UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Simple Protocol for Encrypted Messaging (SPEM)

A Protocol for End-to-End Encrypted Instant Messaging

Bachelor's Thesis

Simon Studer

Supervisor

Prof. Dr. Ulrich Ultes-Nitsche

Department of Informatics, University of Fribourg

August 2016

Abstract

In this thesis a protocol for end-to-end encrypted instant messaging is proposed, Simple Protocol for Encrypted Messaging (SPEM). Not only is SPEM designed to protect the privacy of its users without compromising usability, it is also designed to prevent any messaging provider from achieving a monopoly. Anyone can implement and operate messaging servers, as well as messaging clients, for SPEM. The requirements SPEM has to satisfy are established by analyzing a wide range of existing messaging systems differing in type, popularity, and purpose.

Table of Contents

Chapter

1

Introduction

When it comes to instant messaging, everyone has their own preferences. The majority of people use applications like WhatsApp and Google Hangouts on their mobile phones. Other people use Threema or Signal to benefit from privacy enhancing features. Yet another group of people use Telegram which trades off some of its security for convenience. On personal computers, the variety includes some of the applications for mobile phones but also applications like ICQ and IRC. The list of messaging applications is incredibly long and so is the list of back end systems that are used behind the scenes. This means that usually, there cannot be any communication between users of different applications. Everyone must use a multitude of messaging applications to be able to communicate with all their friends, family, and coworkers.

The fact that there is no de facto standard for instant messaging is rather surprising, considering that for longer messages, there is only one notable standard: e-mail. This has the advantage that everyone can use the application he likes most to send and receive e-mail but is still able to exchange messages with everyone else. The same is true for Short Message Service (SMS). Every mobile phone can exchange SMS messages with all other mobile phones, independently of the vendor or the application that is used to compose and read the messages. Even though they are considered standard, those systems have disadvantages that make them a poor choice for instant messaging.

It is true that the various messaging applications each have their unique set of features, and everyone chooses their preferred application according to which of these features are most important to them. But it would be much more convenient for users if there was only a single protocol behind all instant messaging applications, so they could be used interchangeably.

In Chapter 2 the most popular messaging systems are analyzed to highlight their strong and weak points. This includes systems that are not commonly used for instant messaging but have traits that could be useful in an instant messaging system. Afterwards, in Chapter 3, a set of minimum requirements is established that an instant messaging solution must fulfill to make it a viable competitor in today's messaging landscape. Additionally, a set of optional

features is established, which are not absolutely necessary but might make the protocol more attractive compared to other messaging systems, and features that might be missed by users.

Once the minimum requirements are defined, a protocol is proposed which meets those minimum requirements and includes many of the optional features as well. This protocol leverages existing technologies whenever possible and is kept as simple as possible.

Chapter

2

Analysis of existing messaging systems

Before starting to design an instant messaging protocol, it is vital to look at the various messaging systems that exist already. This chapter lists a handful of the most widely used messaging systems. All of those systems are examined in order to highlight their strong and weak points. If present, other interesting characteristics are discussed as well.

In November 2014, the Electronic Frontier Foundation (EFF) created the *Secure Messaging Scorecard* to compare instant messaging systems.[1] Since then, the EFF have made occasional changes to keep the scorecard up to date. It consists of a comprehensive list of almost 40 instant messaging systems, which are compared according to a set of seven requirements. These requirements are formulated as the following yes-no questions:

1. Is your communication encrypted in transit?
2. Is your communication encrypted with a key the provider does not have access to?
3. Can you independently verify your correspondent's identity?
4. Are past communications secure if your keys are stolen?
5. Is the code open to independent review?
6. Is the crypto design well-documented?
7. Has there been an independent security audit?

The more of these questions can be answered with *yes* for an instant messaging system, the better. Note that the last requirement, whether the security design has actually been audited by an independent security firm, depends on the popularity of the instant messaging system and not only on the security design itself. If a system is popular it is more likely to get examined by independent parties.

The *Secure Messaging Scorecard* gives a good and systematic overview over the different instant messaging systems. However, the requirements are formulated as very specific yes-no questions with a primary focus on privacy. Even with privacy as a main focus, other considerations have to be taken into account if the new instant messaging protocol is to be

[1] The scorecard can be found at `https://www.eff.org/secure-messaging-scorecard`.

adopted on a large scale. Questions about deployment and usability have to be addressed as well. These questions are not easily answered with *yes* or *no*, making a direct comparison between systems rather difficult. Therefore, in this thesis, the messaging systems are examined individually, with only few direct comparisons. Nevertheless, the requirements the scorecard uses serve as a good basis for privacy oriented evaluation.

Note that many of the messaging systems discussed in this chapter are not used for instant messaging but for other forms of messaging. Some of them are not even complete messaging systems but rather protocols that can be used as extensions to other messaging systems.

WhatsApp Facebook's WhatsApp[2] is arguably the most popular instant messaging system with a user base of over 900 million monthly active users (MAUs) [25].

As of April 2016, WhatsApp uses End-to-End Encryption (E2E encryption) for all communications, including media files and voice calls. Since this change, WhatsApp scores six out of the seven points on the Secure Messaging Scorecard, only missing a point for being closed source [4].

Although it's generally a good thing that WhatsApp uses E2E encryption, some questions arise as to why Facebook would allow this. Facebook's business model relies on showing its users a newsfeed filled with content the users actually want to see, laced with advertisements that Facebook wants the users to see. This sort of customized newsfeed requires Facebook to know a lot about its users and how they are connected to other users [16]. Some people argue that Facebook allows WhatsApp to use E2E encryption because they can still gather enough information about its users by looking at the metadata [5], [11].

Another important point to keep in mind is that the messages themselves are not the only way WhatsApp can collect information about its users. Discovering friends, for example, requires personal contacts to be uploaded to WhatsApp servers. Also, making use of WhatsApp's built-in backup feature might upload the message database to third party cloud servers. Even though the backups are encrypted, they become more accessible to attackers [14].

In order to use WhatsApp, users have to create an account linked to their phone number. Account creation is mostly hidden from the user except for a confirmation step, where the provided phone number is verified by an SMS message. A user cannot have multiple WhatsApp accounts without having multiple phone numbers, but it is possible to have multiple phone numbers linked to a single account. Further, WhatsApp cannot be used on multiple devices with the same account at the same time. Even though WhatsApp has a web client, it cannot be used on its own; it has to be activated by scanning a Quick Response Code (QR code) with the phone. When using the web client, messages continue to be sent and received over the phone in the background.

Facebook Messenger Facebook Messenger[3] is another big contender in the instant messaging field with around 800 million MAUs [25]. It it is also closed source and does not have E2E encryption. It's advantage is, that it is not linked to a phone number but rather to a Facebook account, so there is no need to activate the web client by scanning a QR code.

[2]The official website can be found at `https://www.whatsapp.com/`.
[3]The official website can be found at `https://www.messenger.com/`.

Hangouts Google Hangouts[4], formerly known as Google Talk, is an instant messaging application that comes pre-installed on all Android phones. Since most Android users connect their Google account, they can be reached over Hangouts. However, some of these user might not use it on a regular basis. Reliable numbers about the active userbase of Google Hangouts are not readily available.

Similarly to Facebook Messenger, Hangouts is tied to a Google account rather than a phone number. But it also does not offer E2E encryption (messages are only encrypted in transit) and the source code is proprietary [4]. Furthermore, Hangouts has some usability issues. There is no easy way to forward messages and there is no search function.

Hangouts is available for almost all operating systems in one way or another. On mobile phones and tablets, native applications can be installed. A Hangouts client for desktop computers is available as a Google Chrome extension (Google Chrome is installable on all major platforms). Alternatively, there exists a web client, where Hangouts can be used within the browser, without installing additional software.

Note that the "Off-the-record" function of Hangouts is not related to the Off-the-Record (OTR) protocol discussed below and it does not secure the messages in any additional way except that the messages are removed from the device after the conversation has been closed.

iMessage Apple's iMessage[5] used to be very popular among iPhone users because it allowed iPhone users to send messages to other iPhone users over mobile data instead of SMS. When a message is sent to someone that does not use an iPhone, iMessage simply falls back to SMS. Apple's iMessage still ships with iPhones and other Apple products, but it can only be used on Apple products. This is not likely to change, since Apple's main products are the devices and not the software.

E-mail E-mail[6] is one of the oldest and most widely used form of electronic messaging. Although it is not an instant messaging system, it so popular and versatile that it is worth taking a closer look at.

The most noteworthy of its characteristics is its popularity. Whenever we see a successful product in the technology world, it is instantly copied by other organizations that offer incremental improvements. This leads to a fragmentation of the user base. In the instant messaging field this fragmentation is very prominent and is a motivation for this thesis. However, e-mail has not suffered such a fragmentation of its user base. There is a large variety in e-mail server software, in e-mail client software, and in web client software, but the protocol, or rather protocols, they leverage are mostly the same. It is an advantage that users can freely choose a provider to host their e-mail service and freely choose client software to send and receive their e-mail messages. All the wile being able to send messages to people using different providers and different software. Since most computer or smartphone users have one or more e-mail accounts, it is easy to send e-mail messages to family, friends, and coworkers without switching between applications. If users do not feel comfortable having their e-mail hosted by a third party, they can host their own servers.

[4]The official website can be found at `https://hangouts.google.com/`.
[5]The official website can be found at `https://www.apple.com/ios/messages/`.
[6]The protocol specification can be found at `https://tools.ietf.org/html/rfc5321`.

Furthermore, E-mail accounts are generally not linked to a phone number or an account on a social network. E-mail accounts can usually be created anonymously and for free. A user can then simultaneously log into the account from any device through either an e-mail client or a web interface.

E-mail is far from perfect, however. It lacks built-in encryption: Messages may be read by anyone who handles them on the way to their destinations (they are sometimes encrypted in transit between the client and the server). Messages may easily be spoofed to come from another sender and they contain a lot of metadata [9].

There are some tools that can be used to encrypt the body of e-mail messages. Two of them, Secure/Multipurpose Internet Mail Extensions (S/MIME) and Pretty Good Privacy (PGP) are discussed below. However, these tools are usually very cumbersome to set up and only work if the recipient has set them up for their e-mail as well. But even then, only the message body and attachments are encrypted.

Secure/Multipurpose Internet Mail Extensions (S/MIME) S/MIME[7] is a public key encryption and signing scheme that relies upon a centralized Certificate Authority (CA) to handle encryption keys. The keys consist of X.509 certificates, the same sort of certificate that is used for Transport Layer Security (TLS) encrypted web traffic. It is not a standalone messaging system but rather an encryption scheme that can be used in combination with e-mail to encrypt the body and attachments of messages.

Setting up and maintaining such a CA is rather difficult and therefore, S/MIME is mostly used by larger corporations or other institutions. Individuals rarely go through the trouble of setting up such a CA for personal use [13]. Also, to exchange messages that are encrypted with S/MIME, both parties must be registered with a CA from which they obtain their encryption keys.

Using S/MIME requires a certain amount of trust in the CA to manage the keys responsibly. X.509 certificates follow a hierarchical tree-like structure, where the users' encryption keys are the leafs. The root and intermediary keys can be used to spoof messages coming from users which are authenticated with keys that are leafs in the same branch. If the root or intermediary keys were leaked, the unauthorized holder of the keys could spoof messages, perform Man-in-the-Middle (MITM) attacks, and arbitrarily issue keys that will be trusted by anyone who trusts the CA. For TLS, this has already happened on several occasions to high value CA's like the Google Internet Authority G2 [8].

Note that this kind of key escrow may be useful for work environments. The contents of e-mail sent by a company's employees can have a large impact on the company's reputation and is often times monitored for quality assurance. In some cases, employee's e-mail can be used as evidence in legal disputes.

Pretty Good Privacy (PGP) PGP[8] is a program for encrypting and signing content using a public key encryption scheme that is not based on a centralized CA. Instead, PGP relies on

[7] The protocol specification can be found at `https://tools.ietf.org/html/rfc3851`.
[8] The official website can be found at `https://www.pgp.com`.

a decentralized Web of Trust (WOT) for key authentication. Note that the OpenPGP Alliance maintains an open source version of the protocol called OpenPGP[9].

PGP is not a standalone messaging system but rather an encryption scheme that can be used with almost any other messaging system to encrypt and sign the body and attachments of a message. Metadata of such messages usually remains unencrypted. Most of the time, PGP is used to encrypt and sign e-mails. Plugins for various e-mail clients are available to automate this task (e.g. Enigmail for Thunderbird). There is even a browser extension called Mailvelope that allows to en- and decrypt e-mails over popular web clients like GMail.

PGP has been around for a while and has proven itself to be very secure. In his 2013 leak of National Security Agency (NSA) secrets, whistleblower Edward Snowden relied on PGP encrypted e-mail to communicate with journalist Glenn Greenwald [3].

Contrary to S/MIME, PGP is mostly used in private environments because WOT is not very practical for work environments. Unfortunately, PGP has not been adopted by the larger public; it is mostly used by a small fraction of very tech savvy and privacy oriented group of people. This is largely due to the fact, that it is very cumbersome to set up and use. Additionally, the WOT system requires a lot of work by the users to have their public key signed by other people who are already part of the WOT.

Internet Relay Chat (IRC) Internet Relay Chat (IRC)[10] was invented in 1988 which makes it one of the oldest instant messaging systems. The principle functionality of IRC are chatrooms called channels, but private conversations between two users are possible as well. The popularity of IRC has decreased in recent years. With the exception of freenode, all the major IRC networks showed a receding userbase [24].

An interesting aspect of the IRC protocol is the way server federation works. IRC is mostly used by connecting a client to one of the popular IRC networks like freenode[11] or QuakeNet[12]. These networks allow users to communicate with other users within the same network. The networks usually consist of a multitude of IRC servers, or nodes, which are arranged as an acyclic graph. If the connection between two nodes is interrupted, the whole network is split in two and users may be cut off from certain channels. This makes IRC networks very vulnerable to outages.

IRC only supports encryption for messages in transit in form of TLS. Whether or not this encryption is actually used depends on different factors. First, the network must support TLS encryption. If so, the operator managing a channel may force clients to connect over TLS. Then, the client must support TLS encryption as well. If so, users can configure it to only connect over TLS. This leaves scenarios where the traffic is not encrypted.

Extensible Messaging and Presence Protocol (XMPP) Extensible Messaging and Presence Protocol (XMPP)[13], formerly known as Jabber, is an open protocol, maintained by the XMPP Standards Foundation (XSF).

[9] The official website can be found at `http://openpgp.org/about_openpgp/`.
[10] The protocol specification can be found at `https://tools.ietf.org/html/rfc2810`.
[11] The official website can be found at `https://freenode.net/`.
[12] The official website can be found at `https://www.quakenet.org/`.
[13] The official website can be found at `https://xmpp.org/`.

XMPP is also one of the few instant messaging system where anyone can host their own messaging server. Setting up a self-hosted XMPP server is comparable with setting up an e-mail server in terms of complexity. It is not necessary to manage an entire network like it is the case with IRC.

In March 2014, the XSF released a public statement in form of a manifesto, declaring that all client-server as well as server-server connections must be encrypted according to certain guidelines.[14] The manifesto also states that the XSF is working on built-in E2E encryption. However, more than two years after the release of the manifesto, XMPP is still missing said E2E encryption, it has to be implemented by the client. The same problem arises as with e-mail: if the recipient does not use a client that supports E2E encryption, or if the recipient has not set up his encryption keys and made them available to the sender, the conversation is only encrypted in transit.

Google used to have a full XMPP support for Google Talk. But in May 2013, during the rebranding of Google Talk to Google Hangouts and the introduction of new features, this support was limited drastically [10]. According to some sources Google limited XMPP support amongst others because of problems with spam messages [23], [27].

Short Message Service (SMS) Although it is a messaging system, SMS is quite different from the others on the list. It does not send messages over the Internet but rather over the cell phone network.

The characteristic of SMS is that everyone with a cell phone can send and receive SMS messages without needing a smartphone. In 2009, Tomi Ahonen conjectured that at the end of 2010, there would be around 4.2 billion SMS users worldwide which corresponds to roughly 80% of all cellphone users. This conjecture was supposedly confirmed by ABI Research in December 2010 [1].

Another aspect is the decentralized structure. Each wireless carrier can deliver SMS messages in his own network and route the messages to other carriers. There is no single company that has control over the entire SMS infrastructure like it is the case with most instant messaging systems today.

Unfortunately, SMS are inherently insecure. The messages are sent in plain text and carry a lot of metadata. This is, amongst others, a reason why Open Whisper System decided to drop SMS support for TextSecure (now known as Signal) [19].

Telegram Telegram[15] is a privacy oriented cloud messenger created by the Russian brothers Nikolai and Pavel Durov [28]. The Application Programming Interface (API) of Telegram is well documented and accessible online. This allows users to create custom clients that send and receive messages over Telegram.

Telegram distinguishes between two modes of messaging. In standard mode, messages are only encrypted in transit and are synced over all devices. In *secret chat* mode, messages are only exchanged between two devices (not all clients support secret chat) and are encrypted before leaving the device. Secret chat offers a few extra features like self-destructing messages,

[14] The manifesto can be found at `https://raw.githubusercontent.com/stpeter/manifesto/master/manifesto.txt`.

[15] The official website can be found at `https://telegram.org/`.

remote deletion of messages, notifications if a screenshot is taken of a picture, and it is not possible to forward messages from secret chats. These extra features might be very useful in some circumstances, however, they are not guaranteed to work. These features all require some form of action by the client of the chat partner. If the chat partner uses a client which does not support these features, they will not work. This is problematic because these features give the impression of security, even though they are easily circumvented. Some sources claim that the features do not even work correctly under normal circumstances: deleted messages from a secret chat supposedly remain in the message database of the device [2]. Nevertheless, secret chat offers other features as well, which are reliable and virtually impossible to circumvent. Forward secrecy and out-of-band key authentication are two of them.

Off-the-Record (OTR) Protocol Having been around for a while now, OTR[16] has proven itself to be a reliable way to communicate privately over almost any instant messaging system. It is not a messaging system itself but rather a protocol to encrypt the body of messages in other messaging systems.

In addition to E2E encryption with perfect forward secrecy and key authentication, OTR also offers deniability. Even though the chat partner can be authenticated to prevent MITM attacks, transcripts of the conversation can be denied by both parties. This makes it interesting for political journalists and other dissidents in oppressed countries. Because of this, it comes pre-installed in privacy oriented operating systems like Tails [26], the self-proclaimed "amnesic incognito live system".

However, there are some disadvantages with using OTR. While the communication between two devices is considered to be secure, the multi-party version (mpOTR [7]) has encountered some criticism for "weakening the security" [15], especially for leaving out forward secrecy [18]. Also, if a user uses multiple devices, these cannot be used interchangeably, separate conversations have to be used for each device.

Note that conversations (especially the key exchanges) are meant to be synchronous. This makes sense considering the ephemeral nature of OTR conversations. However, for conversations that are asynchronous, this may cause problems such as delaying the delivery of messages [17].

Signal Signal[17], formerly known as TextSecure, is a privacy oriented messaging application, developed by Open Whisper Systems.

Moxie Marlinspike, the founder of Open Whisper Systems, was one of the key developers to bring E2E encryption to WhatsApp. In fact, WhatsApp partnered with Open Whisper Systems to integrate the Signal protocol into their messaging application [22].

Signal itself is a full-featured open-source instant messaging system that was developed from the ground up with privacy in mind. It is one of the few applications to get all the seven points on EFF's Secure Messaging Scorecard [4], and although it is still in its beta phase, a desktop client is available in the form of an extension to the Google Chrome browser. It has to be activated with the phone in a similar fashion as the WhatsApp web interface.

[16]The official website can be found at `https://otr.cypherpunks.ca/`.
[17]The official website can be found at `https://whispersystems.org/`.

Signal's protocol is based on the OTR protocol mentioned above, with a few modifications [21]. Further, it uses a custom key exchange mechanism, which allows messages to be sent asynchronously, something that is not possible with standard OTR [20].

While Signal's protocol is considered to be very secure and has been audited by independent third parties including a research team at the Ruhr University Bochum [6], it is relatively young compared to OTR. This means that there might still be some issues with the protocol that have not been discovered yet. The university researchers, for example, discovered an *unknown key-share attack*. While this issue is relatively unimportant, it nevertheless shows that cryptographic protocols may suffer from small deficiencies in their early phases.

Signal's server and client software is open-source. Anyone could set up their own server and modify the client such that messages are sent over their custom server. Messages sent over such servers could only originate from and be sent to people using the modified client. Signal does not support decentralized servers, so all messages have to pass through their own, centralized server.

There is a conceptual flaw in Signal. The encryption scheme is based on OTR which supposedly makes conversations analogous to ephemeral face-to-face meetings. However, messages are sent to all devices (desktop client and mobile client) and conversations tend to stay on the device indefinitely; few people ever delete messages. To solve this problem, it would suffice to create a second, ephemeral conversation with the same contact alongside the long-lived one; the ephemeral conversation could then simply be deleted when the conversation has ended. But unfortunately, it is currently not possible to create multiple conversations with the same contact.

Another drawback is, that Signal uses the phone number to identify users. Messages can therefore only be sent from the mobile phone with that number and the desktop clients activated with that phone.

Threema Threema[18] is a security oriented instant messaging application developed by the Swiss company Threema GmbH. It is a relatively new contender in the instant messaging field; it was launched in 2012 [29]. Threema is one of the only messaging applications that can keep up with, or even surpass WhatsApp in terms of user interface (UI). It is beautifully designed and very intuitive to operate. Unfortunately, the application is limited to mobile platforms and there is no option to use it on multiple devices with the same account. There is no web interface that relays messages over the cellphone.

From a security point of view, Threema is a good choice as well. It uses E2E encryption and forward secrecy for all messages. The only weak points here are that the application is closed source and all messages must pass through Threema's servers.

Further, Threema suffers from the same conceptual flaw as Signal. While the conversations are protected with E2E encryption and forward secrecy, they are long-lived and only one conversation is possible with each contact.

[18]The official website can be found at `https://threema.ch/`.

Chapter

3

Requirements

This chapter describes the requirements that SPEM has to fulfill. The requirements are based on the analyses of the previous chapter. In addition to the list of requirements, a list of optional features is given as well. These are features that are not in the main focus of SPEM but are considered important nevertheless; SPEM strives to enable as many of these features as possible.

3.1 Requirements

Most of the minimum requirements listed here are established assuming the perspective of the end user. They are to ensure that SPEM effectively protects the users' privacy in a manner which they can verify for themselves.

Let's demonstrate this with an example. A user has full control over the instant messaging client running on their device and can therefore be certain (by checking the source code) that messages are encrypted according to the protocol specifications before leaving the device. If they send a message to a contact with a verified public key, the sender can be certain that the message can only be decrypted by the recipient without having to trust the server delivering the message or the client of the recipient. This is contrary to what we saw in the previous chapter with Telegram, which offers security features that require trust in the recipient's client as well (e.g. self-destructing messages). Such features will not be part of SPEM because they are misleading. The mandatory requirements also ensure that systems building on SPEM remain accessible to non-tech-savvy people.

- **End-to-End Encryption** – The first and most important requirement is E2E encryption. All messages are encrypted before they leave the user's device. This ensures that the content and parts of the metadata remain private.
- **Key authentication** – For E2E encryption to work properly, there must be some way to verify that messages were actually encrypted by or for the intended chat partner.

- **Encryption in transit** – To minimize the amount of metadata that is transmitted in plain text over the Internet, any data exchanged between servers and clients have to be secured by an additional layer of encryption.
- **Decentralized servers** – Any server that relays messages between clients must, to some extent, have access to a message's metadata. If all messages are relayed by the servers of a single provider, this provider can aggregate these data. Using big data techniques, the company could then learn things about its users, which they never intended to reveal. This is why there should not be a centralized server and why users should be able to choose which provider gets to handle their messages. Nevertheless, they must be able to communicate with people who chose a different provider.
- **Off-the-Record** – There may be cases where an ephemeral conversation is needed. For the user, this should be as easy as normal messaging. Yet, ephemeral conversations need to conform to a higher standard of privacy protection. They must be deniable and must be impossible to reconstruct once deleted, even if the messages were intercepted and the long-term encryption keys are compromised some time later.
- **Usability** – For a messaging system to be widely adopted, it has to be easy to use. If there is the least bit of complication, people switch to an easier solution, even if the easier one is less secure. Therefore, the messaging protocol described here must be conceived such that it is possible to implement the clients in a way that they are at least as easy to use as the common messaging systems described in Chapter 2. This is particularly relevant for privacy oriented messaging systems because they require key management, which must be completely hidden from the user. Other aspects, like key verification and account management also have to be kept as simple as possible. Further, it is important that none of the common features are missing. These include group conversations, sending media files, and having client implementations for common platforms.
- **Cross-device** – This requirement is tightly related to usability. A user has to be able to send and receive messages from any number of devices. The current messaging landscape can be divided into two groups (with the exception of PGP-encrypted Email): messaging solutions that encrypt messages and messaging systems featuring full cross-device capabilities. There appears to be an inherent trade-off between accessing messages from multiple devices and encrypting messages before letting them leave the device. However, both are needed for a secure yet usable messaging system.
- **Web interface** – Under the points *Usability* and *Cross-device* it was already mentioned that there must be clients for all platforms and that users must be able to interchangeably use multiple clients on multiple devices. Listing a web interface is therefore redundant. However, web interfaces for messaging systems are slightly different than other clients and must be available as well. Web-interfaces have the advantage that they allow users to send and receive messages without having to install any software, as long as a web browser is available. Users might want to access their messages from devices on which they are not allowed to install software, like a computer at work for example.

3.2 Optional features

This section describes features which should be supported as well as possible by SPEM but which are guidelines rather than strict requirements.

- **Simplicity** – The protocol should be kept as simple as possible. The more complexity is introduced, the less secure the messaging system as a whole could become through unforeseen behavior and bugs in the implementation.
- **Re-use of existing technology** – There are many protocols and libraries related to encryption and messaging which are already available. Many of them have been used in production for a long time and have proven themselves to be reliable and secure. Some of these technologies are backed by large communities which are continuously improving the available protocols and libraries. Whenever possible, SPEM should make use of these technologies instead of inventing something new.
- **Metadata containment** – Even though it is not possible to keep all metadata from being exposed, it should be encrypted whenever possible and only visible if absolutely necessary.
- **Hostile to spam** – Spam is a bane to any messaging system and its users. It is difficult to distinguish real messages from spam messages and spam messages are highly annoying for users. Encrypting messages makes this even more difficult, as spam filters do not get to see the body of the message in transit.
- **Easy adoption by the industry** – To promote widespread adoption, SPEM should integrate seamlessly with existing systems, especially with related systems like e-mail. Also, it might be advantageous for businesses to have the ability to monitor the messages of their employees, as long as there is no way to do it surreptitiously.[1]

[1] An employer monitoring the company issued messaging accounts is not considered a breach of privacy.

Chapter

4

Protocol specification

This chapter describes the inner workings of SPEM. SPEM is based on a classical client-server infrastructure. A messaging server consists of a web server implementing a JavaScript Object Notation (JSON) based Representational State Transfer (REST) API, and the messaging clients are applications issuing requests to this API. To send and receive messages, the clients post to and poll from the servers. All requests and responses are encrypted in transit with TLS encryption. Additionally, the messages are encrypted with either the Rivest-Shamir-Adleman (RSA) algorithm or the OTR algorithm.

To facilitate protocol development, a proof of concept implementation was developed in Python. The implementation is largely incomplete and only served to test certain aspects of SPEM.

Throughout this chapter, the users *Alice* and *Bob* as well as the servers *abc.com* and *xyz.org* are used as placeholders to simplify explanations and illustrations. *Alice* and *Bob* run through the procedure of setting up their messaging applications and exchanging messages. From time to time, *Eve* is used as a placeholder for an attacker. At each step, the protocol, i.e., the JSON API, is explained in detail.

4.1 User and device management

First off, *Alice* and *Bob* need to register an account on a messaging server. *Alice* creates an account on *abc.com*, *Bob* creates one on *xyz.org*. The signup process could be done over a website by filling out a form, or directly from a messaging client. If the user signs up over a website, providers may handle the process however they want, but their messaging servers must still implement the corresponding API route, in case a user wishes to signup from their preferred messaging client.

Signup *Alice* opens her messaging client and fills out the required fields. She registers on *abc.com* with the username 'alice' and the password 'top-secret'. Her messaging address will be

'alice@abc.com'. It is no coincidence that messaging addresses look exactly like e-mail addresses. They follow the same specifications as e-mail addresses. This has two advantages: There is no need to come up with new format specifications, and SPEM can easily be added on to any existing e-mail service. Similar to Google Hangouts, the users only need one account and one address for e-mail and instant messaging.

When *Alice* clicks the signup button, her messaging client issues a `POST` request to `https://abc.com/users`. This is shown as step 1 in Figure 4.1a. The request contains a JSON body with the following field:

- `user` (Object) – The user object to store on the server. The username must be unique for each domain. Only the fields `username` and `password` are required for signup. If any device objects are supplied, they are registered as well. A `user` object generally contains the following fields. Note that depending on the action, the fields are not all present.
  - `username` (String) – The username of the user.
  - `password` (String) – The password of the user. This field can only be submitted by the user, but never returned by the server. The server should not store the password in plain text.
  - `devices` (Array of `device` objects) – The devices which are registered for this user. The array may be empty when no devices have been registered yet.

The response from the server is also in JSON format and contains the following fields:

- `success` (Boolean) – Whether or not the API call was successful.
- `notification` (String) – A notification that can be shown to the user.
- `user` (Object) – The user object as it has been created on the server. It does not contain a password field.
- `error` (Integer) – Error code. Only present if request was unsuccessful.
- `error_description` (String) – Human readably description of the error. Only present if request was unsuccessful.

The fields `success`, `notification`, `error`, and `error_description` make up the standard server response and are always returned. If no response is specified for an API call by the protocol description below, only the standard response is returned. Here, a `user` object is returned in addition to the standard response.

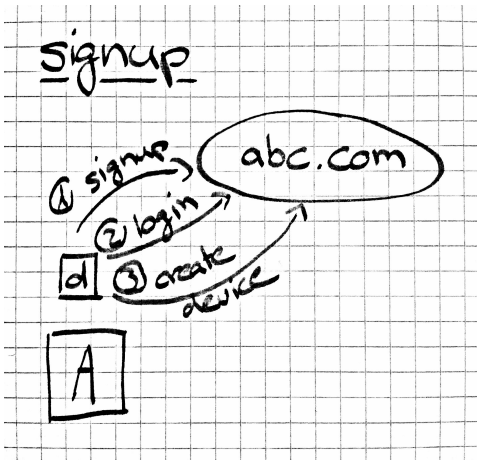
Examples of the JSON objects, which could be sent in the signup request and returned by the server, are available in appendix A.1.

Note that the signup process limits the registration information to the fields specified for the `user` object. This means that a provider cannot ask for more information than the username and password. This problem could be addressed in a future version of the protocol. The API could be adapted to have a route that returns a `user-model` object that contains all the fields that must be submitted by the client to register a new user.

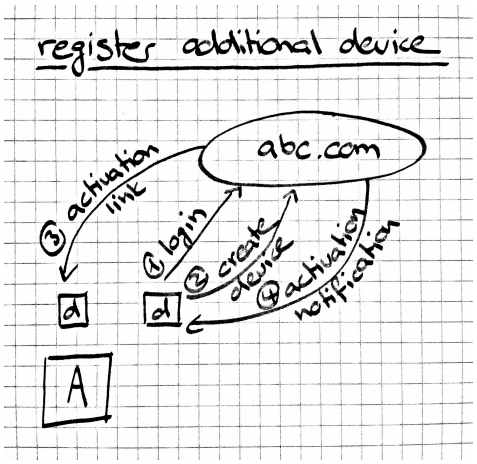
Modify account To modify information she entered during the signup process, *Alice* can always issue a `PUT` request to `https://abc.com/users/alice?password=top-secret` with a new `user` object. The `user` object would only contain the fields that she wishes to change on the server.

Delete account If *Alice* ever wishes to completely delete her account and all associated information from the server, she can send a `DELETE` request to `https://abc.com/users/alice?password=top-secret`.

The account is not removed immediately for security reasons. First, the server sends a message to all her devices, notifying her of the action. Optionally, if her messaging account is associated with an e-mail account, an additional e-mail could be sent containing the same information. This message contains two links. The first link, which has a short expiration of only two hours, immediately disables the account when clicked. The disabled account is not accessible over the API and is therefore prevented from sending or receiving messages. It is removed from the server after 24 hours. The second link disables the first link. If the account has already been disabled, it is enabled again. The second link expires either when the first link has expired or, if the first link was clicked, when the account is removed from the server. Any action that is taken is notified to all devices. The expiration delays listed above can be adapted by the server and should in any case be included in the message.



(a) Alice (A) signs up on a messaging server *abc.com* (step 1), logs in (step 2), and registers her first device *d* (step 3).



(b) Alice (A) logs in from a new device (step 1) and registers it on the server (step 2). An activation link is sent to her other devices (step 3); as soon as it is clicked, her new device is activated (step 4).

Figure 4.1: Procedure for registering users and their devices on a messaging server.

Login Now *Alice* has an account on *abc.com*. The next step is to log in. Normally, *Alice* would have to enter her messaging address and her password to log in, but since she just entered it to create an account, the client can reuse it without *Alice*'s interaction. The client uses the password to authenticate the next API call, which is to store a randomly generated cookie on the server. Once the cookie has been stored on the server, it can be used to authenticate most of the API calls. There are however a handful of privileged actions that still require the password, for example, setting a new password or deleting the account. The advantage of using a cookie like this is that the password never has to be stored on the device.

The API call for logging in is a `POST` request to `https://abc.com/users/alice/login?password=top-secret`. This is shown as step 2 in Figure 4.1a. Note that the information

for authentication is passed as query parameter in the Uniform Resource Locator (URL). Therefore, special characters need to be encoded as `%<refernce_number>...` (the space character is encoded as `%20` for example). The request body needs to contain the following JSON field:

- `cookie` (Object) – The cookie to use for authentication. A `cookie` object contains the following fields:
  - `token` (String) – Token consisting of 256 random bits, represented as hex string.
  - `expiration_time` (Integer) – The time in seconds since epoch[1] when the cookie expires. The server may override this value.

The response of the server is also in JSON format and contains the standard response, as well as the following field:

- `cookie` – The actual cookie that was stored on the server.

Note that cookies are always limited in time; they always have an expiration time. This expiration time can be set arbitrarily by the client. However, the expiration time the client sends with the cookie can be overridden by the server. If the server wants to limit the cookies Time-to-Live (TTL), it can do so by storing a different time than was sent by the client. After the TTL has expired, the server stops accepting authentication with this cookie.

In addition to the password, an optional 2-factor authentication token could be demanded in a future version of the protocol.

Renew cookie Since the cookie expires after a certain amount of time, it has to be renewed regularly by the client. There is a dedicated API route which permits the client to set a new cookie, using the old cookie to authenticate.

The API call to renew the login cookie is a `PUT` request to `https://abc.com/users/alice/login?cookie=<old-cookie>`. The JSON body of the request as well as the response from the server are the same as for the login.

Logout Logging out is as simple as sending a `DELETE` request to `https://abc.com/users/alice/login?cookie=<cookie>`. This makes sure that the cookie is deleted from the server, which makes it useless in case it was compromised.

Register device Registering a device happens behind the scenes when signing up to create an account or when logging in from a new device for the first time. The API call to register a new device requires password authentication for extra security. For *Alice* this does not make any difference because as stated above, devices are only ever registered when logging in or signing up. In these situations she has to enter her password anyway.

To register a new device, the client sends a `POST` request to `https://abc.com/users/alice/devices?password=top-secret`. This is shown as step 3 in Figure 4.1a. The JSON body of the request contains the following field:

- `device` (Object) – The device object to be registered to the user's account. The `device` object must contain the `public_key` field, the other fields are not necessary. A `device` object generally contains the following fields:

[1] The time since epoch is always based on Coordinated Universal Time (UTC).

- `id` (Integer) – The IDentifier (ID) of the device.
- `public_key` (Object) – The `public_key` object contains information related to the RSA public key.

The server's response contains the `device` object that has been created, as well as the standard response.

The exact format of the `public_key` field has not been decided yet. This is partially due to limitations of the *rsa* module and the complexity of the *pyOpenSSL* module in Python, which make progress slow in this regard. Another reason is the difficulties that come with formats like PEM and DER. PEM and DER data are supposed to be stored in a file and therefore consist of multiple lines of text. However, JSON cannot handle line breaks out of the box: they have to be escaped. This makes it much more complex to specify such a format. The easiest way would be to use a JSON-based format that stores the key information analogous to the Extensible Markup Language (XML) format for RSA keys. Unfortunately, such a format does not officially exist and defining one is out of the scope of this thesis. Note that the format for RSA keys, defined by Jones [12], contains more information than just the key itself and is therefore unsuitable. For the proof of concept implementation, a custom format was used, where the public key object contained two integer fields: `e` and `n`.

Note that the public keys do not have an expiration date. This means they should not be cached to reduce traffic, otherwise they might be used even though they have been deleted from the server. A future version of the protocol could introduce a TTL for public keys, making caching possible.

Now that *Alice* has registered her first device, she is ready to start messaging. Before she starts though, she registers a second device. For this, she logs in from the second device and issues the necessary API calls for registering a device. Contrary to the first one, the second device is not activated immediately. The server sends a message to the already registered devices (only one in this case) containing an activation link and the fingerprint of the public key of the new device. The server uses the address 'postmaster@abc.com' to send these messages. They are regular messages like the ones exchanged by users. *Alice* compares the fingerprint in the message with the fingerprint of her second device. If they match, she clicks the link to activate the newly registered device. Then, the server sends a 'welcome' message to the new device. These steps are illustrated in Figure 4.1b.

The format of the activation link is not specified. The messaging providers may choose any format they wish. In a future version of the protocol, activation messages could be standardized. If they are standardized, the messaging clients can parse them to show the user an activation button instead of just the link. But since the server sends the link in each case, standardization is not necessary. Note that the activation links should also expire after a while.

A Public Key Infrastructure (PKI), like the one used for SPEM, only works if the users can trust the keys provided by the server. For this it is essential that there is a way to authenticate the keys of other users. SPEM does not specify how this is to be achieved because it happens entirely in the client. A future version of the protocol could still standardize this because there are some important points to consider. For example, the system for key authentication must be very simple for the user, otherwise users will not use it. A good example is the way Threema handles key authentication. The public key's fingerprint can be displayed on the smartphone

as QR code, which can then be scanned by other smartphones. This method is quick and easy but it causes compatibility problems. While the system with the QR code is very simple and works on every modern smartphone, it is rather cumbersome with other types of devices. If the users want to compare the keys' fingerprints by talking over the phone, a QR code cannot be used either.

Get device info If *Bob* needs information about *Alice*'s devices, he can issue a `GET` request to `https://abc.com/users/alice/devices` which returns the following field:

- `devices` (Array of `device` objects) – The devices which are registered for the user *alice*. The array is empty if no devices have been registered yet.

Alternatively, *Bob* can get information about a single device by issuing a `GET` request to `https://abc.com/users/alice/devices/<device-id>`. This request returns the field:

- `device` (Object) – The device object with ID `<device-id>`.

Note that the ID of a device can be incremental starting at 0 for each user, incremental over all users, or completely random. It is up to the server to decide how IDs are assigned. Giving incremental IDs over all users does not allow an attacker to iterate over all IDs to create a list of registered users. The username of the device's owner has to be known to access a device; the ID alone is not enough.

Change device If *Alice* ever feels that the private key of one of her devices has been compromised, she can generate a new one and update it on the server. The API call for this is very similar to registering a new device. The client needs to issue a `PUT` request to `https://abc.com/users/alice/devices/<device-id>?password=top-secret` with a JSON body containing the new device information. All the fields in the `device` object are updated on the server. If any are missing, they are left as they were before.

Remove device To remove a device from her account, *Alice* can issue a `DELETE` request to `https://abc.com/users/alice/devices/<device-id>?password=top-secret`.

The device is not removed immediately. The procedure to definitely remove the device is the same as for removing the account from the server. A message is sent to all devices, containing links to confirm the removal of the device or to abort it.

4.2 Message exchange

Alice, as well as *Bob* who signed up and registered two devices on *xyz.org*, are now ready to exchange messages.

There are two types of messages. The first is the *standard* type, where the body is encrypted and signed with the RSA keys of the devices. The second type of message is *otr* as in Off-the-Record. In *otr* messages the body is encrypted using the OTR protocol; there is no separate signature.

Note that for SPEM the choice of message type for a conversation has further implications in addition to the way messages are encrypted. Distinguishing between two types of messages allows a clear separation of short- and long-lived conversations. If, for example, *Alice* wishes to

start an ephemeral conversation with *Bob*, she can start an *otr* conversation. This conversation benefits from all the features of the OTR protocol like forward secrecy and deniability. This, however, is unsuitable for long-term conversations.

The OTR protocol requires a synchronous session to be established in order to encrypt messages. Let's imagine a scenario, for example, where two devices establish such a session. Then, one of them goes offline but both devices send a message. The session becomes inconsistent across the two devices and has to be reestablished once both devices are online again. Now, consider that OTR sessions are always limited to two devices, but most users want to use multiple devices interchangeably. This means that separate sessions would have to be created and maintained for all possible pairs of devices. Managing all these sessions becomes a large overhead, especially because not all devices are online all of the time.

Also, for long-term conversations there is nothing to be gained from features like forward secrecy. If the OTR session is closed and the messages are deleted, they become impossible to recover. This is one of the strong points of OTR. But this does not work if the session runs indefinitely and the plain text messages remain on the device. If ever the long-term key should be compromised, the plain text messages could just as well be compromised.

The deniability characteristic of OTR conversation is rendered useless in much the same way as forward secrecy. If the messages remain on the device instead of being deleted once the conversation has ended, deniability would be lost in case the messages are discovered.

Using *standard* messages and leveraging an RSA PKI, we loose deniability and forward secrecy, but get a much simpler system, where we do not have to manage any sessions. This is much more suitable for long-lived conversations. If *Alice* wants to send a message to *Bob*, she can simply encrypt the body with the public keys of all of *Bob*'s devices and submit the messages (one for each device) to *Bob*'s messaging server (see Figure 4.2). In case she wishes to have her messages synchronized across all of her devices, she repeats this process for them as well.

Using the two message types alongside one another has some more subtle advantages. If a user wants to start an ephemeral conversation, they need to do this very explicitly in their messaging client. This makes the user more conscious of the fact that whatever is said in this conversation is supposed to remain confidential. It also makes it more intuitive to delete the messages of this ephemeral conversation; it can be done by closing the conversation. Also, OTR conversations only ever include two devices. When discussing sensitive information, this is an advantage since messages do not end up on all devices. For normal conversations, on the other hand, this characteristic is limiting usability because it does not allow the user to use his devices interchangeably.

Message format Messages are composed and sent to the server of the recipient as `message` objects. These JSON objects contain the following fields:

- `sender` (String) – Messaging address of the sender.
- `sender_device_id` (Integer) – The device from which the message was sent.
- `recipient` (String) – Messaging address of the recipient.
- `recipient_device_id` (Integer) – The device for which the message is intended.
- `type` (String) – Type of the message. Can either be *standard* or *otr*.

- `body` (Object) – Cyphertext of the message body. The format of the message body depends on the type of message. The format for *standard* messages is discussed in section 4.2.1, that of *otr* messages in section 4.2.2.

Since this information is visible to the server, it is important to only include information that is absolutely necessary for the server to deliver the message.

Note that more metadata are leaked than what is above. The server knows for example *when* the message was sent and from *where* (more precisely, from which Internet Protocol (IP) address).

The API calls to exchange messages are the same for both message types, but they are used differently. The API calls to send, fetch, and delete messages are listed below. The way they are used for each message type is explained in sections 4.2.1 and 4.2.2.

In general, message exchanges consist of one device submitting a JSON `message` object to the server of the recipient. Each `message` object is destined for exactly one device of the recipient. If the recipient has more than one device, a `message` object has to be submitted for each of them.

Submit message To submit a message to *Bob*'s server, *Alice* issues a `POST` request to `https://xyz.org/users/bob/devices/<device_id>/messages`. The JSON body of the request needs exactly one field:

- `message` (Object) – The `message` object to be submitted to the server.

The response from the server repeats the `message` object back to the client.

Fetch messages *Bob* can demand a list of messages for each of his devices by issuing a `GET` request to `https://xyz.org/users/bob/devices/<device_id>/messages?cookie=<cookie>`. The response from the server contains the field:

- `messages` (Array of `message` objects) – The messages that are available for a device. This array is empty if no messages are available.

Alternatively, single messages can be fetched by extending the API call with the message ID as follows: `GET` request to `https://xyz.org/users/bob/devices/<device_id>/messages/<msg_id>?cookie=<cookie>`. The server responds with the field:

- `message` (Object) – The message that is stored on the server.

It could be wise to limit the number of messages that are returned in a single response. This prevents the response from becoming too big if a client has not checked its messages for a long period of time. For this version of the protocol, no such limitation is imposed because the implications of such a limit have not been explored in detail.

Delete message Once *Bob* has fetched a message, he can delete it from the server by sending a `DELETE` request to `https://xyz.org/users/bob/devices/<device_id>/messages/<msg_id>?cookie=<cookie>`. Deleting messages is done in a separate step to make sure that the client has time to store the message before it is removed from the server. Otherwise, a message might be lost if the client crashes before the message is saved.

Note that the messaging clients have to constantly poll the server for new messages. Depending on how this is implemented, it can be very energy inefficient. On mobile devices,

this could cause the battery to drain faster. To reduce this effect, a variety of push notification services could be used. These services are often already present on the devices (like Google Cloud Messaging (GCM) on Android for example). Using such services could raise privacy concerns and should therefore be optional.

4.2.1 Standard messages

By default, messages are sent as *standard* type messages. Figure 4.2 lists the steps required to send a message. Let's look at the scenario where *Alice* composes a message to *bob@xyz.com* and presses the 'send' button.

The first thing her messaging client does is to look up the devices that are registered for *Bob*'s messaging address. Each device has its own RSA public key, so the content of the message is encrypted and signed once for each device, in this case twice. The results are two JSON `message` objects, which are then submitted individually to *xyz.org*.

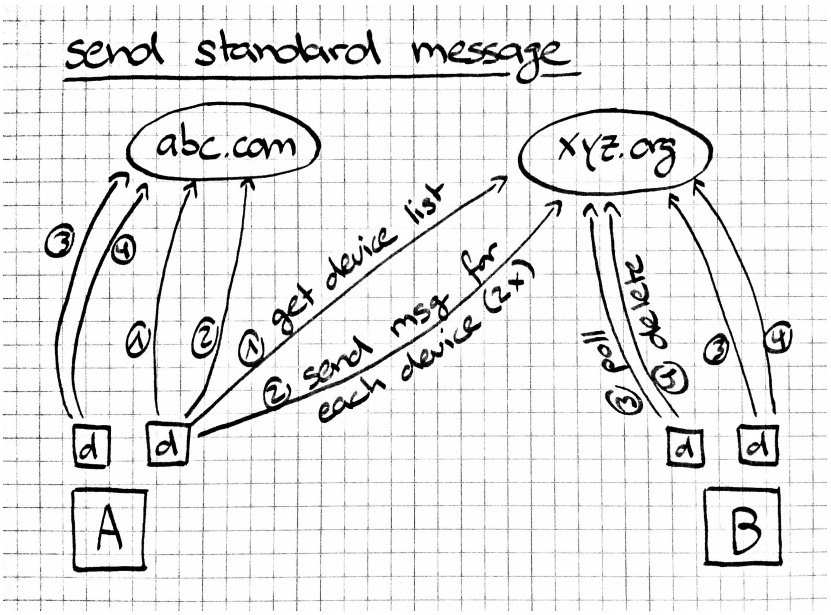


Figure 4.2: Alice fetches a list of *Bob*'s devices (step 1). She encrypts the body of her message for each of *Bob*'s devices and submits them to *Bob*'s messaging server (step 2). *Bob*'s devices can then individually fetch the message that was encrypted for them (step 3) and delete it from the server (step 4). *Alice* repeats steps 1 to 4 for her own devices to keep her conversation logs synchronized.

The server at *xyz.org* performs multiple tests on the messages from *Alice*. First, the server checks whether the recipient and the recipient's device exist and are valid. Then the server checks whether *Alice*'s signature is valid. To do this, her public key is requested from *abc.com*, the signature decrypted, and the hashes compared. If the public key of *Alice*'s device is not available at *abc.com* or if the signature is not correct, the server at *xyz.org* immediately discards the message. If everything checks out, the message is made available for download by the recipient's device.

Similarly to the way a message is sent to each of *Bob*'s devices, a copy of the message is sent to all of *Alice*'s devices as well. If this is not done, the conversation log on the other device would be incomplete. The message *Alice* sends to her own device is similar to the one she sends

to *Bob*, but not exactly the same. First, it is addressed and encrypted for herself. Second, the `data` field of the message contains a copy of the message sent to *Bob*.

Body format As remarked above, the `body` object of *standard* messages are different from those of *otr* messages. The body of *standard* messages contains the following fields:

- `symmetric_key_cypher` (String) – The symmetric key can be decrypted with the recipient's private key and then be used to decrypt the cyphertext. The symmetric key consists of 256 randomly generated bits and is unique for each message.
- `cyphertext` (String) – The cyphertext consists of a `content` object, which is encrypted using the symmetric Advanced Encryption Standard (AES) algorithm. The decrypted `cyphertext` string is a `content` object containing the fields:
  - `sender` (String) – Messaging address of the sender.
  - `sender_device_id` (Integer) – The device from which the message was sent.
  - `recipient` (String) – Messaging address of the recipient.
  - `recipient_device_id` (Integer) – The device for which the message is intended.
  - `data_type` (String) – Multipurpose Internet Mail Extensions (MIME) type of the data. The unregistered MIME type `aplication/x-message-content` can be used to send a `content` object (to synchronize messages across devices).
  - `data` (String) – Actual data that are sent in the message.
  - `sent_time` (Integer) – Time when message was sent (in seconds since epoch).
  - `conversation_info` (Object) – A JSON object containing information about the context of a message:
    - `conversation_token` (String) – An arbitrary 256 bit hex string identifying the conversation.
    - `conversation_members` (Array of Strings) – A list of messaging addresses of users that should receive messages sent in this conversation.
    - `message_number` (Integer) – An integer that is sent and incremented with each message (separately for each user).
- `cyphertext_signature` (String) – Signature of the cyphertext. SHA-256 hash, encrypted with the private key of the sender.

Examples of encrypted and decrypted messages are shown in appendix A.2. If the data field contains a message of the type `aplication/x-message-content`, additional checks should be done. Normally, these messages only come from devices belonging to the same user.

It sticks out that the addresses and device IDs of the sender and recipient are listed twice in every message: once encrypted and once unencrypted. They are listed twice because on one hand, they have to be unencrypted and readable to the server delivering the message, but on the other hand, they have to be encrypted to make sure that they cannot be altered in transit.

If they were not repeated in the encrypted body of the message, the following attack would be possible. *Alice* sends a message as *alice@abc.com* to *bob@xyz.org* containing information that only she could know. *Eve* registers a device with the same public key as *Alice*'s device under the account *aIice@abc.com* (lower case 'l' replaced by capital 'i'). She then intercepts *Alice*'s message and replaces the sender's address with her own address (idem for the device ID). When *Bob* receives the message, he might not notice that the sender address has changed

because the information within the message could only come from *Alice*. But when he replies, the reply goes to *Eve*. So now *Eve* can have a conversation with *Bob*, while he thinks he is writing to *Alice*.

Currently there is another weak point of the protocol as well. There is no guarantee that *Alice* sends the same message to all of *Bob*'s devices and there is no way to check it either. A possible solution to this would be to add an unencrypted field to the `body` object that contains a hash of the plain text content. The server would then accept messages in batches only. If *Alice* wants to send a message to *Bob*, she only makes one call to the server, submitting the messages for all of *Bob*'s devices at the same time. Before accepting the messages, the server checks that they all contain the same hash for the plain text content and that there is a message for each device. Afterwards, when the body of a message is decrypted on the recipient's device, it can be hashed and compared to the hash supplied by *Alice*. If they do not match up, the message is discarded and the other devices are notified to also discard the message.

4.2.2 Off-the-Record conversation

As mentioned above, *otr* messages are exchanged between exactly two devices. This has the advantage that only a single OTR session has to be maintained and that messages are not broadcast over all devices. Also, all the features of the OTR protocol can be leveraged.

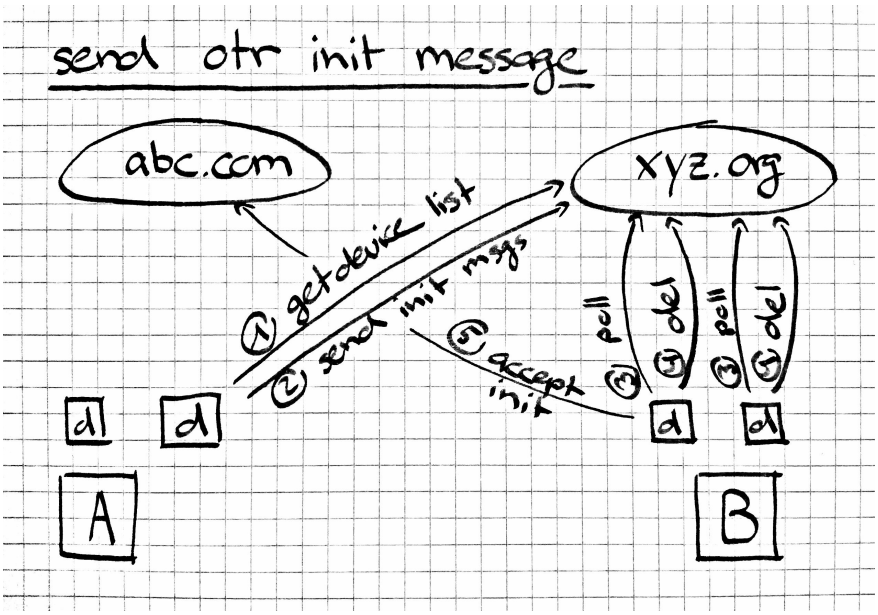


Figure 4.3: Alice fetches a list of *Bob*'s devices (step 1). She then sends an OTR init message to each of them, in this case two (step 2). All of *Bob*'s devices fetch the init message (step 3) and delete it from the server (step 4). The devices show a prompt to *Bob* to start the OTR conversation. The device on which *Bob* accepts the init message continues establishing the OTR session with *Alice*'s device (step 5). Messages are not synchronized across devices.

The process of of starting an OTR conversation is illustrated in Figure 4.3. *Alice* starts out sending an init message to all of *Bob*'s devices (this init message is called *OTR Query Message*). *Bob* is shown a prompt on each of his devices, asking him whether or not he would like to engage in this OTR conversation or not. If he accepts the prompt on one of his devices, the session is

started and the key exchange is made between the two involved devices (a whitespace-tagged plaintext message is used to accept the init message). Once the key exchange has succeeded, *Alice* and *Bob* can start exchanging messages.

On the API level, the only difference compared to *standard* messages is the format of the message's body. The API to submit and fetch messages is the same.

Body format The body of an *otr* message is a JSON object with only one field, which contains the OTR cyphertext.

Note that since the body does not have a cryptographic signature, the server cannot verify the recipients address. This is important because it adds to the deniability. Even though there is no cryptographic signature, *otr* messages are not easily spoofed. More information about why *otr* messages are difficult to spoof can be found in section 4.4.3.

4.3 Presence

A common feature of instant messaging systems is having a presence service. A presence service would allow *Alice* to know whether *Bob* is online, whether he is currently composing a message to *Alice*, or whether he has seen the messages *Alice* has sent.

The current version of the SPEM does not specify how presence information is handled. If such a service is added in a future version, it must respect the minimum requirements that were defined in section 3. For example, presence information must be encrypted end-to-end, which implies that presence information is handled exclusively by the clients and not the servers. This gives users complete control over which presence information is divulged and with whom.

A possible extension of the SPEM could specify a MIME type `application/x-presence-information`, a JSON object to be used as data in a *standard* message.

4.4 Implementation considerations

This section describes some guidelines to consider when creating software for this protocol. Note that a user has no guarantee that all software follow these guidelines. Users can only check the software of the clients running on their own devices or the software of their server if they choose to run their own.

4.4.1 Data as a toxic asset

The goal of this protocol is to enable a messaging system that protects the privacy of its users. The implementation of the messaging servers and clients should reflect this as well. Data should only be stored if absolutely necessary. Even the metadata should be discarded as soon as possible.

If data are stored, they should be protected according to best practice. Passwords, for example, should never be stored; a salted hash is much safer and is sufficient for user authentication.

4.4.2 Integration

For SPEM to become widely adopted, it has to be compatible with existing services that large companies already have in place. As mentioned above, the protocol is designed to be easily added to e-mail services or other services that use e-mail addresses to identify user accounts.

Large companies often wish to monitor employee communications. Whether it is for quality assurance or in case of legal disputes, it makes sense for a company to have access to the messages sent by employees over their company accounts. This is accomplished with a sort of key escrow where instead of generating new encryption keys on the devices, employees use keys that are provided by their employer. This allows the employer to have access to the messages in case there is need for it. The employer should, in this case, make the employees aware of the fact that their messages are under surveillance.

Note that an employer can only access the contents of *standard* messages but not the contents of *otr* messages. If the employer wishes to prevent *otr* messages, the messaging server can be configured to categorically reject them.

4.4.3 Spam

Spam is always a big issue for messaging systems. At first glance, SPEM appears to be more vulnerable to spam because messages are always encrypted (messages cannot be filtered by the servers depending on their content, as it is mostly done with e-mail), and there is no centralized server that could detect anomalous behaviors. Contrary to intuition, this is not the case.

Standard messages The fact that messages have to be signed by the sender (at least *standard* messages, more on *otr* messages below) makes spamming more difficult. Let's say *Eve* is a spammer, sending messages from her address *eve@abc.com* to users at *xyz.org*. All those messages can be traced back to her by verifying her signature. The server at *xyz.org* can keep track of how many messages *Eve* has sent and can refuse messages after a threshold is exceeded. The server at *abc.com*, where eve is registered is not powerless either. If it notices a large amount of requests for *Eve*'s public key, it knows something is wrong. After careful consideration, her public key can be suspended, causing all her spam messages to be discarded by the recipients.

Suspending users' keys to disable their accounts should be considered carefully, taking into account more than just the number of times their public key is looked up over a certain period of time. If the public key is simply removed when a threshold is reached, the users would be open to a Denial-of-Service (DOS) attack. *Eve*, for example, could send a large number of messages with *Alice*'s address as sender. The verification of the messages' signatures would obviously fail, but the recipients would look up *Alice*'s public keys nevertheless, causing them to be suspended.

Otr messages As mentioned above, *otr* messages do not have a standalone signature that can be verified by the server. Without such a signature, it is more difficult to pin a message to a user and block this user if too many messages are sent. But the absence of a signature does not mean that OTR messages are easy to use for spamming.

Otr messages that cannot be decrypted by the recipient are useless and discarded immediately, so there is no need for spammers to send them. This means that first, a spammer has to establish an OTR session by sending an init message. If *Eve* sends an *otr* init message to *Bob* putting *Alice*'s address as sender, *Eve* would never be able to establish the session. *Bob* would send his confirmation message to *Alice*, who would then discard it because she never sent the corresponding init message to *Bob*. In other words, *Eve*'s spam message never reaches *Bob* because the session between *Eve* and *Bob* is never established.

The only thing that *Eve* could do is annoy *Bob* with repeating prompts to start an OTR conversation. But this can be reduced by filtering init messages on a client level. Considering how rarely a normal person has to starts a new OTR conversation with people whose address are not already in their contacts list, the messaging client can use a default configuration where *otr* messages are discarded if they come from an unknown sender. Even if there are a few people who disable this filter, most would leave it on its default setting. This reduces the attack surface considerably, making this attack so ineffective that it is probably not worth the effort for spammers.

CAPTCHA In addition to the above precautions, a server could also demand users to solve a Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) before relaying submitted messages. CAPTCHAs are commonly used this way on websites containing a form the user has to fill out. The use of CAPTCHAs should be kept to a minimum however. Excessive use of CAPTCHAs could deter users.

4.4.4 Client diversity

For the user, the choice of messaging client is very important because it is the software they interact with directly. Since SPEM is open and designed for decentralized deployment, the software for servers and clients can be created by anyone who wishes to do so. This encourages developers to create implementations that suit different target audiences, like it is the case for e-mail. This also allows people to develop their own clients if they want to use a platform which is not yet supported.

Additionally to native clients, web clients can be created for cases where it is not possible or inconvenient to install a client. A web client is especially interesting for developers because with a single implementation, they can reach a large number of users on a large number of platforms.

Web clients, however, leave users vulnerable to attacks that are not possible with native clients. They have to run completely in the web browser and store everything locally. This is feasible in modern browsers with JavaScript and HyperText Markup Language (HTML) local storage. The problem lies in trusting the site providing the web client. Since the website is loaded anew every time the web client is accessed, it is possible that malicious JavaScript code is dynamically inserted, if ever the provider is breached by an attacker. This would most likely go unnoticed by the user, but would nevertheless expose all messages that are stored locally to the attacker.

Chapter

5

Conclusion

There are a lot of messaging systems in use today, especially systems for instant messaging. Unfortunately, users are facing a trade-off between security and usability. Older messaging systems like e-mail and SMS tend to be very insecure because they do not feature any encryption built in natively. The optional encryption protocols that can be layered on top are usually limited to encrypting the body of a message and leave a lot of metadata unencrypted. However, some of these older systems are very widely adopted and people can exchange messages independently of the messaging provider they use. Newer messaging systems like Threema and Signal, on the other hand, have strong encryption already built in. But users can only exchange messages with people that use the same provider and are usually forced to use the proprietary messaging clients.

By sallying out the best and the most important features of existing messaging systems, a set of requirements has been established to guide the development a new instant messaging protocol that combines all the good parts into a single protocol. The result of this synthesis is SPEM. SPEM is a protocol enabling secure and user-friendly instant messaging by fulfilling all the requirements listed in Chapter 3:

- **End-to-End Encryption** – Messages are encrypted end-to-end with a combination of the asymmetric RSA and the symmetric AES algorithms.
- **Key authentication** – The use of RSA keys allows users to authenticate the fingerprints of their chat partners.
- **Encryption in transit** – Since the messaging servers are web servers, traffic can easily be sent over encrypted TLS connections.
- **Decentralized servers** – Decentralization is achieved by leveraging standard routing methods similarly to e-mail. Messages are sent to a messaging address that consists of a username and a domain; the recipient's messaging server can be found using standard Domain Name System (DNS) lookups.

- **Off-the-Record** – The *otr* message type allows users to engage in ephemeral conversations.
- **Usability** – The SPEM provides support for all features of existing messaging systems. Through the use of MIME types, media files can be sent as Base64-encoded binary code. Group conversations are possible as well. But most importantly, message encryption and key management is tightly integrated into the protocol and hidden from the user.
- **Cross-device** – Since the protocol is not closed, anyone can create a messaging client for the platform of their choosing.
- **Web interface** – Web interfaces can be implemented in JavaScript, allowing a web interface to run completely in the user's web browser.

The optional requirements are also fulfilled to a large extent:

- **Simplicity** – The SPEM is kept very simple and can mostly be specified in just a handful of API calls.
- **Re-use of existing technology** – SPEM relies heavily on web servers with a JSON API, which is very common for web services nowadays. The encryption algorithms that SPEM uses are also very common and are available in most programming languages and platforms.
- **Metadata containment** – Only a very small amount of metadata is leaked in each message: the sender's and the recipient's messaging address, as well as the IDs of the devices they use.
- **Hostile to spam** – Since they are signed by the sender, *standard* messages can be traced back to the sender. The sender can therefore be held accountable for spam messages. For *otr* messages, default filter configurations make it unlikely that anyone would even try to send spam messages. Further, users can be asked to solve CAPTCHAs if they are suspected of spamming.
- **Easy adoption by the industry** – The use of e-mail addresses as messaging addresses, allows easy integration into existing services. The possibility of using a key escrow mechanism allows businesses to use messaging systems built on SPEM as well.

There is still potential for growth for the current version of SPEM. The signup process can be extended to allow messaging providers to dynamically add fields to the signup form and additional security can be added with two-factor authentication. The messages coming from the server (links for device activation for example) can be specified by the protocol for more uniformity across providers.

The current version of the protocol does not include a presence service. Especially for an instant messaging system, a presence service is important and should be added in a future version.

While not quite ready for production use, SPEM has the potential to replace insecure and proprietary instant messaging systems, and become a de facto standard in much the same way as e-mail.

Acknowledgements

I would like to thank Ulrich Ultes-Nitsche for supervising this thesis. The bi-weekly meetings we held helped me structure the thesis and helped me focus on the important aspects of the protocol. His regular feedback about the progress of the thesis was appreciated as well.

I would also like to thank Laura Rettig and Jürg Studer for taking the time to proof-read this thesis. They pointed out typographical, as well as grammatical errors, and gave me hints on how to rephrase some sentences to make them more suitable for a scientific paper.

Appendix

JSON examples

This section lists a handful of JSON objects to illustrate what the body of a request or response looks like for SPEM. In particular, JSON objects for the sign-up process and for the message exchange are shown.

A.1 Signup

Listing A.1 shows an example body of a signup request. Listings A.2 and A.3 show the response from the server if the request was successful or if it was unsuccessful, respectively.

```
{
    "user": {
        "username": "alice",
        "password": "top-secret"
    }
}
```

Listing A.1: Body of a signup request.

```
{
    "success": true,
    "notification": "User created successfully!",
    "user": {
        "username": "alice",
        "devices": []
    }
}
```

Listing A.2: Response of a successful signup request.

```
{
    "success": false,
    "notification": "User not created!",
    "user": null,
    "error": 400,
    "error_description": "Bad request: Missing username."
}
```

Listing A.3: Response of an unsuccessful signup request.

A.2 Messages

Listings A.4 and A.5 show an encrypted *standard* message as it would be submitted to the recipients messaging server and the corresponding decrypted message as it would be seen by the recipient.

```
{
    "sender": "alice@abc.com",
    "sender_device_id": 5,
    "recipient": "bob@xyz.org",
    "recipient_device_id": 2,
    "type": standard,
    "body": {
        "symmetric_key_cypher": "134ae...34bbef",
        "cyphertext": "5fa3b...321bc",
        "cyphertext_signature": "ef890...bcd22"
    }
}
```

Listing A.4: Example of an encrypted standard message.

```
{
    "sender": "alice@abc.com",
    "sender_device_id": 5,
    "recipient": "bob@xyz.org",
    "recipient_device_id": 2,
    "type": standard,
    "body": {
        "symmetric_key_cypher": "134ae...34bbef",
        "cyphertext": {
            "content": {
                "sender": "alice@abc.com",
                "sender_device_id": 5,
                "recipient": "bob@xyz.org",
```

```
            "recipient_device_id": 2,
            "data_type": "text/plain",
            "data": "Hello Bob!",
            "sent_time": 1467740684,
            "conversation_ino": {
                "conversation_token": "7a6c6...07535",
                "conversation_members": [
                    "alice@abc.com",
                    "bob@xyz.org"
                ],
                "message_number": 0;
            }
        }
    },
    "cyphertext_signature": "ef890...bcd22"
  }
}
```

Listing A.5: Example of a decrypted standard message.

Bibliography

[1] T. Ahonen, *Time to confirm some mobile user numbers: sms, mms, mobile internet, m-news*, 2011. [Online]. Available: http://communities-dominate.blogs.com/brands/2011/01/time-to-confirm-some-mobile-user-numbers-sms-mms-mobile-internet-m-news.html (visited on Apr. 16, 2016).

[2] Z. Avraham, *Telegram app store secret-chat messages in plain-text database*, 2015. [Online]. Available: https://blog.zimperium.com/telegram-hack/ (visited on Apr. 1, 2016).

[3] M. Calderone, *How glenn greenwald began communicating with nsa whistleblower edward snowden*, 2013. [Online]. Available: http://www.huffingtonpost.com/2013/06/10/edward-snowden-glenn-greenwald_n_3416978.html (visited on Apr. 6, 2016).

[4] Eletronic Frontier Foundation, *Secure messaging scorecard*, 2015. [Online]. Available: https://www.eff.org/secure-messaging-scorecard (visited on Mar. 22, 2016).

[5] R. Falkinge, *Whatsapp encryption shows value of metadata*, 2014. [Online]. Available: https://www.privateinternetaccess.com/blog/2014/11/whatsapp-encryption-shows-value-of-metadata/ (visited on Apr. 15, 2016).

[6] T. Frosch, C. Manika, C. Bader, F. Bergsma, J. Schwenk, and T. Holz, "How secure is textsecure?" 2014, [Online]. Available: https://eprint.iacr.org/2014/904.pdf (visited on Apr. 17, 2016).

[7] I. Goldberg, B. Ustaoğlu, M. D. Van Gundy, and H. Chen, "Multi-party off-the-record messaging," *Proceedings of the 16th ACM conference on Computer and communications security*, p. 358, 2009.

[8] D. Goodin, *Symantec employees fired for issuing rogue https certificate for google*, 2015. [Online]. Available: http://arstechnica.com/security/2015/09/symantec-employees-fired-for-issuing-rogue-https-certificate-for-google/ (visited on Apr. 6, 2016).

[9] R. Greenfield, *What your email metadata told the nsa about you*, 2013. [Online]. Available: http://www.thewire.com/technology/2013/06/email-metadata-nsa/66657/ (visited on Mar. 22, 2015).

[10] P. Higgins, *Google abandons open standards for instant messaging*, 2013. [Online]. Available: https://www.eff.org/deeplinks/2013/05/google-abandons-open-standards-instant-messaging (visited on Apr. 23, 2016).

[11] F. Jansen, *What's up with crypto in whatsapp?* 2015. [Online]. Available: https://myshadow.org/whats-crypto-whatsapp (visited on Apr. 15, 2016).

[12] M. Jones, *Rfc 7517 - json web key (jwk)*, 2015. [Online]. Available: https://tools.ietf.org/html/rfc7517 (visited on Jun. 26, 2016).

[13] P. Lambert, *Email encryption: using pgp and s/mime*, 2013. [Online]. Available: http://www.techrepublic.com/blog/it-security/email-encryption-using-pgp-and-s-mime/ (visited on Apr. 6, 2016).

[14] M. Lee, *Battle of the secure messaging apps: how signal beats whatsapp*, 2016. [Online]. Available: https://theintercept.com/2016/06/22/battle-of-the-secure-messaging-apps-how-signal-beats-whatsapp/ (visited on Jun. 25, 2016).

[15] H. Liu, E. Vasserman, and N. Hopper, "Improved group off-the-record messaging," *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pp. 249–254, 2013.

[16] Y. Lo, *How does facebook make money?* 2016. [Online]. Available: https://www.quora.com/How-does-Facebook-make-money?share=1&redirected_qid=134342 (visited on Apr. 15, 2016).

[17] M. Marlinspike, *Forward secrecy for asynchronous messages*, 2013. [Online]. Available: https://whispersystems.org/blog/asynchronous-security/ (visited on Mar. 31, 2016).

[18] M. Marlinspike, *Private group messaging*, 2014. [Online]. Available: https://whispersystems.org/blog/private-groups/ (visited on Mar. 31, 2016).

[19] M. Marlinspike, *Saying goodbye to encrypted sms/mms*, 2015. [Online]. Available: https://whispersystems.org/blog/goodbye-encrypted-sms/ (visited on Apr. 16, 2016).

[20] M. Marlinspike, *Advanced cryptographic ratcheting*, 2016. [Online]. Available: https://whispersystems.org/blog/advanced-ratcheting/ (visited on Apr. 17, 2016).

[21] M. Marlinspike, *Simplifying otr deniability.* 2016. [Online]. Available: https://whispersystems.org/blog/simplifying-otr-deniability/ (visited on Apr. 17, 2016).

[22] M. Marlinspike, *Whatsapp's signal protocol integration is now complete*, 2016. [Online]. Available: https://whispersystems.org/blog/whatsapp-complete/ (visited on Apr. 17, 2016).

[23] J. C. Perez, *Google defends its use of proprietary tech in hangouts*, 2013. [Online]. Available: http://www.pcworld.com/article/2039820/google-weak-xmpp-support-capabilities-led-us-to-proprietary-tech-in-hangouts.html (visited on Apr. 23, 2016).

[24] Pingdom, *Irc is dead, long live irc*, 2013. [Online]. Available: http://royal.pingdom.com/2012/04/24/irc-is-dead-long-live-irc/ (visited on Apr. 23, 2016).

[25] Statista, *Most popular global mobile messenger apps 2016*, 2016. [Online]. Available: http://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/ (visited on Mar. 22, 2016).

[26] Tails, *Off-the-record (otr) encryption*, n.d. [Online]. Available: https://tails.boum.org/doc/anonymous_internet/pidgin/index.en.html (visited on Mar. 31, 2016).

[27] S. J. Vaughan-Nichols, *Google moves away from the xmpp open-messaging standard*, 2013. [Online]. Available: http://www.zdnet.com/article/google-moves-away-from-the-xmpp-open-messaging-standard/ (visited on Apr. 23, 2016).

[28] Wikipedia, *Telegram (software)*, n.d. [Online]. Available: https://en.wikipedia.org/wiki/Telegram_(software) (visited on Mar. 30, 2016).

[29] Wikipedia, *Threema*, n.d. [Online]. Available: https://en.wikipedia.org/wiki/Threema (visited on Mar. 30, 2016).

Acronyms

AES Advanced Encryption Standard.
API Application Programming Interface.
CA Certificate Authority.
CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart.
DNS Domain Name System.
DOS Denial-of-Service.
E2E encryption End-to-End Encryption.
EFF Electronic Frontier Foundation.
GCM Google Cloud Messaging.
HTML HyperText Markup Language.
ID IDentifier.
IP Internet Protocol.
IRC Internet Relay Chat.
JSON JavaScript Object Notation.
MAUs monthly active users.
MIME Multipurpose Internet Mail Extensions.
MITM Man-in-the-Middle.
NSA National Security Agency.
OTR Off-the-Record.
PGP Pretty Good Privacy.
PKI Public Key Infrastructure.
QR code Quick Response Code.
REST Representational State Transfer.
RSA Rivest-Shamir-Adleman.
S/MIME Secure/Multipurpose Internet Mail Extensions.
SMS Short Message Service.
SPEM Simple Protocol for Encrypted Messaging.
TLS Transport Layer Security.
TTL Time-to-Live.
UI user interface.
URL Uniform Resource Locator.
UTC Coordinated Universal Time.
WOT Web of Trust.
XML Extensible Markup Language.

XMPP Extensible Messaging and Presence Protocol.

XSF XMPP Standards Foundation.