Sandra Holzmann

SEMS for Windows: A Client Implementation for Windows Phone 8 and Security Analysis of the SEMS Protocol

Master's Thesis

Graz University of Technology

Institute for Applied InformationProcessing and Communications
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Assessor: O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch
Supervisor: Dipl.-Ing. Dr.techn. Peter Teufl

Graz, March 2014

Abstract

In the recent years alternative communication opportunities have become increasingly popular as a replacement for SMS (short message service), MMS (multimedia messaging service) and voice calls. These messenger services are nearly free alternatives for text and multimedia messages and in some cases also for voice calls. While the amount of users is increasing rapidly, the providers of mobile messaging applications have paid little attention to the security measures. Therefore, three students of the Graz University of Technology have designed a secure alternative called SEMS. The idea behind SEMS is to provide a secure mobile messaging application with end-to-end encryption for messages. This master thesis aims to analyze the security of the SEMS protocol, and the implementation of an Windows Phone 8 application. The implementation of the Windows Phone 8 client uses the provided API. Therefore, an overview of the structure of SEMS is given and in addition the challenges and implementation for Windows Phone are discussed. The following security analysis addresses the security of the SEMS messenger service. The authentication, the protocol, and the design decisions are analyzed.

Keywords: Mobile Messaging Application, Security Analysis, Secure Messages, SEMS, Windows Phone

Kurzfassung

In den letzten Jahren wurden alternative Kommunikationsmöglichkeiten als Ersatz für SMS (Short Message Service), MMS (Multimedia Messaging Service) und Telefonie immer beliebter. Diese Messenger Dienste sind nahezu kostenfreie alternativen für Text- und Multimedia-Nachrichten und in manchen Fällen auch für Telefonie. Während die Nutzerzahlen rasant steigen, wurde nur wenig Aufmerksamkeit auf die Sicherheit dieser Messenger Anwendungen gelegt. Daher entwarfen drei Studenten der Technischen Universität Graz eine sichere alternative mit dem Namen SEMS. Die Idee von SEMS ist es eine sichere Mobile Messaging Applikation mit Ende-zu-Ende Verschlüsselung für Nachrichten zu bieten. Diese Masterarbeit widmet sich der Sicherheit des SEMS Protokolls und der Implementierung einer Windows Phone 8 Applikation von SEMS. Die Implementierung des Windows Phone 8 Client verwendet die zur Verfügung gestellte API. Dazu wird ein Einblick in die Struktur von SEMS gegeben sowie auf die Herausforderungen und Umsetzung für Windows Phone eingegangen. Die anschließende Sicherheitsanalyse befasst sich mit der Sicherheit des SEMS Messenger Dienstes. Dabei werden die Authentifizierung, das Protokoll und die Designentscheidungen analysiert.

Schlagworte: Mobile Messaging Anwendung, Sicherheitsanalyse, Secure Messages, SEMS, Windows Phone

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Danksagung

An dieser Stelle möchte ich zunächst meinen Eltern danken. Erst durch ihre Unterstützung und Geduld wurde meine schulische und universitäre Ausbildung ermöglicht.

Besonderen Dank gilt auch meinem Betreuer Peter Teufl für die unkomplizierte Betreuung meiner Diplomarbeit.

Des weiteren bedanke ich mich bei Christof Stromberger, Mattias Rauter und Dominik Ziegler für die Unterstützung und Beantwortung aller meiner Fragen.

Zuletzt bedanke ich mich bei meinem Freund Christian, der mich immer geduldig unterstützt hat.

Sandra Holzmann
März 2014

Contents

List of Figures

List of Tables

Listings

1. Introduction

The way how people communicate with each other constantly changes. With the invention of the Internet and e-mail, another milestone was written. Still to this day, e-mail is one of the most common means of communication. Despite the success of e-mail, this was not the end of technical progress. Parallel to e-mail, different approaches have been developed to communicate with people in near real time.
One of the first messaging systems was the Compatible Time-Sharing System (CTSS), a multi-user system developed by the Massachusetts Institute of Technology (MIT) in 1961. The CTSS allowed up to 30 users to log-in at the same time and send messages to each other [1].
In 1982, the Commodore 64 included an Internet service, called Quantum Link or also known as Q-Link, which allowed users to pay a monthly fee to send text messages via modem to other users [2, 3]. The receiver has the option to respond to this messages. In 1991 Quantum Link changed their name to America Online (AOL) [4].
Modern instant messaging programs as they are known today, began to come up in the mid-1990s. One of the first instant messaging programs was ICQ (a homophone for *I seek you*). ICQ was released in June 1996. It was developed by Mirabilis, an Israeli company, which was founded by four students – Yair Goldfinger, Arik Vardi, Sefi Vigiser and Amnon Amir [5]. With this instant messaging program it was possible to use multi-user chats or exchange files. Therefore ICQ enjoyed great popularity. In the following years, other companies developed their own messenger services. e.g. MSN or Yahoo!. Most of these Messenger use a self-developed proprietary protocol.
In 2000, an open standards-based protocol called Jabber was launched. Jabber is an XML-based protocol for instant messaging networks. The Internet Engineering Task Force (IETF) has formed a working group to formalize a standard which was known as Extensible Messaging and Presence Protocol (XMPP) and was based on the Jabber protocol. In 2004 this protocol was raised as a standard [6].
Nowadays, instant messaging services offer, in addition to text messages and usual file transfer, often additional features such as video chat or Voice over IP.

1.1. Mobile Messaging Applications

With the increasing distribution of mobile devices such as smartphones and tablet computers, the interest in free alternatives to common communication protocols including SMS (short message service) or MMS (multimedia messaging service) is growing. In the recent years a new generation of mobile messaging applications for smartphones was introduced. The basic idea behind those communication applications is to provide a better and versatile alternative to common mobile phone messaging applications like an ordinary SMS application. The most widely used mobile messenger is called WhatsApp. WhatsApp was launched in 2009 and is

a cross-platform messaging application for Android, BlackBerry, iOS and Windows Phone [7]. WhatsApp is not the only available mobile messaging application. There are dozen other applications available like Viber, WeChat or Hike. Most of them have a large variety of features.
One advantage of these new messaging applications is that they are easy to configure and afterwards easy to use. Many of them use only the user's phone number for registration. Some Messenger like WeChat provide an alternative log-in with a Facebook account [8]. After registration, the usage is similar to sending an ordinary text message by SMS.
Since many mobile messenger service provider have obviously placed their primary focus on the usability and adding new features, it seems that the security measures have received little attention. Many providers have changed a lot over the last couple of months and years to get rid of these issues, but many problems and attacks are still there.

1.2. Secure Messages (SEMS)

During the lecture Advanced Computer Networks in winter semester 2012/13 the students Christof Stromberger, Mattias Rauter and Dominik Ziegler started a project to develop a secure messenger, called SEMS. It is a cross-platform messaging application for Android, iOS and Windows Phone 8. The Idea behind SEMS is to provide a secure messaging protocol with end-to-end encryption for messages. SEMS uses its own JSON-based messaging protocol and the connection is build up via HTTPS. Additionally to end-to-end message encryption the server itself stores only a minimal set of user information. After registration, each user get its own user identification number which is connected to the registered account information. An account information is a hashed email address of the user. These email hashes are the only "private" user information stored on the server.
The Registration and authentication is based on Google's OAuth 2.0 protocol, no additional authentication mechanisms other than the Google mail address are used. Other authentication methods like Facebook, OpenID or phone numbers will be added in future versions.

1.3. Contribution

The overall contribution of this master thesis consists of two main steps. The first part of this thesis addresses the implementation of the SEMS client for Windows Phone 8. The implementation is based on the provided API of SEMS (see Appendix A). This API was designed by the developers of the SEMS server and iOS client. The second part of this thesis addresses the security analysis of the underlying communication protocol. In detail this master thesis contributes the following:

- A **Windows Phone 8 Client** of the SEMS messenger service.
- A **Security Analysis** of the underlying protocol used by SEMS. We examine the protocol and analyze the possibilities of abuse, as well as general security relevant vulnerabilities. Furthermore, design decisions of SEMS are critical considered in terms of confidentiality of user data.

1.4. Structure of this Document

This introduction Chapter is followed by an overview of security concepts that are used in a secure messenger (Chapter 2). Chapter 3 discusses the related work in the field of mobile messaging applications. This Chapter provides an overview of the currently most used mobile messenger WhatsApp, as well as new upcoming messenger services, pursue the same objectives as SEMS. Chapter 4 gives a description of SEMS, the design concepts, requirements to provide a secure messenger and an outline of the protocol for sending and receiving messages. An overview of the Windows ecosystem is given in Chapter 5. In addition, the differences between those platforms, in case of application development, will be discussed. The implementation of the SEMS client for Windows Phone 8 will be explained in Chapter 6. The topics local encryption, database and limitations of Windows Phone 8 as well as limitations of the implemented SEMS client are discussed. In Chapter 7 the results of the security analysis are described. This covers several areas such as the authentication mechanism, as well as protocol and server related issues. Finally, Chapter 8 presents the conclusion and an outlook of this work.

In Appendix A, a documentation of the used API commands can be found.

2. Secure Messenger

Since many years, Instant Messaging is already been used by a variety of users, to communicate with friends by sending text and multimedia messages in real-time. In recent years, the business world discovered the benefits of Instant Messengers. They are an attractive business tool because many processes can be performed easier, faster and cheaper [9]. However, many popular instant messaging services have been criticized because they have a number of security weaknesses.
This chapter discusses the requirements for a secure messenger. Basic security concepts are discussed and how they can be used for a secure messenger.

2.1. Principles of Information Security

Depending on the application area, there are different requirements are placed on a system. However, it can be said that personal information should be protected. For this purpose, security models for achieving and adherence to information security, and for protection of data can be defined. The most widely used security model is the CIA triad of confidentiality, integrity and availability. These three core principles form the basis for the fulfillment of a secure system [10, 11]. In addition, depending on the field of application and researchers, further properties such as authenticity, safety, privacy, non-repudiation, etc. are defined [11, 12, 13]. Regardless of the classification found in the literature, the individual requirements are defined similarly. For a secure messenger the following principles could be relevant:

- **Confidentiality** – This is the concept of keeping personal information private. Confidentiality is ensured when the information contained in the system are accessible only to authorized persons or systems. If unauthorized persons or systems have access to such information or data, confidentiality can not be ensured.
- **Integrity** – This is the concept of making sure that data is complete and correct. This ensures that stored data or data during transmission were not corrupted.
- **Availability** – This is the concept of making sure that the data is available to all authorized users or systems when they need it.
- **Anonymity** – This is the concept of hiding personal information. Anonymity is assured, if it can not be determined who is sending or receiving the data. A person or system reveals no personal information that could be read by a third party during data transfer.
- **Authenticity** – This is the concept of making sure that the identity of a subject or resource is the identity claimed. Authenticity is assured, by ensuring that information and data are genuine and that the identity of a user or system can be validated.

- **Non-repudiation** – This is the concept of proving that an action has taken place, so that this cannot be repudiated later. The sender of a message can not deny having sent it, as well as the recipient can not deny receiving the message.

In order to assure these requirements, a number of security mechanisms are available. These include encryption methods to ensure confidentiality, as well as signatures or hash functions for fulfilling of integrity.

2.2. Secure Messenger Concepts

The most popular instant messaging services uses little or no security mechanisms to protect user and their personal data. Thus, the users are subjected to eavesdropping and leakage of confidential data. Kim et al. [14] reviewed the security risks of instant messaging services in terms of concept of security, vulnerability, thread and risk, and controls to make it secure. The main focus of their work is placed on the use of instant messengers in enterprise environments. Apart from the business requirements such as control over the transmitted information, also more general problems such as missing message encryption were addressed. Many instant messaging service are vulnerable to eavesdropping because the message is transmitted unencrypted between client and server. In addition, most instant messaging services relay messages through a central server. These messages would be prone to interception at these servers. Especially secure transmission and secure storage of data is an important security aspect for a secure messenger.

2.2.1. Message Transmission

The message transfer, depending on the architecture of the messenger service, can be established between two client applications (peer-to-peer) or via a central server. In both architectures, there exists the risk of interception of the message. This can be for example achieved by a classic man-in-the-middle attack. Using a man-in-the-middle attack, an attacker can intercept the transmitted message and distort them. This kind of attack is a major risk if the attacker is on the same network as the victim. The manipulation of the traffic has an impact on the integrity of the transmitted data. The probability that the Internet traffic is intercepted or even forged is rather low. However, it is possible and should be considered in the planning of a messaging service [10].
In order to ensure confidentiality, integrity and reliability of the underlying communication, the encryption protocol SSL/TLS is used [15]. SSL/TLS was designed to provide a secure connection between two points on the internet. To establish a secure connection, various cipher suites[1]are available. It must be noted, that not all of these cipher suites are considered as secure. In the past, several attacks on SSL and TLS were shown. These include examples such as BEAST [16], CRIME [17] and Lucky Thirteen [18]. Recently, a new attack scenario was proposed by D. J. Bernstein et al., which exploits weaknesses of the RC4 cipher in TLS connections [19]. Therefore, it is recommended to use TLS version 1.2 with AES or Camellia encryption using GCM (Galois/Counter Mode) or CCM (Counter with CBC-MAC) mode [34].

[1] A complete list of cipher suites for TLS can be found at `http://www.iana.org/assignments/tls-parameters/tls-parameters.xml`.

2.2.2. Message Security

The security of message transmission between two clients or between client and server has already been discussed in Section 2.2.1. In this section we consider the security of the message itself. For client-server based messenger architectures where messages relay through a server, it is not enough to ensure a secure communication channel. SSL / TLS provides encryption only between two points. That means that an SSL/TLS encrypted message will be decrypted by the server and then re-encrypted to send the message to the receiver via another secure channel. Thus, the server has the ability to read and capture all incoming messages and perform data mining analysis. Often it is unknown which individuals or institutions have access to the server or how well protected these servers are against attacks by a third party. To ensure, that a sent message can not be read by unauthorized persons or systems, it must be secured with an end-to-end encryption. Such an encryption ensures, that only the person who owns the appropriate key can decrypt the message. With an end-to-end encryption only the message, that has been written by the sender, is encrypted and no protocol or connection relevant data.
In recent years, researchers presented several approaches for messenger systems with end-to-end encryption. One challenge was to ensure a secure protection of the messages and to avoid any significant delay when sending or receiving messages by the use of security mechanisms.

Kikuchi et al. [15] proposed a secure instant messaging protocol in order to hide the message content from the server. On registration, the user communicates with the server to determine a common secret. When a new conversation is created, a fresh random number is generated by the initiator, on which a single message is computed and sent to the server. Then the server modifies the message so that a recipient can perform the Diffie-Hellman key exchange protocol with the initiator's public key. It was also pointed out, that in order to send and receive messages in real-time, a symmetric block cipher would be more suitable for message encryption. The key used to encrypt the message is afterward encrypted with the key obtained by the Diffie-Hellman protocol.

Tung et al. [20] also proposed an approach for a secure messaging protocol, with the name Pandora. As well as the already discussed protocol of Kikuchi et al., Pandora also uses a Diffie-Hellman-based approach. One advantage over the approach of Kikuchi et al. is that there is no modification of the server required to satisfy the protocol requirements and thus it can be used atop of an existing instant messaging service architecture. As an additional security measure they presented a self-data-destructing feature. This feature will delete an encryption key after a specified time. Thus, the encrypted messages can not be decrypted anymore. The protocol uses for message encryption the symmetric encryption algorithm AES and HMAC with SHA-1 for verifying the data integrity and authenticity of a message. For encryption of the AES key, which represents an ephemeral key, OpenPGP RSA with 1024-bit is used. One disadvantage of this method is the overhead, which is the result of generating the ephemeral keys with OpenPGP.

A secure messenger for use in healthcare was presented by Bønes et al. [21]. The proposed MedIMob system is a XMPP based messenger, which uses a hybrid encryption to secure messages. The message encryption uses a one-time session key for AES. Afterwards, this session key is encrypted with the RSA public key of the recipient. In addition to the

proposed MedIMob system, the risks of the use of mobile messengers were analyzed. The use of mobile devices opens new threats compared to messengers on normal personal computer systems. The loss or theft of the device are additional risk factors. To minimize this risks, a messenger should store as little as possible personal information. In addition, these data should only be stored encrypted on the device.

2.2.3. Conclusion

In order to develop a secure messenger, a number of security mechanisms and considerations are necessary. Especially in terms of mobile messaging, as mobile devices are taken everywhere. To protect the communication from third parties, the internet encryption protocol SSL/TLS is used. However, this is not enough to hide the content of messages from the server. Therefore, an additional end-to-end encryption is necessary. In Section 2.2.2 approaches for such an end-to-end encryption were shown.
However, other threats such as loss or theft of the mobile devices as well as the unintentional disclosure of private long-term keys must be taken into account. Encryption of locally stored data might increase security in case of loss or theft, but offers no guarantee that the data are protected from unauthorized persons.

3. Related Work

This chapter presents an excerpt of the most relevant evolution of mobile messaging applications in terms of security. Developers of mobile messaging applications, such as the popular WhatsApp messenger, have placed their primarily focus on usability or new features and less on the security of accounts and confidentiality of the user data. Most of the popular mobile messaging applications have in common, that they do not support any message encryption or only provide weak encryption schemes to protect messages.
The first section contains an overview of WhatsApp, its vulnerabilities and security flaws. An overview of new mobile messaging applications with additional security features, to protect messages and privacy of users, will be given in the following section.

3.1. WhatsApp

The most popular messenger is the WhatsApp mobile messaging application. It is a cross-platform messaging application available for all common smartphone operation systems like Android, BlackBerry, iOS and Windows Phone 8. In August 2013 the vendor reported that on one day over 31 billion messages were sent over WhatsApp [22], growing from 10 billion messages per day in August 2012 [23].

3.1.1. Protocol

WhatsApp uses a customized version of the open standard Extensible Messaging and Presence Protocol[1] (XMPP), a heavily documented XML-based internet messaging protocol for instant messaging.
To reduce the overhead of the standard XML structure, the founder of WhatsApp created a way to express the XMPP syntax using only a few bytes. Therefore they converted common keywords such as a *message* or a *timestamp* tag into a single byte. The modified WhatsApp version, internally called FunXMPP, is not officially documented. A simple example of a XMPP message is shown in Listing 3.1 and the equivalent FunXMPP message in Listing 3.2. For more information about FunXMPP and how it is constructed, we refer the reader to [24].

[1] XMPP Standards Foundation `http://xmpp.org/`

```
<message from="01234567890@s.whatsapp.net"
        id="1234567890-0"
        type="chat"
        timestamp="1383144284">
    <notify xmlns="urn:xmpp:whatsapp"
            name="Alice" />
    <request xmlns="urn:xmpp:receipts" />
    <body>Hello world!</body>
</message>
```

Listing 3.1: A simple example of the XMPP message syntax.

```
\xf8\x0a\x5d \x38 \xfa\xfc\x0b01234567890\x8a
             \x43 \xfc\x0c1234567890-0
             \xa2 \x1b
             \x9d \xfc\x0a1383144284
    \xf8\x03
        \xf8\x05 \x65 \xbd \xae \x61 \xfc\x05Alice
        \xf8\x03 \x83 \xbd\xad
        \xf8\x02 \x16 \xfc\x0bHello world!
```

Listing 3.2: Syntax of a FunXMPP message used by WhatsApp. All keywords are replaced by just one byte. The syntax \xnn represents one byte with the hexadecimal value nn.

3.1.2. Authentication

While classic instant messaging applications like Skype or ICQ identify their user by username and password, WhatsApp followed a more simpler approach for user authentication. To register an account only the current phone number is required.
WhatsApp uses a JID (Jabber ID) and a password to authenticate to the service. The JID is a concatenation of the users country code and mobile pone number (`[country code][phone number]@s.whatsapp.net`). The password is generated on the server and received on the client upon registration. Here, an SMS with a verification code is sent back to the user. This received verification code must be verified within the application. Upon successful verification, the server sends the password to the client. The received password is never visible to the user.
Until the end of 2012 the password was generated using an MD5 hash of the phone's IMEI number on Android. On iOS they used the MAC address of the WiFi-interface and for Windows Phone (7) the DeviceUniqueId. However, this was fixed because IMEI numbers or interface data can be easily obtained and thus did not offer an appropriate level of protection. After registering the phone number all contacts are uploaded to the WhatsApp server and compared with the contact data stored on the server. The WhatsApp client receives as reply a subset of all contacts that have an WhatsApp account registered. This procedure is done automatically, so that the user has no control, what data are uploaded to the server.

3.1.3. Security

WhatsApp was first published in 2009 for iOS. One of the first vulnerabilities was reported in May 2011 when hackers found a security hole that left user accounts open for session hijacking [25]. In recent years, researchers found a variety of security vulnerabilities that exist in the widely used and popular WhatsApp messenger. Some of them have been fixed by the company while others still pose a risk.

Account Hijacking and Free SMS Messages

In [26] the authors analyzed the security of popular smartphone messaging applications, including WhatsApp. They discovered that is was possible to hijack accounts during authentication using a man-in-the-middle attack. As prevention against account hijacking, WhatsApp uses a verification SMS containing a PIN during the registration process. The user has to copy that PIN into the application to finish the registration process. However, this PIN was generated on the user's mobile phone and then sent over an HTTPS connection to the WhatsApp server. In the next step the WhatsApp server sends an SMS message, containing the verification PIN, over an SMS proxy server back to the user's mobile phone. An attacker could easily perform a man-in-the-middle attack to hijack any WhatsApp account by using the victim's phone number and intercepting the communication between the messenger client and the server. In addition, this authentication mechanism could be misused in order to send text messages worldwide for free. As already mentioned, the authentication PIN was generated on the mobile phone and the WhatsApp server sends it back to to the mobile device by an SMS message. It was possible to intercept, and modify the transmission of that authentication code. The modified text was still delivered, so there was the possibility to send almost any text string.
Both vulnerabilities were fixed by the WhatsApp developers after they became public.

Password Security

As authentication mechanism WhatsApp requires the phone number only. As a result, the user needs no additional username or password. The user's phone number is linked to an internal JID (Jabber ID) and a password. Both, the JID and the password are never visible to the user. Sam Granger [27] published a blog post on how passwords are generated on Android devices. The client uses the Unique Device Identification (IMEI) number to generate a password. In order to generate the password, the MD5 hash of the reversed IMEI number was used without additional salt.

```
md5(strrev('imei'))
```

An attacker can easily develop an application that sends the victim's IMEI and phone number to his server in an background task. Many Android Apps found in the official store read the IMEI number and send it to an external server. This shows that an IMEI number is not a secret information, because it is easily obtainable. Therefore, an IMEI number should not be used for authentication.

A few days later Ezio Amodio [28] published in his blog a documentation on how passwords are generated on Apple iOS devices. In fact, that Apple does not allow third-party applications the access to the IMEI number, WhatsApp uses the mac address of the WiFi interface `en0`. To generate the password the MD5 hash was calculated using the mac address of interface `en0` taken twice.

```
md5(AA:BB:CC:DD:EE:FFAA:BB:CC:DD:EE:FF)
```

A similar principle was used on Windows Phone 7 too. On Windows Phone devices they used the reversed MD5 hash of the Unique Device ID [29].
Similar to Android, it is not difficult to read-out these values on iOS and Windows Phone devices. Especially the network interface used on iOS is visible to everyone located in the same WiFi network.
After becoming aware of this vulnerability, WhatsApp reacted and changed the password generation mechanism. From now passwords are not generated locally anymore. Instead, the server now generates the password during the first registration process and sends it to the client after a successful verification of the user.
Following these changes it was still possible with a man-in-the-middle attack to sniff the password, because WhatsApp was not verifying the server certificate [30]. However, this security flaw has already been fixed by the company.

Message Encryption

Another security vulnerability on WhatsApp is related to message encryption. Until August 2012, messages were not encrypted at all. In the case of using WhatsApp in an unprotected WiFi network, anybody was able to read the message transfer. However, there is no official information on how WhatsApp encrypts the data. The company stated in a Frequently Asked Questions (FAQ) entry on their website, that messages are now encrypted [31]. Researchers found out, that they used the stream cipher RC4 to encrypt messages. The stream cipher RC4 has some weaknesses and should not be used anymore. In 2001, Adi Shamir et al. [32] demonstrated that the first N bytes of an RC4 key stream have a predictable structure. One recommendation was to discard the first bytes of the key stream. However, new results of research showed more correlations between the keystream and the key [33]. In October 2013, the ENISA (European Union Agency for Network and Information Security) published recommendations for security algorithms and key sizes [34]. It was noted that the stream cipher RC4 should be replaced, since this cipher is, by modern standards, a weak stream cipher.

Apart from the weaknesses of the RC4 algorithm, another mistake was made in the implementation of the message encryption in WhatsApp. In October 2013, Thijs Alkemade analyzed the encryption mechanism used by WhatsApp [35, 36]. He came up to the conclusion that the encryption of all previous WhatsApp messages must be considered compromised. Alkemade has found out, that the same key was used in both directions of communication. When two messages are encrypted with the same RC4 key, then the encrypted messages can be compared. To obtain the cipher text the message plaintext is xor'ed with the RC4 key stream. By xor'ing the ciphertext with the same key stream, the message is decrypted. Using the same key in both directions, the key stream can be canceled out, since the following applies:

$$(A \oplus X) \oplus (B \oplus X) = (A \oplus B)$$

Since WhatsApp uses the same key in both directions, xor'ing the incoming string with the outgoing string, this yields to the same result as xor'ing both plaintext messages. By xor'ing this with either of the plaintext bytes, the other bytes of the unknown plaintext message can be calculated. This does not directly reveal all bytes of a message but it should be noted that some parts of a message are predictable. These parts can be headers or other parts of the protocol. For example every stanza of the WhatsApp protocol starts with `0xf8`. Even if some bytes are not fully known, assumptions about the value of a specific byte can be made. Sometimes it can be known that such a byte must be alphanumeric or an integer within a specific range. Due to these facts, assumptions can be made about the other byte of the unknown plaintext message.
Therefore it must be assumed that anyone who is able to eavesdrop a WhatsApp connection, is capable of decrypting the transmitted messages.

This was just a short excerpt from a wide variety of vulnerabilities that have occurred in WhatsApp and some are still exploitable.

3.2. New Generation of Mobile Messaging Applications

Since the huge success of WhatsApp a variety of other mobile messenger applications was published. Nearly all of them have similar security flaws or missing encryption. Over the last months, a number of mobile messengers was published with the primary aim of security. Examples of mobile messenger using encryption are Threema, MyEnigma and whistle.im.

3.2.1. Threema

Threema[2] is a mobile messaging application, available for Android and iOS. According to the developers there are no plans at this time for supporting additional mobile platforms such as Windows Phone or Blackberry. Threema is developed by Kasper Systems GmbH in Switzerland. The name is derived from the acronym EEEMA, the short form of End-to-End Encrypted Messaging Application. To create a Threema user account an email address or a phone number is required.
In the current version, only the iOS (version 1.6.7) application supports a group chat with up to 20 group members. There is currently no group chat functionality available for Android (version 1.13) devices. A group chat on Android was announced but at this time, no specific date is known.
The messenger uses Elliptic Curve Cryptography (ECC) with the NaCl Cryptography Library[3] to protect messages between sender and receiver, as well as to encrypt the communication between application and server. According to the company two layers of encryption are used:

- An end-to-end layer between conversation participants.

[2] Threema - Seriously secure mobile messaging. `https://threema.ch/`
[3] NaCl: Networking and Cryptography library `http://nacl.cr.yp.to/`

- A layer to ensure a secure transport between application and server.

Thereby, the user has control over the key exchange. This ensures that no third-party can decrypt the content of the messages. Also, the server operators themselves should not have access to message content, according to the developers.
An ECC-based encryption with a strength of 255 bits is used to secure the data. Elliptic curve Diffie-Hellman in conjunction with a hash function and a nonce is used, to derive a unique symmetric key with 256 bits length for message encryption. To encrypt a message the stream cipher Xsalsa20 is used. In addition, a 128 bit long message authentication code (MAC) is added.
Also, the data stored on the mobile device are encrypted according to the manufacturer's FAQ page on both platforms.

As an additional feature the messenger provides a so called verification level. This three levels (red, yellow and green) indicates the confidentiality of the communication partner. Red indicates that the conversation partner is still unknown because no matching contact was found in the address bock. So we cannot be sure that the person is who he or she claims to be. With a yellow color the server confirms the identity of the communication partner. In addition Threema offers the option to verify a contact by scanning a QR code directly from the mobile phone of another user. Through this process the verification level will change to green.

3.2.2. MyEnigma

MyEnigma[4] is an instant messenger for smartphones and available for Android, Blackberry and iOS. Like Threema it offers a end-to-end encryption for communication. The messenger is developed by Qnective Media Relations located in Switzerland. To register an account an email address and a phone number is required.
The developer provides not much information about the used encryption mechanisms. However, the company promises on its website the following points:

- End-to-end encryption based on highly secure mechanisms and open standards implemented by governments and private organizations.
- Double-layer encryption technology to protect messages and the communication with the server.
- Verification of user accounts by means of two independent channels, preventing from account impersonation.
- MyEnigma staff cannot decrypt or read messages.

The developers of MyEnigma published no detailed information about the used encryption algorithms, and the security of the messenger service was so far no field of research. Therefore, no statement about the security of the service can be made.

[4]MyEnigma - The one and only secure messaging App. `http://www.myenigma.com/`

3.2.3. whistle.im

whistle.im[5] is a mobile messaging application for smartphones. Currently, only a beta version for Android is available. In addition, a browser version[6] is available for desktop computers. A support for iOS and Windows Phone will follow soon. whistle.im is a project of two students from Germany. So far, only sending text messages is possible. Multimedia features such as images or videos are currently not available.
whistle.im uses RSA 2048 and AES 256 in CBC mode to secure messages. Figure 3.1 shows a general overview of the communication between two parties over the central server. The cryptography implementation of whistle.im is open source and can be examined by everyone (see [37]).

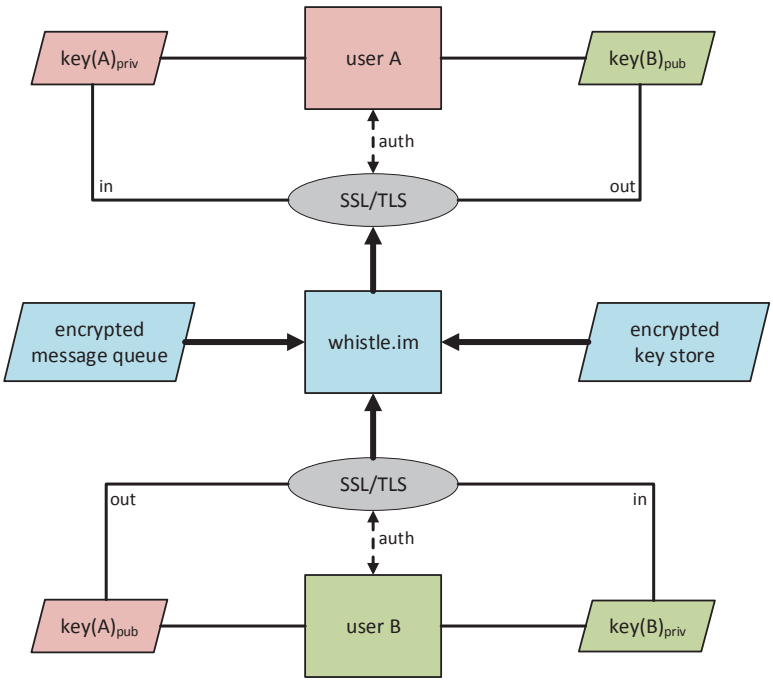

Figure 3.1.: Overview of an encrypted communication on whistle.im [38].

The whistle.im server also acts as a key server on which the RSA keys of all users are stored. Every client must send a request to the server to get the required keys.

As already mentioned before, whistle.im is currently in beta. A few months ago an analysis was made by the hacker Nexus from the Chaos Computer Club Hanover[7]. However, soon after the release of the first beta version he revealed serious shortcomings [39]. As an example, private keys of the users are stored on the server of the service provider. This is done automatically and a user has no possibility to influence it. Although the private keys are encrypted, but if the server provider or an attacker gains knowledge of the user's password, the communication can be decrypted. The developers reacted and made the private key

[5] whistle.im - Privacy for everyone. `https://whistle.im/`
[6] whistle.im browser version `https://whistle.im/browser#login`
[7] Chaos computer Club Hanover `http://hannover.ccc.de/`

upload functionality optional, so that users can decide for themselves whether they want to upload and backup their private keys on the server.

The client connects via HTTPS to the server `switch.whistle.im`, thereby the SSL/TLS certificate was not checked for validity. Nexus remarked that any self-signed certificate was accepted. In addition, no verification of received public keys was made, so an attacker or the service provider itself could insert an arbitrary key by replacing the original one. The client does not notice this forgery. Thus, it can not be sure that a public key received from the server belongs to the message receiver. Performing a man-in-the-middle attack it would be possible to read the entire communication. Also, this security flaw was fixed, according to the developers. From now the server certificate is included in the client software and compared with the certificate received from the server when establishing a connection. In addition, the developers also wanted to take care of the public key distribution problem [40].

It was also criticized that the server stores all messages and opens the possibility to read messages later if the knowledge of the decryption key was obtained. Furthermore, there was no possibility to replace a compromised key. All keys are transferred only once during the initial registration process.

4. SEMS

This chapter discusses the SEMS application, differences and similarities compared to already in Chapter 3 discussed messengers. In addition, the design concepts, as well as the used protocol are explained in detail.

4.1. SEMS for Secure Messages

Messenger for mobile platforms such as smartphones and tablet computers are becoming increasingly popular, as the successful WhatsApp Messenger has shown in the recent years. WhatsApp, and many other Messenger for mobile devices rely on ease of use. Examples are the registration by phone number, so that the user has to remember no addition credentials, or the automatic identification of friends who also have an registered account. Today's messenger offer also a wide range of additional multimedia features like video chat or Voice over IP. The confidentiality of information and the privacy of the user has been given little attention. For users who have placed more importance on privacy and confidentiality of the transmitted data, there were no real alternatives. Thus, three students decide to develop a messenger that puts safety first. The project SEMS was born.
SEMS is an acronym for Secure Messages and offers a messenger service with encrypted message transmission between sender and receiver. In addition to the encryption of messages, the privacy of the user is also an important aspect. Thus, only anonymous user data should be used for communication. This should ensure, that only the person who knows the communication partners, can also identify those.

4.1.1. SEMS in Comparison to other Messengers

In the past year and recent months, much has changed in the market of mobile messenger applications. Since the start of the SEMS project, also other messengers have raised, who also have their primary focus on security. Table 4.1 shows an overview of already in Chapter 3 discussed messengers, compared to SEMS. In this table only the core features of a (secure) mobile messenger are illustrated. It should be noted, that the messenger whistle.im is currently in beta. SEMS, however, is in early development stages. Therefore, not all features that are planned have already been implemented. For instance whistle.im and SEMS do not support multimedia messaging features such as sending an image. The messengers Threema and MyEnigma have been already published but are not yet available for all mobile platforms.

The first step, in order to use a mobile messaging service is the registration process. Registration and authentication should be simple, but also provide a high level of protection of personal data to protect them from unauthorized persons.

	Group Chat	Multimedia Messages	Registration
WhatsApp	yes[1]	yes	Phone number
Threema	yes[2]	yes	Phone number or e-mail
whistle.im	yes	no	whistle ID and password
MyEnigma	yes[3]	yes	Phone number, e-mail and password
SEMS	yes	no	OAuth / Google Account

	SSL/TLS	End-to-End Encryption	Addressbook Access
WhatsApp	yes	no	yes
Threema	yes	yes	no[4]
whistle.im	yes	yes	no
MyEnigma	yes	yes	yes
SEMS	yes	yes	yes

[1] Up to 50 group members.
[2] Up to 20 group members and available for iOS only.
[3] Up to 30 group members.
[4] Optional Feature.

Table 4.1.: Overview of selected mobile messaging applications, their required information for registering an account and supported features.

Large differences can be observed in the way in how users can register an account for the different services. Threema for example, uses the phone number or the e-mail address of the user, in order to link this user data to a server generated Threema ID. However, users are responsible to make backup copies of their Threema ID, because they can not be recovered. In case of loss, the user has to create a new account. In comparison, by registering a whistle.im account, the user can choose his or her own whistle ID and password. Registering an WhatsApp account only requires the user's phone number, as already explained in Section 3.1.2.
SEMS uses for registration and authentication the OAuth 2.0 protocol. In order to register, a valid Google account is required. This type of authentication has the advantage that the log-in credentials are managed separately from the messenger service, by a third, independent party. For users, this system has also the advantage that they do not have to create a new account, if they already have a supported account. Especially when several OAuth service providers are supported. Currently, only Google's OAuth service is available for SEMS, but other service providers will follow. Also a registration using the user's phone number, similar to WhatsApp is planned.

In Chapter 2, the properties have already been discussed, which are necessary to provide a secure messenger. In order to prevent that a communication can be read by third parties, the internet encryption protocol SSL/TLS is used. All messenger services listed in Table

4.1 use an encrypted connection. However, this is not enough to keep a message private. The content of a message can still be read by the server provider. To keep these messages confidential, an end-to-end encryption must be used. Such an encryption is used by the messengers Threema, whistle.im, MyEnigma and SEMS. Only WhatsApp does not provide end-to-end encryption of messages.

The management of contacts works, except whistle.im, similar in all messenger applications. An advantage of using the address book is that no own buddy lists have to be created by the user. This method has both its advantages and disadvantages. In order to use WhatsApp, access to the phone's address book must be granted. All stored phone numbers are then uploaded to the server and compared with the server-side existing data. In other words, private information of the user's address book, are uploaded to an external server over the internet. A similar approach is also used by MyEnigma and SEMS. According to the developers of MyEnigma, only anonymous hash values of the phone number are sent to the server [41].
An automatic synchronization of all contacts of the phone's address book, does not exist for SEMS. Here, the hashed e-mail addresses are sent to the server to obtain the corresponding user ID, when a new conversation is created. Only the hashed e-mail addresses are transmitted, which have been specified as recipients of the created message. Nevertheless, for this approach, the access to the phone's address book must be granted.
The transmission of address book data must be considered as very critical, since personal information is passed through the internet and probably be stored on an external server. Often, it is completely unclear what data are transmitted and stored on the server of the message service provider. Therefore, Threema offers the users the choice of whether they want to automatically synchronize their contacts. If the automatic synchronization is enabled, the client sends the hash values of the phone number or e-mail address, which are stored in the phone's address book, to the server. The server compares the hash values with the stored data in its database, and sends the found Threema IDs back to the client. Alternatively, contacts can also be added manually. This is done by stating the Threema ID or scanning of a QR code of the respective user.

4.1.2. Supported Platforms

For a messenger service, it is important that as many platforms as possible are supported. Object of this master thesis is to develop a messenger for the mobile platforms of the Windows ecosystem. The effort to support a client for Windows Phone 8 and Windows RT was difficult to estimate at the beginning of this thesis. During development, it turns out that the development of both platforms would exceed the scope of this thesis. Therefore, we decided to support only Windows Phone 8.
Besides of the development of an Windows Phone 8 client in this master thesis, SEMS is also developed for Apple's iOS and Google's Android. A version for BlackBerry is currently not planned.

4.2. Requirements

To develop a messenger which protects the privacy of individual users, a number of different requirements are necessary. It must be ensured that the message can only be read by the recipient. In addition, it must also be ensured that the content of a message can not be forged on the transmission between sender and receiver.
Another challenge is the protection of user data. Only anonymous user identities should be used for the communication, to protect the real user identities against possible espionage, so that an eavesdropper obtains no valuable information. The server should also store no personal data such as names, e-mail addresses or phone numbers.
Nevertheless, the application should be easy to use and offer no disadvantages compared to other messengers.

4.3. Design Concepts

For the implementation of the SEMS Messenger, a number of different encryption mechanisms, protocols and standards were used, based on the requirements described in Section 4.2, which are considered in more detail in the following Sections.

4.3.1. Authentication

When using the SEMS client, a user is confronted first with the registration process or the authentication against the server. For authentication of users, SEMS uses the Google OAuth 2.0[1] service. In our case, the SEMS server requests the access to the user's e-mail address from the Google API.
Using OAuth offers several advantages for users and for the development of SEMS. For registration or authentication, no personal information of the user is required. An exception might be the user's e-mail address, which is required by the SEMS Server to calculate the hash value to generate the required account information. The password used to log-in, is not communicated to the SEMS server. Conversely, less data must be stored in the database of the SEMS server. In addition, it simplifies the development of the service, because no own authentication mechanism needs to be implemented. For example, no support for password renewal, forgotten password or authentication of users need to be implemented.

In order to authenticate a user against the SEMS service, a number of steps are required as shown in Figure 4.1. When the application gets started and the Google service was selected for authentication, the user is redirected to the log-in page of Google. After entering the user credentials, the user is prompted to grant the SEMS application access to the Google API. Figure 4.2 shows the granting page, to grant SEMS the access to the API provided by Google.
After the user has granted access, the Google server replies with an authorization code, which the application can exchange for an access token. Once the SEMS server has obtained the access token, the server can use this access token to access the Google API and query the

[1] Detailed information about using OAuth 2.0 to access the Google API – `https://developers.google.com/accounts/docs/OAuth2?hl=de`

approved user data. In our case, the server configuration allows to query this data only as long as the obtained access token is valid. Normally, such an access token is valid for 3600 seconds. Because the server does not receive a refresh token, no new access token can be issued. In addition, users will have to grant the access of SEMS to the Google API at every log-in.
The obtained e-mail address is used to calculate the account information which is necessary for the communication protocol of SEMS. No other personal information of a user are stored.

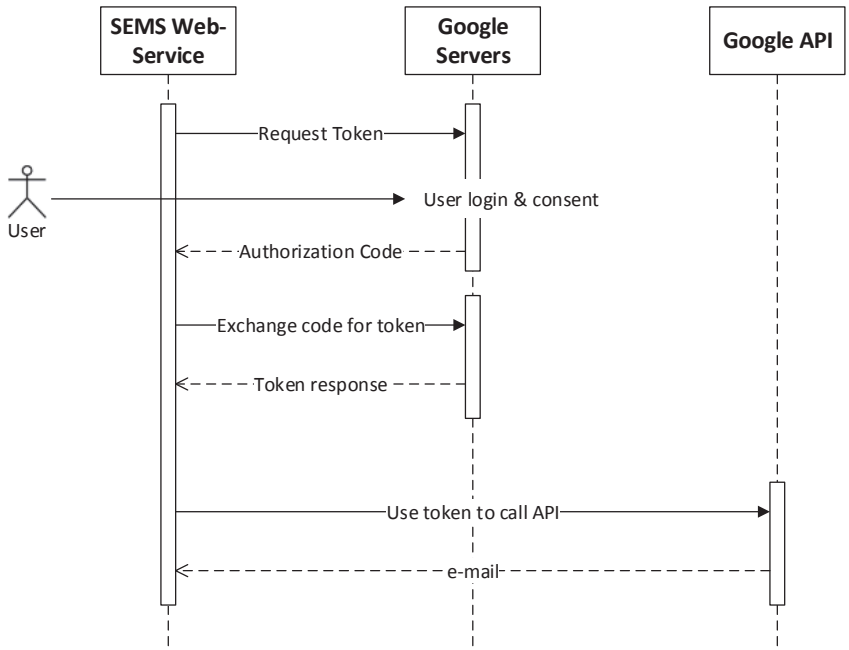

Figure 4.1.: Illustration of the SEMS authentication process using the Google OAuth 2.0 protocol.

Currently SEMS supports only an authentication via Google OAuth 2.0. Additional OAuth providers such as Facebook and Twitter or an authentication via OpenID or the user's phone number are planned for future versions.

4.3.2. Message Security

Messages are always transmitted as encrypted strings. Such a message string is a concatenation of several components:

- AES Key Wrap: $E_{RSA}(K_{AES})$
- Message Signature: $S_{RSA}(SHA_{256}(IV||E_{AES}(message)))$
- Initialization Vector: IV

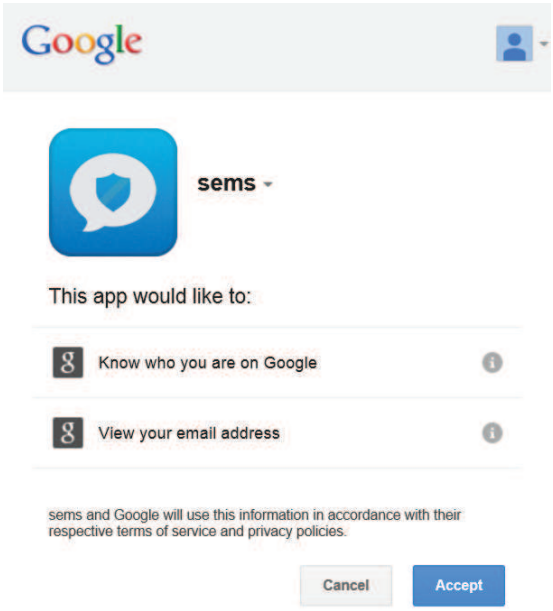

Figure 4.2.: Example of Google's granting page to grant an application access to the Google API.

- Encrypted Message: $E_{AES}(message)$

The concatenation of these four components results in the following message string:

$$E_{RSA}(K_{AES})||S_{RSA}(SHA_{256}(IV||E_{AES}(message)))||IV||E_{AES}(message)$$

AES Key Wrap The AES key wrap is found at the beginning of the message string and contains an AES key, encrypted with RSA. To encrypt the AES key, the public key of the recipient is required. For the RSA key pair SEMS uses a length of 2048 bits and PKCS#1 v1.5 padding.

Message Signature The message signature is used to ensure the integrity of the sent data. The hash value for the signature is generated from the concatenation of an initialization vector and the AES encrypted message. As a hash function SHA_{256} was chosen. Using this hash value and the private signature key of the sender, the message signature can be generated. Just like the encryption keypair, SEMS also uses a key length of 2048 bits and PKCS#1 v1.5 padding for generating an RSA signature keypair.

Initialization Vector The initialization vector, as well as the AES key are generated anew for each message. Thus no two messages are encrypted with the same key an initialization vector. As a result it can be ensured that if an AES key was leaked, a third party can not decrypt the entire communication.

Encrypted Message The plaintext message is encrypted using an AES key and a specified initialization vector. The encryption used by SEMS is based on AES 128 bits in CBC mode and PKCS#7 padding. As already mentioned, a new key and a new AES initialization vector are generated for each message.

4.3.3. Privacy

SEMS should know as little as possible about the registered users. Therefore, only a minimal set of anonymous user data is stored on the SEMS server. For authentication, as already explained in Section 4.3.1, the OAuth protocol is used. Thus, the server does not know any user passwords. The only information that is used to identify a user is the user's e-mail address. The e-mail address is obtained via Google API and is stored on the server only as a hashed value. For this purpose, the SHA_{256} hash of the e-mail address is calculated.

Text messages, stored on the server, are always secured with an asymmetric encryption scheme. This means that only a person who has the private key, can decrypt the message. Owner of such a private key is only the recipient of this message. Therefore, the server can not decrypt the stored messages and thus not read the contents of the message.
On the client side, the sender's name of a message can be displayed only, when the receiver knows that person. This means that he or she needs to know the receiver's e-mail address, in order to associate the sender's user ID with his or her name.

4.4. The Protocol

The SEMS messenger is using a self-designed communication protocol. All Clients communicate via a central server using an HTTPS connection to `sems.iaik.tugraz.at/sems/`. Thereby, all client requests and server responses are stated as JSON strings. Furthermore, applies to the server responses that only a minimum set of return codes are used. If the server was able to process the incoming data correctly, or had found the requested data in its database, it responds with the required data and the return code *200 OK*. For the case, that the specified session is no longer valid, the server responds with the error code *401 Unauthorized*. Otherwise, the server responds with *403 Forbidden*.

In the following two Sections, we consider the communication process between client and server when sending or receiving a message.

4.4.1. Sending Messages

When a message is sent with SEMS, this message is not sent directly to the recipient. A message is always sent to the server and afterward the recipient can download this messages from the server. The structure of such a message is shown in Listing 4.1.
The first parameter represents the own session. This session ID is specified for each request. The server uses this session ID to identify the user who has sent the request. With the *to*

parameter, the user ID of the recipient is specified. In addition, a server-generated conversation ID is required. The *preview* parameter is optional and can be used for push notifications.

```
{
    "session":"41062242-7896-48df-b79d-2a439371e960",
    "to":"66e2e95f-c578-4075-9aab-584ed878f8b1",
    "message":"SGFsbG8gU2Vtcw==",
    "conversation":"ivnu3q5qvf8g@sems-conv.com",
    "preview":"MessagePreview"
}
```

Listing 4.1: JSON string of an outgoing message request, used by the SEMS protocol.

The sequence of API calls is shown in Figure 4.3, that are required to send a message with SEMS. Parameters which are sent from the client to the server are shown in blue. Server responses are shown in green. This diagram represents a rough outline of the API calls and can deviate from the implementation of the different operating system versions. It is merely intended to give an outline of the protocol used by SEMS.

A basic distinction is made between two cases. In the first case a new conversation is created. The second case covers the situation when a user replies on an already, on the client existing conversation and thus the required data is already stored in the client's database.
When a new conversation is created, the receivers are specified within the application by the user. Then the hash value of the e-mail address is calculated and sent with the API command `usr/getUser` to the server to obtain the corresponding user ID. This user ID is required to specify the recipient of a message (see Listing 4.1 at parameter *to*). This API call has to be done for each specified recipient. In addition to the User IDs also a conversation ID is required. If no corresponding conversation ID, for the current user IDs, is stored in the client's database, the server is queried with `msg/newConversation`. For the case that the user responds to a conversation that is already available on the client, these two just mentioned API calls are not performed. To query the server is not necessary because the required data is already stored in the client's database.
Once the user IDs of the recipients are known, the public keys can be queried from the server, in order to encrypt the message for each individual recipient. Because SEMS provides an end-to-end encryption, the message must be encrypted for each recipient separately. With the command `key/getPublicKeys` the public keys are downloaded from the server for the specified user. The received encryption key is then used to, as already explained in Section 4.3.2, to encrypt the message. Thereafter the massage is sent to the server using the command `msg/sendMessage` and is available for download.
For the public key of the recipient it must be noted, that this value has to be requested for each message, since these public keys may change.

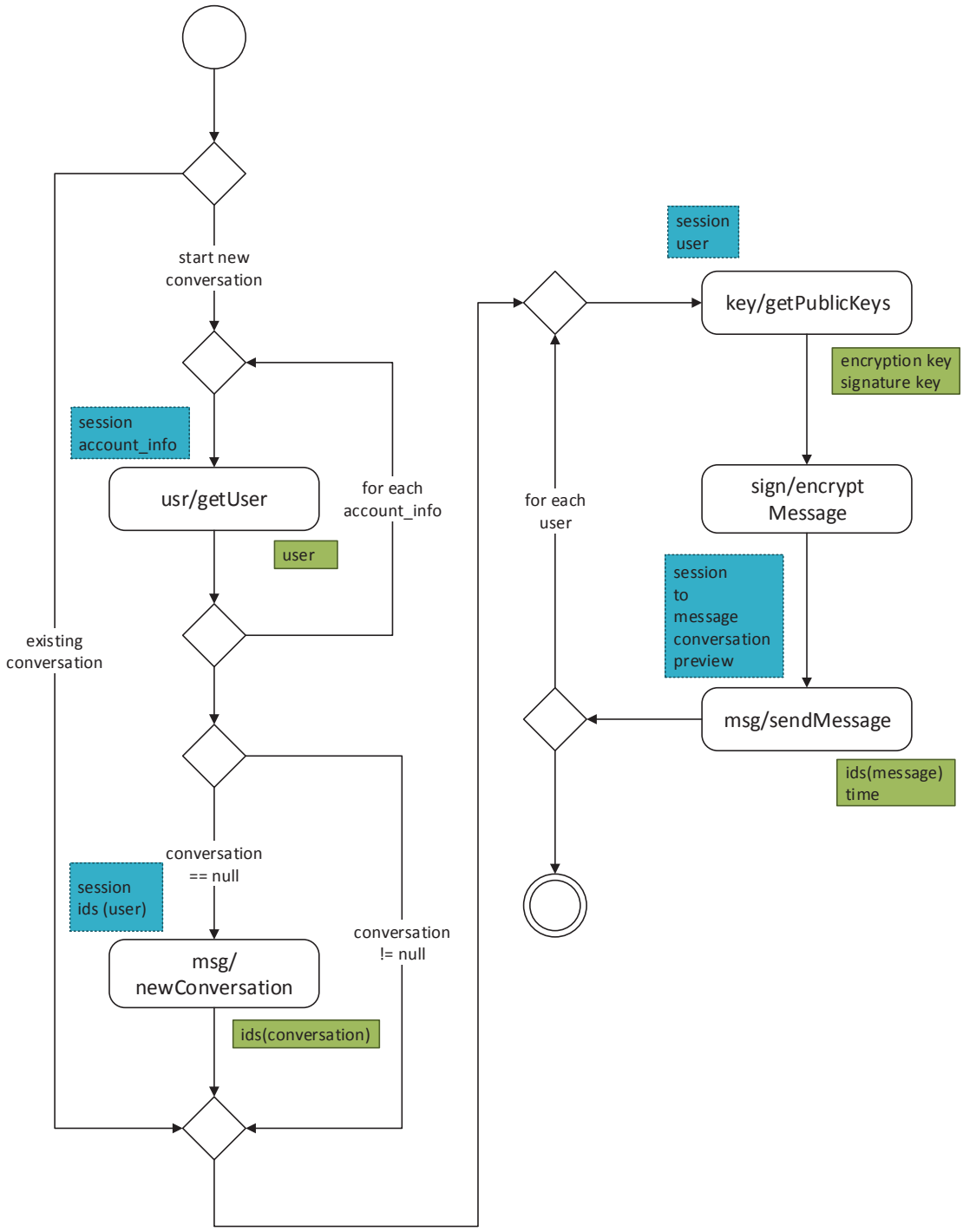

Figure 4.3.: Illustration of the used API calls and the resulting information flow when sending a text message with SEMS.

4.4.2. Receiving Messages

The process to receive messages using SEMS is graphically illustrated in Figure 4.4. Again, a blue parameter represents a client request and a green parameter represents a server response. As with the previous diagram for sending messages, also applies to this diagram that the sequence of API calls can vary by platform dependent implementation, because this diagram is a simplified illustration to obtain an overview of the receiving messages process.

Before a message can be received, the client has to query the server if new messages are available. With the command `msg/getMessageIDS` and the contained *since* parameter, the client specifies which messages are relevant for downloading. The server response includes all messages that match the specified timestamp, which means that the timestamp of the message must be greater than or equal to the specified value. This server response contains a list of message IDs with the associated conversation IDs.
If the conversation ID contained in the server response is not yet known, the client queries the server with `usr/getUserIDsConversation` to obtain all user IDs which belongs to this conversation. As response, the server returns the assigned user IDs. Are these user IDs also unknown, the client queries the server with `usr/getAccountInfo` to get the associated account information for each unknown user. The obtained account information are the hashed e-mail addresses of the user. Thus, the client can compare its address bock with the response, received from the server. If this account information can not be assigned to a contact in the address book, then the client shows "Unknown" instead of the user name. If the conversation ID already exists in the client's database, these two API calls are not executed.
With the obtained message IDs, each message can be downloaded from the server with the `msg/gestMessage` command. The structure of a message receive from the server is shown in Listing 4.2. In order to verify the integrity of the received message, the public signature key of the sender is required. This is again downloaded with the `key/getPublicKeys` command from the server. After a successful verification of the message, it can be decrypted with the private key.

```
{
    "message":"SGFsbG8gU2Vtcw==",
    "from":"f3045940-47fd-4c10-80d4-158024fca262",
    "time":"1360867257000",
    "conversation":"ivnu3q5qvf8g@sems-conv.com",
    "read":"false"
}
```

Listing 4.2: JSON string of an incoming message response, used by the SEMS protocol.

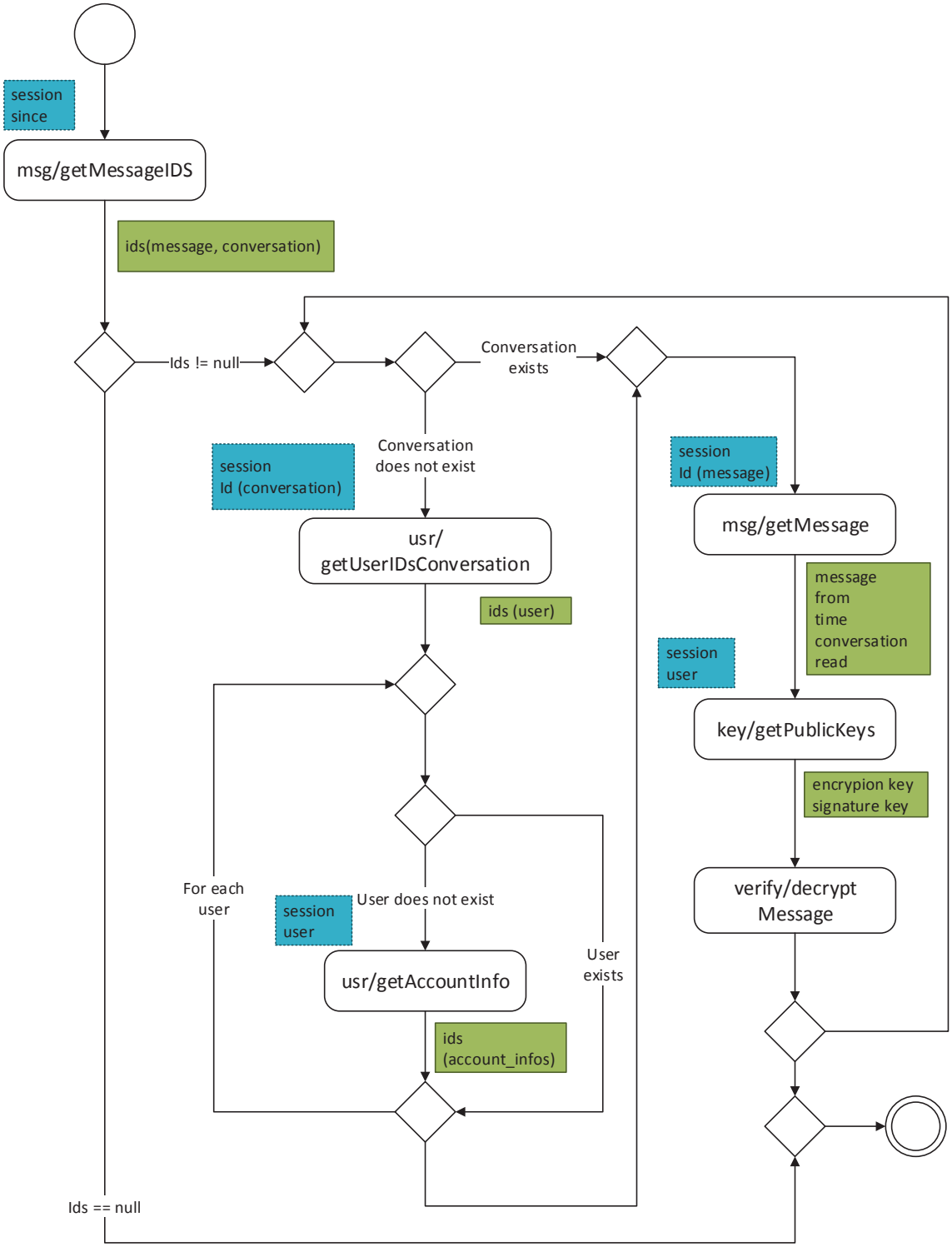

Figure 4.4.: Illustration of the used API calls and the resulting information flow when receiving a text message from the SEMS server.

5. App Development for Windows Platforms

In this chapter, we give a general overview of the Windows ecosystem in terms of the different platforms, Windows Phone 8 and Windows RT. In addition, we give an overview of the difference between these two platforms with respect to application development.

5.1. Windows Phone 8

Windows Phone 8 is an operating system from Microsoft, which was developed for smartphones. Compared to Windows Phone 7, Windows Phone 8 is no longer based on Windows CE. Instead, Windows Phone 8 uses the same Windows NT kernel as the desktop version. It was released in October 2012.
Windows Phone 8 is a closed system like iOS. Therefore, it is only possible to install apps from the official store. Also, it is not possible to directly access the file system.

5.2. Windows RT

Windows RT is an operating system for mobile devices that use an ARM architecture. Also, it uses, as well as Windows Phone 8, the same Windows NT kernel as the desktop version of Windows 8. The appearance of Windows RT is similar to Windows 8 for desktop computers. However, Windows RT allows only an installation of apps from the official Windows Store. Windows RT was released in October 2012 and can be purchased only in combination with a device.

5.3. Platform Differences in Case of App Development

With Windows 8, Microsoft has introduced three new platforms, Windows 8 for desktop computers, Windows Phone 8 for smartphones and Windows RT for ARM devices such as tablet computers. For all these platforms, Microsoft also provides an official store for applications. However, a distinction is only be made between apps for Windows Phone 8 and Windows 8 Store Apps. Windows 8 Store Apps can be installed on desktop computers, as well as on Windows RT devices.
Although all three platforms are based on the same Windows NT kernel, there are some significant differences in the API, the supported programming languages and the design guidelines from Microsoft. Table 5.1 shows an overview of programming languages supported by Windows Phone and Windows 8 (Store Apps).

	.NET (C#, VB.NET)	C++	JavaScript
Windows Phone 8	yes	yes	no
Windows 8	yes	yes	yes

Table 5.1.: Overview of the supported programming languages on Windows Phone and Windows 8 (Store Apps).

The API can be basically classified into three components, the native (Win 32 and COM) API, the Windows Runtime and the .NET API. Windows Phone and Windows (Store Apps) share the same .NET framework engine. However, not all namespaces are represented in both platforms. The native API and the Windows Runtime for both platforms share only a small part of the API [42].

In the following paragraphs, examples of differences between the platforms are pointed out. These differences also affect the development of SEMS for Windows platforms.

User Interface Both platforms, Windows Phone 8 and Windows 8 (Store Apps) are using the Extensible Application Markup Language (XAML) for declaring an user interface. Typically a XAML file in an Windows Phone or Store App project represents an user interface page. Although for both platforms the same markup language is used, different namespaces for Phone and Store Apps are used. The namespace for Windows Phone is called `System.Windows` and for Windows 8 (Store Apps), the namespace `Windows.UI.Xaml` is used. In addition, some elements of the user interface have different names or are missing on one platform. Missing elements can be found in the Windows Phone Toolkit[1], but with different names. Table 5.2 shows a summary of differences in UI elements between these two platforms.

Windows Phone 8	Windows 8 (Store Apps)
ListPicker[1]	ComboBox
LongListSelector	ListView
RichTextBox	RichTextBlock
Toggle[1]	ToggleSwitch
WrapPanel[1]	VariableSizedWrapGrid / Wrap Grid
WebBrowser	WebView

[1] Available in the Windows Phone Toolkit

Table 5.2.: Overview of different user interface elements of Windows Phone 8 and Windows 8 (Store Apps).

[1]The Windows Phone Toolkit (formerly the Silverlight for Windows Phone Toolkit) is an additional package of user interface controls and functionalities, provided by Microsoft. `http://phone.codeplex.com/`

Security Windows Phone 8 and Windows 8 (Store Apps) are using different namespaces for their cryptography library. For Windows Phone 8 the namespace `System.Security.Cryptography` of the .NET API is used. In comparison, Windows 8 (Store Apps) uses the namespace `Windows.Security.Cryptography` of the Windows Runtime API.
On Windows Phone 8 only a basic set of cryptographic algorithms are available. As an example on Windows Phone 8 the library for AES supports only CBC mode and PKCS#7 padding. On Windows 8 (Store Apps) much more different algorithms and options are available.

Threading The both platforms use different Dispatcher classes. For Windows Phone 8, the Dispatcher class can be found in `System.Windows.Threading`. On the other hand, the Dispatcher class on Windows 8 (Store Apps) is located in `Windows.UI.Core` and is called CoreDispatcher.

Local Database On Windows Phone 8 we already have a Local Database. These Local Database *LINQ to SQL* API is not available for Windows 8 (Store Apps).

6. SEMS Client Implementation

In this chapter we discuss the implementation of SEMS for Windows Phone 8. We do not focus on implementation details. Instead we treat design decisions, database structure and the limitations that arise with Windows Phone 8. In addition, the mechanisms, used to protect the personal data within the application, are described.

6.1. SEMS for Windows Platforms

In this thesis it was originally intended to develop a SEMS client for Windows Phone 8 and Windows RT or rather Windows 8 Store App. However, it has shown during the implementation of the Windows Phone 8 client, that due to missing functionalities, limited availability of external libraries and large differences between these two platforms (see Section 5.3 in Chapter 5), the development of a Windows Phone and RT client would exceed the effort of this thesis.
To implement a client for Windows Phone 8 and Windows 8 App Store, the user interface for each platform would have to be implemented individually, as they managed in a different namespaces. In addition, the Windows Phone Toolkit was used for the Windows Phone 8 client implementation. This toolkit already provides useful user interface elements such as an *AutoComplete Box*. Since such a toolkit is not available for Windows 8 Store Apps, these elements would have to be additionally implemented.

Many external libraries are not available for Windows Phone 8 or were published during the ongoing development of the Windows Phone client. Thus, we had to implement for example a parser, so that we could export and import RSA keys. The Windows libraries allow only to export an RSA key as XML formatted string (see Listing 6.1). Therefore, it is not possible using the Windows Phone internal libraries to export an RSA key into an ASN.1 formatted string, which is used by SEMS. Listing 6.2 shows the in Listing 6.1 given RSA key as ASN.1 formatted string. There were also no external libraries for Windows Phone 8, which could be used for this task.
This points have led, among other things, that we have focused only on a Windows Phone 8 version of the SEMS client.

The motivation for the development of a SEMS client for Windows Phone 8 was to provide the same user experience as if a user send a ordinary SMS. Therefore, the application has been implemented in a way that the user interface of SEMS is very similar to the SMS Application interface of Windows Phone 8.

```
<RSAKeyValue>
    <Modulus>mzwtt3058e4gxjtEVtJ774HqF62m9HY/
       s3xGwZ94BPgvJmJjRQSTWL9dbnd5WhUNDMOk2XnJaxgCpYGTmEXutQ
       ==</Modulus>
    <Exponent>AQAB</Exponent>
    <P>zRud5pgkWIJjz+vnLGy46ughld6LZYM5ujE0TCVFBkE=</P>
    <Q>wcCocvG1+KyyFC71rkW7oGUhWhnzo8yTwPD7kn/zU3U=</Q>
    <DP>ABAGGE/rkutXRHT5+RV/aPLxL66FtF454kipj2xjfQE=</DP>
    <DQ>YfZ3NSbhSwqGP44+yg6X/1Eiu9vDRF57lmJB0KDA9sE=</DQ>
    <InverseQ>gCLdmd3cQzEu4i3MIYrXYCDZAJOxaD21o/rlOnYViaA=</
       InverseQ>
    <D>idNNV8t0RgvBx57ZVyi8ZvL2r507TXMrraUagVCf0/BQclwge6/
       SXIEeuo7ebEAC0s9oSRP4uoJafMpvmRinAQ==</D>
</RSAKeyValue>
```

Listing 6.1: Example of a 512-bit RSA key formatted as an XML string, which is supported by the Windows Phone API. This 512-bit key is used for illustration purposes only. SEMS uses a key length of 2048 bits.

```
MIIBOwIBAAJBAJs8Lbd9OfHuIMY7RFbSe++B6hetpvR2P7N8RsGfeA
T4LyZiY0UEk1i/XW53eVoVDQzDpNl5yWsYAqWBk5hF7rUCAwEAAQJB
AInTTVfLdEYLwcee2VcovGby9q+dO01zK62lGoFQn9PwUHJcIHuv0l
yBHrqO3mxAAtLPaEkT+LqCWnzKb5kYpwECIQDNG53mmCRYgmPP6+cs
bLjq6CGV3otlgzm6MTRMJUUGQQIhAMHAqHLxtfisshQu9a5Fu6BlIV
oZ86PMk8Dw+5J/81N1AiAAEAYYT+uS61dEdPn5FX9o8vEvroW0Xjni
SKmPbGN9AQIgYfZ3NSbhSwqGP44+yg6X/1Eiu9vDRF57lmJB0KDA9s
ECIQCAIt2Z3dxDMS7iLcwhitdgINkAk7FoPbWj+uU6dhWJoA==
```

Listing 6.2: Example of the in Listing 6.1 illustrated 512-bit RSA key as an ASN.1 formatted string. This format is used by SEMS to send these keys via the API.

6.2. Local Data Encryption

One requirement of the SEMS client is to protect all private data of the users. Therefore, the data, stored in the device's isolated storage, has to be protected against unauthorized access. In this first client version, three files are created. One file with the session ID, one with the user's own user ID and a file containing the database. Both, the session ID and the own user ID are encrypted and stored in the isolated storage of the application. About encrypting the database, we refer the reader to Section 6.3.2.
To create an encryption key in order to encrypt the data would not be sufficient, since the key would have to be stored as well. The Windows Data Protection API (DPAPI) solves this problem. It provides a mechanism to create and store a key by using the device and user credentials. On Windows Phone 8 each application gets its own key, which is created when the application runs for the first time. With this encryption key, provided by the DPAPI, the session and the user ID are encrypted and stored in the isolated storage of the device.

6.3. Database

In the database, all contact data, conversations and messages, used by the SEMS application, are stored. Since, as already mentioned, the application should be implemented for Windows Phone 8 and Windows RT/Store App, we decided to use a SQLite database. This database engine can be used for both platforms. Since SQLite is written in C/C++, an additional wrapper is necessary. As a wrapper, *sqlite-net* and the Windows Phone extension *sqlite-net-wp8* was chosen, because this wrapper provides the same access for both Windows platforms. Therefore the source code needs to be written once for both platforms.

6.3.1. Database Model

In Figure 6.1 the database model used by the SEMS client is shown. The database consists of five tables – *ContactModel*, *ConversationModel*, *MessageModel*, *MessageSendModel* and *UserSettingsModel*. As can be seen in Figure 6.1, all *varchar* fields have no maximum length, since SQLite ignores this information[1]. Also fields containing a Boolean value are specified with an integer, as SQLite does not support a separate Boolean data type. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

The ContactModel Table

The table *ContactModel* is used to store the contact data used by SEMS. Basically, this table is a mapping between the e-mail addresses (*contact*), of contacts stored in the phone's address book, and the resulting account information (*accountInfo*). As a primary key an internally consecutive ID will be used. The column *userID* contains the user ID for the respective contact, which was assigned from the server. If this user ID is not known to the client, this field remains empty.
The first time the application is started, all the contacts that are stored in the phone's address book are written into this table. Each time the application is started again, the table is updated when new entries are added in the phone's address book.

The ConversationModel Table

The table *ConversationModel* contains all conversation relevant data. As primary key again an internally consecutive ID is used, since the conversation ID, which is assigned by the server, can not be regarded as unique. For each participant in a conversation an entry in this table is created. For example, if there is a conversation of three participants, there are also three entries in the *ConversationModel* table.
The *ContactModel* table is in a 1:n relationship with the entries in the *ConversationModel* table. Thus, a relationship between conversation and conversation participants is established.

[1] Numeric arguments that following a type name are ignored by SQLite. SQLite does not consider any length specifications. The only exception is the global `SQLITE_MAX_LENGHT` limit, with a default value of 1 billion. For more information see `http://www.sqlite.org/datatype3.html` and `http://www.sqlite.org/limits.html#max_length`

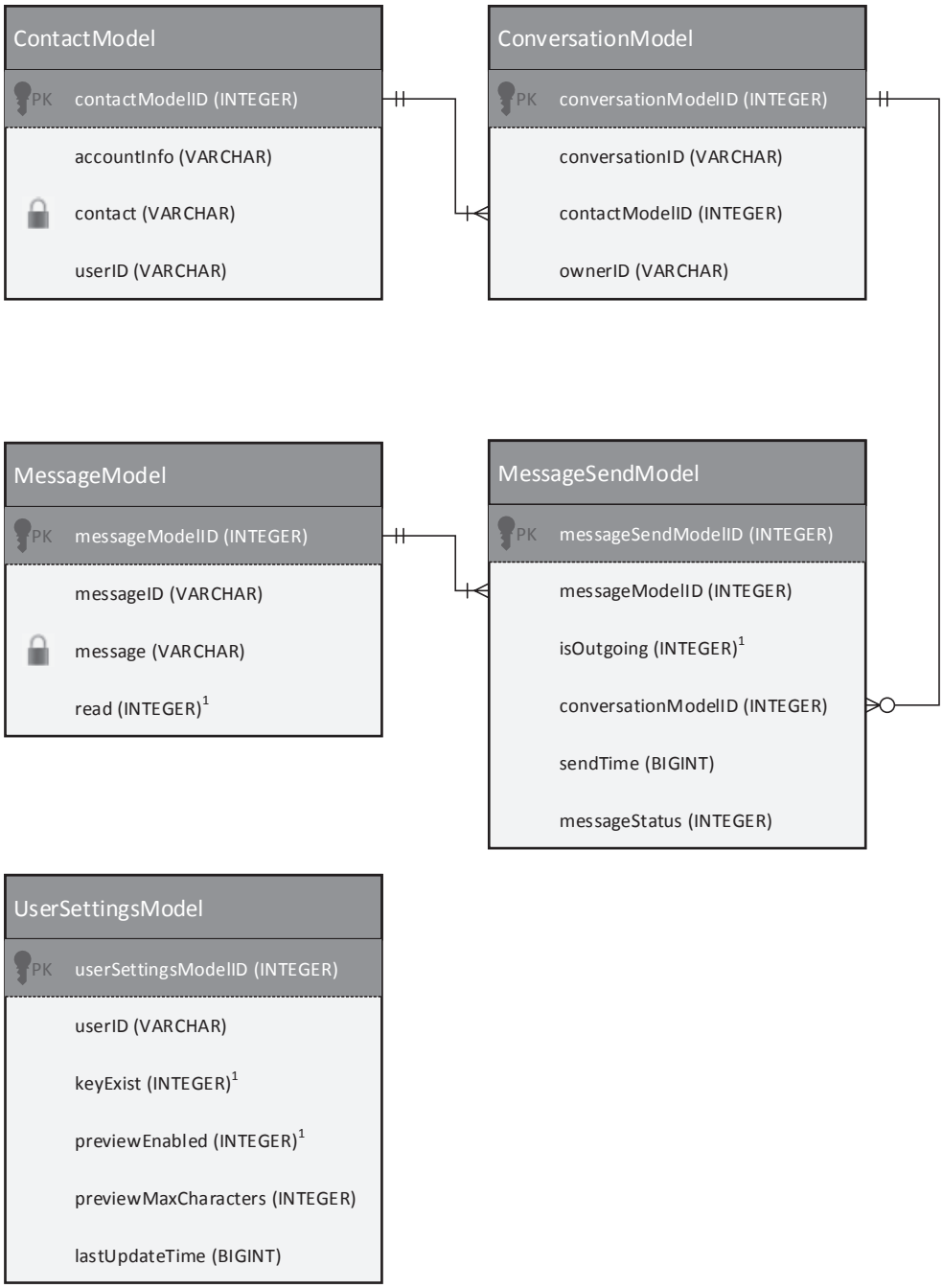

Figure 6.1.: Database schema that was used for the implementation of the Windows Phone 8 client of SEMS. The padlock symbol indicates that these fields are stored encrypted in the database.
[1] Boolean values are stored as integer 0 (false) and 1 (true).

With the *ownerID* the owner of this conversation is specified, since theoretically, multiple users can use the application on a single device. This should ensure that only those con-

versations are displayed within the application, which could be assigned to the logged in user. The case, that a single device is used by different users is relatively low, but has been considered in the database structure.

The MessageModel Table

In table *MessageModel* all incoming and outgoing messages are stored. As primary key again an internally consecutive ID is used. The message ID represents a server generated ID for each individual message. This message ID only has a meaning for incoming messages. For outgoing messages this ID is only stored for the sake of completeness. When a message is sent to multiple individuals (group conversation), then only one ID is stored in the database, although a unique ID is generated for each message by the server. The *read* field indicates whether a message has been read by the user.

The MessageSendModel Table

The table *SendMessageModel* is the link between *ConversationModel* and *MessageModel*. As in the three previous described tables also this table has a internally consecutive ID as primary key. The tables *MessageModel* and *ConversationModel* are both in a 1:n relationship with the *MessageSendModel* table.
Since, in a group conversation the message must be sent separately for each participant, the details of these messages are stored in this table, while the message text is stored in the *MessageModel* table only once. The Boolean field *isOutgoing* indicates whether a message was sent or received by the client. With the *sendTime* field the client stores the server-assigned time at which the message was sent. With the message status field, the status of each message is stored. This field is encoded as an integer. Thus, it can be determined whether a message has already been sent or is currently only stored on the client and is ready to be sent. The following status codes are currently available:

- 0 – messageSent: Message has been successfully sent to the server.
- 1 – messageStored: Message could not be sent yet and will only be displayed locally on the client. The client continues to try to send this message.
- 2 – messageUnsent: Message could not be sent due to an error. The client will not try again to send these messages. Thus, such messages will only appear locally on the client. These messages do not have a timestamp and thus will be apparent to the user that these messages were not sent.
- 7 – messageDeleted: This message was deleted locally and is therefore no longer displayed in the user interface. Messages will be stored in the database until they were successfully deleted from the server. This only applies to received messages.

The UserSettingsModel Table

In the table *UserSettingsModel* the client-specific settings for each user are stored. As primary key an internally consecutive ID is used. The field *userID* represents the respective user

account. The *keyExsits* field indicates whether the necessary private keys are available on this device. Also, the settings for the optional message preview for push notifications are stored in this table. The *previewEnabled* field specifies whether the preview parameter is to be specified, when a message is being sent. With *previewMaxCharacters* the maximum number of characters used for the preview is specified. The *lastUpdateTime* field contains a timestamp that is required for receiving messages using the API command `msg/getMessageIDS`.

6.3.2. Database Encryption

As discussed in Section 6.2, all personal data stored on the phone, should be protected from unauthorized access. This includes the data stored in the database.
The core SQLite library does not support encryption of the entire database. An encryption of the database file in the isolated storage of the device is not possible, because this file must be decrypted, before it can be accessed with the provided SQLite API. Also, there are no external available libraries for Windows Phone 8, which could encrypt the entire SQLite database. Therefore, we have decided to encrypt not the entire database and instead encrypt only the relevant columns in the database. This includes those columns that contains personal information about communication partners and message content. In Figure 6.1 all encrypted columns are marked with a padlock. These columns are encrypted using the Data Protection API (DPAPI) and the resulting encrypted string is stored in the database.

6.4. Local Key Management

Each SEMS client requires two RSA key pairs for communication. The public keys are sent to the SEMS server. The private keys remain on the device on which they were created. Private keys are managed by the client using the operating system provided libraries. Windows Phone 8 provides the possibility to create a key container in which cryptographic keys can be stored. These key containers provide a secure way to store persist cryptographic keys an keep them secret from malicious third parties. Each key container has to be assigned a name. With these names, the associated key container can be called and the included key can be accessed. In the implementation of the Windows Phone 8 client of SEMS, the name of a key container is a concatenation of the user ID of the logged-in user and the intended use of this key. For example, a private signature key gets the name `[userID]_signatureContainer`.

6.5. Limitations on Windows Phone

In this section, limitations on Windows Phone 8 are treated, which led to certain limitations in the implementation of the SEMS client. This includes bugs that were found on Windows Phone 8, as well as limitations due to the provided API.

6.5.1. Required Windows Phone Version

The Windows Phone 8 client of SEMS requires at least Windows Phone version 8.0.10322.0. This corresponds to the General Distribution Release 2 (GDR2) update which was rolled out in July 2013 for the first devices.
To control the access of multiple tasks to the database and the data structure for displaying the content in the user interface, the class *ReaderWriterLockSlim* is used. This class provides a locking system which allows multiple tasks or threads simultaneously read access, or allows a single task or thread exclusive write access. This class is used in several situation in the program code of SEMS, e.g. to control the SQLite database access.
In previous versions of Windows Phone 8, this class throws a *MethodAccessException* exception when two readers tried to enter the lock. This behavior is unintended and was fixed with the, previously mentioned, version update [43].

6.5.2. Phone Numbers in Text Messages

In text messages, elements such as URLs or phone numbers may occur. Therefore, it would be desirable if the user can interact with such elements. On Windows Phone, is it possible to make URLs clickable, so that the website is opened. When a user taps on a URL in a text message, it opens the Internet Explorer.
For phone numbers, it would be desirable, that when a user taps on such a number, either the address book opens with the detail page for this person, or if the phone number can not be assigned to a contact within the address book, then the user should get the possibility, to add this phone number to the phone's address book. On Windows Phone 8, it is not possible to open the Contacts/People Hub of the operating system for a phone number. The only possibility offered by the API, is to invoke a *PhoneCallTask*. A *PhoneCallTask* can be used to make phone call directly within the application. In this case, the user will be asked with a message box if he or she wishes to call this phone number. With this method, only phone calls are possible, but not sending SMS messages. In addition, a capability that allows phone calls is necessary[2]. However, this is not what we imagine for our messenger application.
A workaround would be to export the contacts from the phone's address book as vCard and store them in the isolated storage of the application, in order to import them again. With this procedure the Contacts/People Hub opens and offers the possibility to save this contact. However, this is not necessarily the behavior of what we want to achieve. If there already a contact exists in the phone's address book, that matches the specified phone number, no option to save the contact should appear.
Therefore, it is on the Windows Phone 8 client of SEMS currently only possible to interact with URLs of a website, which appear in text messages.

6.5.3. Sending Files

SEMS currently only supports text messages, multimedia messages, however, are now a standard in today's messengers. Therefore we give a small outlook of the limitations of Windows Phone 8 in terms of multimedia data.

[2] To allow an application to run a phone call task, the capability `ID_CAP_PHONEDIALER` is required.

The API of Windows Phone currently offers only access to the image library [44]. Thus it is only possible to send pictures stored in this image library. Any stored video, music or other files like PDFs can not be assessed. Videos and music files are stored in the *Music + Videos* application of Windows Phone 8, on which there is currently no API access. As Windows Phone 8 is a closed system, this also applies to other file types. These are usually stored in the isolated storage of the application. An example would be a PDF file. In order to read a PDF file, an application is required that can open such a file. In this context, the PDF file is stored in the isolated storage of the application which opens this file. This isolated storage can be only accessed by the application which owns this storage area. From the outside, by another application, this storage can not be accessed.
According to these limitation, the SEMS application on Windows Phone 8 has to take over the management of videos and other file types. For example, that only those videos that have been recorded by the user can be sent or received videos can only be stored within the isolated storage of the application. However, this assumes that the SEMS messenger has the necessary file management. This would have to be additionally implemented.

6.5.4. Certificate Verification

Windows Phone 8 offers with the current API no possibility to verify certificates manually. The general Windows API provides with the *ServicePoinManager* and a *ICertificatePolice* the ability to manually check HTTPS certificates. However, these classes are missing in the Windows Phone 8 API as well as in the API for Windows 8 Store Apps.
On Windows Phone 8, a certificate is automatically trusted when the following conditions are complied[3]:

- The certificate is derived from a root CA installed on the phone.
- The client must have access to all certificates in the chain – i.e. the server should include all intermediate certificates in the server certificate response.
- The server certificate's CN must match the host name of the URI.
- The server certificate must include the server authentication extended key usage.

In the case of Windows Phone 8, there is no possibility to override the certificate validation logic in code.

6.6. SEMS Client Limitations

In this Section the limitations of the SEMS client are treated. These are features and functionalities that were planned, but could not be implemented for the first version of SEMS.

[3]This definition was given in the context of a forum discussion in the official MSDN forum and was marked by a Microsoft employee as a correct answer. `http://social.msdn.microsoft.com/Forums/en-US/596247f8-68c1-4a10-83a2-9ec67f036f1b/which-library-support-https-in-windows-phone-8`, visited on December 2013

6.6.1. Multi-Client Support

For SEMS a multi-device support was intended. This means that a user can use several devices simultaneously with only one account. During the implementation, this feature turn out to be more complex than initially suspected. Therefore, the developers of the server and iOS version of SEMS have decided for the first version of SEMS, that only a single client can be logged-in at the same time.

6.6.2. Deletion of Conversations

The server of SEMS stores all incoming messages that have been sent to an account. The user has the option on the client application to delete these messages. In this case, the API command `msg/deleteMessage` is used. This command accepts a single message ID as argument. Therefore, only deleting single messages can be realized. If a whole conversation should be deleted, this API command is not a good choice. However, since the SEMS API does currently not support a command to delete a whole conversation, the Windows Phone 8 client of SEMS also supports only the deletion of single messages.

6.6.3. Contact Handling

When a conversation is created, all the recipients must be specified by the user. This is done by choosing the contact from the phone's address book. For the case that a contact has multiple e-mail addresses, each of these e-mail addresses is sent to the server, until the server responses that one of these addresses is assigned to an SEMS account. Once such an e-mail address was found, this address is used for sending the message. If a contact has multiple e-mail addresses which are a valid SEMS account, the first e-mail address is always used in the current version of SEMS. So far, no uniform solution was defined, how to deal with this situation. Therefore, the Windows Phone 8 client behaves in this case like the iOS client.

6.6.4. Multiple Account Information for a single Contact

For SEMS it was panned that an User ID can have multiple account information. Due to the fact that there is no API support for adding an e-mail address to an already existing account, there are also no implementations of the client-side for this feature.

7. Security Analysis of SEMS

In this chapter we analyze the SEMS API provided by the server that is used to communicate between client and server. In addition the data stored on the server are analyzed and the risk that arises when they leak out.

7.1. Methodology

The SEMS Messenger is based on the idea to use as little personal information as possible. Figure 7.1 shows a general illustration of the involved components within the messenger. The two main components of these are the SEMS server and the application on the mobile device itself.
The server stores for each registered user the respective user identity, consisting of the currently valid session identifier, the hashed e-mail address and an user identifier. In addition, the public keys for message encryption and signature verification as well as all communication-relevant data and the encrypted messages are stored on this server. These data can be queried using the API.
The application installed on the mobile device has an additional protected area. This means that all data that are assigned to this area are stored encrypted on the device. This includes the database[1], the private keys for decrypting and signing messages as well as the user data of the currently logged-in user. In addition, the application has read access to the phone's address book.
The Google OAuth server is required for the log-in. Also the respective push notification server for the different mobile phone operating systems are necessary for the implementation of push notifications. Both, the Google OAuth server and the push notification servers are not part of this analysis.

For this security analysis, we consider the following security-related aspects:

- Authentication
- Vulnerability, if some secret data were leaked
- Protocol Related Issues
- Server Related Issues

Authentication As already discussed in Section 4.3.1 in Chapter 4, SEMS uses the Google OAuth protocol for user authentication. Upon a successful authentication and if the user has granted the access, the SEMS server gets the permission to query the requested data from

[1]The encryption of the database differs in the various implementations. While Windows Phone 8 only encrypts individual columns of the database, on iOS the entire database is encrypted.

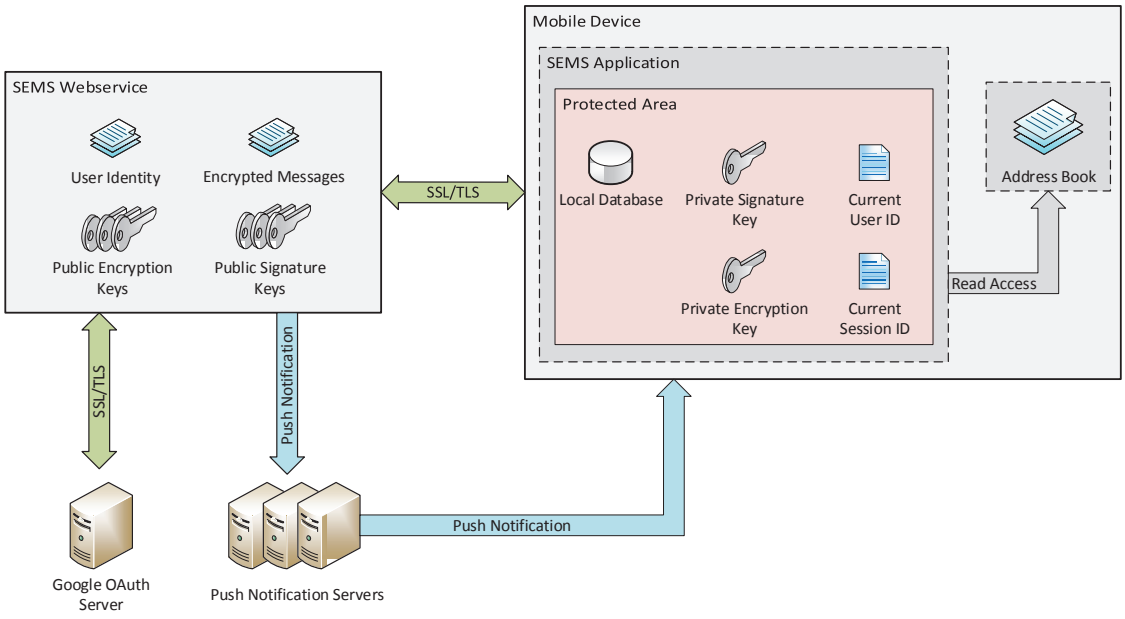

Figure 7.1.: Illustration of the components involved in SEMS and the data stored in the individual components.

the Google server. This data may include personal information of the user. The OAuth 2.0 access restrictions are well defined, but the configuration of the access type is defined by the SEMS service provider. The data required by SEMS are limited to the user's e-mail address, which can be found only as hashed value on the server. Nevertheless, it must be analyzed whether the server provider or an attacker is able to restore the original data subsequently by using the OAuth protocol.

Vulnerability, if some secret data were leaked Normally secret data are protected by mechanisms against access from malicious third parties. As part of this security analysis the risks are assessed that may arise when confidential data become known to a third person. Such data can for example be used to feign identities or decrypt already sent messages.

Protocol Related Issues The API commands of the SEMS protocol consist of several parameters, whose values can be chosen by an attacker. Therefore, the effects of such manipulations must be analyzed. This applies to both, a manipulation by the sender as well as a manipulation during transmission, by malicious third party.

Server Related Issues Due to incorrect implementation or missing verification of incoming data, there exists a risk that the server can reveal user or communication related information to an unauthorized person. In addition, it must be ensured that an attacker or the server provider itself is unable to access confidential data.

7.1.1. Assets

Before we can start to analyze SEMS, we have to define first, what kind of information must be kept secret and which may be publicly known. SEMS aims to securely transmit messages from a sender to the receiver. Therefore, it is necessary to ensure that an encrypted message can be decrypted only by the receivers of that message and thus only the receivers can read the plaintext message. In order to ensure this, private keys used for encryption and signing of messages have to be accessible to the owner only. Furthermore, for the communication with the server a session ID is needed. This Session ID will be set on each log-on and is valid until a successful log-out of the client application. With this session ID, a client can be identified by the server and therefore the server can assign this session to a registered user who sent that request. Therefore, the session ID as well as the private keys must be kept secret.
It is also important to protect the identities of the individual users. SEMS currently only supports authentication via a valid Google account. As a result SEMS currently uses only the respective e-mail addresses. These e-mail addresses are stored on the server only as hash values.

This results in the following assets for our security analysis:

- Messages
- Private encryption key
- Private signature key
- E-mail addresses
- Session ID

7.1.2. Assumptions

This security analysis of SEMS focuses on the underlying API protocol and the provided features of SEMS. Therefore, the following assumptions were made for this analysis:

- An attacker has full access to the network communication. This means, that an attacker can read and manipulate the transmitted messages.
- The SEMS server is treated as a Black Box. This means that no exact information on the processing of the server-side stored data are known and thus the security of the server-side data is not taken into consideration.
- Device-specific security mechanisms are not considered as part of this analysis. It is assumed that these mechanisms are working correctly. It is also assumed that the used devices are not compromised by Malware or an Jailbreak.
- For authentication, it is assumed that it works correctly according to the documentation of OAuth 2.0 and Google.

7.2. Authentication

For authentication SEMS currently only supports Google's OAuth 2.0 protocol. So, the security of the stored user credentials such as username (e-mail addresses) and passwords are not part of the SEMS service and therefore not relevant for this security analysis. Due to the lack of access to the implementation and the undocumented implementation-specific features, the Google OAuth system must be considered as a black box. The security of OAuth protocols, including popular OAuth service providers such as Google or Facebook, is a popular field of research [45, 46].

To log-on to SEMS, the API command `auth/login` is called on the client. This opens a so called WebView and redirects the user to the page of Google's OAuth 2.0 service. The used URL is shown in Listing 7.1.

```
https://accounts.google.com/o/oauth2/auth?response_type=code&
   client_id=502218450784.apps.googleusercontent.com&redirect_uri=
   https://sems-test.iaik.tugraz.at/sems/auth/oauthcallback&scope=
   https://www.googleapis.com/auth/userinfo.email&access_type=
   online&approval_prompt=force
```

Listing 7.1: Google OAuth 2.0 URL used by SEMS with the marked security relevant parameters scope, access_type and approval_prompt.

Security-relevant parameters are `scope`, `access_type` and `approval_prompt`. The parameter `scope` with value `userinfo.email` specifies that the server would receive the e-mail address of a user using the Google API. Users who wants to log-in are also informed about this on a granting page inside the WebView. The parameter `access_type=online` indicates that the server does not receive a refresh token. Refresh tokens are required to get a new access token from the Google service. An access token gives the server the right to obtain the previously stated information from the Google API. With `approval_prompt=force`, the user is prompted to confirm the SEMS application and thus grant access to the Google API at every single log-in.

The configuration of this URL is made by the SEMS service provider. Thus, it is theoretically possible that the service provider subsequently changes the configuration. This modification would be visible only in the used URL. An exception is the modification of the scope parameter, because the requested data must be authorized by the user and are therefore displayed on the granting page. If the SEMS service provider changes the `access_type` to offline (example shown in Listing 7.2), the SEMS server also receives a refresh token. With this refresh token he could subsequently request a new access token and thus request the data again using Google's API. This offers the possibility to query the identity and the email address of the users again at a later date and then link them to the data stored on the server.

```
https://accounts.google.com/o/oauth2/auth?response_type=code&
   client_id=502218450784.apps.googleusercontent.com&redirect_uri=
   https://sems-test.iaik.tugraz.at/sems/auth/oauthcallback&scope=
   https://www.googleapis.com/auth/userinfo.email&access_type=
   offline&approval_prompt=auto
```

Listing 7.2: Example of a modified Google OAuth 2.0 URL. The access_type was set from online to offline and approval_prompt changes from force to auto.

These changes can only be seen at the URL string but such an URL is usually not displayed within the application. When the API command `auth/login` is called via a conventional Browser, the URL used for the Google authentication can be viewed. Moreover this changes can only be used when the users log-on again and confirm the access of SEMS to the Google API, otherwise no refresh token is issued.

Over all, it must be trusted on the information provided by the service provider, because he or she theoretically has the opportunity to store all e-mail addresses and identities of the users from the beginning and not only the hash values of the e-mail addresses, like on the SEMS server.

7.3. Vulnerability in Case of Leaked Secret Data

In this section we consider the case when any confidential information is known to an attacker. In particular, the session ID provides several ways to manipulate the communication or get access to private data. On the other hand private keys offers the option to decrypt an previously encrypted message.

7.3.1. Messages

Messages are always transmitted as an encrypted string and can only be read by the receiver of the message. As already discussed in Section 4.3.2 in Chapter 4, a message string is constructed as follows:

$$E_{RSA}(K_{AES})||S_{RSA}(SHA_{256}(IV||E_{AES}(message)))||IV||E_{AES}(message)$$

To avoid that a sent message was tampered on the transport, it is additionally signed by the sender. To decrypt text messages, an attacker must have knowledge of the private encryption key of the message receiver, since for each text message a new AES key is used.

7.3.2. Private Keys

For each SEMS account two private keys are stored on the mobile device, an encryption key and a signature key. The private encryption key is used to decrypt the wrapped AES key within the message string, in order to decrypt the text message with the obtained AES key. In contrast, the private signature key is used to sign an encrypted text message.

All private keys are stored locally on the device. To ensure, that third parties do not have access to these keys they are stored encrypted. The used mechanisms in order to protect private keys is thereby platform-dependent.
Basically, private keys never leave a device. However, this would imply that if this mobile device was replaced by another device or the user logs on to a different device, the user has to generate new key pairs for SEMS. As a result, all previous messages can not be decrypted anymore.

To give the user the possibility to make private keys accessible to other devices, it is planned to provide an optional feature for storing the private keys on the server. A password based key derivation function shall be used to store private keys encrypted on the server. So, the private keys are encrypted, but therein there is also the risk that an attacker or even the service provider obtains knowledge of the password, and can decrypt the entire communication which is stored on the server. In addition, messages can be signed with the identity of the victim. However, it should be noted that the usage of this feature is optional and each user can decide for themselves whether they want to use it.

7.3.3. Email Addresses

When creating a SEMS account, the user must agree that his or her e-mail address can be used by the SEMS server. With this e-mail address the account information is derived using a hash function. This account information is linked with the server generated user ID. Theoretically, several account information, and thus more e-mail addresses can be assigned to a single user ID. However, this possibility is not yet supported by SEMS.
In all communications of SEMS, never the email address itself is used. Even if a new message is created and the e-mail addresses of the recipients are specified, only the hash values of the e-mail addresses are sent to the server to get the User ID of the associated users. This should ensure that by observing the communication protocol it can not determined who communicates with whom.

7.3.4. Session ID

A session ID is a UUID string generated on the SEMS server and is sent at every log-in via a cookie to the client. This session ID is valid up to a successful log-out. Currently, this is the only situation in which a session ID can lose their validity. In future server versions, even during periods of inactivity or a log-on on another device, a session ID will lose their validity.
Session IDs are stored on the server in hashed form, so an attacker has no way to manipulate or abuse user accounts if he has access to the data stored in the server's database. On client side, the current session ID is also stored encrypted on the phone's isolated application storage.
A session ID is assigned uniquely to a user and is therefore used as an identification of an user on every request. If an attacker has the knowledge of such an ID, several attack scenarios would be possible, as discussed in the following sections.

Public Key Upload

If an attacker has the knowledge of the victim's current session ID, he could replace the public keys that are stored on the server, by his own public keys. To upload the public keys, only the session ID is used as a identification of the user on the server. Listing 7.3 shows the API command `key/uploadPublicKeys` for uploading public keys to the server.

```
{
    "session":"24ae7058-4cdf-484b-a1cf-5f6ec5b88b29",
    "encryptionKey":"MIIBCgKCAQEAzTZyTJ8Pne...DAQAB",
    "signingKey":"MIIBCgKCAQEA1M+3co7ah+...DAQAB"
}
```

Listing 7.3: API command `key/uploadPublicKeys` for uploading public keys to the SEMS server.

This would make it possible for an attacker to send text messages with the victim's identity. In addition, he could decrypt all messages that have been sent to the victims account, if they have been already encrypted with the attacker's public encryption key. The victim would not notice the exchange directly. Additionally, other users cannot proof the owner of the received keys and therefore the keys can not be verified whether they were manipulated by an attacker or the server provider itself. However, it should be noted that if public keys are uploaded to the server by a third person, the respective private keys on the client no longer fit together. As a result, decrypting messages yields in wrong values or the entire decryption and verification process fails.
There is no direct way to determine this manipulation for the victim's client or the victim's themselves. This would only becomes apparent when the victim receives messages that can not be verified and decrypted or the decrypted text results in no meaningful text.
However, is should be noted that this also depends on the implementation of the client, whether a user is notified when a message could not be correctly verified or decrypted.

Sending Messages with a Wrong Identity

As already discussed in Section 4.3.2 in Chapter 4, an encrypted message consists of a wrapped (encrypted with RSA) AES key, the signed message, an AES initialization vector and the encrypted text message using the AES key.

When a user wants to send a message to another user, he needs the user's public key. This public key is stored on the SEMS server and can be requested via API command `key/getPublicKeys`. Figure 7.2 shows a general sequence of a typical communication between two parties.

As illustrated in Figure 7.2, Alice wants to send a text message to Bob. With the API command `key/getPublicKeys` Alice receives the public keys of Bob from the server. Alice encrypts the text message with a random generated AES key K_{AES} and signs the encrypted message with their private signature key. Thereafter, the AES key is encrypted with Bob's

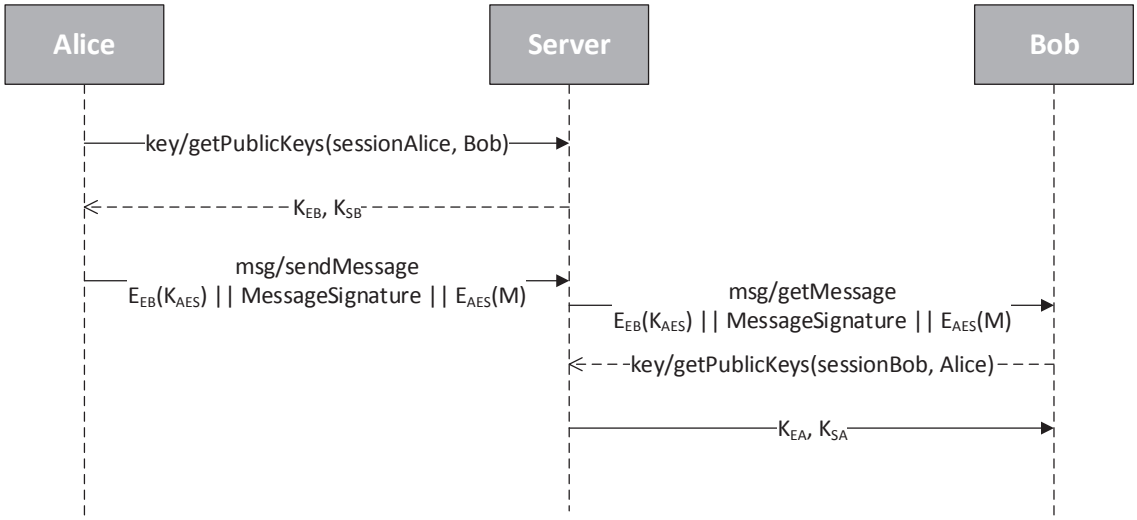

Figure 7.2.: Simplified illustration of a communication flow between the two communication participants, Alice and Bob, over the SEMS server.

public encryption key K_{EB} received from the server. Then Alice sends the resulting message string to the server.
When Bob receives this message, he queries the server for the public keys of Alice. With the obtained signature key K_{SA} of Alice, he can verify the message string and afterward decrypt the wrapped AES key with his private RSA key in order to recover the plaintext message.

If an attacker knows the victim's session ID, it allows him or her to send messages with the victim's identity. In order that these messages can be verified and afterward decrypted by the receiver, an attacker must also have the knowledge of the victim's private signature key. As already described in Section 7.3.4, only the current session ID of the victim is necessary to upload new public keys to the server. This allows an attacker to sign the AES encrypted massage with its own private signature key. Additionally, he or she encrypts the used AES key with the public encryption key of the recipient and send the entire message string to the server. The recipient of the message gets the matching signature key from the server and verifies the signature of the incoming message. Afterward he or she decrypts the AES key with his or her private encryption key and then the text message with the obtained AES key. For the recipient, it sill looks as if the message had been sent by the victim.

Receive Server-side Stored Messages

With a known session ID of the victim, it would be possible to retrieve all messages stored on the server. As shown in Figure 7.3, all message IDs and associated conversation ID's, stored on the server, can be downloaded using the API command `msg/getMessageIDS`. The timestamp parameter defines which messages should be downloaded from the server. As the server in the current version stores all messages that have been sent to an account, it

is theoretically possible to download all these messages. The contained conversation ID of the server response may provide clues to the conversation partner of the victim. In the next step, the encrypted message can be downloaded using the `msg/getMessage` command with the respective message ID.
However, the encrypted message can not be decrypted without knowing the corresponding private key of the victim. Considering that the session ID is not publicly available, and messages can only be decrypted with the corresponding private key of the message recipient, there is no high risk that an attacker could get to the content of such messages.
Alternatively, the received message IDs can be used to delete the messages stored on the server (see next section 7.3.4).

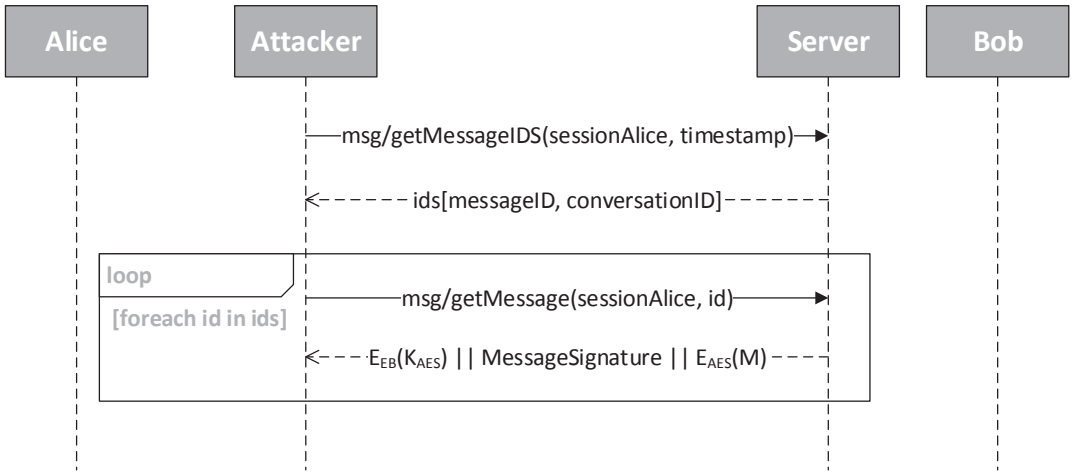

Figure 7.3.: With the knowledge of the session ID, all messages that have been sent to the victim and are stored on the server, can be downloaded.

Deletion of Messages

The session ID allows a third person, who has knowledge about the victim's session ID, to delete all messages stored on the server. For deleting a message, the session ID and the message ID are required. The necessary IDs of the messages can be obtained from the server using the API command `msg/getMessageIDS`. The server returns all stored message IDs including the associated conversation IDs starting at the specified timestamp.
By deleting single messages, an attacker can possibly prevent that these messages are received by the victim's client. An existing message preview, which will be send via push notification to the client, can not be suppressed or deleted.

7.4. Protocol Related Issues

In this section we analyzed general risks affecting the SEMS API. We focused on parameter tampering of an API call, so that these manipulations where not noticed by the server or the involved clients. This would be allow to establish communication between individuals with wrong data or would allow an attacker to read along sensitive data.

7.4.1. Public Key Distribution

Besides the delivery of message related data and authentication of users, the SEMS server acts additionally as PKI. For each communication with another user, his or her public encryption key is required. Also, for receiving of a message, the public signature key of the sender is needed for message verification. Both public keys are stored on the server and can be retrieved from any client by using the API command `key/getPublicKeys` for a given user. The server responds with the public encryption and signature keys (see Listing 7.4). Both keys of the response are encoded in Base64 and contains the respective keys in ASN.1.

```
{
    "encryptionKey":"MIIBCgKCAQEAzTZyTJ8Pne...DAQAB",
    "signingKey":"MIIBCgKCAQEA1M+3co7ah+...DAQAB"
}
```

Listing 7.4: API command `key/getPublicKey` for downloading public keys from the SEMS server. Each key string is encoded in Base64 and contains a public key in ASN.1 format.

Because the keys are delivered without proof of the owner, this results in the threat of a man-in-the-middle attack, as shown in Figure 7.4. An attacker could intercept the server response of a `key/getPublicKeys` request and sends a forged message back to the client. The attacker exchanges the public keys, that have been sent from the server, against its own keys and sends it to the requesting client. As can be seen in Listing 7.4, the receiver of the public keys has no possibility to verify the owner of the public keys. Therefore, the client has to trust that the received keys are belonging to the communication partner. With the received encryption key, the used AES key is encrypted and then the entire message string is send to the SEMS server. The attacker intercepts the message, decrypts the AES key and encrypts it with the originally key sent by the server. Afterward he or she forwards this new message string to the server. By obtaining the AES key, the attacker can decrypt the message. With this procedure, an attacker can repeat it for each communication between the victim's client and the server and thus read the outgoing communication of the victim.

Not only a third person between client and server is a risk that messages can be read from an unauthorized person. Also, the service provider can responds with an incorrect public key to the requesting client (see Figure 7.5). In such a case, the server provider can also read the communication without that sender or receiver of the message being aware of it.

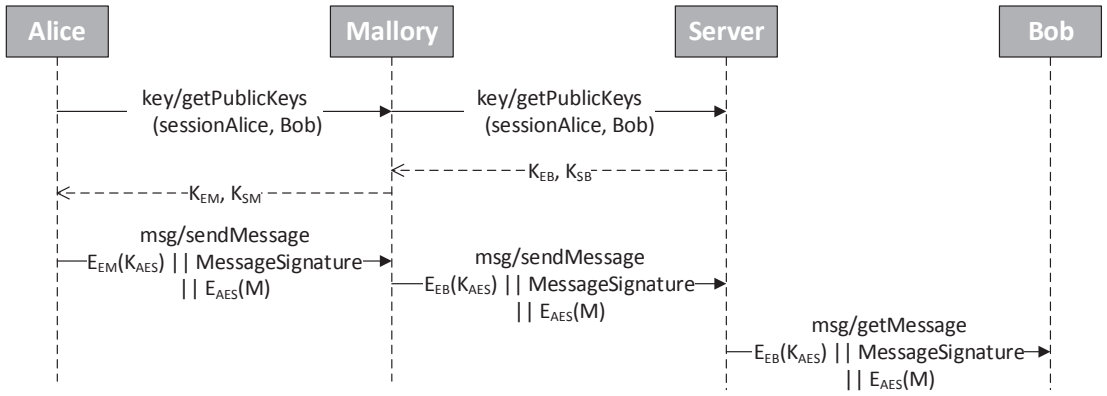

Figure 7.4.: Example of a Man-in-the-Middle attack to read the outgoing messages of Alice. The attacker (Mallory) intercepts the `key/getPublicKeys` server response and forwards his or her own public keys to Alice. So he can obtain all necessary information to decrypt the message sent by Alice.

7.4.2. Message Preview for Push Notifications

The preview parameter is part of the API command `msg/sendMessage` and is used to send a preview within push notifications. This parameter is optional an can be set on the client options. An additional parameter is necessary to provide a preview for push notifications, because all messages are always sent encrypted. Thus, the SEMS server has no way to read the content of the encrypted messages.
Due to the fact that the preview for push notifications must be specified separately and the SEMS server has no possibility to read the content of the sent messages, it is possible to forge this preview. It can be stated for the preview any arbitrary string. As a result, a completely different text, than the original text message, would be displayed as preview of a push notification. The only limitation is the maximum string length that is accepted by the SEMS server. According to the documentation of the API, the server accepts a maximum string length of 32 characters. Experimental tests have shown, that the server has accept a maximum length of 36 characters.
As already mentioned, there is no possibility for the server to verify the data of the preview parameter. Therefore, there remains a certain risk that the preview and the text message are not related.

7.4.3. Sending Messages with Forged Data

Text messages are sent via the API command `msg/sendMessage` to the SEMS server. In addition to the encrypted message string, other parameters such as receiver or the own session ID must be specified. Listing 7.5 shows a example of a possible send message request.

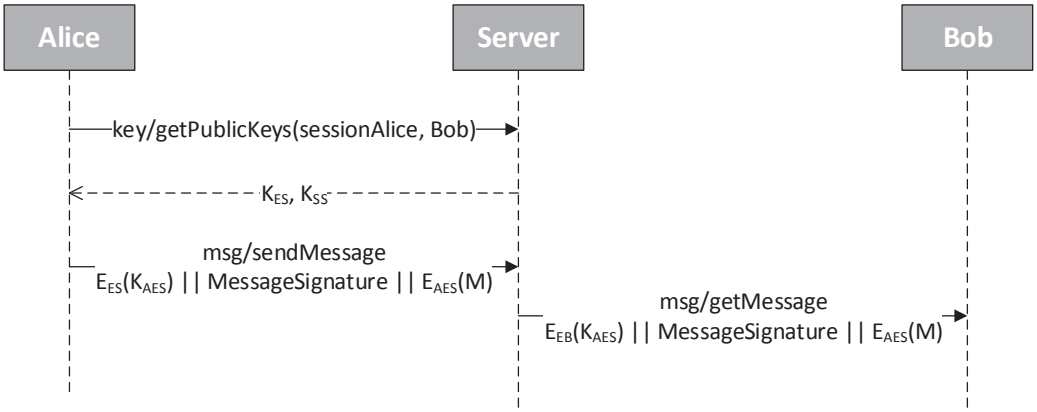

Figure 7.5.: The service provider acts as man in the middle to read the communication between Alice and Bob.

```
{
    "session":"41062242-7896-48df-b79d-2a439371e960",
    "to":"66e2e95f-c578-4075-9aab-584ed878f8b1",
    "message":"SGFsbG8gU2Vtcw==",
    "conversation":"ivnu3q5qvf8g@sems-conv.com",
    "preview":"MessagePreview"
}
```

Listing 7.5: API command `msg/sendMessage` for sending text messages.

As can be seen, in addition to the receiver (*to* parameter) a conversation identifier is necessary. This means that these two parameters must be related to each other. However, the SEMS server accepts messages in which the specified conversation ID is not related to the recipient of the message. This means that the recipient of the message is not a member of this conversation. For instance, the following scenario would be possible:

1. Alice sends a text message to the user Bob. Usually, the common conversation ID `c68l1mf9gyi5@sems-conv.com` should be specified. But the wrong conversation ID `efv1071pmx73@sems-conv.com` was used. This conversation ID is assigned to the users Alice and Carol. Listing 7.6 shows the transmitted message string to the server. The server responds with the message ID `dracbx6wa1on@sems-msg.com`.

2. Bob receives this message and obtains the following data:

   - Message ID `dracbx6wa1on@sems-msg.com`
   - From `eda91939-2a7d-4db0-b08c-40705dac58dc` - This user ID corresponds to the user Alice.
   - Conversation ID `efv1071pmx73@sems-conv.com`

3. Because the received conversation ID is unknown to Bob, the client queries the server to get the associated user IDs. As a response Bob receives that the users Alice and Carol belong to the specified conversation ID. Since the response of the server never includes the own user ID, even if it is associated with the conversation, Bob believes that he is a member of the conversation with Alice and Carol.

```
{
    "session":"5afb695b-5a55-4a28-87c7-827545bcd3f2",
    "to":"f92270af-accb-4d3d-8781-6e1c4959335f",
    "message":"SGFsbG8gU2Vtcw==",
    "conversation":"efv1071pmx73@sems-conv.com",
    "preview":"Message Preview"
}
```

Listing 7.6: Message that is sent from Alice to Bob with a wrong conversation ID.

When Bob would now reply to the message, he not only would send it to Alice, also Carol would receive it. As a result, the communication between Alice and Bob has been extended to a third person. However messages sent by Carol can only be received from Alice and not from Bob.

Similarly, it would be theoretically possible that the sender of a message is not part of the given conversation. The following scenario shows a possible sequence:

1. User Eve has knowledge about the common conversation ID of Alice and Bob (e.g. `c68l1mf9gyi5@sems-conv.com`).

2. Eve sends a message to Alice and uses the common conversation ID of Alice and Bob.

In this case, the server responds with a *403 Forbidden* error code. This means, that the server does not accept the message.

The fact that the attempt to send a message with a session that has no relation to the specified conversation ID has failed, it is clear that the SEMS server checks the incoming message to the accuracy of the incoming data. However, the first test has shown that the data regarding recipient and conversation ID will not be checked. Correctly, it should also be verified whether the receiver and the conversation ID relate to each other. This problem should be easy to fix with an additional check.

7.4.4. Tampering of server responses

So far we have only considered the situation when request data was forged. When an attacker controls the communication between client and server, it is also possible that the server responses are distorted. In the following sections we consider the impact on the security of the SEMS application if server responses have been manipulated by a third party.

User Identities

When a new conversation is created and the recipients are be specified, for every e-mail address of a recipient the hash value of the e-mail addresses is calculated by the client application. This hash value forms the account information and is sent via API call `usr/getUser` to the SEMS server in order to check if this user has a registered SEMS account. As response, the client receives the user ID of the queried account information, which is needed in further consequence to send a message to this user. The requesting client has to rely on the correctness of the server response.

A third person who controls the communication between client and server and thus can also manipulate it, could forge the server response and send a wrong user ID back to the client. As a result, the message would be encrypted with a wrong key. Also, the message would not be sent to the intended recipient. Messages could thus be redirected by an attacker. However, it should be noted that this situation can only come about when a new conversation is created. When replying to an already on the client existing conversation, the server will not be queried for the associated user IDs.
To reduce the risk of forged server responses the client should validate the received user ID. If the client stores the previous received user IDs along with their account information and every time the server responds with a user ID, the client compares this value with the stored one. In the case that the data is not identical, the user would be informed. An alternative approach would be to renounce the server request, if the user ID already exists in the database.

Conversations

For every conversation, in addition to the user identities, a server generated conversation ID is required. These can be obtained from the server via `msg/newConversation`. As with the user IDs in Section 7.4.4, must be trusted here on the correctness of the server response. An attacker who again controls the communication channel between client and server could intercept the server response and forge the conversation ID. As a result, an attacker can interfere with the communication of two or more parties. The risks that are associated with a wrong conversation ID have already been discussed in Section 7.4.3.
Again, this is only a threat if a new conversation is created. It is also implementation dependent if the server is contacted at all if a corresponding conversation ID is already stored locally.

User Identity Impersonation

Received messages always contain the user ID of the sender, to be able to determine the user's identity. If this user ID to a client is unknown, the client uses the API command `usr/getAccountInfo` to obtain the associated account information (hash value of an e-mail address). An attacker who controls the communication could forge this response and modify or extend the account information sent by the server. By replacing its own account information with another one, an attacker could impersonate someone else. This could be a person familiar to the victim. A possible attack could be as follows:

1. Eve sends a message with their own SEMS account to Alice.
2. Alice receives this message, but the user ID is unknown to her.
3. Alice queries the server using `usr/getAccountInfo` to get the associated account information.
4. Eve intercepts the server response and replaces her account information by the account information of Bob. She then forwards the forged server response to Alice.
5. Alice thus receives a reply that this messages was sent from Bob. Since Alice knows the user Bob, she can combine Bob's account information with his e-mail address and thinks that the received message comes from Bob. However, if Alice should answer, the message will not be sent to Bob but to Eve.

Disable the Reception of Messages

All timestamps within a conversation are assigned by the server. This means that when a message was sent, the server sets the time at which this message was sent. Before a client can receive individual messages, it must first use the command `msg/getMessageIDS` to get all message IDs since a specified time from the server. Listing 7.7 shows an example request to get the message IDs. The *since* parameter in the request defines which messages are relevant for download. So that only messages are downloaded, where the timestamp is greater than or equal to the specified parameter. The server response includes a list of message IDs and the associated conversation ID's.

```
{
    "session":"24ae7058-4cdf-484b-a1cf-5f6ec5b88b29",
    "since":"1360866142000"
}
```

Listing 7.7: API command `msg/getMessageIDS` for receiving all message ID's since the specified timestamp.

In order to avoid to receive all messages every time, the timestamp has to be updated regularly. As already mentioned, the timestamps are managed by the SEMS server. Therefore, for the *since* parameter, the highest timestamp of the already received messages is used.
If an attacker is able to intercept a `msg/getMessage` server response and manipulates the time parameter within, then he can disable receiving messages on the victim's client. This can be achieved, by replacing the *time* parameter with a relatively high value, which represents a date far away in the future. Listing 7.8 shows a sample response with a future timestamp.

```
{
    "message":"SGFsbG8gU2Vtcw==",
    "from":"eda91939-2a7d-4db0-b08c-40705dac58dc",
    "time":"2147483648000",
    "conversation":"efv1071pmx73@sems-conv.com",
    "read":"false"
}
```

Listing 7.8: Response of a `msg/getMessage` request with a forged timestamp. The value 2147483648000 represents the date in January 19, 2038 at 03:14:08 UTC.

7.5. Server Related Issues

Not only the knowledge of private information or the tampering of API parameters provide a security risk, also the SEMS server itself stores user information and provides the possibility to request some of those data via API calls. Usually a user should only receive the information from the server if they concern him or her, or they are publicly available.

7.5.1. API Requests

Whenever the required data is not available on the client, the client sends a request to the SEMS server to obtain the missing information. For example, if a message is received on the client, only the information about the sender and the associated conversation ID is included in the server response. If the received conversation ID is unknown to the client, the client calls the API command `usr/getUserIDsConversaion` to get more information about the members of that conversation. Also for a given user ID, the associated account information can be queried with the respective API command, or vice versa.
Tests have shown that the server can be abused to get information of conversation participants of an observed conversation ID or to enumerate registered SEMS users by sending a huge number of parallel requests to the server.

Enumeration of Users within a Conversation

With the API command `usr/getUserIDsconversation` it is possible to ask the server, to obtain a list of users which are a member of the specified conversation. Listing 7.9 shows the structure of such a request. For this purpose, the own session ID and a conversation ID is required. As a response, the server returns the list of associated user IDs which are a member of the specified conversation.

In our tests, the server did not responds only to requests in which the specified session was related to the conversation ID. The server responded as well when the session ID and the conversation ID did not belong together. The test setup requires three users, Alice, Bob and Eve. The conversation ID for a conversation between Alice and Bob is defined as `efv1071pmx73@sems-conv.com`. A third person, in our case Eve, knows this ID but she does

```
{
    "session":"24ae7058-4cdf-484b-a1cf-5f6ec5b88b29",
    "id":"efv1071pmx73@sems-conv.com"
}
```

Listing 7.9: Example of the API command `usr/getUserIDsConversation` to get all members of a conversation ID. The specified session ID relates to the user Eve while the conversation id (*id* parameter) relates to the users Alice and Bob.

not know which users belongs to it. Therefore, Eve sends with their own SEMS account a `usr/getUserIDsConversation` request to the server (see Listing 7.9). Although Eve is not a member of the specified conversation, she receives a response with the corresponding user IDs from the server as shown in Listing 7.10.

```
{
    "ids":["7c51949e-dcea-413b-b049-e990f2a44897",
        "eda91939-2a7d-4db0-b08c-40705dac58dc"]
}
```

Listing 7.10: Server response of the API request shown in Listing 7.9 The included IDs relates to the users Alice and Bob.

So, Eve found out, that these two users belong to the specified conversation ID. These user ID's can also be connected via the API command `usr/getAccountInfo` to the associated account information. In our case the account information are the hashed values of the e-mail addresses. Since this are only hash values of the respective e-mail addresses, no personal information, without further knowledge, can be gained from it.
This issue can very easily be resolved, if the server only accepts requests in which the session ID and the conversation ID are related to each other. In the case that the requesting user is not a member of the specified conversation, the server should not respond with the related user IDs and instead deny the request.

Enumeration of Registered SEMS Users

In SEMS, users are identified via their account information. This means that a client sends the hash value of the e-mail address of an user to the server and receives the associated user ID. The API of SEMS allows only one account information per request. Thus there is no possibility to upload the complete address book to the server and receive all registered users as it would be possible in other messengers such as WhatsApp. However, the individual requests can be sent in sequence or in parallel to the server. An example of such a single request is shown in Listing 7.11.

```
{
    "session":"24ae7058-4cdf-484b-a1cf-5f6ec5b88b29",
    "account_info":"hd5Q6ZbX+J0yYiS+
       JBtyF2JknBclB98vpB99C2483uw="
}
```

Listing 7.11: API command `usr/getUser`.

In several test runs, it turned out that the server allows a non-limited number of request within a short time. In this tests, the same session ID was used for every single request. The experimental setup was performed with more than 2000 contacts that have been sent over single requests to the SEMS server. All requests sent to the server where completely answered in under one minute with the associated user IDs or an error code if the user has no registered SEMS account.

To make the enumeration of registered SEMS users harder, the server should only answer a limited number of requests with the same session ID within a specified time. This issue can not prevented entirely, but significantly complicates the enumeration of accounts.
Basically it should also be thought about to limit the number of participants in a group conversation, so that, for example, a maximum of 30 users can participate within a conversation.

7.5.2. Data Stored by the Server

A main goal of SEMS was to store as little personal user information as possible. Therefore, the SEMS server uses only anonymous user data. In the database of the server according to the documentation, the following user data are stored:

- Unique server generated user identifier
- Hash value of the session identifier – $SHA_{256}(session)$
- Account information of the user – $SHA_{256}(e-mail)$
- Public keys
- Encrypted messages

Due to the lack of access to the implementation code of the SEMS server and the lack of documentation, no detailed analysis of the SEMS server can currently be performed. In the following Sections, only design features are critically examined.

Server-side Stored Messages

In the current version of SEMS, all messages are stored on the server. Due to the message encryption, it is only possible to store those messages, which were sent to the respective SEMS account. Because of the encryption with the public key of the recipient, the message sent by the client can not be stored for the respective user on the server. In order to decrypt

the message, the private key of the recipient would be required.
For stored messages in the current server version, there is no maximum time period, how long they are stored on the server. This design feature has the advantage that users can receive incoming messages on multiple devices, even if they have already received it some time ago on another device. However, this feature has the disadvantage that if the user's private key should be known to a third person, e.g. the service provider or an attacker, this opens the possibility to decrypt all messages that have been sent to the victim's account.
In addition, because only the incoming messages can be stored, this feature should be reconsidered again, as this is only a one-sided backup. Instead, an alternative backup function could be implemented, in which the local data are stored encrypted within the mobile phone backup or a backup in an external cloud service. It is important that the decision is left to the user whether he or she wants to use this feature.

Private Key Backup

Currently, there is no feature in SEMS to backup its private keys. This means that if the device is changed, or SEMS is used on multiple devices, the user has no possibility to transfer his or her private keys to another device. In these cases, the user needs to generate new key pairs. One disadvantage is, that any sent message can not be decrypted anymore and thus be deleted from the server. This applies to already read and unread messages and can lead to the potential loss of unread data.
As an optional feature it is planned to provide a private key backup functionality. In this feature, the user should have the opportunity, to encrypt his or her private keys with a password and store it on the server of the SEMS service provider. The idea behind is to encrypt the private keys with a PBKDF2 derived key and then upload these encrypted keys to the server.
Basically, this contradicts the rule that private keys should never be given out of hand, but it is hard to provide a uniform solution for all supported operating systems (Android, iOS and Windows Phone 8). Nevertheless, it should be noted that the password, chosen by the user, has a great impact on the security of the encrypted private keys.
An alternative might be, to provide the user the ability to manage his or her private keys locally. Instead of uploading the private keys to the server in the background, one possibility would be to provide a copy and paste functionality within the SEMS application. By choosing a password the private keys will be encrypted and afterward the encrypted keys will be displayed in text format so that the user can copy and paste it to a safe location.

8. Conclusion and Outlook

8.1. Summary

This thesis treated the implementation of a Windows Phone 8 client of the secure messenger SEMS and discussed the security of the underlying protocol and the design decisions. We had a closer look at WhatsApp, the market leader of mobile messaging applications. Furthermore we had a look at new messenger with similar approaches and objectives as SEMS has. We compared the introduced messenger with the functionalities of SEMS. Also an insight into the design of SEMS was given.
With an overview of the Windows ecosystem which includes the Windows Phone 8 operating system and the description of the design concepts of the Windows Phone 8 client of SEMS, we took a closer look into the implementation of a client for Windows Phone.
Finally, with the security analysis we have uncovered a number of vulnerabilities and security flaws. In addition have critical analyzed some design decisions. Furthermore, we have identified the risks that remain when using this protocol.

8.2. Discussion

In this section we discuss the secure messenger SEMS, about the insight gained during the implementation of the Windows Phone 8 client as well as the results of the security analysis. Additionally we discuss the implementation of the client, and potential improvements.

8.2.1. SEMS

For this master thesis, the first nearly complete version of the SEMS API protocol was used. This protocol and the server as well as the client for iOS, which were developed by the three founders of SEMS, were also not completed at the beginning of this thesis and are still in development. Thus we had to make independent decisions regarding some implementation details of individual processes.
In the security analysis, some weaknesses were identified. Many of them, however, can be very easily fixed. This mainly includes vulnerabilities where the server processes the request, without checking the received data. Examples thereof are described, inter alia, in the sections 7.4.3, 7.5.1 and 7.5.1. Other problems, however, can not be easily fixed. This includes the public key distribution, discussed in Section 7.4.1, but other messengers have shown good approaches which can increase the level of security. For this case, the messenger Threema (see Section 3.2.1 in Chapter 3) offers a good approach. Threema provides, as an additional verification of the public keys, with the possibility to verify them by a QR code. The QR code is generated on the device and can be scanned by other users. Thus, the user, of the

scanned QR code, is considered to be trustworthy.
The storage of messages on the server must be considered critically. In the current server version, it stores all incoming messages. This messages will not be deleted when they were successfully delivered. To delete these messages from the server, the user must delete these messages on his or her client. This feature should be considered again, as it is only a one-sided backup of incoming messages. In addition, a permanent storage of message data is probably not desirable for a secure messenger. Furthermore, the backup of message data should be left to the user.
Missing features such as a private key backup, still need to be implemented. In the security analysis in this thesis, the originally planned key backup was already discussed. However, this was not implemented on server-side, until the completion of this thesis, or no alternatives designed.
Among the non-implemented features, in addition to the mentioned private key backup, also includes an authentication via different OAuth server provider and a registration using the phone number. A multi-device support is also not implemented. The implementation of these features has been postponed to a later release.

8.2.2. Windows Phone Client Implementation

When we started to implement the client during this thesis, Windows Phone 8 was still a relatively new operating system. Although it is the successor of Windows Phone 7, but many new approaches in Windows Phone 8 were implemented and a new operating system was created. During the implementation of the SEMS client thereby arose some challenges. In many cases, we had to fall back to documentation for Windows Phone 7 and Silverlight. In addition, we noticed that the MSDN documentation of Microsoft did not include all important information. An example would be the *HTTPWebRequest* and the behavior in the dormant state of the application. The documentation does not address the behavior of an web request with respect to the dormant state.
Since the Windows Phone operating system was still relatively new and not so widely used, such as Android, we were also very limited in the choice and usage of external libraries. For the export and import of public and private key, we had to write a separate parser since Windows and .NET does not support cryptographic keys in the required format and there were no external libraries available for Windows Phone, which could be used for this task. The only alternative would be the *BouncyCastle* library. However, this library has no direct Windows Phone support and no documentation is available for C#.

For the implementation of the client for Windows Phone, we had only a documentation of the API commands and the definition of the cryptographic algorithms. A general overview of the processes of SEMS was missing. During the ongoing implementation, many new insights into the underlying protocol and the processes of SEMS were gained. Therefore, improvements are still possible in many processes. This involves especially the process for receiving messages and the database structure.
Additionally, it should be thought about a unification of the clients. Currently there is no uniform design concept for all platforms how the different processes should be done. This applies especially to the API calls. It should be well-defined what data is stored on the device and does not need to be queried again. At present it is also not defined, whether registered users can delete their account and what should happen in such a case on the client when

trying to write a message to such a user. Also, the procedure for sending messages should be unified. Discussions with the developers of the iOS client have shown, that the client versions of iOS and Windows Phone 8 using different approaches. It would be important for the case that a message could not be sent to the server, to define a standardized behavior, so that the same user experience is guaranteed on any platform. This would be particularly important in group conversations, since they must be sent separately for each participant.

8.3. Future Work

8.3.1. Improvements and Additional Features

In addition to general improvements of the performance and processes of the application some extensions for the client are conceivable. This includes a way to delete conversations. This would require the support of an additional SEMS API command to delete the messages from the server. Also a way to backup the private keys should be added in a later version of SEMS. This could include an option for a local backup in addition to an optional backup on the server.

For push notifications the Windows Phone client of SEMS uses a tile and a toast notification. A toast Notification is used to display the number of unread messages on the Live Tile of the application. On the other hand a toast notification displays at the top of the screen and a additionally sound to notify the user of a incoming message. In addition, a possibility was introduced with the current GDR3 update of Windows Phone 8 to define a personalized sound for a toast push notification.
Besides these two types of notifications, Windows Phone 8 provides the possibility of a raw notification. Thus, the user could get an additional option with which he could disable toast notifications without disabling the receiving messages process. Raw notifications will be considered only when the application is running and can be used to receive messages without informing the user by a sound signal.

8.3.2. Additional Windows Phone 8 Features

Windows Phone 8 offers a number of new features that can be used for an enhanced user experience. In the following paragraphs, some features are discusses which would be quite conceivable for SEMS.

People Hub Integration Windows Phone 8 provides the possibility to integrate an application into the People Hub. This feature could also be implemented for SEMS. The user has to choose the desired contact and select the SEMS option in the contact's details, then the SEMS client would start and the user can begin to write the text message.

Live Tiles Live Tiles are tiles on Windows, which are displayed on the home screen of the phone. The content of this Live Tiles can be changed dynamically. Windows Phone provides three different tile sizes, as shown in Figure 8.1. With these Live Tiles it would be possible

to display some content of SEMS within the largest tile. This could be used for a preview of unread messages. Currently the Windows Phone 8 SEMS client supports only the small and the medium tile.

Figure 8.1.: Illustration of three different sizes of Iconic Tiles on Windows Phone 8.

Lock Screen Integration With Windows Phone 8 it is possible to display information of an application within the lock screen. Figure 8.2 shows a typical Windows Phone 8 lock screen structure. In the notification area of the lock screen, SEMS could display the number of unread messages. Also, a preview of the last incoming message could be displayed within the notification area.

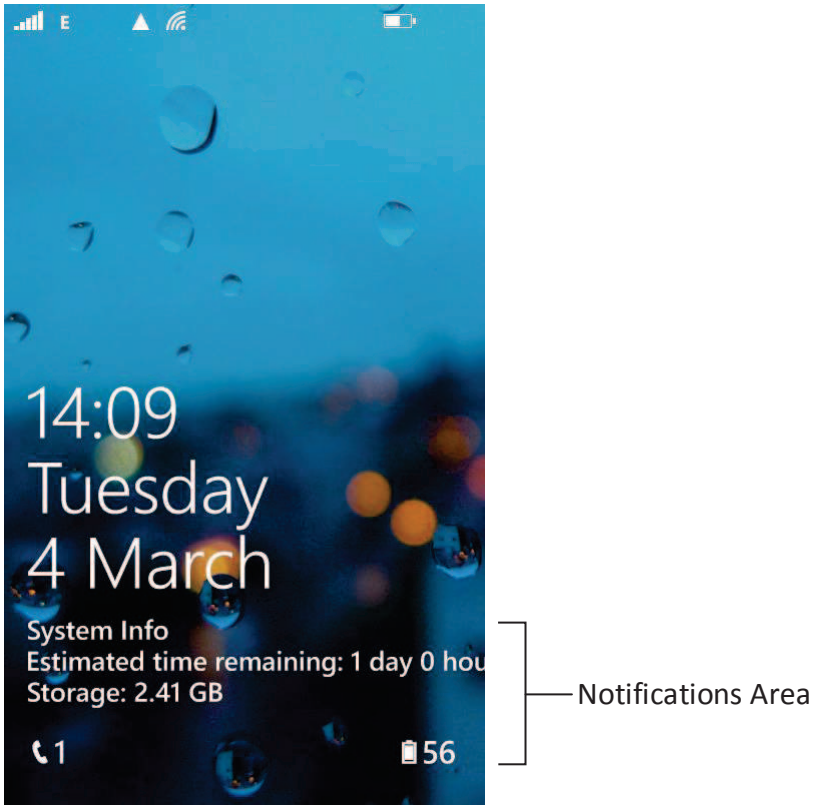

Figure 8.2.: Illustration of a lock screen on Windows Phone 8.

Bibliography

[1] Tom Van Vled; The History of Electronic Mail; URL: http://www.multicians.org/thvv/mail-history.html; Last visited on 2. December 2013

[2] The Retorist; Quantum Link for the Commodore 64; URL: http://www.retroist.com/2007/03/05/quantum-link-for-the-commodore-64/; Last visited on 2. December 2013

[3] Commodore Power/Play Magazine; October/November 1986

[4] AOL; About AOL: Overview; URL: http://corp.aol.com/about-aol/overview; Last visited on 14. February 2014

[5] ICQ; The ICQ Story; http://www.icq.com/info/story.html; Last visited on 2. December 2013

[6] XMPP Standards Foundation; History; URL: http://xmpp.org/about-xmpp/history/; Last visited on 2. December 2013

[7] El País; Inside the world of WhatsApp; URL: http://elpais.com/elpais/2012/07/09/inenglish/1341836473_977259.html; Last visited on 2. December 2013

[8] WeChat; Facebook Connect; URL: http://www.wechat.com/en/features.html#facebook; Last visited on 14. February 2014

[9] Tirus Muya Maina; Instant messaging an effective way of communication in workplace; 2013

[10] John Stone, Sarah Merrion; Instant Messaging or Instant Headache?; Symantec; 2004

[11] Algirdas Avizienis, Jean-Claude Laprie Brain Randell and Carl Landwehr; Basic Concepts and Taxonomy of Dependable and Secure Computing; IEEE Transactions on Dependable and Secure Computing; 2004

[12] Cherdantseva Y. and Hilton J.; A Reference Model of Information Assurance & Security; Eighth International Conference on Availability, Reliability and Security (ARES); 2013

[13] Eckert Claudia; IT-Sicherheit: Konzepte Verfahren Protokolle; Oldenburg Verlag; 8. Auflage; 2013

[14] Sangkyun Kim, Choon Seong Leem; Security of the internet-based instant messenger: Risks and safeguards; Internet Research, Vol. 15 Iss: 1, pp.88 98; 2005

[15] Hiroaki Kikuchi, Minako Tada, Shohachiro Nakanishi; Secure instant Messaging Protocol Preserving Confidentiality against Administrator; Dept. Of Information Media Technology Tokai University; 2004

[16] Thai Duong, Juliano Rizzo; Here Come The $\oplus$ Ninjas; May 2011

[17] Thai Duong and Juliano Rizzo; The CRIME attack; Presentation at ekoparty Security Conference; 2012.

[18] Nadhem J. AlFardan, Kenneth G. Paterson; Lucky Thirteen: Breaking the TLS and DTLS Record Protocols; February 2013

[19] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, Jacob C. N. Schuldt; On the Security of RC4 in TLS and WPA; July 2013

[20] Tsai-Yeh Tung, Laurent Lin, D. T. Lee; Pandora Messaging: An Enhanced Self-Message-Destructing Secure Instant Messaging Architecture for Mobile Devices; 26th International Conference on Advanced Information Networking and Applications Workshops; 2012

[21] Erlend Bønes et al., Risk analysis of information security in a mobile instant messaging and presence system for healthcare, International Journal of Medical Informatics (2006), doi:10.1016/j.ijmedinf.2006.06.002.

[22] The Verge; WhatsApp adds voice messaging as it hits 300 million monthly active users; URL: http://www.theverge.com/2013/8/6/4595496/whatsapp-300-million-active-users-voice-messaging-update; Last visited on 21. October 2013

[23] BGR; WhatsApp now delivers 10 billion messages each day, URL: http://bgr.com/2012/08/23/whatsapp-stats-10-billion-messages/; Last visited on 21. October 2013

[24] Koen Koning; FunXMPP; URL: https://github.com/koenk/whatspoke/wiki/FunXMPP; Last visited on 31. October 2013

[25] Brad McCarty; Signup goof leaves WhatsApp users open to account hijacking; URL: http://thenextweb.com/apps/2011/05/23/signup-goof-leaves-whatsapp-users-open-to-account-hijacking/#!pjdG5; The Next Web; Last visited on 9. December 2013

[26] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leitner, M. Mulazzani, M. Huber and E. Weippl; Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications; SBA Research gGmbH; 2011

[27] Sam Granger; WhatsApp is using IMEI number as passwords; URL: http://samgranger.com/whatsapp-is-using-imei-numbers-as-passwords/; Last visited on 7. November 2013

[28] Ezio Amodio; Whatsapp - iOS password generation; URL: http://www.ezioamodio.it/?p=29; Last visited on 7. November 2013

[29] Ezio Amodio; Whatsapp & Windows Phone; URL: http://www.ezioamodio.it/?p=49; Last visited on 7. November 2013

[30] Philipp C. Heckel; How To: Sniff the WhatsApp password from your Android phone or iPhone; URL: http://blog.philippheckel.com/2013/07/05/how-to-sniff-the-whatsapp-password-from-your-android-phone-or-iphone/; Last visited on 7. November 2013

[31] WhatsApp; Frequently Asked questions - Are my messages secure?; URL: http://www.whatsapp.com/faq/en/general/21864047; Last visited on 21. October 2013

[32] Fluhrer, Mantin and Shamir; Weaknesses in the Key Scheduling Algorithm of RC4; 2001

[33] Andreas Klein; Attacks on the RC4 stream cipher; 2008

[34] ENISA (European Union Agency for Network and Information Security); Algorithms, Key Sizes and Parameters Report - 2013 recommendations; Version 1.0; October 2013

[35] Thijs Alkemade; Piercing Through WhatsApp's Encryption; URL: https://blog.thijsalkema.de/blog/2013/10/08/piercing-through-whatsapp-s-encryption/; Last visited on 7. November 2013

[36] Thijs Alkemade; Piercing Through WhatsApp's Encryption (2); URL: https://blog.thijsalkema.de/blog/2013/10/08/piercing-through-whatsapps-encryption-2/; Last visited on 7. November 2013

[37] whistle.im; Github Repository; URL: https://github.com/whistle-im/whistle-im; Last visited on 8. November 2013

[38] whistle.im; Encryption Mechanism; URL: https://github.com/whistle-im/whistle-im/wiki/Encryption-Mechanism; Last visited on 6. November 2013

[39] Nexus; whistle.im: FaaS - Fuckup as a Service; URL: http://hannover.ccc.de/~nexus/whistle.en.html; Last visited on 31. October 2013

[40] heise Security; Krypto-Messenger whistle.im soll jetzt sicher sein; URL: http://www.heise.de/security/meldung/Krypto-Messenger-whistle-im-soll-jetzt-sicher-sein-1940415.html; Last visited on 8. November 2013

[41] MyEnigma FAQ; Do you access the data from my address / phone book? Do you store this data?; February 2014

[42] Windows Phone Dev Center; Windows Phone 8 and Windows 8 platform comparison; URL http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj681690(v=vs.105).aspx; Last visited on 4. March 2014

[43] Windows Phone Dev Center; ReaderWriterLockSlim class; URL: http://msdn.microsoft.com/en-us/library/windowsphone/develop/system.threading.readerwriterlockslim(v=vs.105).aspx; Last visited on 17. December 2013

[44] Windows Phone Dev Center; Data for Windows Phone; URL: http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402541%28v=vs.105%29.aspx#BKMK_Medialibrary; Last visited on 17. December 2013

[45] Feng Yang and Sathiamoorthy Manoharan; A security analysis of the OAuth protocol; Department of Computer Science, University of Auckland, New Zealand; 2013

[46] San-Tsai Sun and Konstantin Beznosov; The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems; Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada; 2012

A. SEMS Protocol Documentation

The following appendix lists all API commands that were used in the implementation and the security analysis. The API of SEMS has besides these, additional API commands, but they are not used at this time and still offer no additional functionality.

A.1. Authorization

- `auth/login`: This command opens up a WebView and redirects the user to the OAuth Server. The SEMS Server has access to the provided e-mail address and can create a new user, or authenticate an already existing user. The client receives as a response a session cookie and the log-in type (login, register).
  - **Request**: WebView with a request to the given URLs.
  - **Response**: On success a session cookie will be set (`session:...`). Additionally the log-in type will be set (`login:register` or `login:login`). On failure a cookie `error=unauthorized` will be returned.

A.2. Key

- `key/uploadPublicKeys`: Upload the public encryption and the public signature key. Both keys should be in Base64 encoding.
  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "encryptionKey":"MIGJAoGBALSopst...MBAAE=",
    "signingKey":"MIGJAoGBAPaB30PPx...MBAAE="
    }
    ```

  - **Response**:
    - 200 – OK
    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `key/getPublicKeys`: Get the encryption and signature key of the specified user. Return both keys in Base64 encoding.

  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "user":"26664f32-d885-44ea-b483-755eb38f2131"
    }
    ```

  - **Response**:

    - 200 – OK

      ```
      {
      "encryptionKey":"MIGJAoGBALSopst...MBAAE=",
      "signingKey":"MIGJAoGBAPaB30PPx...MBAAE="
      }
      ```

    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

A.3. Push

- `push/ios/uploadPush`: Upload a push token for iOS devices. The sound parameter indicates the name of the push sound to be played.

  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "token":"JdH\/atJqedT7RL9KxR4FlP4R+GVQQuo6m4GoH7RW2Oc=",
    "alert":true,
    "OS":"iOS",
    "sound":"default",
    "badge":true
    }
    ```

  - **Response**:

    - 200 – OK
    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `push/windows/uploadPush`: Upload the subscription URI for Windows devices.

  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "subscriptionURI":"http://sn1.notify.live.net/throttledthirdparty...
                       ...lVTU0MwMQ"
    ```

```
}
```

– **Response**:

- 200 – OK
- 401 – Unauthorized (session mismatch)
- 403 – Forbidden (else – we do not want to give any hints what error occurred.)

A.4. Message

- `msg/newConversation`: Start a new conversation or get an existing if it has already been created. The *ids* parameter contains a list of all users the conversation should contain.

  – **Request**:

  ```
  {
  "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
  "ids":["66e2e95f-c578-4075-9aab-584ed878f8b1",
         "83342daf-2ae7-421a-a74a-9630f1cb029c"]
  }
  ```

  – **Response**:

  - 200 – OK

    ```
    {
    "id":"7hsc2n94o36f@sems-conv.com"
    }
    ```

  - 401 – Unauthorized (session mismatch)
  - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `msg/sendMessage`: Send a message to the user within the specified conversation. The message string should be Base64 encoded. The preview length is max 32 characters.

  – **Request**:

  ```
  {
  "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
  "to":"66e2e95f-c578-4075-9aab-584ed878f8b1",
  "message":"SGFsbG8gU2Vtcw==",
  "conversation":"7hsc2n94o36f@sems-conv.com",
  "preview":"MessagePreview"}
  }
  ```

  – **Response**:

  - 200 – OK

```
{
"id":"nqzmtr0mwtns@sems-msg.com",
"time":1360866142000
}
```

The time is in milliseconds since 1970.

- 401 – Unauthorized (session mismatch)
- 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `msg/getMessageIDS`: Get all message IDs since specified date.
  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "since":"1360866142000"
    }
    ```

    The time is in milliseconds since 1970.
  - **Response**:
    - 200 – OK

      ```
      {
      "ids":{
            "6tawab6v2exq@sems-msg.com":"7hsc2n94o36f@sems-conv.com",
             ...,
            "8cj945njafhd@sems-msg.com":"7hsc2n94o36f@sems-conv.com"
            }
      }
      ```
    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `msg/getMessage`: Get the message with the specified ID. Returned message is in Base64 encoding.
  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "id":"8cj945njafhd@sems-msg.com"
    }
    ```
  - **Response**:
    - 200 – OK

```
{
"message":"SGFsbG8gU2Vtcw==",
"from":"f3045940-47fd-4c10-80d4-158024fca262",
"time":1360867257000,
"conversation":"7hsc2n94o36f@sems-conv.com",
"read":false
}
```

    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `msg/deleteMessage`: Delete message with the specified ID from the server.
  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "id":"7hsc2n94o36f@sems-msg.com"
    }
    ```

  - **Response**:
    - 200 – OK
    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

A.5. User

- `usr/getAccountInfo`: Returns all hashed e-mail addresses of an account for the given user. The hash values are in Base64 encoding.
  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    "user":"26664f32-d885-44ea-b483-755eb38f2131"
    }
    ```

  - **Response**:
    - 200 – OK

      ```
      {
      "ids":[
            "hd5Q6ZbX+J0yYiS+JBtyF2JknBclB98vpB99C2483uw=",
            ...,
            "hasdafjkh12387bnma/asy123e8ASDasdasd9ad2sd+="
            ]
      ```

```
}
```

- 401 – Unauthorized (session mismatch)
- 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `usr/getUser`: Returns an user ID for a given hash value (account info). The parameter *account_info* should be in Base64 encoding.
  - **Request**:

```
{
"session":"95013818-0bff-4123-b162-3c6666e7fcd8",
"account_info":"hd5Q6ZbX+J0yYiS+JBtyF2JknBclB98vpB99C2483uw="
}
```

  - **Response**:
    - 200 – OK

```
{
"user":"67da8700-e23c-4b68-a37a-2147284a841d"
}
```

    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `usr/getUserIDsConversation`: Returns the user IDs of the given conversation, excluding the own user.
  - **Request**:

```
{
"session":"95013818-0bff-4123-b162-3c6666e7fcd8",
"id":"7hsc2n94o36f@sems-conv.com"
}
```

  - **Response**:
    - 200 – OK

```
{
"ids":[
      "f3045940-47fd-4c10-80d4-158024fca262",
      "67da8700-e23c-4b68-a37a-2147284a841d"
      ]
}
```

    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)

- `usr/logout`: Log-out of the user.
  - **Request**:

    ```
    {
    "session":"95013818-0bff-4123-b162-3c6666e7fcd8",
    }
    ```

  - **Response**:
    - 200 – OK
    - 401 – Unauthorized (session mismatch)
    - 403 – Forbidden (else – we do not want to give any hints what error occurred.)