SECTP.1
Specification

Secure Electronic Communication
Transfer Protocol, v. 1

Abstract

Purpose of this document is to be an authoritative specification for the Secure Electronic Communication Transfer Protocol, which is being developed as part of a Jugend Forscht project by the document author.

The Internet and its associated protocols haven't been developed with security in mind, originally. Adding secure communication has mostly been an afterthought and moving to the secure protocol variants isn't close to being completed, as can bee seen in the fact that the migration from HTTP to the more secure HTTPS is still ongoing.

On the other hand, the Internet has one property, which could make it more secure and which enhance privacy: it's decentralized. There is no central "Internet Government", and there are no central servers, on which everything must be hosted. Unfortunately, this plurality has been lost in the last few years due to the monopolization of Internet services in the hand of a few large companies. Previously, everyone hosted their own mail server and could hence be assured, that nobody else would be able to read the emails. However, nowadays a large part of the humans use few, centralized services, such as Google Mail, where they must assume, that the providers are able to read their messages.

Furthermore, using encryption for emails (e.g. GPG) is still not accessible to the general Internet user. The available tools are mostly complex, visibly outdated and not very user friendly.

This protocol - SECTP.1 - aims to solve these issues by being a the standard for a privacy by default and by design, distributed system, where everyone should be

able to send encrypted messages to every other user of the system. The protocol shall also be extendable, to allow for useful additions, such as identity verification, which might be useful for example to allow legal documents to be sent via the system.

Definitions

Within the context of this document,

the key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "**OPTIONAL**" are to be interpreted as described in RFC 2119;

the "**Service**" is anyone who provides a web server, which is compliant with this protocol specification and which may be used to communicate securely with other Services;

the "**User**" is any account registered at a Service, able to receive and send Messages;

the "**Message**" is any text, document, image or other file, which has been sent between two Users of the same or different Services;

the "**Service Certificate**" is a self-signed certificate, following the rules of this specification, which identifies a service without doubt;

the "**User Certificate**" is a certificate signed by the Service the user is registered at, which identifies a User without doubt;

a Service "**trusts**" another Service, if it deems the other Service trustworthy, assumes that it follows all the rules of this specification and that its identity verification process is at least as proper as its own.

General Interface

Every Service MUST provide a general interface, described by the following rules:

The Service MUST be accessible under a valid URL (the “Root URL”), where requests are responded by a HTTP-compliant web server. The Service web server MUST use the secure variant HTTPS for every request and it SHALL NOT respond to requests which are made only in unsafe HTTP, except for rejecting them.

The Service MAY host other pages under the Root URL, including, without limitation, an accessible User interface to it’s functions.

The Server MUST respond to a virtual directory, which is located at “/api” under the Root URL. This is the “Interface Root”. Unless specified otherwise in this document, all URL routes are given relative to that Interface Root.

Certification

Every Service SHALL follow the following rules for generating certificates for itself and for its users.

Service Certificate

Every server MUST generate two pairs of PGP keys, (pubkey_sign, privkey_sign) and (pubkey_recv, privkey_recv), which SHALL be cryptographically strong. Their length SHOULD be at least 4096 bytes. The two private keys privkey_sign and privkey_recv MUST be treated with the highest possible confidentiality. They may only be accessible to the Service.

Now, every Service has to generate the following certificate template file:

```
+++SECTP.1/Certfile+++
Type: Service
Name: (Service name)
Handle: (Root URL)
```

```
PubkeySign: |
(pubkey_sign as b64)
PubkeyRecv: |
(pubkey_recv as b64)
Flags: -
IssuedDate: (Issued Date)
Authorize: self
```

In this template, replace “(Service name)” with a self-chosen, likely to be unique, name of the Service provider; “(Root URL)” with the Root URL for the Service; “(pubkey_sign as b64)” with a Base64 representation of pubkey_sign, with a forced linebreak after every 42 characters; “(pubkey_recv as b64)” with a Base64 representation of pubkey_recv, with a forced linebreak after every 42 characters; “(Issued Date)” with a ISO 8601 timestamp for the date, the certificate was issued.

Next, the Service MUST generate a secure hash of the template file by using the SHA-256 algorithm. They must RSA-encrypt that hash with privkey_sign. The encrypted hash shall be written down in a Base64 representation and be put at the end of the template file like this: (“(hash)” is the encrypted hash, “(date)” the date ISO 8601 timestamp of the signature)

```
***signed: (hash) at (date)
```

This file SHALL be kept securely and SHALL NOT be lost. This new file is the Server Certificate.

Server Certificate Verification

Every Service MUST provide their Certificate under the “/certfile” route.

Every Service MUST be able to prove the correctness of their Certificate by providing a “/certfile/challenge” route, which SHALL be called together with a GET parameter named “task” containing a random twenty digits number. Upon request, the Service will generate a secure SHA-256 hash of that number, encrypt it with its own privkey_sign and provide that encrypted value as response.

User Certificate

For every user, the server SHALL generate two pairs of PGP keys, (pubkey_sign, privkey_sign) and (pubkey_recv, privkey_recv), which SHALL be cryptographically strong. Their length SHOULD be at least 2048 bytes.

The two private keys privkey_sign and privkey_recv MUST be treated with the highest possible confidentiality. The server MUST safely disclose them to the User and then it SHALL NOT possess them any longer. It MAY warn the User, that the keys cannot be restored if lost. It MAY store the keys for the User, if it uses a safe encryption format and if they can only be decrypted by the User.

Now, the Service MUST generate a template certificate file for the user. The file MUST look like this:

```
+++SECTP.1/Certfile+++
Type: User
Name: (User's name)
Handle: (a unique handle)@(Root URL)
PubkeySign: |
(pubkey_sign as b64)
PubkeyRecv: |
(pubkey_recv as b64)
Flags: (Flags or -)
IssuedDate: (Issued Date)
Authorize: (Root URL)
```

In this template, replace "(User name)" with the name of the User; "(a unique handle)" with a unique alphanumeric (including ".", "_" and "-") handle for each User of the Service, possibly self-chosen; "(Root URL)" with the Root URL for the Service; "(pubkey_sign as b64)" with a Base64 representation of pubkey_sign, with a forced linebreak after every 42 characters; "(pubkey_recv as b64)" with a Base64 representation of pubkey_recv, with a forced linebreak after every 42 characters; "(Issued Date)" with a ISO 8601 timestamp for the date, the certificate was issued.

Furthermore, replace “(Flags or -)” either with “-“ if you don’t want to add specific flags to the certificate, elsewise with a comma-and-space-separated list of the following applicable flags:

> “**verified**”, if the Service was able to verify the user’s identity and if the User name is surely correct;
>
> “**authority**”, if the Service was able to verify that the User is a government authority;
>
> “**readconfirm**”, if the Service and the User commit to informing the Message senders, when it has been read;
>
> “**temporary**”, if the Service has only granted a temporary certificate (the certificate will expire 180 days after being issued with this remark).

The Service MAY issue any other flags for internal purposes, but other Services SHOULD ignore those.

Next, the Service MUST generate a secure hash of the template file by using the SHA-256 algorithm. They must RSA-encrypt that hash with *their own* privkey_sign. The encrypted hash shall be written down in a Base64 representation and be put at the end of the template file like this: (“(hash)” is the encrypted hash, “(date)” the date ISO 8601 timestamp of the signature)

```
***signed: (hash) at (date)
```

This file SHALL be kept securely and SHALL NOT be lost. A copy of this file SHOULD be given to the User. This new file is the User Certificate.

User Certificate Verification

Every Service MUST provide every Users’ Certificate under the “/user/(handle)/certfile” route, where “(handle)” is the unique user handle also in the certificate.

Access and Verification

Every User MUST be assigned a locally unique handle by its Service. This handle MAY contain the following characters; it MUST NOT contain any other character:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789.-_
```

The handle MUST be unique within one Service, no two users may be able to register the same two handles at one Service.

For all purposes, the handle shall be used to identify the user.

To access a user on any other Service, an "@" and the other Service's Root URL will be appended. For example, if your handle is "luap42" and you are registered on "example.com", you may be accessed as

"**luap42**" on example.com;

and "**luap42@example.com**" everywhere else.

To prevent confusion with email addresses, you MAY also prepend the Service's Root URL followed by a backslash ("\") to identify a User:

"**example.com\luap42**"

Web of Trust

Every Service MAY declare, that it trust's any other Service. This is called "direct trust".

Within the context of this specification, a Service is considered to trust itself and any Service, that fulfills at least one of these conditions:

the other Service is directly trusted by the Service, or

a Service directly trusted by the Service trusts the other Service directly.

Furthermore, within the context of this specification, a Service is considered to be "trust candidate", or "possibly trustworthy", if it is trusted by you or if it is trusted by at least two Services you directly trust.

Sending and Receiving Messages

Sending and Receiving Messages according to the following rules is the core feature of this protocol.

Interface

Every Service MUST provide the following route for every User: "/user/(handle)/recv", where "(handle)" is the User's handle. Any Message sent to that route SHALL be verified for formal correctness and then it MUST be considered an inbox Message, which is displayed to the User.

Every Service MAY provide the following route for every or for a specific subset of Users: "/user/(handle)/confirm", where "(handle)" is the User's handle. Every user, for which this route is available SHOULD have the "readconfirm" flag in their certificate. This route shall be used for confirming, whether a Message has been read.

Preparing Messages

A Message SHALL only be sent, when it has been prepared in accordance with the following procedure.

First of all, we calculate a Base64 representation of the Message's body. Then we generate the following message template file:

```
Subject: (subject)
DataType: (data type)
Body: |
(message as b64)
```

In this template file, “(subject)” is the chosen message subject; “(data type)” is a textual representation of the data type of the message body (example: text, html or pdf); “(message as b64)” is the Base64 representation of the Message body.

Next, we generate a cryptographically strong session key. It SHOULD be at least 4096 characters long. We then encrypt the message template file with that key using the AES encryption. Furthermore, we take the receiver’s pubkey_recv and use it to encrypt the session key.

Now we use these two information to generate the following wrapper template:

```
+++SECTP.1/Message+++
Author: (author)
Key: (encrypted key)
Message: |
(encrypted message)
MessageDate: (date)
```

Here, “(author)” is the sender’s handle with an “@” and the sender’s Service’s URL root appended (see above); “(encrypted key)” is the encrypted session key; “(encrypted message)” is the message template AES-encrypted with the session key; “(date)” is the ISO 8601 formatted current timestamp.

Then we calculate the SHA-256 hash of the wrapper template file and encrypt it with the sender’s privkey_sign; the encryption is stored as a Base64 representation. We take that encrypted hash and append to the wrapper template file: (“(hash)” is the hash of the wrapper template file; “(date)” the current date as ISO 8601 timestamp)

```
***signed: (hash) at (date)
```

This is our finally packaged message.

Sending and Receiving

Every Service MUST be able to receive messages according to the following protocol. It SHOULD be able to send messages according to it.

To send a message to a specific recipient User, the Service from which the Message is to be sent, starts a POST request to the "/user/(handle)/recv" route. The recipient Service replies with a two-line message containing the status code and a reply body. These status codes can be

"**OK**", when the Message has been successfully sent; the message body SHALL contain an ID with which the message can be uniquely identified;

"**REJECTED**", when the Message is formally valid, but a specific reason or security rule caused it to be rejected; the reply body SHOULD mention the reason for the rejection;

"**ERROR**", when the Message is formally valid, but an error on the recipient's side caused the Message to be discarded; the reply body MAY contain useful information;

"**INVALID**", when the Message cannot be received, because it's formally invalid; the reply body MAY contain useful information.

The recipient Service MUST accept any incoming Message coming from a server it trusts. It SHOULD accept any incoming Message coming from a possibly trustworthy server, and it MAY accept any other incoming Message, but this is a question of it's security policy.

If the Service accepts the Message, it MUST validate it's signature. If the signature is valid, it SHALL store it safely locally for the User to read. Otherwise, it MUST reject the Message as invalid.

Read Confirm

If the recipient account has activated read confirm, per the flags on its certificate, their Service is expected to ("MUST") notify the sender of the message by submitting a reply to "/user/(handle)/confirm", when

- the Message has been successfully stored locally by using the code "**STORED**";

- the Message has been decrypted for the first time by using the code "**DECRYPTED**";
- and when the Message has been opened for the first time by using the code "**READ**".
- If the Service informs the User of received messages, for example by email, it SHOULD also inform the sender of the Message of the successful delivery of that notification by using the code "**NOTIFIED**".

The confirmation message is sent to the Service from which the Message originated, by using a two-lined POST request to the route mentioned above. In the first line, the code from above MUST be put, in the second line the Message ID, as given upon the receipt of the Message).

The Service from which the Message originated MUST verify, whether the confirmation is sent from the proper server. It SHALL store all confirmations, together with their time. And it SHOULD display these information, upon request, to the sender.

Extensions

Services MAY define extensions to this protocol. An extension SHALL NOT cause an otherwise legitimate message to be rejected, just because it is not in compliance with the extension.

An extension MAY add rules of the form

```
X-[Option]: [Value]
```

to the schema of a Certfile or a Message. Option must be an alphanumeric string, which may use "-" to separate words. Value must be a single-line value.