String phone Documentation

Release 0.1

Stavros Korokithakis

November 15, 2015

Contents

Nothing is as secure as a string phone. – The NSA

Danger: This is alpha-quality software, and it's alpha-quality *security* software, at that, which is at least ten times more dangerous. Don't use it for anything where people are going to die if I got something wrong, or even where people are going to be mildly inconvenienced. Just use it for your quantified self dashboards until it gets super famous and is reviewed a bit more.

CHAPTER 1

Introduction

String phone is a secure communications protocol and library geared towards embedded devices. Its goal is to allow, for example, your mobile phone to communicate with your home automation devices in a secure manner, even over an insecure channel. It also allows for authentication of devices, so you can be sure that the only device whose commands will be accepted is the phone.

String phone isn't a communication layer itself. Rather, it sits over your communication layer, encrypting and signing messages as required before they are sent over your channel. It also verifies and decrypts incoming messages, ensuring that devices are who they claim to be, and that no third party can read your communications.

Since complicated things tend to be less secure, string phone aims to have a very simple interface:

```
>>> key = stringphone.generate_topic_key()
>>> alice = stringphone.Topic(topic_key=key)
>>> bob = stringphone.Topic(topic_key=key)
>>> message = alice.encode(b"Hi bob!")  # "s\xa7\xdf\xc3x\x19\x96\xd4..."
>>> bob.decode(message, naive=True)  # `naive` skips signature verification
b"Hi bob!"
```

CHAPTER 2

Motivation

With embedded devices, the internet of things, and ubiquitous computing becoming so common that I bet you didn't notice those three words refer to the exact same thing, there's an increasing need for security when these devices communicate with each other. The most common approach so far has been to just not use security at all, because it's more convenient. String phone aims to make security so convenient that you won't have an excuse to not use it. All you need to do to secure your devices' communications in the simple case is to generate a key and store it on each one, and call two methods to encode/decode messages before transmission. That's it.

This specific library is written in Python as a proof-of-concept and initial implementation of the underlying protocol, so it can be refined and improved. Python isn't very appropriate for running on embedded devices, as it's suitable for pretty much only the ones running Linux, like Raspberry Pi and the like. However, the intention is that the library will be ported to other languages like Java, Objective C/Swift and C, so it can be used on other platforms.

CHAPTER 3

Basics

(If you are too impatient for the theory, skip to *Getting started* to... get started)

String phone's communication primitive is a `Topic`. Think of a topic as a room where many devices are shouting at each other. This can be an MQTT queue, a pub/sub channel, an IRC channel, or even a single socket (one-to-one communication is a subset of many-to-one).

Each device in the topic is called a *participant*. Each participant has its own, persistent elliptic curve key, that is kept secret from other participants and anyone else. This key is top secret, and should not be shared with any person or device. It should never leave the participant's storage. This key is used to identify the participant and to sign the participant's messages so other participants are sure of who is sending them.

Each topic also has a persistent encryption key, called a *topic key*. The topic key ensures that all communcations between participants are securely encrypted. The topic key should only be known to the participants. Anyone with the topic key can read all messages exchanged in the topic without being detected.

Since you've made it this far, you can continue to the Introduction.

CHAPTER 4

Weaknesses

No security library can be complete without a list of its weaknesses, so you know what to avoid. Since the protocol is geared towards embedded devices, it strives to be simple, so it lacks many bells and whistles that may not be necessary or useful to everyone. You may have to implement these yourself, on top of string phone, or just be aware of them.

Here's what you need to watch out for:

- There is no replay protection at all. Anyone can record and replay a message at any time. If you want to guard against replays, make sure to include a sequence number or timestamp in your message, and discard commands that are too old or out of sequence.
- There is no forward secrecy. Once a participant has joined a topic, they can read all future *and* past messages that they may have. The only way to get rid of them is to create a new topic with a new key and move everyone over.
- Salsa20 has a 64-bit nonce, which may be too small when sending many small messages. This may be worth keeping in mind if you're worried about nonce reuse.
- Many more things that I'm sure will come up soon.

4.1 Introduction

4.1.1 Getting started

We'll start with a simple case. We will create two participants, one called *Alice* and one called *Bob*, both of whom know the topic key already (they have communicated beforehand over a secure channel and shared the topic key. Let's get them to exchange messages securely:

```
>>> from stringphone import Topic, generate_signing_key_seed, generate_topic_key

# Let's generate a topic key to share between the participants.
#  generate_topic_key() uses a cryptographically secure RNG, so
# you're safe using it to generate your keys.
>>> topic_key = generate_topic_key()

>>> topic_key
']j\x9b\xf7\xe77\x07h\xdcF\x82\x95\x0fo\x06\x90\xe1]R\xff\x8a\xeal\xd0\xef\x89J\xbd\x97\xf1[\xb4'

# Each participant generates a seed for their signing key and stores it.
# This key is their identity, so they must keep it completely secret from
# everyone, and safe.
>>> alice_seed = generate_signing_key_seed()
```

```
>>> bob_seed = generate_signing_key_seed()

# Give Alice and Bob their seeds, and the shared topic key.
>>> alice = Topic(alice_seed, topic_key)
>>> bob = Topic(bob_seed, topic_key)

# Alice encodes a message to send to Bob. encode() encrypts and signs it.
>>> alice_message = alice.encode("Hi Bob!")

# Bob will try to decode the message.
>>> bob.decode(alice_message)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "stringphone/topic.py", line 166, in decode
    "Verification key for participant not found."
UntrustedKeyError: Verification key for participant not found.

# String phone has raised an exception, as Bob has never seen Alice before
# and does not trust her. We can decrypt the message anyway by disabling
# signature verification, which is VERY VERY BAD.
# If you don't care about making sure that devices are who they claim to be,
# you can just use this and you're done.
>>> bob.decode(alice_message, naive=True)
'Hi Bob!'

# If we don't want to be hassled by unkown messages, we can ignore
# messages from untrusted participants:
>>> bob.decode(alice_message, ignore_untrusted=True)

# A much better way to communicate is to have Bob trust Alice's public key.
# This is done offline, after receiving the public key from Alice in some
# secure manner. It can also be done through the discovery process, which is
# detailed later on.
>>> bob.add_participant(alice.public_key)

# Strict mode will work now.
>>> bob.decode(alice_message)
'Hi Bob!'

# Let's see what Alice thinks by having Bob reply back:
>>> bob_message = bob.encode("Hey Alice!")

# Alice will try to decrypt.
>>> alice.decode(bob_message)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "stringphone/topic.py", line 166, in decode
    "Verification key for participant not found."
UntrustedKeyError: Verification key for participant not found.

# Alice doesn't trust Bob either. We can fix this the same way as before and
# restore order in the universe:
>>> alice.add_participant(bob.public_key)

# That's it for simple communication! We can look into discovering participants
# and how to trust them in the "Discovery" section.
```

4.1.2 Discovery

Discovery is a way for participants to join the topic without any prior knowledge, and for them to trust each other. In short, when a participant joins a topic, it can request the topic key from the other participants already in that topic. It can also request that new participants trust its public key so they can later verify its messages.

The flow is:

- Call *construct_intro* on the participant that needs the topic key (or at least wants the other participants to acknowledge it and trust its key). This returns the message, which can then be sent down the channel.
- Call *construct_reply* on one or more participants that already have the topic key. This creates the reply message that contains the encrypted topic key, which can then be sent.
- Call *parse_reply* to extract the `topic_key <stringphone.topic.Topic.topic_key` from the response and save it in the *Topic*. The *encode* and *decode* methods will then be able to encrypt and decrypt messages so other participants can read them and reply.

Here's a quick annotated demonstration of how to do this:

```
>>> from stringphone import Message, Topic, generate_topic_key

# Instantiate two participants, Alice with a topic key and Bob without one.
# Bob will use the discovery protocol to request the key from Alice.
>>> alice = Topic(topic_key=generate_topic_key())
>>> bob = Topic()

# Bob doesn't have a key, so he must ask for one. The way to do this is by
# constructing and sending an intro message to Alice.
>>> intro = bob.construct_intro()

# Alice will receive the message and try to decode it, but an exception will
# be raised, since the message is an introduction.
>>> alice.decode(intro)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "stringphone/topic.py", line 371, in decode
    raise IntroductionError("The received message is an introduction.")
IntroductionError: The received message is an introduction.

# Alice wraps the message in the Message convenience class and retrieves
# the sender's (Bob's) public key.
>>> message = Message(intro)
>>> message.sender_key
'\x0f\x83\xc7\xcb52\xe5,q\xba\xed\x94\xab\xd9\xb5\xfc=\x8d\x13\xa2\xeb\x19\x84\x0f9\xba\xeb\xa2\tR\x(

# Realistically, Alice will decide to reply to the intro because of a
# message dialog that will ask the user whether they want to trust the
# Bob, or because of a pairing period where Alice will trust all devices
# that introduce themselves in the next 10 seconds.
# Never unconditionally trust devices, or you will let anyone join the topic
# and security will be invalidated.
>>> alice.add_participant(message.sender_key)

# Construct and send the reply.
>>> reply = alice.construct_reply(message)

# Bob will try to decode, producing another exception, which is how he will
# will realize that this is a reply.
>>> bob.decode(reply)
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "stringphone/topic.py", line 375, in decode
    "The received message is an introduction reply.")
IntroductionReplyError: The received message is an introduction reply.

# Bob will parse the reply, which will populate the topic with the
# topic key and return True to indicate success.
>>> bob.parse_reply(reply)
True

# Bob decides to trust the participant that sent him the key (i.e. Alice).
# Never trust participants unconditionally.
>>> bob.add_participant(reply.sender_key)

# Now the participants can freely and securely talk to each other.
>>> message = bob.encode(b"Hey, Alice! Thanks for the key!")
>>> alice.decode(message)
'Hey, Alice! Thanks for the key!'
>>> message = alice.encode(b"Hey Bob! No problem!")
>>> bob.decode(message)
'Hey Bob! No problem!'
```

This was a short summary of how discovery works. You should now be able to use all of string telephone to exchange encrypted messages between participants and announce your clients to the world, as well as send the encryption key between them.

From here, you can continue to the Protocol documentation to learn more details about how string phone works at a lower level, or go to the API documentation to find more information about how the code is structured.

4.2 Protocol

This page details the messaging protocol and the format of the messages. It is an in-depth explanation of the internals of string phone. If all you want to do is use the library, you don't need to know any of this.

4.2.1 Cryptography

This is going to be short. String phone uses NaCl, as the "gold standard" in cryptography, and does whatever NaCl does. Keys are generated using NaCl's functions, signing is done using NaCl's signing methods, and symmetric and asymmetric encryption are as well. In short, it's NaCl all the way, with minimal novelty.

To delve into the lower layers a bit, NaCl uses Salsa20 for symmetric encryption and Poly1305 for authentication. Each message uses a new, randomly-generated nonce, which may not be enough when sending many short messages. A potential future improvement would be to use XSalsa20 for the longer nonce.

For specifics, please refer to the PyNaCl documentation.

4.2.2 Discovery

In the simple case, every participant already has the shared topic key and the keys and IDs of every other participant that they are interested in. Obviously, if you don't care about receiving or authenticating a participant's messages, you don't need their public key. More frequently, though, a participant will start out not knowing anyone, or the key. This is where discovery comes in.

Introduction

Discovery is done through introductions. When a participant joins a channel, it can send an introduction message (obtained through `construct_introduction()`) to request the topic key from other participants. This contains the new participant's signing key (so it can identify subsequent messages to others) and its encryption key, with which the topic key will be encrypted. The encryption key is signed with the signing key, to prevent attackers from sending arbitrary signing keys along with their own encryption key and enticing participants to give them the topic key.

Reply

A participant may choose to reply to an introduction. It can also just ignore the introduction, if it's not expecting any new participants. If the participant chooses to reply to the introduction, it can construct a reply with `construct_introduction_reply(introduction)`. The reply contains the encrypted topic key, the encryption key (for verification) and the signing key of the replying participant, as the new participant may want to trust the former.

The new node parses this and decrypts the topic key, which it then uses to post messages to and read messages from the topic.

4.2.3 Message format

Messages are delimited by size. All message formats start with a single-byte header that indicates the type of the message. The rest of the message varies according to its type. Details on all message types are elaborated on below.

Message

The simple message is the main way of communication. It contains:

- The ID of the sender of the message (for identification and as a way to select the right public key for verification).
- The ciphertext of the intended message.
- A signature of all of the above, signed with the participant's signing key.

Part	Type ("m")	Signature	Participant ID	Ciphertext
Size	1 byte	64 bytes	16 bytes	Variable

Introduction

The introduction contains:

- The sender's signing key (from which the sender's ID can be derived).
- An ephemeral encryption key to which replies with the topic key can be encrypted. The ephemeral encryption key is signed with the signing key.

Part	Type ("i")	Signing key	Signature	Encryption key
Size	1 byte	32 bytes	64 bytes	32 bytes

Reply

The introduction reply contains:

- The ID of the intended recipient (i.e. the participant that sent the original introduction that this reply is for).

- The encrypted topic key for the current topic, so the recipient can participate in the topic.
- The ephemeral public encryption key that the sender used to encrypt the topic key (for verification purposes).
- The sender's signing key (from which the sender's ID can be derived).

Part	Type ("r")	Recipient ID	Encrypted topic key	Encryption key	Signing key
Size	1 byte	16 bytes	72 bytes	32 bytes	32 bytes

4.3 stringphone package

This is the autogenerated API documentation. Use it as a reference to the public API of the project.

4.3.1 stringphone.crypto module

Symmetric and asymmetric cryptography- and signing-related classes and methods.

class `stringphone.crypto.`**`AsymmetricCrypto`**
Bases: `object`

`decrypt`(*ciphertext*, *public_key*)
Asymmetrically decrypt and verify the ciphertext.

Parameters

- **`plaintext`** (*bytes*) – The ciphertext to decrypt.
- **`public_key`** (*bytes*) – The sender's public encryption key.

Returns The plaintext.

Return type bytes

`encrypt`(*plaintext*, *public_key*)
Asymmetrically encrypt and sign the plaintext to the given public key.

Parameters

- **`plaintext`** (*bytes*) – The plaintext to encrypt.
- **`public_key`** (*bytes*) – The recipient's public encryption key.

Returns The ciphertext.

Return type bytes

`public_key`
The public encryption key of the AsymmetricCrypto object.

Return type bytes

class `stringphone.crypto.`**`Signer`**(*private_key*)
Bases: `object`

Instantiate a new Signer.

Parameters **`private_key`** (*bytes*) – The private signing key to use. Use *`generate_signing_key_seed`* to generate this.

`public_key`
The public key of this Signer object.

Return type bytes

sign(*plaintext*)
Sign the given plaintext.

Parameters **plaintext** (*bytes*) – The plaintext to sign.

Returns The signed plaintext.

Return type bytes

class stringphone.crypto.**SymmetricCrypto**(*key*)
Bases: object

Instantiate a new SymmetricCrypto object.

SymmetricCrypto performs symmetric encryption and decryption of byte arrays.

Parameters **key** (*bytes*) – The key to use for encryption and decryption. Use *generate_topic_key* to generate this.

decrypt(*ciphertext*)
Decrypt the ciphertext.

Parameters **ciphertext** (*bytes*) – The ciphertext to decrypt.

Returns The ciphertext.

Return type bytes

encrypt(*plaintext*)
Encrypt the plaintext.

Parameters **plaintext** (*bytes*) – The plaintext to encrypt.

Returns The ciphertext.

Return type bytes

class stringphone.crypto.**Verifier**(*public_key*)
Bases: object

Instantiate a new Verifier.

Parameters **public_key** (*bytes*) – The public signing key to use.

verify(*signed*)
Verify the signature of a signed bytestring.

Returns The plaintext that was signed with a valid signature.

Return type bytes

Raises BadSignatureError The signature was invalid.

stringphone.crypto.**generate_signing_key_seed**()
Generate and return a new signing key seed. The generated seed is cryptographically secure.

Returns A cryptographically secure random signing key seed.

Return type bytes

stringphone.crypto.**generate_topic_key**()
Generate and return a new topic key. The generated key is cryptographically secure.

Returns A cryptographically secure random topic key.

Return type bytes

4.3.2 stringphone.exceptions module

exception stringphone.exceptions.**BadSignatureError**
Bases: Exception

Raised when a signature did not verify.

exception stringphone.exceptions.**IntroductionError**
Bases: Exception

Raised when a message is an introduction.

exception stringphone.exceptions.**IntroductionReplyError**
Bases: Exception

Raised when a message is an introduction reply.

exception stringphone.exceptions.**MalformedMessageError**
Bases: Exception

Raised when attempting to decode a malformed message.

exception stringphone.exceptions.**MissingTopicKeyError**
Bases: Exception

Raised when trying to encode data without the topic key.

exception stringphone.exceptions.**UntrustedKeyError**
Bases: Exception

Raised when the verification key for a signed message could not be found.

4.3.3 stringphone.topic module

lasses and methods relating to the topic and its participants.

class stringphone.topic.**Message**(*message*)
Bases: bytes

ciphertext
The ciphertext.

Return type bytes

Raises ValueError if the given message type does not have this property.

encrypted_topic_key
The encrypted topic key.

Return type bytes

Raises ValueError if the given message type does not have this property.

encryption_key
The encryption key that encrypts the topic key.

Return type bytes

Raises ValueError if the given message type does not have this property.

recipient_id
The ID of the recipient.

Return type bytes

Raises ValueError if the given message type does not have this property.

sender_id
The ID of the sender.

Return type bytes

Raises ValueError if the given message type does not have this property.

sender_key
The public key of the sender.

Return type bytes

Raises ValueError if the given message type does not have this property.

signed_encryption_key
The signed encryption key.

Return type bytes

Raises ValueError if the given message type does not have this property.

signed_payload
The signed payload (signature + ciphertext).

Return type bytes

Raises ValueError if the given message type does not have this property.

type
The type of the message.

Return type int

class stringphone.topic.**Topic**(*signing_key_seed=None*, *topic_key=None*, *participants=None*)
Bases: object

A topic is the main avenue of communication. It can be any one-to-many channel, such as an MQTT topic, an IRC chat room, or even a TCP socket (one-to-one is a subset of one-to-many communication).

Various amounts of state can be passed to initialize according to each use case.

Parameters

- **signing_key_seed** (*bytes*) – The optional seed for our signing key. If you require identity persistence of this participant across restarts, save and provide this. A participant's public and private key and ID are generated from this seed, so passing the same seed will result in the same keys and ID. **Keep this completely secret from everyone**.

  If this is not provided, one will be generated. You will not be able to retrieve the generated seed, so only do this if you don't care about persistent identities.

- **topic_key** (*bytes*) – The optional symmetric encryption key this topic uses. If this is known when instantiating, we can start sending and receiving messages immediately, and discovery will only be useful to get other participants to trust us.

  If this is not provided, encryption and decryption will fail.

- **participants** (*dict*) – The optional dictionary of trusted participants. This should have the form {b"participant_id": b"participant_key"}. Participant keys in this dictionary will be trusted when verifying messages signed with them.

add_participant(*public_key*)
Add a participant to the list of trusted participants.

Parameters **public_key** (*bytes*) – The public key of the participant to add.

construct_intro()
Generate an introduction of ourselves to send to other devices.

Returns The message to broadcast.

Return type bytes

construct_reply(*message*)
Generate a reply to an introduction. This gives the party that was just introduced **FULL ACCESS** to the topic key and all decrypted messages.

Parameters **message** (*bytes*) – The raw introduction message from the channel.

Returns The reply message to broadcast.

Return type bytes

Raises BadSignatureError if the signature of the encryption key is invalid.

decode(*message*, *naive=False*, *ignore_untrusted=False*)
Decode a message.

If `naive` is True, signature verification will not be performed. Use at your own risk.

Parameters

- **message** (*bytes*) – The plaintext to encode.
- **naive** (*bool*) – If `True`, signature verification **IS NOT PERFORMED**. Use at your own risk.
- **ignore_untrusted** (*bool*) – If `True`, messages from unknown participants will be silently ignored. This does not include introductions or introduction replies, as those are special and will still raise an exception.

Returns The decrypted and (optionally) verified plaintext.

Return type bytes

encode(*message*)
Encode a message from transmission.

Parameters **message** (*bytes*) – The plaintext to encode.

Returns The encrypted ciphertext to broadcast.

Return type bytes

id
Our ID.

Return type bytes

parse_reply(*message*)
Decode the reply to an introduction. If the reply contains a valid encrypted topic key that is addressed to us, add it to the topic. If we already know the topic key, ignore this message.

Parameters **message** (*bytes*) – The raw reply message from the channel.

Returns Whether the retrieval of the topic key was successful.

Return type bool

participants()
Return all trusted participants.

Return type dict

public_key
Our public key.

Return type bytes

remove_participant (*participant_id*)
Remove a participant from the list of trusted participants.

Parameters **participant_id** (*bytes*) – The ID of the participant to remove.

topic_key
Return the topic encryption key.

Return type bytes

4.4 Code examples

These are some code examples, meant to illustrate the various uses of string phone.

4.4.1 Simple

The simple example shows how to exchange messages securely using a pre-shared key. Signature verification is disabled, since this example assumes that any participant that knows the pre-shared key can be trusted completely.

`Simple example`.

4.4.2 MQTT

The MQTT example uses an MQTT server for discovery and messaging.

`MQTT example`.

S

V