime
- `mix descriptor` - A database record which describes a component mix
- `family` - Identifier of security domains or entities operating one or more mixes in the network. This is used to inform the path selection algorithm.
- `nickname` - simply a nickname string that is unique in the consensus document; see Chapter 5.

- `layer` - The layer indicates which network topology layer a particular mix resides in.
- `Provider` - A service operated by a third party that Clients communicate directly with to communicate with the Mix-net. It is responsible for Client authentication, forwarding outgoing messages to the Mix-net, and storing incoming messages for the Client. The Provider MUST have the ability to perform cryptographic operations on the relayed messages.

6.1.2 Security Properties Overview

This Directory Authority system has the following feature goals and security properties:

- All Directory Authority servers must agree with each other on the set of Directory Authorities.
- This system is intentionally designed to provide identical network consensus documents to each mix client. This mitigates epistemic attacks against the client path selection algorithm such as fingerprinting and bridge attacks [FINGERPRINTING] [BRIDGING].
- This system is NOT byzantine-fault-tolerant, it instead allows for manual intervention upon consensus fault by the Directory Authority operators. Further, these operators are responsible for expelling bad acting operators from the system.
- Enforces the network policies such as mix join policy: Note that closed mix-nets will want to prevent arbitrary hosts from joining the network.
- The Directory Authority/PKI system for a given mix network is essentially the root of all authority in the system. This implies that if the PKI as a whole becomes compromised then so will the rest of the system (the component mixes) in terms of providing the main security properties described as traffic analysis resistance. Therefore a decentralized systems architecture is used so that the system is more resilient when attacked, in accordance with the principle of least authority which gives us security by design not policy. [SECNOTSEP] Otherwise, reducing the operation of the PKI system to a single host creates a terrible single point of failure where attackers can simply compromise this single host to control the network consensus documents that mix clients download and use to inform their path selection.
- We do not require cryptographic authenticity properties from the network transport because all our messages already have a cryptographic signature field that MUST be checked by the receiving peer. Confidentiality is not required because clients should all receive the exact same consensus file with all the signatures to prove its origins.
- Constructing this consensus protocol using a cryptographically malleable transport could expose at least one protocol parser to the network, this represents a small fraction of the attack surface area.

6.1.3 Differences from Tor's and Mixminion's Directory Authority systems

In this document we specify a Directory Authority system which is different from that of Tor's and Mixminion's in a number of ways:

- The schema of the mix descriptors is different from that used in Mixminion and Tor, including a change which allows our mix descriptor to express n Sphinx mix routing public keys in a single mix descriptor whereas in the Tor and Mixminion Directory Authority systems, descriptors are used.

- The serialization format of mix descriptors is different from that used in Mixminion and Tor.
- There is no non-directory channel to inform clients that a node is down, so it will end up being a lot of packet loss, since clients will continue to include the missing node in their path selection till keys published by the node expire and it falls out of the consensus.

6.2 Overview of Mix PKI Interaction

Each Mix MUST rotate the key pair used for Sphinx packet processing periodically for forward secrecy reasons and to keep the list of seen packet tags short. [SPHINX09] [SPHINXSPEC] The Katzenpost Mix Network uses a fixed interval (`epoch`), so that key rotations happen simultaneously throughout the network, at predictable times.

Each Directory Authority server MUST use some time synchronization protocol in order to correctly use this protocol. This Directory Authority system requires time synchronization to within a few minutes.

Let each epoch be exactly `10800 seconds (3 hours)` in duration, and the 0th Epoch begin at `2017-06-01 00:00 UTC`.

To facilitate smooth operation of the network and to allow for delays that span across epoch boundaries, Mixes MUST publish keys to the PKI for at least 3 epochs in advance, unless the mix will be otherwise unavailable in the near future due to planned downtime.

Thus, at any time, keys for all Mixes for the Nth through N + 2nd epoch will be available, allowing for a maximum round trip (forward message + SURB) delay + transit time of 6 hours. SURB lifetime is limited to a few hours because of the key rotation epoch, however this shouldn't present any useability problems since SURBs are only used for sending ACK messages from the destination Provider to the sender as described in [KATZEND2END].

6.2.1 Directory Authority Server Schedule

Directory Authority server interactions are conducted according to the following schedule, where `T` is the beginning of the current epoch.

`T` - Epoch begins

`T + 2 hours` - Vote exchange

`T + 2 hours + 7.5 minutes` - Tabulation and signature exchange

`T + 2 hours + 15 minutes` - Publish consensus

6.2.1.1 Mix Schedule

Mix PKI interactions are conducted according to the following schedule, where T is the beginning of the current epoch.

`T + 2 hours` - Deadline for publication of all mixes documents for the next epoch.
`T + 2 hours + 15 min` - This marks the beginning of the period where mixes perform staggered fetches of the PKI consensus document.
`T + 2 hours + 30 min` - Start establishing connections to the new set of relevant mixes in advance of the next epoch.

`T + 3 hours - 1MSL` - Start accepting new Sphinx packets encrypted to the next epoch's keys.

`T + 3 hours + 1MSL` - Stop accepting new Sphinx packets encrypted to the previous epoch's keys, close connections to peers no longer listed in the PKI documents and erase the list of seen packet tags.

As it stands, mixes have ~2 hours to publish, the PKI has 15 mins to vote, and the mixes have 28 mins to establish connections before there is network connectivity failure.

6.2.2 Scheduling Mix Downtime

Mix operators can publish a half empty mix descriptor for future epochs to schedule downtime. The mix descriptor fields that MUST be populated are:

- `Version`
- `Name`
- `Family`
- `Email`
- `Layer`
- `IdentityKey`
- `MixKeys`

The map in the field called "MixKeys" should reflect the scheduled downtown for one or more epochs by not have those epochs as keys in the map.

6.3 Voting for Consensus Protocol

In our Directory Authority protocol, all the actors conduct their behavior according to a common schedule as outlined in section 6.2. The Directory Authority servers exchange messages to reach consensus about the network. Other tasks they perform include collecting mix descriptor uploads from each mix for each key rotation epoch, voting, signature exchange and publishing of the network consensus documents.

6.3.1 Protocol Messages

There are only two document types in this protocol:

- `mix_descriptor`: A mix descriptor describes a mix.
- `directory`: A directory contains a list of descriptors and other information that describe the mix network.

Mix descriptor and directory documents MUST be properly signed.

6.3.1.1 Mix Descriptor and Directory Signing

Mixes MUST compose mix descriptors which are signed using their private identity key, an ed25519 key. Directories are signed by one or more Directory Authority servers using their authority key, also an ed25519 key. In all cases, signing is done using JWS [RFC7515].

6.3.2 Vote Exchange

As described in section 6.2.1, the Directory Authority servers begin the voting process 2 hours after epoch beginning. Each Authority exchanges vote directory messages with each other.

Authorities archive votes from other authorities and make them available for retrieval. Upon receiving a new vote, the authority examines it for new descriptors and fetches them from that authority. It includes the new descriptors in the next epoch's voting round.

6.3.3 Vote Tabulation for Consensus Computation

The main design constraint of the vote tabulation algorithm is that it MUST be a deterministic process that produces that same result for each directory authority server. This result is known as a network consensus file. Such a document is a well formed directory struct where the "`status`" field is set to "`consensus`" and contains 0 or more descriptors, the mix directory is signed by 0 or more directory authority servers. If signed by the full voting group then this is called a fully signed consensus.

1. Validate each vote directory: - that the liveness fields correspond to the following epoch - status is "`vote`" - version number matches ours
2. Compute a consensus directory: Here we include a modified section from the Mixminion PKI spec [MIXMINIONDIRAUTH]:
   - **For each distinct mix identity in any vote directory:** – If there are multiple nicknames for a given identity, do not include any descriptors for that identity.
     - If half or fewer of the votes include the identity, do not include any descriptors for the identity. [This also guarantees that there will be only one identity per nickname.]
     - If we are including the identity, then for each distinct descriptor that appears in any vote directory:
       - Do not include the descriptor if it will have expired on the date the directory will be published.
       - Do not include the descriptor if it is superseded by other descriptors for this identity.
       - Do not include the descriptor if it not valid in the next epoch.
       - Otherwise, include the descriptor.
   - Sort the list of descriptors by the signature field so that creation of the consensus is reproducible.
   - Set directory "`status`" field to "`consensus`".

6.3.4 Signature Collection

Each Authority exchanges their newly generated consensus files with each other. Upon receiving signed consensus documents from the other Authorities, peer signatures are appended to the current local consensus file if the signed contents match. The Authority SHOULD warn the administrator if network partition is detected.

6.3.5 Publication

If the consensus is signed by all members of the voting group then it's a valid consensus and it is published. Otherwise if there is disagreement about the consensus directory, each authority collects signatures from only the servers which it agrees with about the final consensus.

Upon consensus failure detection, the Directory Authority SHOULD report to its administrator that the consensus has failed, and explain how. Passive consumer clients downloading the network consensus documents SHOULD also receive a warning or error message.

6.4 PKI Protocol Data Structures

6.4.1 Mix Descriptor Format

Note that there is no signature field. This is because mix descriptors are serialized and signed using JWS. The `IdentityKey` field is a public ed25519 key. The `MixKeys` field is a map from epoch to public X25519 keys which is what the Sphinx packet format uses.

```
{
    "Version": 0,
    "Name": "",
    "Family": "",
    "Email": "",
    "AltContactInfo":"",
    "IdentityKey": "",
    "LinkKey":"",
    "MixKeys": {
        "Epoch": "EpochPubKey",
    },
    "Addresses": ["IP:Port"],
    "Layer": 0,
    "LoadWeight":0
  }
```

6.4.2 Directory Format

Note

replace the following JSON example with a JWS example

```
{
    "Signatures": [],
    "Version": 0,
    "Status": "vote",
    "Lambda" : 0.274,
    "MaxDelay" : 30,
    "Topology" : [],
    "Providers" : [],
}
```

6.5 PKI Wire Protocol

The wire protocol is built using HTTP. The following URLs for publishing and retrieving are constructed using `SERVER` and `EPOCH` where SERVER is the address of the Directory Authority server and EPOCH is the monotonically increasing integer indicating the epoch as described in section "6.2. Overview of Mix PKI Interaction".

6.5.1. Retrieving a directory

A directory may be retreived from a Directory Authority server with a URL of the form:

http://SERVER/katzenpost-v0/get/EPOCH

If the request is made with an old epoch or one too far in the future, then authority will return `Gone`, `Internal Server Error`, `Not Found`, with "Internal Sever Error" being extremely unlikely past the initial bootstrapping.

6.5.2. Publishing a mix descriptor

A mix descriptor may be uploaded to a Directory Authority server with a URL of the form:

http://SERVER/katzenpost-v0/post/EPOCH

The Authority replies with either `Accepted` and `Forbidden` http error codes.

7. Sphinx Message Format Specification

7.1 Introduction

The Sphinx cryptographic packet format is a compact and provably secure design introduced by George Danezis and Ian Goldberg [SPHINX09]. It supports a full set of security features: indistinguishable replies, hiding the path length and relay position, detection of tagging attacks and replay attacks, as well as providing unlinkability for each leg of the packet's journey over the network.

7.1.1 Terminology

- `Message` - A variable-length sequence of octets sent anonymously through the network.
- `Packet` - A fixed-length sequence of octets transmitted anonymously through the network, containing the encrypted message and metadata for routing.
- `Header` - The packet header consisting of several components, which convey the information necessary to verify packet integrity and correctly process the packet.
- `Payload` - The fixed-length portion of a packet containing an encrypted message or part of a message, to be delivered anonymously.
- `Group` - A finite set of elements and a binary operation that satisfy the properties of closure, associativity, invertability, and the presence of an identity element.
- `Group element` - An individual element of the group.
- `Group generator` - A group element capable of generating any other element of the group, via repeated applications of the generator and the group operation.

The "C" style Presentation Language as described in [RFC5246] Section 4 is used to represent data structures, except for cryptographic attributes, which are specified as opaque byte vectors.

`x | y` denotes the concatenation of x and y.

`x ^ y` denotes the bitwise XOR of x and y.

"`byte`" is an 8-bit octet.

`x[a:b]` denotes the sub-vector of x where a/b denote the start/end byte indexes (inclusive-exclusive); a/b may be omitted to signify the start/end of the vector x respectively.

`x[y]` denotes the y'th element of list x.

`x.len` denotes the length of list x.

`ZEROBYTES(N)` denotes N bytes of 0x00.

RNG(N) denotes N bytes of cryptographic random data.

LEN(N) denotes the length in bytes of N.

CONSTANT_TIME_CMP(x, y) denotes a constant time comparison between the byte vectors x and y, returning true iff x and y are equal.

7.2 Cryptographic Primitives

This specification uses the following cryptographic primitives as the foundational building blocks for Sphinx:

- H(M) - A cryptographic hash function which takes an octet array M to produce a digest consisting of a HASH_LENGTH byte octet array. H(M) MUST be pre-image and collision resistant.
- MAC(K, M) - A cryptographic message authentication code function which takes a M_KEY_LENGTH byte octet array key K and arbitrary length octet array message M to produce an authentication tag consisting of a MAC_LENGTH byte octet array.
- KDF(SALT, IKM) - A key derivation function which takes an arbitrary length octet array salt SALT and an arbitrary length octet array initial key IKM, to produce an octet array of arbitrary length.
- S(K, IV) - A pseudo-random generator (stream cipher) which takes a S_KEY_LENGTH byte octet array key K and a S_IV_LENGTH byte octet array initialization vector IV to produce an octet array key stream of arbitrary length.
- SPRP_Encrypt(K, M)/SPRP_Decrypt(K, M) - A strong pseudo-random permutation (SPRP) which takes a SPRP_KEY_LENGTH byte octet array key K and arbitrary length message M, and produces the encrypted ciphertext or decrypted plaintext respectively.

  When used with the default payload authentication mechanism, the SPRP MUST be "fragile" in that any amount of modifications to M results in a large number of unpredictable changes across the whole message upon a SPRP_Encrypt() or SPRP_Decrypt() operation.
- EXP(X, Y) - An exponentiation function which takes the GROUP_ELEMENT_LENGTH byte octet array group elements X and Y, and returns X ^^ Y as a GROUP_ELEMENT_LENGTH byte octet array.

  Let G denote the generator of the group, and EXP_KEYGEN() return a GROUP_ELEMENT_LENGTH byte octet array group element usable as private key.

  The group defined by G and EXP(X, Y) MUST satisfy the Decision Diffie-Hellman problem.
- EXP_KEYGEN() - Returns a new "suitable" private key for EXP().

7.2.1 Sphinx Key Derivation Function

Sphinx Packet creation and processing uses a common Key Derivation Function (KDF) to derive the required MAC and symmetric cryptographic keys from a per-hop shared secret.

The output of the KDF is partitioned according to the following structure:

```
struct {
    opaque header_mac[M_KEY_LENGTH];
    opaque header_encryption[S_KEY_LENGTH];
```

```
    opaque header_encryption_iv[S_IV_LENGTH];
    opaque payload_encryption[SPRP_KEY_LENGTH]
    opaque blinding_factor[GROUP_ELEMENT_LENGTH];
} SphinxPacketKeys;

Sphinx_KDF( info, shared_secret ) -> packet_keys
```

Inputs: `info` The optional context and application specific information.

`shared_secret` The per-hop shared secret derived from the Diffie-Hellman key exchange.

Outputs: `packet_keys` The SphinxPacketKeys required to handle packet creation or processing.

The output packet_keys is calculated as follows:

```
kdf_out = KDF( info, shared_secret )
packet_keys = kdf_out[:LEN( SphinxPacketKeys )]
```

7.3 Sphinx Packet Parameters

7.3.1 Sphinx Parameter Constants

The Sphinx Packet Format is parameterized by the implementation based on the application and security requirements.

- `AD_LENGTH` - The constant amount of per-packet unencrypted additional data in bytes.
- `PAYLOAD_TAG_LENGTH` - The length of the message payload authentication tag in bytes. This SHOULD be set to at least 16 bytes (128 bits).
- `PER_HOP_RI_LENGTH` - The length of the per-hop Routing Information (Section 7.4.1.1) in bytes.
- `NODE_ID_LENGTH` - The node identifier length in bytes.
- `RECIPIENT_ID_LENGTH` - The recipient identifier length in bytes.
- `SURB_ID_LENGTH` - The Single Use Reply Block (Section 7.7) identifier length in bytes.
- `MAX_HOPS` - The maximum number of hops a packet can traverse.
- `PAYLOAD_LENGTH` - The per-packet message payload length in bytes, including a `PAYLOAD_TAG_LENGTH` byte authentication tag.
- `KDF_INFO` - A constant opaque byte vector used as the info parameter to the KDF for the purpose of domain separation.

7.3.2 Sphinx Packet Geometry

The Sphinx Packet Geometry is derived from the Sphinx Parameter Constants (Section 7.3.1). These are all derived parameters, and are primarily of interest to implementors.

- `ROUTING_INFO_LENGTH` - The total length of the "routing information" Sphinx Packet Header component in bytes:

  ```
  ROUTING_INFO_LENGTH = PER_HOP_RI_LENGTH * MAX_HOPS
  ```

- `HEADER_LENGTH` - The length of the Sphinx Packet Header in bytes:

  ```
  HEADER_LENGTH = AD_LENGTH + GROUP_ELEMENT_LENGTH +
  ```

ROUTING_INFO_LENGTH + MAC_LENGTH

- PACKET_LENGTH - The length of the Sphinx Packet in bytes:

  PACKET_LENGTH = HEADER_LENGTH + PAYLOAD_LENGTH

7.4 The Sphinx Cryptographic Packet Structure

Each Sphinx Packet consists of two parts: the Sphinx Packet Header and the Sphinx Packet Payload:

```
struct {
    opaque header[HEADER_LENGTH];
    opaque payload[PAYLOAD_LENGTH];
} SphinxPacket;
```

- **header - The packet header consists of several components,** which convey the information necesary to verify packet integrity and correctly process the packet.
- payload - The application message data.

7.4.1 Sphinx Packet Header

The Sphinx Packet Header refers to the block of data immediately preceding the Sphinx Packet Payload in a Sphinx Packet.

The structure of the Sphinx Packet Header is defined as follows:

```
struct {
    opaque additional_data[AD_LENGTH]; /* Unencrypted. */
    opaque group_element[GROUP_ELEMENT_LENGTH];
    opaque routing_information[ROUTING_INFO_LENGTH];
    opaque MAC[MAC_LENGTH];
} SphinxHeader;
```

- additional_data - Unencrypted per-packet Additional Data (AD) that is visible to every hop. The AD is authenticated on a per-hop basis.

  As the additional_data is sent in the clear and traverses the network unaltered, implementations MUST take care to ensure that the field cannot be used to track individual packets.

- group_element - An element of the cyclic group, used to derive the per-hop key material required to authenticate and process the rest of the SphinxHeader and decrypt a single layer of the Sphinx Packet Payload encryption.
- routing_information - A vector of per-hop routing information, encrypted and authenticated in a nested manner. Each element of the vector consists of a series of routing commands, specifying all of the information required to process the packet.

  The precise encoding format is specified in Section 7.4.1.1.

- MAC - A message authentication code tag covering the additional_data, group_element, and routing_information.

7.4.1.1 Per-hop routing information

The routing_information component of the Sphinx Packet Header contains a vector of per-hop routing information. When processing a packet, the per hop processing is set up such that the first element in the vector contains the routing commands for the current hop.

The structure of the routing information is as follows:

```
struct {
    RoutingCommand routing_commands<1..2^8-1>; /* PER_HOP_RI_LENGTH bytes */
    opaque encrypted_routing_commands[ROUTING_INFO_LENGTH - PER_HOP_RI_LENGTH];
} RoutingInformation;
```

The structure of a single routing command is as follows:

```
struct {
    RoutingCommandType command;
    select (RoutingCommandType) {
        case null:              NullCommand;
        case next_node_hop:     NextNodeHopCommand;
        case recipient:         RecipientCommand;
        case surb_reply:        SURBReplyCommand;
    };
} RoutingCommand;
```

The following routing commands are currently defined:

```
enum {
    null(0),
    next_node_hop(1),
    recipient(2),
    surb_reply(3),

    /* Routing commands between 0 and 0x7f are reserved. */

    (255)
} RoutingCommandType;
```

The null routing command structure is as follows:

```
struct {
    opaque padding<0..PER_HOP_RI_LENGTH-1>;
} NullCommand;
```

The next_node_hop command structure is as follows:

```
struct {
    opaque next_hop[NODE_ID_LENGTH];
    opaque MAC[MAC_LENGTH];
} NextNodeHopCommand;
```

The recipient command structure is as follows:

```
struct {
    opaque recipient[RECIPEINT_ID_LENGTH];
} RecipientCommand;
```

The surb_reply command structure is as follows:

```
struct {
    opaque id[SURB_ID_LENGTH];
} SURBReplyCommand;
```

While the `NullCommand`'s padding field is specified as opaque, implementations SHOULD zero fill the padding. The choice of '0x00' as the terminal NullCommand is deliberate to ease implementation, as `ZEROBYTES(N)` produces a valid NullCommand RoutingCommand, resulting in "appending zero filled padding" producing valid output.

Implementations MUST pad the routing_commands vector so that it is exactly `PER_HOP_RI_LENGTH` bytes, by appending a terminal NullCommand if necessary.

Every non-terminal hop's `routing_commands` MUST include a `NextNodeHopCommand`.

7.4.2 Sphinx Packet Payload

The Sphinx Packet Payload refers to the block of data immediately following the Sphinx Packet Header in a Sphinx Packet.

For most purposes the structure of the Sphinx Packet Payload can be treated as a single contiguous byte vector of opaque data.

Upon packet creation, the payload is repeatedly encrypted (unless it is a SURB Reply, see Section 7.7) via keys derived from the Diffie-Hellman key exchange between the packet's `group_element` and the public key of each node in the path.

Authentication of packet integrity is done by prepending a tag set to a known value to the plaintext prior to the first encrypt operation. By virtue of the fragile nature of the SPRP function, any alteration to the encrypted payload as it traverses the network will result in an irrecoverably corrupted plaintext when the payload is decrypted by the recipient.

7.5 Sphinx Packet Creation

For the sake of brevity, the pseudocode for all of the operations will take a vector of the following PathHop structure as a parameter named path[] to specify the path a packet will traverse, along with the per-hop routing commands and per-hop public keys.

```
struct {
    /* There is no need for a node_id here, as
       routing_commands[0].next_hop specifies that
       information for all non-terminal hops. */
    opaque public_key[GROUP_ELEMENT_LENGTH];
    RoutingCommand routing_commands<1...2^8-1>;
} PathHop;
```

It is assumed that each routing_commands vector except for the terminal entry contains at least a RoutingCommand consisting of a partially assembled NextNodeHopCommand with the `next_hop` element filled in with the identifier of the next hop.

7.5.1 Create a Sphinx Packet Header

Both the creation of a Sphinx Packet and the creation of a SURB requires the generation of a Sphinx Packet Header, so it is specified as a distinct operation.

```
Sphinx_Create_Header( additional_data, path[] ) -> sphinx_header,
                                                    payload_keys
```

Inputs: `additional_data` **The Additional Data that is visible to** every node along the path in the header.

`path` **The vector of PathHop structures in hop** order, specifying the node id, public key, and routing commands for each hop.

Outputs: `sphinx_header` The resulting Sphinx Packet Header.

`payload_keys` **The vector of SPRP keys used to encrypt** the Sphinx Packet Payload, in hop order.

The `Sphinx_Create_Header` operation consists of the following steps:

1. Derive the key material for each hop.

```
num_hops = route.len
route_keys = [ ]
route_group_elements = [ ]
priv_key = EXP_KEYGEN()

/* Calculate the key material for the 0th hop. */
route_group_elements += EXP( G, priv_key )
shared_secret = EXP( path[0].public_key, priv_key )
route_keys += Sphinx_KDF( KDF_INFO, shared_secret )
blinding_factor = keys[0].blinding_factor

/* Calculate the key material for rest of the hops. */
for i = 1; i < num_hops; ++i:
    shared_secret = EXP( path[i].public_key, priv_key )
    for j = 0; j < i; ++j:
        shared_secret = EXP( shared_secret, keys[j].blinding_factor )
    route_keys += Sphinx_KDF( KDF_INFO, shared_secret )
    route_group_elements += EXP( group_element, keys[i-1].blinding_factor )

At the conclusion of the derivation process:
    route_keys           - A vector of per-hop SphinxKeys.
    route_group_elements - A vector of per-hop group elements.
```

2. Derive the routing_information keystream and encrypted padding for each hop.

```
ri_keystream = [ ]
ri_padding = [ ]

for i = 0; i < num_hops; ++i:
    keystream = ZEROBYTES( ROUTING_INFO_LENGTH + PER_HOP_RI_LENGTH ) ^
                  S( route_keys[i].header_encryption,
                     route_keys[i].header_encryption_iv )
    ks_len = LEN( keystream ) - (i + 1) * PER_HOP_RI_LENGTH

    padding = keystream[ks_len:]
    if i > 0:
        prev_pad_len = LEN( ri_padding[i-1] )
        padding = padding[:prev_pad_len] ^ ri_padding[i-1] |
```

```
            padding[prev_pad_len]

    ri_keystream += keystream[:ks_len]
    ri_padding += padding

At the conclusion of the derivation process:
   ri_keystream - A vector of per-hop routing_information
                  encryption keystreams.
   ri_padding   - The per-hop encrypted routing_information
                  padding.
```

3. Create the routing_information block.

```
/* Start with the terminal hop, and work backwards. */
i = num_hops - 1

/* Encode the terminal hop's routing commands. As the
   terminal hop can never have a NextNodeHopCommand, there
   are no per-hop alterations to be made. */
ri_fragment = path[i].routing_commands |
   ZEROBYTES( PER_HOP_RI_LENGTH - LEN( path[i].routing_commands ) )

/* Encrypt and MAC. */
ri_fragment ^= ri_keystream[i]
mac = MAC( route_keys[i].header_mac, additional_data |
               route_group_elements[i] | ri_fragment |
               ri_padding[i-1] )
routing_info = ri_fragment
if num_hops < MAX_HOPS:
    pad_len = (MAX_HOPS - num_hops) * PER_HOP_RI_LENGTH
    routing_info = routing_info | ZEROBYTES( pad_len )

/* Calculate the routing info for the rest of the hops. */
for i = num_hops - 2; i >= 0; --i:
    cmds_to_encode = [ ]

    /* Find and finalize the NextNodeHopCommand. */
    for j = 0; j < LEN( path[i].routing_commands; j++:
        cmd = path[i].routing_commands[j]
        if cmd.command == next_node_hop:
          /* Finalize the NextNodeHopCommand. */
          cmd.MAC = mac
        cmds_to_encode = cmds_to_encode + cmd /* Append */

    /* Append a terminal NullCommand. */
    ri_fragment = cmds_to_encode |
        ZEROBYTES( PER_HOP_RI_LENGTH - LEN( cmds_to_encode ) )

    /* Encrypt and MAC */
    routing_info = ri_fragment | routing_info /* Prepend. */
    routing_info ^= ri_keystream[i]
    if i > 0:
```

```
        mac = MAC( route_keys[i].header_mac, additional_data |
                   route_group_elements[i] | routing_info |
                   ri_padding[i-1] )
    else:
        mac = MAC( route_keys[i].header_mac, additional_data |
                   route_group_elements[i] | routing_info )

At the conclusion of the derivation process:
   routing_info - The completed routing_info block.
   mac          - The MAC for the 0th hop.
```

4. Assemble the completed Sphinx Packet Header and Sphinx Packet Payload SPRP key vector.

```
/* Assemble the completed Sphinx Packet Header. */
SphinxHeader sphinx_header
sphinx_header.additional_data = additional_data
sphinx_header.group_element = route_group_elements[0] /* From step 1. */
sphinx_header.routing_info = routing_info   /* From step 3. */
sphinx_header.mac = mac                     /* From step 3. */

/* Preserve the Sphinx Payload SPRP keys, to return to the
   caller. */
payload_keys = [ ]
for i = 0; i < nr_hops; ++i:
    payload_keys += route_keys[i].payload_encryption

At the conclusion of the assembly process:
   sphinx_header - The completed sphinx_header, to be returned.
   payload_keys  - The vector of SPRP keys, to be returned.
```

7.5.2 Create a Sphinx Packet

```
Sphinx_Create_Packet( additional_data, path[], payload ) -> sphinx_packet
```

Inputs: **`additional_data` The Additional Data that is visible to** every node along the path in the header.

`path` The vector of PathHop structures in hop order, specifying the node id, public key, and routing commands for each hop.

`payload` The packet payload message plaintext.

Outputs: `sphinx_packet` The resulting Sphinx Packet.

The `Sphinx_Create_Packet` operation consists of the following steps:

1. Create the Sphinx Packet Header and SPRP key vector.

```
sphinx_header, payload_keys =
    Sphinx_Create_Header( additional_data, path )
```

2. Prepend the authentication tag, and append padding to the payload.

```
payload = ZERO_BYTES( PAYLOAD_TAG_LENGTH ) | payload
payload = payload | ZERO_BYTES( PAYLOAD_LENGTH - LEN( payload ) )
```

3. Encrypt the payload.

```
for i = nr_hops - 1; i >= 0; --i:
    payload = SPRP_Encrypt( payload_keys[i], payload )
```

4. Assemble the completed Sphinx Packet.

```
SphinxPacket sphinx_packet
sphinx_packet.header = sphinx_header
sphinx_packet.payload = payload
```

7.6 Sphinx Packet Processing

Mix nodes process incoming packets first by performing the `Sphinx_Unwrap` operation to authenticate and decrypt the packet, and if applicable prepare the packet to be forwarded to the next node.

If `Sphinx_Unwrap` returns an error for any given packet, the packet MUST be discarded with no additional processing.

After a packet has been unwrapped successfully, a replay detection tag is checked to ensure that the packet has not been seen before. If the packet is a replay, the packet MUST be discarded with no additional processing.

The routing commands for the current hop are interpreted and executed, and finally the packet is forwarded to the next mix node over the network or presented to the application if the current node is the final recipient.

7.6.1 Sphinx_Unwrap Operation

The `Sphinx_Unwrap` operation is the majority of the per-hop packet processing, handling authentication, decryption, and modifying the packet prior to forwarding it to the next node.

```
Sphinx_Unwrap( routing_private_key, sphinx_packet ) -> sphinx_packet,
                                                       routing_commands,
                                                       replay_tag
```

Inputs: `private_routing_key` A group element GROUP_ELEMENT_LENGTH bytes in length, that serves as the unwrapping Mix's private key.

`sphinx_packet` A Sphinx packet to unwrap.

Outputs: `error` Indicating a unsuccessful unwrap operation if applicable.

`sphinx_packet` The resulting Sphinx packet.

`routing_commands` A vector of RoutingCommand, specifying the post unwrap actions to be taken on the packet.

`replay_tag` A tag used to detect whether this packet was processed before.

The `Sphinx_Unwrap` operation consists of the following steps:

0. (Optional) Examine the Sphinx Packet Header's Additional Data.

If the header's `additional_data` element contains information required to complete the unwrap operation, such as specifying the packet format version or the cryptographic primitives used examine it now.

Implementations MUST NOT treat the information in the `additional_data` element as trusted until after the completion of Step 3 ("Validate the Sphinx Packet Header").

1. Calculate the hop's shared secret, and replay_tag.

```
hdr = sphinx_packet.header
shared_secret = EXP( hdr.group_element, private_routing_key )
replay_tag = H( shared_secret )
```

2. Derive the various keys required for packet processing.

```
keys = Sphinx_KDF( KDF_INFO, shared_secret )
```

3. Validate the Sphinx Packet Header.

```
derived_mac = MAC( keys.header_mac, hdr.additional_data |
                  hdr.group_element |
                  hdr.routing_information )
if !CONSTANT_TIME_CMP( derived_mac, hdr.MAC):
    /* MUST abort processing if the header is invalid. */
    return ErrorInvalidHeader
```

4. Extract the per-hop routing commands for the current hop.

```
/* Append padding to preserve length-invariance, as the routing
    commands for the current hop will be removed. */
padding = ZEROBYTES( PER_HOP_RI_LENGTH )
B = hdr.routing_information | padding

/* Decrypt the entire routing_information block. */
B = B ^ S( keys.header_encryption, keys.header_encryption_iv )
```

5. Parse the per-hop routing commands.

```
cmd_buf = B[:PER_HOP_RI_LENGTH]
new_routing_information = B[PER_HOP_RI_LENGTH:]

next_mix_command_idx = -1
routing_commands = [ ]
for idx = 0; idx < PER_HOP_RI_LENGTH {
     /* WARNING: Bounds checking omitted for brevity. */
     cmd_type = b[idx]
     cmd = NULL
     switch cmd_type {
        case null: goto done  /* No further commands. */

        case next_node_hop:
            cmd = RoutingCommand( B[idx:idx+1+LEN( NextNodeHopCommand )] )
            next_mix_command_idx = i /* Save for step 7. */
            idx += 1 + LEN( NextNodeHopCommand )
            break

        case recipient:
            cmd = RoutingCommand( B[idx:idx+1+LEN( FinalDestinationCommand )] )
            idx += 1 + LEN( RecipientCommand )
            break
```

```
        case surb_reply:
            cmd = RoutingCommand( B[idx:idx+1+LEN( SURBReplyCommand )] )
            idx += 1 + LEN( SURBReplyCommand )
            break

      default:
            /* MUST abort processing on unrecognized commands. */
            return ErrorInvalidCommand
    }
    routing_commands += cmd /* Append cmd to the tail of the list. */
}
done:
```

At the conclusion of the parsing step: `routing_commands` - A vector of SphinxRoutingCommand, applied at this hop.

`new_routing_information` - The routing_information block to be sent to the next hop if any.

6. Decrypt the Sphinx Packet Payload.

```
payload = sphinx_packet.payload
payload = SPRP_Decrypt( key.payload_encryption, payload )
sphinx_packet.payload = payload
```

7. Transform the packet for forwarding to the next mix, iff the routing commands vector included a NextNodeHopCommand.

```
if next_mix_command_idx != -1:
    cmd = routing_commands[next_mix_command_idx]
    hdr.group_element = EXP( hdr.group_element, keys.blinding_factor )
    hdr.routing_information = new_routing_information
    hdr.mac = cmd.MAC
    sphinx_packet.hdr = hdr
```

7.6.2 Post Sphinx_Unwrap Processing

Upon the completion of the `Sphinx_Unwrap` operation, implementations MUST take several additional steps. As the exact behavior is mostly implementation specific, pseudocode will not be provided for most of the post processing steps.

1. Apply replay detection to the packet.

   The `replay_tag` value returned by Sphinx_Unwrap MUST be unique across all packets processed with a given `private_routing_key`.

   The exact specifics of how to detect replays is left up to the implementation, however any replays that are detected MUST be discarded immediately.

2. Act on the routing commands, if any.

   The exact specifics of how implementations chose to apply routing commands is deliberately left unspecified, however in general:

   - Iff there is a `NextNodeHopCommand`, the packet should be forwarded to the next node based on the `next_hop` field upon completion of the post processing.

The lack of a NextNodeHopCommand indicates that the packet is destined for the current node.

- Iff there is a `SURBReplyCommand`, the packet should be treated as a SURBReply destined for the current node, and decrypted accordingly (See Section 7.2).
- If the implementation supports multiple recipients on a single node, the `RecipientCommand` command should be used to determine the correct recipient for the packet, and the payload delivered as appropriate.

  Note: It is possible for both a RecipientCommand and a NextNodeHopCommand to be present simultaneously in the routing commands for a given hop. The behavior when this situation occurs is implementation defined.

3. Authenticate the packet if required.

   Iff the packet is destined for the current node, the integrity of the payload MUST be authenticated.

   The authentication is done as follows:

```
derived_tag = sphinx_packet.payload[:PAYLOAD_TAG_LENGTH]
expected_tag = ZEROBYTES( PAYLOAD_TAG_LENGTH )
if !CONSTANT_TIME_CMP( derived_tag, expected_tag ):
    /* Discard the packet with no further processing. */
    return ErrorInvalidPayload

/* Remove the authentication tag before presenting the
   payload to the application. */
sphinx_packet.payload = sphinx_packet.payload[PAYLOAD_TAG_LENGTH:]
```

7.7 Single Use Reply Block (SURB) Creation

A Single Use Reply Block (SURB) is a delivery token with a short lifetime, that can be used by the recipient to reply to the initial sender.

SURBs allow for anonymous replies, when the recipient does not know the sender of the message. Usage of SURBs guarantees anonymity properties but also makes the reply messages indistinguishable from forward messages both to external adversaries as well as the mix nodes.

When a SURB is created, a matching reply block Decryption Token is created, which is used to decrypt the reply message that is produced and delivered via the SURB.

The Sphinx SURB wire encoding is implementation defined, but for the purposes of illustrating creation and use, the following will be used:

```
struct {
    SphinxHeader sphinx_header;
    opaque first_hop[NODE_ID_LENGTH];
    opaque payload_key[SPRP_KEY_LENGTH];
} SphinxSURB;
```

7.7.1 Create a Sphinx SURB and Decryption Token

Structurally a SURB consists of three parts, a pre-generated Sphinx Packet Header, a node identifier for the first hop to use when using the SURB to reply, and cryptographic keying material by which to encrypt the reply's payload. All elements must be securely transmitted to the recipient, perhaps as part of a forward Sphinx Packet's Payload, but the exact specifics on how to accomplish this is left up to the implementation.

When creating a SURB, the terminal routing_commands vector SHOULD include a SURBReplyCommand, containing an identifier to ensure that the payload can be decrypted with the correct set of keys (Decryption Token). The routing command is left optional, as it is conceivable that implementations may chose to use trial decryption, and or limit the number of outstanding SURBs to solve this problem.

```
Sphinx_Create_SURB( additional_data, first_hop, path[] ) ->
                                                 sphinx_surb,
                                                 decryption_token
```

Inputs: **`additional_data` The Additional Data that is visible to** every node along the path in the header.

`first_hop` The node id of the first hop the recipient must use when replying via the SURB.

`path` The vector of PathHop structures in hop order, specifying the node id, public key, and routing commands for each hop.

Outputs: `sphinx_surb` The resulting Sphinx SURB.

`decryption_token` The Decryption Token associated with the SURB.

The Sphinx_Create_SURB operation consists of the following steps:

1. Create the Sphinx Packet Header and SPRP key vector.

```
sphinx_header, payload_keys =
      Sphinx_Create_Header( additional_data, path )
```

2. Create a key for the final layer of encryption.

```
final_key = RNG( SPRP_KEY_LENGTH )
```

3. Build the SURB and Decryption Token.

```
SphinxSURB sphinx_surb;
sphinx_surb.sphinx_header = sphinx_header
sphinx_surb.first_hop = first_hop
sphinx_surb.payload_key = final_key

decryption_token = final_key + payload_keys /* Prepend */
```

7.7.2 Decrypt a Sphinx Reply Originating from a SURB

A Sphinx Reply packet that was generated using a SURB is externally indistinguishable from a forward Sphinx Packet as it traverses the network. However, the recipient of the reply has an additional decryption step, the packet starts off unencrypted, and accumulates layers of Sphinx Packet Payload decryption as it traverses the network.

Determining which decryption token to use when decrypting the SURB reply can be done via

the SURBReplyCommand's id field, if one is included at the time of the SURB's creation.

`Sphinx_Decrypt_SURB_Reply( decryption_token, payload ) -> message`

Inputs: **`decryption_token` The vector of keys allowing a client to** decrypt the reply ciphertext payload. This decryption_token is generated when the SURB is created.

`payload` The Sphinx Packet ciphertext payload.

Outputs: **`error` Indicating a unsuccessful unwrap** operation if applicable.

`message` The plaintext message.

The Sphinx_Decrypt_SURB_Reply operation consists of the following steps:

1. Encrypt the message to reverse the decrypt operations the payload acquired as it traversed the network.

```
for i = LEN( decryption_token ) - 1; i > 0; --i:
    payload = SPRP_Encrypt( decryption_token[i], payload )
```

2. Decrypt and authenticate the message ciphertext.

```
message = SPRP_Decrypt( decryption_token[0], payload )

derived_tag = message[:PAYLOAD_TAG_LENGTH]
expected_tag = ZEROBYTES( PAYLOAD_TAG_LENGTH )
if !CONSTANT_TIME_CMP( derived_tag, expected_tag ):
    return ErrorInvalidPayload

message = message[PAYLOAD_TAG_LENGTH:]
```

7.8 Single Use Reply Block Replies

The process for using a SURB to reply anonymously is slightly different from the standard packet creation process, as the Sphinx Packet Header is already generated (as part of the SURB), and there is an additional layer of Sphinx Packet Payload encryption that must be performed.

`Sphinx_Create_SURB_Reply( sphinx_surb, payload ) -> sphinx_packet`

Inputs: **`sphinx_surb` The SphinxSURB structure, decoded from** the implementation defined wire encoding.

`payload` The packet payload message plaintext.

The Sphinx_Create_SURB_Reply operation consists of the following steps:

1. Prepend the authentication tag, and append padding to the payload.

```
payload = ZERO_BYTES( PAYLOAD_TAG_LENGTH ) | payload
payload = payload | ZERO_BYTES( PAYLOAD_LENGTH - LEN( payload ) )
```

2. Encrypt the payload.

```
payload = SPRP_Encrypt( sphinx_surb.payload_key, payload )
```

3. Assemble the completed Sphinx Packet.

```
SphinxPacket sphinx_packet
sphinx_packet.header = sphinx_surb.sphinx_header
```

```
sphinx_packet.payload = payload
```

The completed `sphinx_packet` MUST be sent to the node specified via `sphinx_surb.node_id`, as the entire reply `sphinx_packet`'s header is pre-generated.

7.9 Anonymity Considerations

7.9.1 Optional Non-constant Length Sphinx Packet Header Padding

Depending on the mix topology, there is no hard requirement that the per-hop routing info is padded to one fixed constant length.

For example, assuming a layered topology (referred to as stratified topology in the literature) [MIXTOPO10], where the layer of any given mix node is public information, as long as the following two invariants are maintained, there is no additional information available to an adversary:

1. All packets entering any given mix node in a certain layer are uniform in length.
2. All packets leaving any given mix node in a certain layer are uniform in length.

The only information available to an external or internal observer is the layer of any given mix node (via the packet length), which is information they are assumed to have by default in such a design.

7.9.2 Additional Data Field Considerations

The Sphinx Packet Construct is crafted such that any given packet is bitwise unlinkable after a Sphinx_Unwrap operation, provided that the optional Additional Data (AD) facility is not used. This property ensures that external passive adversaries are unable to track a packet based on content as it traverses the network. As the on-the-wire AD field is static through the lifetime of a packet (i.e., left unaltered by the `Sphinx_Unwrap` operation), implementations and applications that wish to use this facility MUST NOT transmit AD that can be used to distinctly identify individual packets.

7.9.3 Forward Secrecy Considerations

Each node acting as a mix MUST regenerate their asymmetric key pair relatively frequently. Upon key rotation the old private key MUST be securely destroyed. As each layer of a Sphinx Packet is encrypted via key material derived from the output of an ephemeral/static Diffie-Hellman key exchange, without the rotation, the construct does not provide Perfect Forward Secrecy. Implementations SHOULD implement defense-in-depth mitigations, for example by using strongly forward-secure link protocols to convey Sphinx Packets between nodes.

This frequent mix routing key rotation can limit SURB usage by directly reducing the lifetime of SURBs. In order to have a strong Forward Secrecy property while maintaining a higher SURB lifetime, designs such as forward secure mixes [SFMIX03] could be used.

7.9.4 Compulsion Threat Considerations

Reply Blocks (SURBs), forward and reply Sphinx packets are all vulnerable to the compulsion threat, if they are captured by an adversary. The adversary can request iterative decryptions or

keys from a series of honest mixes in order to perform a deanonymizing trace of the destination.

While a general solution to this class of attacks is beyond the scope of this document, applications that seek to mitigate or resist compulsion threats could implement the defenses proposed in [COMPULS05] via a series of routing command extensions.

7.9.5 SURB Usage Considerations for Volunteer Operated Mix Networks

Given a hypothetical scenario where Alice and Bob both wish to keep their location on the mix network hidden from the other, and Alice has somehow received a SURB from Bob, Alice MUST not utilize the SURB directly because in the volunteer operated mix network the first hop specified by the SURB could be operated by Bob for the purpose of deanonymizing Alice.

This problem could be solved via the incorporation of a "cross-over point" such as that described in [MIXMINION], for example by having Alice delegating the transmission of a SURB Reply to a randomly selected crossover point in the mix network, so that if the first hop in the SURB's return path is a malicious mix, the only information gained is the identity of the cross-over point.

7.10 Security Considerations

7.10.1 Sphinx Payload Encryption Considerations

The payload encryption's use of a fragile (non-malleable) SPRP is deliberate and implementations SHOULD NOT substitute it with a primitive that does not provide such a property (such as a stream cipher based PRF). In particular there is a class of correlation attacks (tagging attacks) targeting anonymity systems that involve modification to the ciphertext that are mitigated if alterations to the ciphertext result in unpredictable corruption of the plaintext (avalanche effect).

Additionally, as the PAYLOAD_TAG_LENGTH based tag-then-encrypt payload integrity authentication mechanism is predicated on the use of a non-malleable SPRP, implementations that substitute a different primitive MUST authenticate the payload using a different mechanism.

Alternatively, extending the MAC contained in the Sphinx Packet Header to cover the Sphinx Packet Payload will both defend against tagging attacks and authenticate payload integrity. However, such an extension does not work with the SURB construct presented in this specification, unless the SURB is only used to transmit payload that is known to the creator of the SURB.

8. Mix Network End-to-end Protocol Specification

8.1 Introduction

Fundamentally a mix network is a lossy packet switching network on which we can build reliable protocols. We therefore utilize a variety of designs from both the mix network and classical internet protocol literature to design an end to end reliability protocol that utilizes the mix network.

8.1.1. Terminology

- `ACK` - A protocol acknowledgment of a previously sent Block.
- `ARQ` - Automatic Repeat reQuest is an error correction method which requires two-way communication and incurs a delay penalty when used.
- **Classes of traffic - We distinguish the following classes of traffic:** – ACKs
    - Small messages
    - Large messages (big attachments)
- `Block` - A fragment of a message that fits into a single packet of a specified class of traffic.
- `Block ID` - A unique identifier for a Block.
- `Client` - Software run by the human being on its local device.
- `E2E Encrypted message` - An encrypted message.
- `Message` - A variable size end-to-end message, transmitted from one location to another. The message can be classified into one of the classes of traffic, depending on the message size, and transported as a single packet or divided into several packets.
- `Message ID` - A unique identifier for a message.
- `Mix` - A server that provides anonymity to clients by accepting messages encrypted to its public key, which it then decrypts, delays for a given amount of time, and transmits either to another mix or to a provider (as specified in the messages). Those operations provide bitwise unlinkability between input and output messages as well as long term correlation resistance.
- `Provider` - The provider is a client's single point of failure for participating in the mix network because it is responsible for authorising sent messages as well as storing received messages on behalf of the user. Provider MUST perform the same cryptographic operations as the Mix.

- `Packet` - A Sphinx packet.
- `SURB-ACK` - A short message notifying that a packet was delivered; transmitted via a Single Use Reply Block [DANEZISG09].
- `SURB_SIZE = sizeof(SphinxSURB)` where `SphinxSURB` is a Single Use Reply Block defined in the "Sphinx Mix Network Cryptographic Packet Format Specification".

8.1.2 Constants

`BLOCK_LENGTH` The maximum payload size of a block (message fragment). The value of `BLOCK-LENGTH` depends on the class of traffic.

8.2 Mix Network Packet Format Considerations

As the mix network message packet format we use Sphinx. The detailed Sphinx specification is described in: "Sphinx Mix Network Cryptographic Packet Format Specification" [SPHINXSPEC].

The Sphinx cryptographic primitives and parameters are specified in Section 3 of: "The Katzenpost Mix Network Specification"

8.3 Client and Provider Core Protocol

All client mixnet interaction happens through their Provider, reusing the existing trust relationship any given user may have with an e-mail service provider, and all client to Provider interaction will use the Katzenpost Mix Network Wire Protocol, described in "Katzenpost Mix Network Wire Protocol Specification" [KATZMIXWIRE].

8.3.1 Handshake and Authentication

Let the contents of the wire protocol AuthenticateMessage's `additional_data` field consist of the local-part component of a client's e-mail address if the client is authenticating, padded with NUL bytes to exactly 64 bytes in length.

In the case that the authenticating party is a Provider instance, let the `additional_data` field contain the domain name that the Provider is responsible for mail for.

8.3.2 Client Retrieval of Queued Messages

Clients periodically poll their Provider for messages that may have been enqueued in that user's mailbox. All wire protocol commands including these defined commands MUST come after the above described handshake and authentication. We define two additional wire protocol commands:

```
enum {
    /* Extending the wire protocol Commands. */
    retrieve_message(16),
    message(17),
} Command;
```

The structures of these commands are defined as follows:

```
struct {
    uint32_t sequence_number;
} RetrieveMessage;

enum {
    ack(0),
    message(1),
    empty(2),
} message_type;

struct {
    opaque surb_id[SURB_ID_LENGTH];
    opaque encrypted_payload[SURB_PAYLOAD_LENGTH];
} Ack;

struct {
    opaque encrypted_payload[PAYLOAD_LENGTH];
    opaque padding[sizeof(Ack) - PAYLOAD_LENGTH];
} MessageCiphertext;

struct {
    message_type type;
    uint8_t queue_size_hint;
    uint32 sequence_number;
    select (message_type) {
        case ack:    Ack;
        default:     MessageCiphertext;
    };
} Message;
```

8.3.2.1 The retrieve_message and message Commands

Once a client is connected to the Provider and has entered the data transfer phase after completing the handshake and authentication, the client may start to retrieve messages from the provider via issuing the `retrieve_message` command.

The `retrieve_message` command contains a sequence number which the client initially sets to `0` at the beginning of each session. This sequence number is incremented each time the client receives a message from the provider (as a message command), except if the `message_type` is `empty` indicating that the client's inbound message queue is empty, as no message has been received.

Clients MUST NOT have more than one outstanding retrieve_message command at a given time.

The Provider MUST respond to retrieve_message commands, in the following manner:

1. Validate that the `sequence_number` is in the expected range, and that there are no other `retrieve_message` commands originating from a particular session being serviced. If the `sequence_number` is unexpected, or the client is issuing multiple `retrieve_message` commands, the session MUST be terminated.

2. If the sequence_number has been incremented, indicating that the client has received the last `message` reply, remove the 0th message from the client's message queue and delete it securely.
3. Send a message command as a response, with the following values for the `Message` fields (as the command's payload).

   `type` - The type of the message that is being transported.

   `queue_size_hint` - The size of the client's inbound message queue, excluding the message currently being sent, clamped to 255.

   `sequence_number` - The sequence number of the retrieve_message.

   If the 0th message is a SURB-ACK:

   `surb_id` - The SURB's identifier taken from the SURBReplyCommand in the Sphinx packet header that delivered the SURB.

   If the message type empty, a `MessageCiphertext` is still embedded in the Message structure, however the contents MUST be zero filled (filled with `0x00` bytes).

Clients MAY use the `queue_size_hint` to determine if additional retreive_message commands should be issued soon, or if they can delay the next retreive_message under the assumption that the queue is empty.

Providers SHOULD attempt to service `retrieve_message` commands in a timely manner.

8.4 Client and Provider processing of received packets

This section describes the protocol that reliably transmits messages across the mix network to the destination Provider.

It is assumed that all clients have a long lived X25519 keypair, the public component of which is known in advance to all peers who wish to communicate securely with them. How to distribute such keying information is beyond the scope of this document.

Messages begin at the sender as byte strings containing an e-mail in the Internet Message Format (IMF) [RFC5322].

Preparing a message for transport takes the following steps:

1. The message is fragmented into block(s).

   The block structure is as follows:

```
struct {
    opaque message_id[16];
    uint16_t total_blocks;
    uint16_t block_id;
    uint32_t block_length;
    opaque block[block_length];
    opaque padding[BLOCK_LENGTH-block_length]; /* 0x00s */
} Block;
```

   Where:

   `message_id` - A unique identifier, consistent across all Block(s) belonging to a given message.

`total_blocks` - The number of Block(s) that make up the fully reassembled message.

`block_id` - The sequence number of the Block as a component of a stream of Block(s) making up a message, starting at `0`.

`block_length` - The length of the Block's message fragment.

`block` - The Block's message fragment.

`padding` - Padding, applied to the terminal Block.

The padding if any MUST contain `0x00s` (ie: be zero padded).

The `message_id` SHOULD be trivially collision resistant, and SHOULD NOT be reused while there is a possibility that the recipient can end up Block(s) belonging to multiple messages with a colliding `message_id`.

2. Encrypt and authenticate each block.

   Each Block is encrypted and authenticated as a Noise protocol [NOISE] handshake plus transport message, using the recipient's long term X25519 public key, the sender's long term X25519 keypair, and a freshly generated ephemeral X25519 keypair.

   `Noise_X_25519_ChaChaPoly_Blake2b` is used as the Noise protocol name and parameterization for the purpose of Block encryption.

   Let the encrypted and authenticated Block be refered to as the following:

```
struct {
    /* Noise protocol fields. */
    opaque noise_e[32];     /* The Noise handshake `e`. */
    opaque noise_s_mac[16]; /* The Noise handshake `s` MAC. */
    opaque noise_s[32];     /* The Noise handshake `s`. */
    opaque noise_mac[16];   /* The Noise ciphertext MAC. */

    opaque ciphertext[BLOCK_LENGTH];
} BlockCiphertext;
```

3. Derive the path(s) and delays for each block.

   Prior to the creation of the Sphinx packet(s) that will transport each message, it is necessary to pre-calculate the forward and optional return path(s), for each BlockCiphertext and it's optional associated SURB-ACK.

   While the sender's provider is not, strictly speaking a "mix", it will apply Sphinx packet processing as if it is a mix, and therefore MUST have a delay.

   The recipient's provider MUST NOT have a delay.

   See Section 8.5.1 and Section 8.5.2 for details.

4. (Optional) Create the SURB-ACK's Single Use Reply Block for each block.

   To allow for reliable transmission we use acknowledgments encapsulated in the Single-User Reply Blocks (SURB) of the Sphinx packet format (see "The Sphinx Packet Format Specification"). We refer to these as SURB-ACKs.

   In order to create a SURB-ACK the Client uses the input obtained from the PKI with all the addresses and public keys of the nodes, where nodes include both providers and mixes.

The new path and set of delays for each SURB-ACK are selected independently following Step 4.

This SURB-ACK is included in the Sphinx packet of the forward message, in the payload that is received by the egress provider.

5. Assemble each BlockCiphertext and (Optional) SURBs into Sphinx packet payload.

   Let the Sphinx packet payload consist of the following:

```
struct {
   uint8_t flags;
   uint8_t reserved; /* Set to 0x00. */
   select (flags) {
   case 0:
       opaque padding[sizeof(SphinxSURB)];
   case 1:
       SphinxSURB surb;
   }
   BlockCiphertext ciphertext[];
} BlockSphinxPlaintext;
```

   All non-terminal hops MUST have a `NodeDelayCommand` and `NextNodeHopCommand` command in the per-hop routing command vector.

   The terminal hop for all forward Sphinx packets MUST have a recipient command in the per-hop routing command vector containing the recipient's identifier (the local-part of the recipient's e-mail address).

   The terminal hop of all SURB-ACKs MUST have a recipient command in the per-hop command vector containing the sender's identifier, and additionally have a surb_reply command containing the ID of the SURB.

6. Send each Sphinx packet via the `send_packet` command.

   Each Sphinx packet is then send out via the sender's Provider into the mixnet, using the `send_packet` wire protocol command.

   The sender SHOULD impose a random delay between each packet, and if the sender chooses to implement this functionality such delay MUST be factored into the path and delay derivation done in step 3.

7. (Optional) Retransmit lost blocks as needed.

   If the SURB-ACK functionality is used, the sender will receive a SURB, containing an ACK, per block from the recipient's provider signalling that the Sphinx packet has arrived, was successfully processed, and queued for delivery to the recipient.

   As the sender specifies all mixing delays in advance, the time that a SURB-ACK should arrive for any given block is known to reasonable accuracy in advance.

   If the sender determines that a Sphinx packet was lost (for example by the lack of a SURB-ACK at around the expected time, factoring in potential additional network delays), it SHOULD retransmit the block. The exact ARQ strategy used to determine when a block is considered lost, and which blocks to retransmit is left up to the implementation, however the following rules MUST be obeyed:

   - All retransmitted blocks MUST be re-encrypted, and have a entirely new set of paths and delays. In simple terms, this means re-doing the packet creation/transmission

from step 2 for each retransmitted block.

- Senders MUST NOT retransmit blocks at a rate faster than one block per 3 seconds.
- Senders MUST NOT attempt to retransmit blocks indefinitely, and instead give up on the entire message after it fails to arrive after a certain number of retransmissions.

8.4.1 Provider Behavior for Receiving Messages from the Mix Network

All Providers MUST accept inbound connections from the final layer of the mix network, and receive Sphinx packets. Upon receiving a Sphinx packet, the provider MUST do the following things:

1. Unwrap the Sphinx packet.

   All unwrapped packets MUST have at least a recipient command in the per-hop command vector specifying which client the packet is destined for.

   Providers MUST discard all packets that are either missing recipient information, or that are addressed to unknown recipients with no additional processing.

2. Handle the unwrapped packet.

   Iff the Sphinx packet did not have a `surb_reply` command in the per-hop command vector, then the payload MUST be interpreted as a `BlockSphinxPlaintext` as follows:

   (a) The Provider queues the packet's ciphertext field for later delivery to the client (via the retrieval mechanism specified in section 8.3.2).

   (b) After the ciphertext has been queued into persistent storage, the Provider MUST generate the ack's payload, concatenate with the received SURB-ACK header and transmit a SURB-ACK, iff the `BlockSphinxPlaintext`'s flags is equal to `1`, and a valid SURB is present in the payload.

   The SURB-ACK payload MUST be completely zero filled (contain only `0x00` bytes).

   Providers MUST NOT generate and transmit a SURB-ACK unless the ciphertext has been successfully queued for delivery.

Iff the Sphinx packet has a `surb_reply` command in the per-hop command vector, then the entire Sphinx packet payload, along with the `surb_id` value from the `surb_reply` command is queued for later delivery to the client.

8.4.2 Client Receive Message Behavior

Clients periodically poll their Provider with a retreive_message command. This section describes the client behavior upon receiving messages from their Provider, based on type.

8.4.2.1 Client Message Processing

When a client receives an inbound message from their provider, denoted as such by virtue of not being a SURB payload, the ciphertext will contain a BlockCiphertext, that is first decrypted as per the Noise protocol using the private component of their long term X25519 keypair, into a Block.

It is then each client's responsibility to:

- Queue, and reassemble multi-block messages as necessary based on the BlockCiphertext s field (sender's long term public key), and the `message_id`, `total_blocks`, and `block_id` fields in the Block structure.

  When reassembling messages, the values of `s`, `message_id`, and `total_blocks` are fixed for any given distinct message. All differences in those fields across Blocks MUST be interpreted as the Blocks belonging to different messages.

  It is important to keep in mind that both the message and ACK delivery mechanisms are fundamentally unreliable, and that it is possible to receive blocks containing identical payload in the event of a spurious transmission. Clients MUST validate that such Blocks (overlapping `block_id`) are in fact spurious retransmissions by doing a bitwise compare of the block payloads, and take appropriate action such as warning the user if an anomaly is detected.

- Present the IMF format message to the user.

Clients MUST discard messages that fail to authenticate or decrypt, and MUST warn the user at a minimum, if the long term public key used by the sender to encrypt messages is different from a previously known value.

Clients MAY impose a reasonable deadline for the reassembly process, after which partially received messages are discarded.

8.4.2.2 Client Protocol Acknowledgment Processing (SURB-ACKs).

When a client receives a message from their provider carrying a SURB payload, the message is a SURB-ACK for a Block that the client previously sent, signaling that the recipient's provider has received and queued the Block successfully.

The SURB ID is used to identify which Block the SURB-ACK corresponds to, along with the SURB payload decryption key (generated at the time of SURB creation).

Clients MUST discard SURB-ACKs corresponding to unknown Blocks, and MUST discard SURB-ACKs with invalid (non-zero filled) payload, with no additional processing.

Ordinarily, reliable protocols MUST use exponential backoff for retransmissions [CONGAVOID] [SMODELS] [RFC896], however if and only if the round trip time is greater than X seconds then exponential backoff is not needed.

8.5 Sphinx Packet Composition Considerations

Here we describe important facets of how clients construct Sphinx packets. This section assumes the client interacts with the mix network PKI as well as a universal time facility (rough time).

8.5.1 Choosing Delays: for single Block messages and for multi Block messages

The Client generates a delay for the ingress provider and for each of the mixes in the route, though not for the egress provider. The delays are drawn from the exponential distribution independently for each node. For a class of traffic `TRAFFIC_X`, the parameter `LAMBDA_X`, which is the inverse of the mean of the exponential distribution in milliseconds, is public by the mix network PKI and the same for all clients. Given `LAMBDA_X`, the sender just draws a random value from `Exp(LAMBDA_X)`.

For multi-Block messages, the client trickles the Blocks rather than sending them all in a burst. This mitigates e2e correlation attacks that look at bursts of multiple sent/received packets, and use that information to link the sender and receiver of a multi-Block message.

8.5.2 Path selection algorithm

The path selection algorithm is composed of four steps:

1. Sample all forward and SURB delays.
2. Ensure total delays doesn't exceed `(time_till next_epoch) + 2 * epoch_duration`, as keys are only published 3 epochs in advance.
3. Pick forward and SURB mixes (Section 8.5.2.1).
4. Ensure that the forward and SURB mixes have a published key that will allow them to decrypt the packet at the time of it's expected arrival.

If either step 2 or 4 fails due to lack of keying, or excessive delay, the entire path selection process MUST be restarted from the beginning.

8.5.2.1 Other Path Selection Considerations

The route contains the ingress and egress providers and a sequence of randomly selected mixes. The sequence of mixes is chosen independently for each Block.

Katzenpost uses the Layered topology, thus the selected path MUST contain one and only one mix per layer, and MUST traverse all layers. Within a layer, the mix is selected with probability proportional to its bandwidth/capacity. Thus, if a mix has a fraction `f` of the total capacity of its layer, it will be selected with probability `f`.

8.6 E-mail Client Integration Considerations

The e-mail client is a distinct component from the mix network client because we want to avoid having to heavily modify an e-mail client just to get it to work with our mix network. Instead we outline an e-mail integration strategy below. The main functionalities of a mix network client are:

1. send a message,
2. download the encrypted messages stored by the egress provider,
3. decrypt the messages using the private key (or universal private key if the client do not have a key, or if the sender didn't know the client's key),
4. reassemble multi-Block messages.

8.6.1 Message Retrieval

A local POP service can act as the mix network client, and decrypt the final layer of Sphinx packet encryption. The K9-Mail and other e-mail clients will download plaintext e-mail from this service. In this way we avoid having to make large code changes to existing e-mail clients.

8.6.2 Message Sending

A local SMTP proxy will perform the Sphinx encryption; the user's e-mail client will send messages to this local proxy. This avoids having to perform the Sphinx encryption natively in the e-mail client.

8.7 Client Integration Considerations

This section specifies additional design considerations other than the core reliability protocol design.

8.7.1 Message Retrieval

The mix network client component can utilize any of the above mentioned reliability protocol and therefore can receive:

- a single Block message
- a multi-Block message

8.7.2 Information available to clients

Clients download Mix Descriptors from the PKI, also known as the Mix Directory Authority service. More details about the PKI system and the Mix Descriptors can be found in the Katzenpost Mix Network PKI Specification.

Clients will have the following information available to them:

- Katzenpost Mix Network Parameters via the PKI:
  - topology information,
  - packet sizes for different classes of traffic,
  - parameter of the exponential delay (lambda) for Poisson mix strategy [KESDOGAN98], [LOOPIX17]
  - the list of public keys and addresses of the providers,
  - the list of public keys and addresses of the active mixes,
- Mix Network Consensus Document containing Mix Descriptors as described in the Katzenpost Mix Network PKI Specification
- Current mix network time via Rough Time protocol with mixes

8.8 Anonymity Considerations

- The reliability protocol will allow for active confirmation attacks, especially if using ARQ schemes which are more efficient than Stop and Wait ARQ. [CYA2013]
- Between two communicating parties at least one Provider must be honest to maintain send/receiver anonymity with respect to third party observers.
- Usage of SURBs for message ACKs present deanonymization vulnerability via compulsion attacks. We can later build defences against these attacks. [COMPULS05]

- There is no specified defence against n-1 attacks [TRICKLE02] at this time. In future versions we may utilize decoy traffic or heartbeat traffic to detect such attacks. [HEARTBEAT03]

8.9 Security Considerations

- Client endpoint public keys must be distributed in order to maintain confidentiality and integrity.

9. Mix Network Wire Protocol Specification

9.1 Introduction

The Katzenpost Mix Network Wire Protocol (KMNWP) is the custom wire protocol for all network communications to, from, and within the Katzenpost Mix Network. This protocol provides mutual authentication, and an additional layer of cryptographic security and forward secrecy.

The "C" style Presentation Language as described in [RFC5246] Section 4 is used to represent data structures, except for cryptographic attributes, which are specified as opaque byte vectors.

`x | y` denotes the concatenation of x and y.

9.1.1 NewHope-Simple Key Encapsulation Mechanism

This protocol uses the NewHope-Simple Key Encapsulation Mechanism, as specified in the original NewHope [NEWHOPE] and NewHope-Simple [NHSIMPLE] papers. All references to the NewHope-Simple shared secret in this document are to be interpreted as the final `mu` output for each party.

Note that while the NewHope and NewHope-Simple papers describe Alice as the "server" and Bob as the "client", for the purposes of this protocol, Alice is to be interpreted as the initiator, and Bob as the responder.

9.2 Core Protocol

The protocol is based on NewHope-Simple and Trevor Perrin's Noise Protocol Framework [NOISE] with the Hybrid Forward Secrecy extension [NOISEHFS] and can be viewed as a prologue, Noise handshake, followed by a stream of Noise Transport messages in a minimal framing layer, over a TCP/IP connection.

`Noise_XXhfs_25519+NewHopeSimple_ChaChaPoly_Blake2b` is used as the Noise protocol name, and parameterization for the purposes of this specification. As a non-standard modification to the Noise protocol, the 65535 byte message length limit is increased to 1048576 bytes.

As NewHope-Simple is not formally defined for the purpose of this version of the `hfs` handshake variant, let the following be the parameters:

`FLEN1 = 1824`, the size of the `m_a` output of the NewHope-Simple encodeA operation.

FLEN2 = 2176, the size of the m_b output of the NewHope-Simple encodeB operation.

FFLEN = 32

It is assumed that all parties using the KMNWP protocol have a fixed long lived X25519 keypair [RFC7748], the public component of which is known to the other party in advance. How such keys are distributed is beyond the scope of this document.

9.2.1 Handshake Phase

All sessions start in the Handshake Phase, in which an anonymous authenticated handshake is conducted.

The handshake is a unmodified Noise handshake, with a fixed prologue prefacing the initiator's first Noise handshake message. This prologue is also used as the `prologue` input to the Noise HandshakeState `Initialize()` operation for both the initiator and responder.

The prologue is defined to be the following structure:

```
struct {
    uint8_t protocol_version; /* 0x00 */
} Prologue;
```

As all Noise handshake messages are fixed sizes, no additional framing is required for the handshake.

Implementations MUST preserve the Noise handshake hash (h) for the purpose of implementing authentication.

Implementations MUST reject handshake attempts by terminating the session immediately upon any Noise protocol handshake failure and when, as a responder, they receive a Prologue containing an unknown protocol_version value.

Implementations SHOULD impose reasonable timeouts for the handshake process, and SHOULD terminate sessions that are taking too long to handshake.

9.2.1.1 Handshake Authentication

Mutual authentication is done via exchanging fixed sized payloads as part of the `Noise_XX` handshake consisting of the following structure:

```
struct {
    uint8_t ad_len;
    opaque additional_data[ad_len];
    opaque padding[255 - ad_len];
    uint32_t unix_time;
} AuthenticateMessage;
```

Where:

- `ad_len` - The length of the optional additional data.
- **`additional_data` - Optional additional data, such as a username,** if any.
- **`unix_time` - 0 for the initiator, the approximate number of** seconds since 1970-01-01 00:00:00 UTC for the responder.

The initiator MUST send the `AuthenticateMessage` after it has received the peer's response

(so after `-> s, se` in Noise parlance).

The contents of the optional `additional_data` field is deliberately left up to the implementation, however it is RECOMMENDED that implementations pad the field to be a consistent length regardless of contents to avoid leaking information about the authenticating identity.

To authenticate the remote peer given an AuthenticateMessage, the receiving peer must validate the `s` component of the Noise handshake (the remote peer's long term public key) with the known value, along with any of the information in the a“dditional_data“ field such as the user name, if any.

Iff the validation procedure succeeds, the peer is considered authenticated. If the validation procedure fails for any reason, the session MUST be terminated immediately.

Responders MAY add a slight amount (+- 10 seconds) of random noise to the unix_time value to avoid leaking precise load information via packet queueing delay.

9.2.2 Data Transfer Phase

Upon successfully concluding the handshake the session enters the Data Transfer Phase, where the initiator and responder can exchange KMNWP messages.

A KMNWP message is defined to be the following structure:

```
enum {
    no_op(0),
    disconnect(1),
    send_packet(2),

    (255),
} Command;

struct {
    Command command;
    uint8_t reserved;    /* MUST be '0x00' */
    uint32_t msg_length; /* 0 <= msg_length <= 1048554) */
    opaque message[msg_length];
    opaque padding[];    /* length is implicit */
} Message;
```

Notes:

- The padding field, if any MUST be padded with `'0x00'` bytes.

All outgoing Message(s) are encrypted and authenticated into a pair of Noise Transport messages, each containing one of the following structures:

```
struct {
    uint32_t message_length;
} CiphertextHeader;

struct {
    uint32_t message[ciphertext_length-16];
} Ciphertext;
```

Notes:

- The `ciphertext_length` field includes the Noise protocol overhead of 16 bytes, for the Noise Transport message containing the Ciphertext.

All outgoing Message(s) are preceded by a Noise Transport Message containing a `CiphertextHeader`, indicating the size of the Noise Transport Message transporting the Message Ciphertext. After generating both Noise Transport Messages, the sender MUST call the Noise CipherState `Rekey()` operation.

To receive incoming Ciphertext messages, first the Noise Transport Message containing the CiphertextHeader is consumed off the network, authenticated and decrypted, giving the receiver the length of the Noise Transport Message containing the actual message itself. The second Noise Transport Message is consumed off the network, authenticated and decrypted, with the resulting message being returned to the caller for processing. After receiving both Noise Transport Messages, the receiver MUST call the Noise CipherState `Rekey()` operation.

Implementations MUST immediately terminate the session any of the `DecryptWithAd()` operations fails.

Implementations MUST immediately terminate the session if an unknown command is received in a Message, or if the Message is otherwise malformed in any way.

Implementations MAY impose a reasonable idle timeout, and terminate the session if it expires.

9.3 Predefined Commands

9.3.1 The no_op Command

The `no_op` command is a command that explicitly is a No Operation, to be used to implement functionality such as keep-alives and or application layer padding.

Implementations MUST NOT send any message payload accompanying this command, and all received command data MUST be discarded without interpretation.

9.3.2 The disconnect Command

The `disconnect` command is a command that is used to signal explicit session termination. Upon receiving a disconnect command, implementations MUST interpret the command as a signal from the peer that no additional commands will be sent, and destroy the cryptographic material in the receive CipherState.

While most implementations will likely wish to terminate the session upon receiving this command, any additional behavior is explicitly left up to the implementation and application.

Implementations MUST NOT send any message payload accompanying this command, and MUST not send any further traffic after sending a disconnect command.

9.3.3 The send_packet Command

The `send_packet` command is the command that is used by the initiator to transmit a Sphinx Packet over the network. The command's message is the Sphinx Packet destined for the responder.

Initiators MUST terminate the session immediately upon reception of a `send_packet` command.

9.4 Anonymity Considerations

Adversaries being able to determine that two parties are communicating via KMNWP is beyond the threat model of this protocol. At a minimum, it is trivial to determine that a KMNWP handshake is being performed, due to the length of each handshake message, and the fixed positions of the various public keys.

9.5 Security Considerations

It is imperative that implementations use ephemeral keys for every handshake as the security properties of the NewHope-Simple KEM are totally lost if keys are ever reused.

NewHope-Simple was chosen as the KEM algorithm due to it's conservative parameterization, simplicty of implementation, and high performance in software. It is hoped that the addition of a quantum resistant algorithm will provide forward secrecy even in the event that large scale quantum computers are applied to historical intercepts.

10. Bibliography

RFC2119 Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

RFC5246 Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

RFC6234 Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

SP80038A Dworkin, M., "Recommendation for Block Cipher Modes of Operation", SP800-38A, 10.6028/NIST.SP.800, December 2001, <https://http://dx.doi.org/10.6028/NIST.SP.800-38A>

AEZv5 Hoang, V., Krovetz, T., Rogaway, P., "AEZ v5: Authenticated Encryption by Enciphering", March 2017, <http://web.cs.ucdavis.edu/ rogaway/aez/aez.pdf>

RFC7748 Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, January 2016.

KATZMIXWIRE Angel, Y., "Katzenpost Mix Network Wire Protocol Specification", June 2017. <https://github.com/katzenpost/docs/blob/master/specs/wire-protocol.txt>.

KATZMIXE2E Angel, Y., Danezis, G., Diaz, C., Piotrowska, A., Stainton, D., "Katzenpost Mix Network End-to-end Protocol Specification", July 2017, <https://github.com/katzenpost/docs/blob/master/specs/end_to_end.txt>.

KATZMIXPKI Angel, Y., Piotrowska, A., Stainton, D., "Katzenpost Mix Network Public Key Infrastructure Specification", December 2017, <https://github.com/katzenpost/docs/blob/master/specs/pki.txt>.

SPHINXSPEC Angel, Y., Danezis, G., Diaz, C., Piotrowska, A., Stainton, D., "Sphinx Mix Network Cryptographic Packet Format Specification" July 2017, <https://github.com/katzenpost/docs/blob/master/specs/sphinx.txt>.

LOOPIX Piotrowska, A., Hayes, J., Elahi, T., Meiser, S., Danezis, G., “The Loopix Anonymity System”, USENIX, August, 2017 <https://arxiv.org/pdf/1703.00536.pdf>

KESDOGAN98 Kesdogan, D., Egner, J., and Büschkes, R., "Stop-and-Go-MIXes Providing Probabilistic Anonymity in an Open System." Information Hiding, 1998, <https://www.freehaven.net/anonbib/cache/stop-and-go.pdf>.

MIXTOPO10 Diaz, C., Murdoch, S., Troncoso, C., "Impact of Network Topology on Anonymity and

Overhead in Low-Latency Anonymity Networks", PETS, July 2010,
<https://www.esat.kuleuven.be/cosic/publications/article-1230.pdf>.