RelaySMS

SMSWithoutBorders

Technical paper

Afkanerd

August 2026

Version 2.0

Table of Contents

Terminology

AEAD_ENCRYPT: This implies an ChaCha-Poly1305 implementation

DR: Double Ratchet implementation as specified by the Signal protocol

AD: Associated data - This is the associated data used during encryption

Publisher: The server side software in charge of storing and publishing the messages on the users behalf. This is used interchangeably with **server**.

1. Introduction

1.1 Background

RelaySMS is a transport layer that bridges SMS with online platforms, allowing users to securely communicate across both. RelaySMS is part of the SMSWithoutBorders project which began in 2021. It works best in the context of a complete internet shutdown; SMS messaging is amongst the only available alternatives to communicating long distance.

RelaySMS relies on the users using their online accounts in publishing messages online. It also provides aliases for users who intend to publish using a phone number without using their online accounts. Being an open source project, the aim is for more security and technically oriented users to host their own versions.[1] We are taking steps to make the clients more suitable for custom hosting and not require developers fully rebuild the entire app to use with customly hosted servers.

This project is funded by the Open Technology Fund.

1.2 SMS Ecosystem

SMS developed in 1993 has come a long way as a messaging service provided by mobile phone operators. The service allows users to send and receive text messages of up to 160 characters; 140 bytes of data. When multiple messages are sent, the sending is switched to segmentation mode. Each message is sent independently and is reconstructed on the receiving end making a single message.[2] In cases where the message can grow fast, segmentation is not the right call as bytes are lost in the process; custom headers could grow beyond the optimized size of segmentation's headers.

[1] *https://github.com/smswithoutborders*

[2] The cost of sending still remains the same and also the payload size is reduced to account for the new header details.

2. Architecture

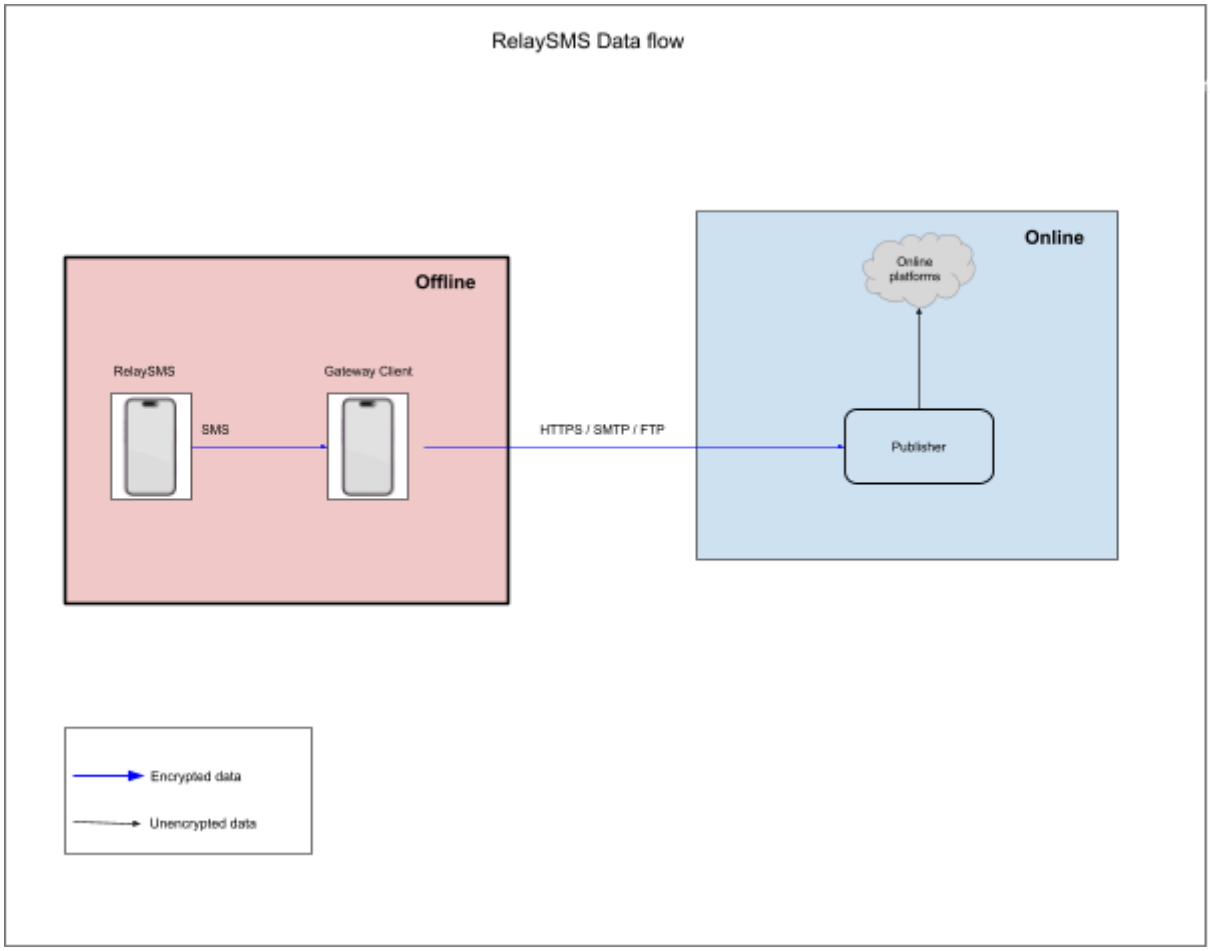

Figure 1.0

3. Infrastructures

3.1 Clients (Android & iOS app)

The clients provide the entire user interface for the service for the user. They provide users with the capability to compose and send out secured SMS messages and manage their accounts. The officially supported clients are the RelaySMS Android and iOS apps.

The value of the clients is the availability of the Keystore/Keychain for securely storing secrets and private keys which they provide. They also have inbuilt modems given the ability to send out SMS messages from the device. They are also portable and provide the user with the ability to use variant SIM cards.

3.2 Gateway client

The Gateway clients are Android devices tasked with receiving incoming SMS messages and forwarding them to the cloud. This can be done with either of the supported protocols such as HTTPS, SMTP and FTP; the reason for the protocols is to be resilient in case of a protocol blockage. The currently officially supported Gateway client app is the open source DekuSMS[3]. The Gateway clients are built to have queuing functionalities with connectivity listeners; this means the apps queue the incoming messages until the Gateway client is connected to the internet to resume forwarding. .

This acts as a bridge between the offline users and the online platforms by receiving and forwarding the incoming SMS messages to the server.

3.3 Publisher (Server-side)

The Publisher is a server-side application that receives encrypted payloads from the Gateway Clients, decrypts and authenticates them, and publishes the content to the requested platforms (Gmail, Bluesky, Telegram etc.) using the user's stored OAuth 2.0 or PNBA (Phone Number Based Authentication) tokens.

Messages forwarded from the Gateway Client to the Publisher include a header used in determining if a message is to be published with the user's OAuth 2.0 tokens or with a PNBA token. The Publisher decrypts the payload, then parses it and sends it to the service the user requested.

Failure is usually caused by:

[3] *Those hosting their own versions can use whichever SMS forwarding mechanism deemed preferable*

- Error in content formation by client
- Issues with SDK (usually exceptions at implementation) - traceable by logs
- Revoked/expired OAuth 2.0 token
- Rejection by the target platform during publishing (e.g., invalid addresses, platform rate limits, or policy restrictions).

The specific nature of the failure is not included in the SMS message sent to the user.

Request authentication

Official clients have the Publisher's long-term **Server Identity public key (SI_pk)** hardcoded. This pinned static key (s) is used to authenticate ensuring MITM protection.[4]

For every gRPC request, the client generates an X25519 key; eC. The client also generates and sends along a nonce and timestamp to help prevent a replay attack.

Requesters:

protocol = "Noise_NK_0RTT_25519_AESGCM"
salt = "RelaySMS v1"
ds = "RelaySMS Encryption Key v1"

Method name begins with a trailing /
LEN(MethodName) is fixed to 1 byte
Timestamp = 8 bytes

request_string = MethodName || Timestamp || <arbitrary payload>
dh = DH(eC, sS_kid_pk)
prk = HKDF-Extract(salt=salt, ikm=dh)
aes_key = HKDF-Expand(prk=prk, info = protocol || 0x00 || ds)
nonce = <random 128 bits value>
associated_data = eC_pk || eS_kid_pk
ciphertext = AEAD_ENCRYPT(input = request_string, key = aes_key, ad = associated_data, nonce=nonce)

Client headers always looks like this:

[4] Users self hosting Vault would need to generate and rebuild their clients replacing the official Vault keys with their generated copy.

X-Payload-bin: <ciphertext>

X-Public-Key-bin: <eC_pk>

X-Key-ID: <Key-ID>

X-Nonce: <Nonce>

X-Timestamp: <Timestamp>

Tokens

RelaySMS supports storing tokens for platforms that support OAuth2.0 and open platforms that provide a way to authenticate and act on the user's behalf (such as Telegram); we call the former oauth2 and the latter "phone number based authentication" (pnba).

RelaySMS generates a unique TokenID, TokenHash and an associated list of 256 X25519 key pairs for every token stored; each token is independent and is not linked to a user account. TokenID are 32 bit values used only when the user is publishing a message for that token. TokenHash are 256 bit random values which identify the token for online based requests, such as refreshing the tokens keys and deleting the tokens. They are transmitted encrypted at all times and would be decrypted on the client and server side when needed.

TokenID and TokenHashes are reused when the user deletes their token; key pairs are associated with Token pairs making its reuse safe.

Standards

Tokens are either from OAuth2.0 platforms or platforms with support for phone number based authentication. The protocols are used in determining what form of authentication the clients would present to the users and how the publisher would process and store the incoming token. Here are reference list of supported protocols and their specifications identification number;

Protocols

OAuth2.0	oauth2	0
Phone number based authentication	PNBA	1

Tokens come from various platforms which provide varying messaging categories; for example Gmail supports the emailing category. The categories are used in determining what inputs the client would

expect from the users and how the publisher would parse and reconstruct the incoming message; referenced in the specifications document. Here are reference list of supported categories and their specifications identification number;

Categories

Email	0	(must contain a *to* address and *body,* optional values are *subject*)
Messages	1	(must contain a *to* address - usually phone numbers in E.164 standards and *body*)
Text	2	(must contain a *body*)

Examples:

Gmail	Email
Messages	Telegram
Text	Bluesky

Storing tokens

The clients sends their authorization code (or OTP code) along with their X25519 ephemeral prekeys[5] (eC_kid) to the server. Keys should therefore be transmitted only with tokens guaranteeing the existence of both.[6] Successful uploads should return a **TokenID** and a **TokenHash**, which would be used for publishing.

The Client also receives the server's X25519 ephemeral keys (eS_kid); The protocol specs support sending up to 256 keys (8 bits).

TokenHash should always be encrypted using a new ephemeral key when being transmitted. It should also be stored encrypted to prevent extraction on any level.

OAuth2 considerations

RelaySMS relies on OAuth2.0 tokens to publish on the user's behalf. Users choose a supported platform to add their OAuth 2.0 token which is saved against their account on the server. The OAuth2.0 scopes required for publishing is the 'write' scope. This aligns with the security principle of **least privilege**, ensuring applications only have the minimum access necessary for their function.

[5] Should be the same amount as the server's ephemeral keys to support key index matching
[6] In case of OAuth2.0 the tokens would be when the Client is sending the tokens to the Server

PNBA considerations

Phone number based platforms do not exercise a shared protocol such as OAuth2.0. Telegram for example provides a SQLCipher database for storing user details when using the Telethon SDK. The security considerations when keeping files secured on a server is out of scope for this document.

Making token based requests

A key is selected from the pool at random and the index becomes the Key ID (Key-ID); this key is known as eC_kid_pk and eS_kid_pk for the Client's and Server's public key respectively. The server's keys would always be the same amount as the client's keys therefore they use the same Key-ID when selecting keys.

The Token can be encrypted for transmission as follows:

protocol = "3DH_25519_AESGCM"
salt = "RelaySMS v1"
ds = "RelaySMS Encryption Key v1"
// client:
    dh = DH(eC_kid, sS_kid_pk)
    dh1 = DH(eC_kid, eS_kid_pk)
// server:
    dh = DH(sS_kid, eC_kid_pk)
    dh1 = DH(eS_kid, eC_kid_pk)
prk = HKDF-Extract(salt=salt, ikm=dh || dh1)
aes_key = HKDF-Expand(prk=prk, info = protocol || 0x00 || ds)
*nonce = 0x00 * 12 // safe because aes_key is one-time*
associated_data = Key-id || eC_kid_pk || eS_kid_pk || sS_kid_pk
encrypted_token = AEAD_ENCRYPT(input = Token, key = aes_key, ad = associated_data, nonce=nonce)

The encrypted token is used as an arbitrary payload in the header request. The key Ids are sent through the body of the request.

Refreshing tokens

Users have 256 key pairs stored against each token. Each is used only once and destroyed immediately after use. This gives the user the ability to send approx. 256 messages before needing to refill their key pair store with more keys. The process of refreshing the keys deletes all previous keys and generates a fresh set of 256 keys. The duration of refreshing would be left to the client, but should be done before the user runs out of every key (one key should always be kept back).

Publishing

RelaySMS supports publishing messages in 2 modes:

- Online first publications
- Offline first publications

Online-first publications

These publications are for users who have stored tokens on the publisher. This is online first because an internet connection is required to store tokens on the server. Online-first publications provide the users with the server's ephemeral keys which provides greater forward secrecy guarantees. This is the most secure and recommended form of publishing with RelaySMS.

The encryption for the payload is constructed as follows (*values are derived like tokens above)*:

```
...
ciphertext = AEAD_ENCRYPT(
        input = plaintext,
        key = chacha_key,
        ad = associated_data,
        nonce = nonce
)
```

The most important security practice here is keys are one time used and must be deleted from both Client and Server immediately after use.

Offline-first publications

These publications are for users who do not have tokens stored on the Publisher (mostly due to the inability to do so). This is the less secure method of publishing messages with RelaySMS as forward secrecy is not guaranteed; compromising the Publisher's static private keys would break all previously sent messages. Implementations of platforms to support offline-first publications should also implement measures to guarantee user's privacy; for RelaySMS-Mail (Rmail), the default offline first publication for RelaySMS, we randomize the address and do not associate the incoming phone number in any way. This brings more issues such as the potential for collision of addresses, but the solutions are currently not scoped in this version of the paper.

The encryption for the payload is constructed as follows (*values are derived like tokens above)*:

info = "RelaySMS C2S DR v1"
h = SHA256("Noise_IK_25519_ChaChaPoly1305_SHA256")
ck = h
h = SHA256(h || sS_pk)
h = SHA256(h || eC_pk)

Sender derivation (→ e, es, s, ss)
dh_es = DH(eC, sS_pk)
ck, k = HKDF(salt = ck, input = dh_es, info = info)
nonce = <96 bits random>
sC_pk_enc = ENCRYPT(key = k, input = sC_pk, ad = h, nonce=nonce)[7] *// send*
h = SHA256(h || sC_pk_enc)
dh_ss = DH(cS, sS_pk)
ck, k = HKDF(salt = ck, input = dh_ss, info = Info)
nonce1 = <96 bits random>
ciphertext = ENCRYPT(key = k, input = ***payload****, ad = h, nonce=nonce)*
tx_h = SHA256(h || tx_payload)
tx_payload = eC_pk || nonce || sC_pk_enc || nonce1 || ciphertext

The client sends a ***tx_payload*** *to the server*

[7] A zeroed 128 bits can be used as nonce for the ENCRYPT methods when using AES-GCM

Recipient derivation

eC_pk = tx_payload[...32]

dh_es = DH(sS, eC_pk)

ck, k = HKDF(salt = ck, input = dh_es, info = info)

nonce = tx_payload[32...44]

sC_pk_enc = tx_payload[44…92]

sC_pk = DECRYPT(key = k, input = sC_pk_enc, ad = h, nonce=nonce)

h = SHA256(h || sC_pk_enc)

dh_ss = DH(sS, sC_pk)

ck, k = HKDF(salt = ck, input = dh_ss, info = Info)

nonce1 = tx_payload[92…104]

ciphertext = tx_payload[104...]

payload[8] *= DECRYPT(key = k, input =* ***ciphertext****, ad = h, nonce=nonce1)*

[8] *The server should proceed to publish the payload.*

4. Considerations

4.1 Key selection

Keys should be selected at random from what is available in the database. This can be achieved by letting the SQL queries select randomly, which means indexes should not rely on but rather the key ID as an identification property.

4.2 Token deletion policy

Tokens are deleted once older than 90 days on the system. This count down is reset to 0 once the token is used for publishing. The count down is also reset to 0 once the user refreshes their keys - showing activity on the client side.

4.3 Deletion of used keys

Ephemeral keys should always be deleted immediately after use; usage includes in cases of failures too, once observed. This is both on the client and server implementations.

4.4 Backup and restores

Tokens are stored on the publisher and their associated public keys on the server and the key pairs on the clients. The TokenHash is the identifiable information linked to those keys and therefore must be backed up to continue usage.
The users can export their Token's information which includes:

- TokenHash
- TokenID
- All associated keypairs

The encryption process for generating the backup file include:

- A 64 character passphrase is generated
- The passphrase is run through using Argon2id creating a cryptographic key
- The encryption Tokens and their associated key pairs are then encrypted using ChaCha20-Poly1305 and written to the file.

The user is handed this file for safe keeping and also their passphrase for decrypting the content of the file. How they secure both the file and passphrase is beyond the scope of this document.

4.5 Privacy

While phone numbers are required for sending SMS messages, they are not tied to the sending account; any phone number with the right token information can send and publish a message. This provides the users with the capability of switching phone numbers (or using burner numbers) to meet their threat models.

5. Limitations

- Rooted/Jailbroken devices are out of scope
- Key management systems are out of scope