SEEPost: A Secure Electronic Messaging System
Bachelor's thesis

Anton F. Jongsma
s1686534

Dirk N. Nederveen
s1706284

July 20, 2012

Abstract

Over the last years email has become a communication medium used by almost everyone. However, the protocols involved in sending or receiving email messages don't account for security measures like message confidentiality and integrity.

In this thesis, we propose Security Enabled Electronic Post, a system of protocols for a secure asynchronous messaging system. This system reminds of email and functions for the user in the same way, but unlike email, security measures are required for proper functioning. We designed protocols that provide the general communication method and a proof of concept application that demonstrates the possibilities.

The protocols provide secure communication and secure message handling, focussing on confidentiality and data integrity. In SEEPost, the servers are given as little knowledge as possible about the contents of the messages they transmit.

The proof of concept application implements the SEEPost protocols in a transparent way, only exposing the security layers in case any error occurs.

Primary supervisor: Prof. dr. G.R. Renardel de Lavalette
Secondary supervisor: dr. F.B. Brokken

Contents

1 Introduction

Since the Internet started, Internet email use has increased to the point that almost everyone uses it daily. Often email is used to communicate privacy sensitive data like government and business communication, financial information and passwords. The risks of using email to exchange these kinds of information are not widely known with the users. An example of this risk is shown by the compromise of over two million email messages from and to several Syrian government divisions by the independent news site `WikiLeaks.org`. The compromise of these documents shows that email messages do not guarantee confidentiality. In the default case, an email message is sent in plain text: any attacker who intercepts the message can effortlessly read the message.

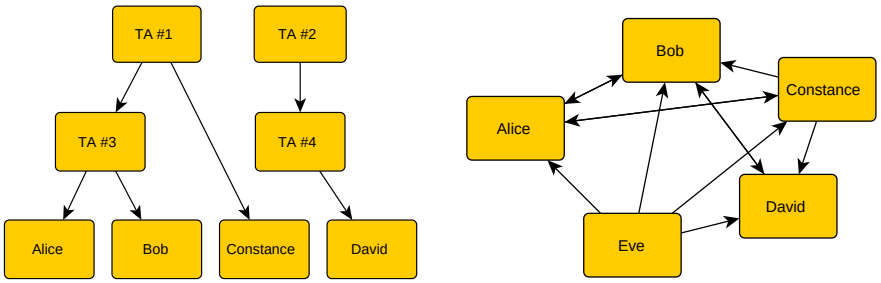
Another problem is message integrity, because the traditional email system provides no mechanism to ensure that a message remains unaltered in transit. It is not uncommon for email servers to alter the message by changing line breaking characters and adding a header line stating that the message passed that server.

In this paper, we present Security Enabled Electronic Post (SEEPost), an alternative message passing system with security concepts integrated in the communication protocols. Section 1.1 covers the history of email, some trust models that should be considered when constructing a secure system and several existing methods that add security layers to conventional email. In section 1.2 we point out system goals and how SEEPost should satisfy these goals. From the observations in Section 1.1 and the goals in Section 1.2 we continue with the proposal of a localized trust model in Section 2.1 for use in SEEPost. Section 2.2 addresses the protocol specification and Section 3 elaborates on choices that were made for the implementation of the demo application's client and server programs.

1.1 Related work

1.1.1 History of email

In 1973 Bhushan et *al.* proposed a standard for storing electronic messages that could be transported between computers connected to ARPANET [2]. The format messages are regularly stored in, Multipurpose Internet Mail Extention (MIME), was not specified until 1996 [8]. When the internet emerged in the early 1980's a protocol called Simple Mail Transport Protocol (SMTP) for handling messages between servers was published by Postel in 1982 [16]. Until 1999 there was no means of authentication for the SMTP protocol; Authenticated SMTP was invented to solve this problem [15, 20, 12]. In 1984, the Post Office Protocol (POP) protocol was invented, allowing a user to fetch messages from a server. POP added a bit of security in the form of user authentication [17]. When users access their messages via POP, these messages are usually transferred to the client program and deleted from the server. POP was followed by the Internet Mail Access Protocol (IMAP) in 1988 [4], which takes a different approach by keeping the messages on the server instead of deleting the messages after they are transferred to the client.



(a) Centralized trust model (b) Distributed trust model

Figure 1: Trust models as directed graphs

1.1.2 Trust models

The basis for a secure system is trust, because security cannot be guaranteed when the other party has not provided proof that its identity claim is trustworthy. There are several strategies to determine if the presented credentials are trustworthy. [14] For this reason, it is desirable to model these strategies as *trust models*. These trust models can be grouped in two categories, the models with a centralized structure and those with a distributed structure.

In centralized models one or more Trust Authorities (TAs) express trust towards an encryption key or identity. When the encryption key of user A is presented to user B, user B verifies the authenticity of the key if the key carries a trust expression of a TA that is acknowledged by user B. If such a trust expression exists and the key is verified, user B trusts the presented key.

Centralized trust models can also specify several layers of TAs. In that case TAs can express second order trust towards another, thereby stating "trust keys that are trusted by that TA". In Figure 1a is a simple diagram that visualises a trust graph in a centralized system with two root TAs.

In the second category, the distributed trust models, no central agent is trusted ultimately, but users express trust towards each other. Abdul-Rahman & Halles propose two elements of trust in their distributed trust model: *direct trust* and *recommender trust* [1]. An expression of direct trust from key A towards key B means that user A trusts that key B belongs to user B. Recommender trust expresses that user A recommends the trust expressions issued by user B. Figure 1b shows a trust graph according to a distributed trust model with only direct trust relations.

1.1.3 Security characteristics

For a system to be secure, it must provide several characteristics. The most important of characteristics are *confidentiality*, *integrity* and *availability*. [19]

A secure system should provide confidentiality, meaning that the messages cannot be read by anyone other than the intended recipient. Integrity means that messages that are transmitted can be guaranteed to originate from the correct sender and are not altered during transport. Availability is harder to define, but includes that if some participants in a system fail in any way, the system keeps functioning.

A fourth important characteristic is *usability*, because if the security system is difficult to use, the chance that the system is used in an insecure way rises. Whitten and Garfinkel show from several studies [21, 11] that many security failures are the result of users who make mistakes in handling the system, instead of faulty protocols or weak keys. From these studies we conclude that usability is a critical element in a secure communication system.

1.1.4 Existing security measures

In 1991 Zimmerman introduced Pretty Good Privacy (PGP) as a way to securely exchange email [22], providing confidentiality and message integrity to users of the system. PGP has a distributed trust model very much alike the distributed trust model described by Abdul-Rahman & Halles [1]. In 1993 Kent et *al.* published Privacy Enhanced Mail (PEM) [13] a method of exchanging mail in a secure fashion. PEM was never widely used, probably because its design relied on a single central Trusted Authority [10]. It was not until 1998 that Secure MIME (S/MIME) was proposed [7] as an extension to the MIME protocol. S/MIME uses X.509 certificates like the Secure Sockets Layer (SSL) and its successor Transport Layer Security (TLS) that are widely used in secure internet connections and uses a centralized trust model.

PGP and S/MIME are still used for the exchange of secured email, but although usage figures are unknown, we estimate that these systems are only marginally used.

1.2 Motivation

Email is widely used to exchange sensitive data: financial details, contracts and other private information, but the email system provides no intrinsic security aspects like confidentiality and integrity. This information is interesting for attackers, who have several possibilities to gain access to the information. The different types of attacks are distinguished in two classes, the passive attacks and the active attacks.

Passive attacks allow an attacker to read parts of the communication or all of it. This is regularly called eavesdropping, and compromises the confidentiality of the communication.

Another kind of attack is the active eavesdropping attack or man-in-the-middle attack, similar to the eavesdropper but with the difference that the attacker has the ability to insert, remove or alter messages without being detected by one or more of the communicating parties, in that way compromising the confidentiality and integrity of the system. A third way a communication system can be attacked, is an impersonation attack, where the attacker falsely identifies themselves as a legit user and sending messages under the name of the victim and make recipients believe the message is genuine. With this attack, the integrity is compromised, because false information is presented as being true. Impostors may use various strategies to gain access to computer systems. The simplest way for an impostor to gain access to a system is by forging credentials of a legit user, e.g. by guessing the password of another user. Another way of becoming a successful impostor is the replay attack, in which an attacker can gain access to the system by repeating a legit authentication sequence that was captured previously.

A denial of service attack aims to make a service unavailable to legitimate users. A common method to accomplish this is flooding the server running the service with more traffic than it can handle.

In a secure system these attacks are very hard or even impossible, rendering it uninteresting for an attacker to gain access to the system.

OpenPGP and S/MIME provide several mechanisms for confidentiality and integrity, but they seem to be only marginally used. This marginal use adds to the chance that an attacker successfully forges a message without the forgery being detected, because it is uncommon for email programs to give a warning at the absence of security measures. This implies that no warnings are given when an attacker forges a message without these security measures.

OpenPGP and S/MIME both have other properties that may inhibit users from using it securely. OpenPGP provides a fine-grained key management system, in which end users are responsible for obtaining and verifying public keys of other users. This bookkeeping is mandatory for both the sending and the receiving party in order to exchange secured messages, and this extra work could be a factor that keeps users from exchanging secured messages with OpenPGP. S/MIME has the problem that the keys in the system must be validated by a TA, a process that raises the cost for users who start to use the system.

For these reasons, we chose not to design a security layer on top of email, but to create a set of protocols that provide secure messaging over insecure connections. In this way, the security layers are not an optional addition to the communication protocol, but an integral part of it. Omitting security measures therefore should be a violation of the protocol, resulting in the violating message being ignored.

We will apply several security techniques in order to achieve a system which provides security for all its users in a way that is easy to use without technical knowledge while making it hard for attackers to compromise the confidentiality, integrity and availability of the system. These techniques can be summarized as *strong encryption*, *simple protocols* and *trusted connections*.

1.2.1 Strong encryption

If a system needs to be secure, it is essential that the security layers are easy to use. One approach is to enforce the use of strong cryptographic encryption in all communication by disallowing unencrypted connections in the protocol. This will prevent that implementations provide weak encryption or no encryption at all, for those implementations will not be able to connect to the network. The use of strong encryption schemes prevents attackers from eavesdropping on connections and from altering any of the communication without being noticed.

Transparent encryption The cryptographic encryption should be as transparent as possible to make sure an end user will be warned only when an insecure situation occurs. A situation like this could occur when another agent demands to use a weak encryption scheme or any other recoverable error. Transparent

encryption means that the encryption is applied without requiring special action from the user, because properly applied encryption should be the default situation in the SEEPost system and therefore should not attract the attention of a user. Notifications and warnings should only be given in cases where the confidentiality of the communication cannot be guaranteed because strong encryption cannot be applied. Connections with weak or absent encryption should be regarded as communication with an attacker and therefore as untrusted connections. End user implementations should not try to continue with an untrusted connection, as a safeguard against security compromise.

1.2.2 Simple protocols

Another important feature is the use of a small set of simple protocols, so that it is easy to create a correct implementation. Current email implementations differ from each other in handling, forcing different implementations to adapt to the errors of other implementations. The protocols should therefore not be ambiguous to make sure all implementations can interoperate with each other.

Current email clients need to support at least two protocols, one for sending mail and one for receiving mail, the connections for these protocols are authenticated individually. Although this allows for greater flexibility in authentication, there are few situations that require different credentials for sending messages and receiving messages. SEEPost should allow sending and receiving messages over a single connection, to simplify the protocol.

The authentication protocol should provide methods against impersonation attacks by requiring a different authentication sequence every time, preventing replay attacks, and should require proof of credentials without exchanging the credentials themselves as to eavesdropping on a password. Furthermore the credentials should be too large for a human to generate and remember, to prevent weak credentials that are easy to forge.

1.2.3 Trusted connections

Conventional email programs implicitly trust any other host that connects to the program, because the other agent can comply with the protocols without applying any security measures. This is because conventional email was developed to allow users of a single computer system to send messages to each other, and was extended later to allow for messaging between computers. The email system was intended for use within a trusted environment like a mainframe within a company, but was scaled without modification to an untrusted environment, i.e. the Internet. Some security measures were added to the email system, but for the system to function each implementation must accept all connections that conform to the protocol, regardless of security measures. By including security measures like integrity checks for messages the complete chain of hosts can be verified and it can be assured that no host has altered the message in any way, this prevents eavesdropping and man-in-the-middle attacks.

2 SEEPost system characteristics

2.1 Localized trust

To create an automated system little to no human effort should be required for the system to function. To achieve this, key exchange and key assessment should be done as much as possible in an automated fashion.

In a centralized trust model, it is necessary for a user to find a TA that expresses trust towards the key of the user. A distributed trust model requires the user to verify the keys of people they communicate with. Both models require in-depth knowledge and some action of the user to participate in the system. This makes sending secured messages a more complex task than is necessary, which may affect the secure handling of messages in a negative way. Since maintaining an automated system is a different task than participating in one, another level of in-depth knowledge may be required of system maintainers than may be required from users of the system. We believe that more in-depth knowledge of the system may be expected from maintainers and that persons with more in-depth knowledge of a system use the system more correctly.

Under these assumptions, a trust model should be formulated in which users of the system do not need in-depth knowledge to use the system without error and system maintainers have the responsibility of key exchange and key assessment.

We propose a layered trust model that combines features from the centralized trust model and the distributed trust model. In the localized trust model we propose the servers exchange keys with each other via a centralized system, whereas the account keys are exchanged with a distributed approach.

In this model, each account is present on exactly one server. It is assumed that the account holder and the corresponding server know each other's public key, this assumption is feasible because service provider and account holder can exchange keys upon account creation.

2.1.1 Server key exchange

When two servers connect to each other for the first time, they do a simple key exchange where both hosts send their public keys towards each other. In a naive implementation this would allow an attacker, using a man-in-the-middle attack, to substitute the real keys with their own fake key. To avoid this threat, SEEPost servers should publish a fingerprint of their public key in the Domain Name System (DNS). With that fingerprint, each server can verify that the key that they received is genuine and can be used to set up a trusted connection. This leaves open the possibility for a DNS cache poisoning attack, where an attacker substitutes DNS information with their own fake addresses and key fingerprints. To avoid this threat, it is highly desirable to protect the DNS information with the DNS Security Extensions (DNSSEC), which eliminate the possibility for attackers to insert false information in DNS servers.

The server key exchange is an implementation of a centralized trust model, with the DNSSEC system and ultimately the private key of the DNS root zone as central Trusted Authority.

2.1.2 Account key exchange

Under the assumption that any two servers can set up a trusted connection, it becomes trivial to exchange account keys. In the localized trust model each server is responsible for the distribution of public keys belonging to all accounts present on that server. The sender of a message delegates the responsibility for obtaining the required keys to their corresponding server. The server relays the key request to the server that is responsible for the distribution of the recipient's account key. This method ensures that all key requests can be satisfied with at most two lookup operations.

2.1.3 Conclusion

These properties satisfy the goals of the localized trust model, as the user only need to exchange keys with the server that provides access to the system for that user. The server maintainer is responsible for obtaining expressions of trust towards the server they maintain.

2.2 Protocols

To accomplish the security of a protocol set, the protocols should be designed with the security characteristics *confidentiality*, *integrity*, *availability* and *usability* in mind; this section explains how we designed the SEEPost protocols and how these protocols provide the necessary security characteristics.

2.2.1 A minimalist approach

To accomodate *availability* and *usability*, SEEPost's design is kept as simple as possible and has two main components: the transport layer and the communication protocol between hosts. This division of components allows the implementation to have clearly defined functions with little uncommon combinations of functions. This minimalist design accomodates testability and with a minimalist design implementation errors are more likely to be detected and avoided. When a program has few implementation errors, the usability will be better, because users quickly get accustomed to the system and are not required to understand many different situations.

The functions for message handling do not rely on many external functions or services. This increases availability, as many components keep functioning correctly when other components fail.

2.2.2 Communication between hosts

The communication protocols cover the aspects of confidentiality and integrity. To provide confidentiality, the communication should be protected with strong encryption involving perfect forward secrecy. Each user of the system has a unique key pair and all messages that are sent should be encrypted with the recipient's public key to ensure that only the intended recipient can read the messages.

Every connection between two hosts in the system starts with a session key exchange via a mutually-authenticated Diffie-Hellman Key Exchange protocol. This protocol provides confidentiality by requiring the connecting parties to

authenticate themselves. The Diffie-Hellman Key Exchange protocol allows two parties to negotiate a shared secret in such a way that even if an attacker can read all communication between the parties, it is computationally hard for the attacker to calculate the shared secret [5]. Both hosts should remove the Diffie-Hellman parameters after the handshake, doing so guarantees full perfect forward secrecy, meaning that if in the future an attacker can access the decryption keys of both parties and the communication, the attacker can not decrypt the communication. Section 2.2.4 covers this protocol part in more detail.

Every message sent to another end user of the system should be encrypted using public-key cryptography. Even if an attacker is somehow able to intercept the communication between client and server, the confidentiality of the communication is ensured as the messages cannot be read by the attacker.

Messages should be stored in an encrypted container to prevent that an attacker with access to the server can read the messages. In this way an attacker will not be able to determine with whom a user has communicated previously, nor the contents of the communication.

SEEPost also ensures the integrity of messages by adding cryptographic signatures to messages.

1. Every message to another end user should be timestamped to avoid replay attacks.
2. Every message to another end user should be cryptographically signed to ensure data integrity and authenticity.

2.2.3 Protocol diagrams

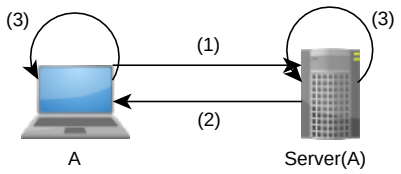
Notation Throughout this section we use the following notation:

A, B	accounts
PK_A	Public key of account A
SK_A	Secret key of account A
g, p, a, b	Diffie-Hellman parameters
k	Session key
$ASC(PK, m)$	Asymmetric encryption of message m with public key PK
KDF	Key derivation function

In the diagrams a variety of braces are used, angular braces denote tuples or lists of values, whereas parentheses denote the argument list for functions.

2.2.4 Authentication

Prior to manipulating messages, the user with account A should get authenticated with server S_A in a way similar to the STS protocol as described by Diffie et *al.* [6]. The hosts already know each other's public key, therefore all messages



(1) Send:
$ASC(PK_{S_A}, \langle A, p, g, g^a \pmod p \rangle)$

(2) Send:
$ASC(PK_A, \langle g^b \pmod p \rangle)$

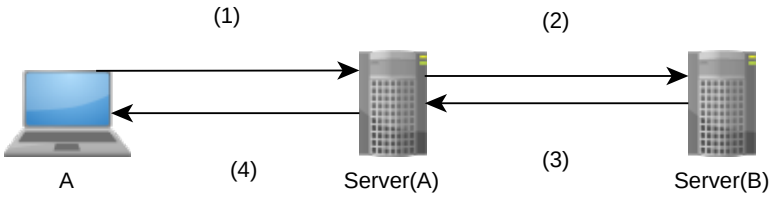
(3) A and S_A both compute
$k = KDF(g^{a*b})$

Figure 2: The authentication protocol

are encrypted with a public-key cryptographic algorithm. The prerequisites for this authentication protocol are that A knows PK_{S_A} and S_A knows PK_A.

The authentication protocol is drawn in Figure 2.

Send message Another important facility is sending messages between end points. Suppose A sends message m to B via S_A. This process is visualized in Figure 3, each communication channel uses a symmetric encryption as negotiated in Section 2.2.4:



(1) $M = \left\langle B, ASC(PK_B, \langle m, ASC(SK_A, m) \rangle) \right\rangle$
M is the message envelope containing the addressee (B), the signed message m ($\langle m, ASC(SK_A, m) \rangle$), which is encrypted to B.

(2) M

(3) Response r: confirmation that the message is delivered if B is present at Server(B), or an error message.

(4) Response r

Figure 3: The protocol for sending a message

2.2.5 Retrieval of account public keys

Under the assumption that servers can establish a secure connection with each other, the retrieval of account public keys is relatively simple. The protocol is visualized in Figure 4 on page 12.

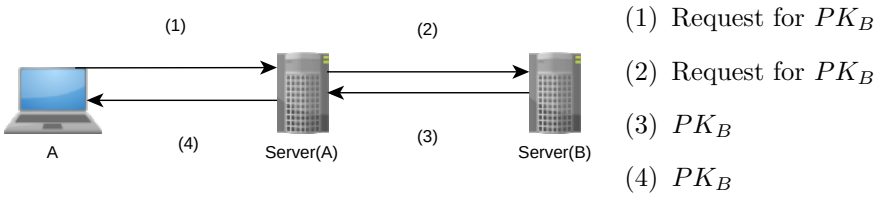


Figure 4: The protocol for retrieving an account public key

2.2.6 Key retrieval of servers

The most difficult security characteristic is authentication, because exchanging keys is an intrinsically hard task if there has been no prior communication between two parties. SEEPost specifies a Localized Trust model that has two levels in the trust hierarchy. In this model, each host, client and server alike, has its own key pair, but only the retrieval of end user keys is specified in the protocol. The first level is the level of inter-server trust. To set up a trusted path, each connection in that path should be mutually authenticated. SEEPost intentionally has defined no specific method of interserver trust, to allow implementations to use a strong basis of trust. There are several techniques to exchange keys with unfamiliar hosts.

- Galvin proposes an infrastructure for key distribution via secure DNS [9]. The drawback of this method is that the DNS KEY record has a limited length, which makes it impossible to exchange keys of arbitrary length.
- Servers could exchange keys at the start of a connection in a way similar to the SSH protocol. In common SSH implementations the key verification is a manual process, which makes it useless for interserver key exchange. With the key fingerprints in a DNS record similar to the SSHFP record [18] and the DNS record protected with DNSSEC, this process can be automated.

The second level is trust between server and client. Each account is associated with a server and the account holder has a trust relation with the person or organization who provides the server. To set up a trusted connection, SEEPost mandates that account holder and server provider have exchanged keys via a trusted channel beforehand. In the Localized Trust model each server is responsible for distributing the public keys of the accounts present on that server. This model provides a two-link trust chain in which a trusted path can be established with the receiving server and the key of the receiving user can be obtained via that trusted path.

3 Implementation

To demonstrate the SEEPost system we implemented it in two demo programs, a client program and a server program. In this section we will first explain how the different parts of the system where implemented. Second we explain both programs in more detail. The source code can be found at `https://github.com/felrood/seepost`.

3.1 Building blocks

The SEEPost system has four major parts, the Transport Layer, the Protocol, the Storage system and the message format. These form the building blocks for the client and server programs.

This section explains the commands that are exchanged between two hosts in the SEEPost network, throughout these commands the words between anglular brackets should be replaced with the correct value. Words between square brackets are optional values, that may be omitted in the command.

3.1.1 Transport Layer

The transport layer provides a secure communication channel and simultaneously authenticates both parties. The connection can be initiated by both the client and server program. The listing program is always a server program.

The protocol for establishing a shared secret is as follows:

The client sends the SEEPost header. In the header both the server name and the client name are stated. A version number is used to allow revisions of the protocol. The header format:

`SEEPOST <seepost version> HELLO <server name> AUTH <client name>`

The server should return `ERROR <error code> <reason>` if it is not the server stated in the header, the client has no account on the server or the protocol version is not supported. Otherwise `OK` should be returned.

The client then sends his Diffie-Hellman parameters encrypted with the server public key, and the server sends his parameters to the client, encrypted with the public key of the client.

For this we used the Botan C++ encryption library[1]. This library implements Diffie-Hellman Key Exchange with the `DL_Group` and `PK_Key_Agreement` classes.

The secret key and initialization vector are then derived from the session key with the *KDF2(SHA-256)* protocol. All communication from this point on is then encrypted with AES using the secret key.

3.1.2 Server key retrieval

During the handshake we assume that both parties know each others public key in advance. In case of client to server communication this condition may be assumed. With server to server communication this might not be true. It could happen that a server needs to deliver a message to a server it has never seen before. For this we implemented a simple key lookup service. This service does not use a secure connection, which leaves opportunity for a man-in-the-middle attack.

The service supports only two commands:

- `GET <hostname>`
- `QUIT <reason>`

[1] `http://botan.randombit.net/`

The response to the `GET` command should be `FOUND <public key>` if the public key that is asked for is indeed the one of the server, otherwise `NOTFOUND` is returned. This syntax differs from responses `OK <public key>` and `ERROR <error code> <reason>` due to historical reasons. Using the `OK`/`ERROR` syntax also in this situation would be more consistent and elegant.

3.1.3 Protocol

The protocol specifies the format of commands that may be issued to a server. The commands are always initiated from the client side of the connection, and a single response is given back. In this protocol `identifier` means an hexadecimal encoded unique identifier. The term *blob* is used for a piece of encrypted data of arbitrary length for the storage system. The commands can be divided in four categories.

General commands

- `ERROR <error code> <reason>`, Indicate an error occurred
- `QUIT [reason]`, Quit

Queue manipulation

- `PEEK [identifier]`, Retrieve message `[identifier]` from the queue without removing it. If the value `[identifier]` is omitted, the first element of the queue is retrieved.
- `DROP <identifier>`, Drop message <identifier> from the queue
- `LENGTH`, Query the length of the queue

Storage manipulation

- `GET <identifier>`, Get the *blob* <identifier> from the data storage
- `PUT <identifier> <data>`, Store <data> at *blob* <identifier> in the data storage
- `DELETE <identifier>`, Delete *blob* <identifier>
- `LIST`, List all the keys in the data storage

Key retrieval and sending a message

- `SEND <address> <message>`, Send message <message> to <address>
- `ENCPUBKEY <address>`, Retrieve the encryption public key from <address>
- `SIGNPUBKEY <address>`, Retrieve the sign public key from <address>

In case of a client - server connection all of the above commands are available for the client. In case of a server to server connection only the *general commands* and *Key retrieval and sending a message* commands are available. This makes the server to server protocol a subset of the client to server protocol.

The response to a command is either `OK [data]` if the command succeded or `ERROR <error code> <reason>` if an error occurred. For some commands the syntax for the `OK` response is actually a little different due to historical reasons. To increase consistency this should be unified to `OK [data]`.

3.1.4 Storage system

The storage system of SEEPost provides a permanent storage facility for the client and transparently handles the encryption of its contents. The server-side part of the storage system is a simple key-value store. The commands `GET`, `PUT`, `DELETE` and `LIST` from the protocol are used for manipulating this key-value store. The values, also known as *blob*s, are encrypted client-side before being stored on the server.

The AES-256 encryption key for the data is encrypted with the users public key and signed with its private key using the RSA algorithm. This ensures that only the user can read the storage key and can verify that he himself only could have created this key. Since the user is the only entity with knowledge of the key and is sure that they created that key themselves, the encryption and signature on the storage provide both confidentiality and data integrity, because any alteration in the data makes the data unreadable.

Inside this key-value store resides a minimalistic file system like structure to store messages efficiently, known as the *storage format*. The current version of the format can only store recieved messages, it does not support common features like folders and storing attachments.

Because the storage format is seperate from the storage system on the server-side, it can store arbitrary information without any server modifications and is therefore not restricted to concepts like messages or attachments. For example, some sort of index could be stored to implement fast search queries, or an addressbook with adresses and corresponding public keys. Due to time contraints we did not implement this.

The storage system supports these functions:

- List messages
- Retrieve message contents
- Store a message
- Delete a message
- Wipe all messages

This last function is mainly used for debugging purposes.

The storage format has several *blob*s with a special purpose.

- 0, *blob* containing the version number in plain text and the storage encryption key, itself encrypted to the owner
- 1, list of index *blob*s
- 2, last message identifier
- 3, usualy the first index *blob*

Blob 1 is a meta-index. It contains a reference to all index *blob*s. This was implemented so the size of a index *blob* would not grow too large. This reduces the transmission overhead of having to downloading the whole message list each time the client connects.

Blob 2 contains the number of the last message. When a message is stored it is assigned a unique number to identify the message.

An index *blob* has a newline seperated `key0=value0;key1=value1;...` format. Each line represents a message. All headers of a message are stored in such a single line. The mandatory keys are:

- `id`, number uniquely identifing the message
- `body`, *blob* id of the message body
- `from`, address of the sender
- `name`, name of the sender
- `subject`, subject line
- `timestamp`, an RFC 2822 timestamp
- `status`, status field. U for unread, R for read.
- `from_server`, the server from which the message originated
- `server_timestamp`, RFC 2822 timestamp of when the server recieved the message
- `digest`, the signature hexadecimal encoded
- `sign_verified`, an RFC 2822 timestamp of when the signature was last verified

Storing a message When a message is added to the structure the message headers are appended to the last index *blob*. If this *blob* then exceeds a certain length or size a new index *blob* is created.

3.1.5 Message format

The message format specifies how a message should be formatted when send from one client to another. A message consists of a header line, the message contents and a hexadecimal encoded signature. This message is then encrypted with the AES-256 algorithm and passed in a hexadecimal encoding. The encryption key is encrypted to the recipient with the RSA algorithm and prepended to the encrypted message.

A more formal definition:

```
envelope := <encryption key> <newline> <encrypted message>
message := <header> <newline> <message content> <newline> <signature>
header := <key> '=' <value> ';' <header>
```

The header line has some required fields:

- `from`, The senders address
- `name`, The name of the sender
- `subject`, Subject line
- `timestamp`, An RFC 2822 timestamp

If these fields are not present the message is invalid and will be dropped by the client.

When the message is put on the queue, the server prepends it with a header line with the following keys:

- `from_server`, the server from which the message originated
- `timestamp_server`, an RFC 2822 timestamp when the server recieved the message

This adds some information for the client program to check the validity of the message, although in the current demo implementation this is not implemented.

3.2 Client program

The client program uses the building blocks which we described in the previous sections and adds an interface to make a usable program.

3.2.1 User Interaction

The client program has a commandline interface to interact with humans, due to time constraints a GUI was not implemented. However, since the commandline interface has only a few easy to understand commands, we still consider it userfriendly.

The demo client supports the following commands:

- `help`, help message
- `retr`, retrieve new messages

- `list [length]`, show a list of the messages in the inbox
- `write`, write a message
- `read <identifier>`, read a message
- `delete <identifier>`, remove message
- `info`, show some info about the system
- `wipe`, destroys the storage and initializes an new one
- `quit`, quit client

A typical interaction with a user is shown in Appendix B. The example program exposes no security or cryptograpy related terms to the user. As the program handles security measures transparently, user security errors are prevented [21, 11].

Dealing with attacks When an attacker attempts to perform a man-in-the-middle attack between the client and the corresponding server, the client cannot establish an connection, because the attacker cannot present a valid response to the Diffie-Hellman parameters the client sent. The user is presented an error message stating: `An error occurred: cannot establish connection`.

If the user receives an message with an invalid signature or with some of the required header fields missing the message is discarded by the client program and the user is presented a messages stating `Signature did NOT verify, discarding message` or `Corrupted message received, discarding`.

3.2.2 Keyfile

The private key of the user and some configuration data is stored in a configuration file. This keyfile can be secured with a passphrase. In this case we have two-factor authentication: both the passphrase and the configuration file are needed to connect to the server.

The data stored in the configuration file:

- `name`, the users own name
- `address`, the users own address
- `server`, the address of the server
- `port`, the servers port
- `sign_pub`, the public key for signing
- `sign_priv`, the private key for signing
- `enc_pub`, the public key for encrypting
- `enc_priv`, the private key for encrypting
- `server_pub`, the public key of the server

3.3 Server program

The server program consists of three actual servers. The `clientserver`, the `interserverserver` and the `keyserver`.

3.3.1 Clientserver

This server listens on port 9011 and serves clients. It implements the full protocol as explained in Section 3.1.3. This is the most important server and the largest in lines of code.

3.3.2 Interserverserver

This server listens on port 9010 and it serves other servers that connect to it. It implements only a subset of the whole protocol as explained in Section 3.1.3. In retrospect it would have been possible to merge the `clientserver` and `interserverserver`, but we realised this too late. Both servers implement the whole or a part of the protocol, so they share a lot of functionallity.

3.3.3 Keyserver

The `keyserver` is very simple. Its purpose is to make the public key of the server freely available, as explained in Section 3.1.2 it only has to deal with two commands. Also, the connection is in plaintext. The server listens on port 9012.

3.3.4 Configuration and data storage

The server program stores its configuration file and all its data in a directory. The public key and some other configuration data from all its users is in the subdirectory `accounts`. The storage system is located in `blobstore/<address>`. The queue for incoming messages is stored in `inqueue/<address>`

4 Conclusion

In Section 1.1.3 the security characteristics *confidentiality*, *integrity*, *availability* and *usability* were identified, this section will evaluate the SEEPost protocols and programs with respect to these characteristics.

Confidentiality The SEEPost protocols provide confidentiality in different parts of the design, most importantly in the connection between any two hosts and in the transport of messages between users. Each connection starts with the authentication protocol as described in 2.2.4, which also provides a session key exchange to set up an encrypted channel which attackers can not decrypt. This way the confidentiality of all communication in the network is guaranteed.

In the demo program this part of the protocol is implemented in the `connection` classes, which perform the key exchange and send all further communications encrypted with the AES-256 algorithm.

As an extra layer of confidentiality the messages are encrypted in such a way that only the recipient can read the messages, so even if attackers are able

to read the communication between two hosts, they can not read the messages that are being transferred.

The `clientproto` class performs this function transparently. When the function for sending a message is called, the recipient's key is looked up and the message is encrypted to that recipient. After the encryption, the message is sent to the recipient.

The storage system provides confidentiality, as users store all information encrypted to themselves, only the correct user can decrypt the stored information. This prevents any attacker with access to the stored information from obtaining the decrypted information.

The demo program implements this part of the protocols in the `clientfs` class, which applies all encryption and decryption operations transparently.

Integrity The protocols provide message integrity as each message that is sent to another user is cryptographically signed, this way any modification to the message will be detected.

The `send` function in the `clientproto` class adds this signature to the message before encrypting and sending the message. On the receiving end, the `verifysignature` function checks if the signature that is found on the message is correct and belongs to the sender.

In the storage system, the encryption key is signed by the client and checked each time the encryption key is requested. This guarantees that the stored information cannot be replaced with false information without being detected.

Availability The security characteristic *availability* is provided by the fact that no third host is needed if two hosts connect to each other. This means that if a server stops functioning, all communication continues in the normal way, except for the communication with that server.

Usability The demo program tries to provide usability by hiding all security measures for the user. If the SEEPost system functions as intended, the user only sees plain text messages and no information about keys and encryption. When the system cannot function as intended, only error messages are shown.

5 Discussion

In this section we discuss some of the fundamental assumptions that were made for the SEEPost system, the weaknesses we see in the protocol design and some topics that require further research.

5.1 Fundamental assumptions

5.1.1 Trust between account holder and server

The Localized Trust model assumes the account holder has a trust relation with their server. This trust relation is necessary, because the server is responsible for the distribution of the account holder's public key and keeping their data. This

trust relation is also feasible, because having someone else provide a service in which they play an active part requires that the server knows where the messages must be sent to.

5.1.2 Use of DNSSEC

The demo implementation does not use DNSSEC due to the lack of DNSSEC on the testing network. For DNSSEC to work widespread coverage is required, which is not yet the case. Until then we cannot depend on it, this makes our design much weaker.

5.2 SEEPost weaknesses

5.2.1 Self-designed security protocol

Although it is generally considered a bad idea to "roll your own" security algorithm [3], we chose to diverge from existing secure messaging protocols. This decision was made to avoid the complex SSL and TLS protocols and the Centralized Trust model that comes with X.509 certificates. The complexity makes it difficult to implement the protocols correctly, which would easily result in security errors.

5.2.2 Transport layer protocol

In retrospect, designing a secure transport layer protocol was harder than we anticipated. There are many important details which we overlooked when we choose not to use TLS. Next time we would consider TLS and X.509 more carefully. Using a different PKI for X.509 certificates would drop one of our main objections against using TLS.

5.2.3 Cipher choice

For the transport layer we choose the block cipher AES for the encryption. Being a block cipher this caused quite some problems with properly padding variable length data. In the current implementation the same plaintext data results in the same cipher texts send over the network. This leaks some information about the higher level protocol to an eavesdropper. Using a stream cipher would have been a more natural choice. With stream ciphers padding is not needed and sending a plain text twice would not result in the same cipher text.

5.2.4 Trust model

The centralized trust model poses a problem, because when a Trust Authority is compromised, no certificate can be trusted. The attacker who compromised the TA could express trust toward any certificate, deceiving all parties that are presented that certificate, since it is trusted by a TA.[2] When a server key is compromised in the localized trust model, all communication with that server may be considered compromised, but communication with other servers is unaffected.

[2]This happened in 2011 with the Dutch X.509 Certificate Authority DigiNotar

5.2.5 Server compromise

In the event that a server is compromised by an attacker, the possible damage that can be done is limited by the SEEPost design. When an attacker is in control of a server all communication from and to that server is potentially compromised, as the attacker may be able to insert their own malevolent software. This means that the attacker can have access to all communication starting from that point in time. For example, any public key that is requested from that server may be replaced by a non-authentic attacker key. This would compromise the messages send to clients of that server, as well as messages send by any of its clients.

The past communication of the server remains secret due to the perfect forward secrecy characteristic of the Diffie Hellman key exchange.

The stored messages of all the clients also remain secret, because the encryption and decryption of storage is performed by the client application.

5.3 Further research

In developing this system we came across several limitations that require further research.

5.3.1 Shared inboxes

The files on the server are encrypted with the public key of a single user. This make inbox sharing impossible. To make inbox sharing possible, a shared key must be established. We think this is possible but would require extension of the protocol.

5.3.2 Webmail

A webmail client is not trivial to implement with the same level of security because the private key is needed to authenticate the connection and to read messages. This means the user should upload its private key to the webserver, which we consider risky behaviour. Another way would be to store the private key always on the webserver, and securing it with a passphrase. This way the private key is never transferred over the internet, but it makes the authentication process weaker. An attacker only has to guess the passphrase, a token (the private key) is already available.

5.3.3 Server side search

The contents of the inbox are encrypted so the server can't read the contents. This makes server side search impossible. This makes searching through your messages very slow because all content needs to be downloaded from the server, decrypted and then inspected for the search query. A possible solution is storing an index of some sort on the server. The client could download this once and use it for searching. This could be implemented completely in the client program.

5.3.4 Unreachable server

Even if a server would only need to send outgoing messages, it needs to be reachable from the outside. This is a result of the server key lookup as explained in Section 3.1.2.

We could remove this problem by incorporating the key exchange in the handshake. The downside of that approach would be that the client–server handshake would differ from server–server handshake, making the protocol more complex and making correct implementation harder.

5.3.5 Key revocation

The current implementation has no method for key revocation. Key revocation should be present for both client keys and server keys.

If the private key of a client is compromised by an attacker or the key is lost, some method should exist change the compromised keypair and replace it with a new keypair. In the demo implementation of the client program, the public key of a recipient is not cached. This more or less compensates for the lack of a key revocation mechanism. However, implementing an addressbook function in the storage system was an original goal early in the project.

A key revocation method for server keys is also not present. Server keys are cached by other serves, so this is a much bigger problem than with client keys. If a server changes its publickey because of an compromise, it becomes impossible for other servers to establish an connection if they have the old publickey in cache.

A possible method would be a "freshness check" in the handshake. The connecting party now only specifies his own name and the name of the party it want to connect to. It could also specify the fingerprint of the key it has in cache, as well as the fingerprint of his own current keys. The receiving party can then detect if it has a different key of the connecting party, or the connecting party has his old key still in cache.

This would however create a new possibility for an attacker, who could successfully perform a man-in-the-middle attack by claiming the other party has an old key. Verifying the new key with DNSSEC or a different Central Trust Authority could solve this.

References

[1] Alfarez Abdul-Rahman and Stephen Hailes. A distributed trust model. In *Proceedings of the 1997 workshop on New security paradigms*, NSPW '97, pages 48–60, New York, NY, USA, 1997. ACM.

[2] A.K. Bhushan, K.T. Pogran, R.S. Tomlinson, and J.E. White. Standardizing Network Mail Headers. RFC 561, September 1973. Updated by RFC 680.

[3] Wm. Arthur Conklin and Glenn Dietrich. Secure software engineering: A new paradigm. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, page 272, jan. 2007.

[4] M.R. Crispin. Interactive Mail Access Protocol: Version 2. RFC 1064, July 1988. Obsoleted by RFCs 1176, 1203.

[5] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[6] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Des. Codes Cryptography*, 2(2):107–125, June 1992.

[7] S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, and L. Repka. S/MIME Version 2 Message Specification. RFC 2311 (Historic), March 1998.

[8] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. Updated by RFCs 2184, 2231, 5335, 6532.

[9] J.M. Galvin. Public key distribution with secure DNS. In *Proceedings of the 6th USENIX Security Symposium*, pages 161–170, 1996.

[10] S. Garfinkel. Signed, sealed and delivered. *CSO Online*, 2004. http://www.csoonline.com/article/219173/email-security-signed--sealed-and-delivered-, accessed July 4, 2012.

[11] Simson L. Garfinkel and Robert C. Miller. Johnny 2: a user test of key continuity management with s/mime and outlook express. In *Proceedings of the 2005 symposium on Usable privacy and security*, SOUPS '05, pages 13–24, New York, NY, USA, 2005. ACM.

[12] T. Hansen and J. Klensin. A Registry for SMTP Enhanced Mail System Status Codes. RFC 5248 (Best Current Practice), June 2008.

[13] Stephen Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8):48–60, 1993.

[14] J. Linn. Trust models and management in public-key infrastructures. 2000.

[15] J. Myers. SMTP Service Extension for Authentication. RFC 2554 (Proposed Standard), March 1999. Obsoleted by RFC 4954.

[2]RFC is short for "Request For Comments", the document format in which proposals for computer interactions are specified

[16] J. Postel. Simple Mail Transfer Protocol. RFC 821 (Standard), August 1982. Obsoleted by RFC 2821.

[17] J.K. Reynolds. Post Office Protocol. RFC 918, October 1984. Obsoleted by RFC 937.

[18] J. Schlyter and W. Griffin. Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints. RFC 4255 (Proposed Standard), January 2006.

[19] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1995.

[20] R. Siemborski and A. Melnikov. SMTP Service Extension for Authentication. RFC 4954 (Proposed Standard), July 2007. Updated by RFC 5248.

[21] Alma Whitten and J. D. Tygar. Why johnny can't encrypt: a usability evaluation of pgp 5.0. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[22] P. Zimmerman. *PGP User Guide*. MIT Press, 1994.

Appendices

A List of Terms

AES Advanced Encryption Standard

DHKX Diffie-Hellman Key Exchange

DNSSEC DNS Security Extension

DNS Domain Name System

GUI Graphical User Interface

IMAP Internet Mail Access Protocol

KDF Key derivation function, a function to generate cryptographic keys from an input

key part of encryption algorithms that is specific to a user, also called **certificate** in X.509-based protocols like **SSL** and **TLS**

MIME Multipurpose Internet Mail Extension

PEM Privacy Enhanced Mail, a system to send secured messages via email

PGP Pretty Good Privacy, a system to send secured messages via email

PK Public Key

POP Post Office Protocol

RSA The asymmetric encryption algorithm by Rivest, Shamir and Adleman

S/MIME Secure MIME, a system to send secured messages via email

SEEPost Security Enabled Electronic Post

SK Secret Key

SMTP Simple Mail Transport Protocol

SSH Secure Shell

SSL Secure Sockets Layer, the protocols for secured Internet communication

TA Trust Authority

TLS Transport Layer Security, the successor of SSL

B Interaction sample

This appendix illustrates the use of the client demo program in a typical interaction session with a user. In this session the user responds on behalf of *Tagware Corp.* and has the address *info@tagware.nl*. The user communitates in this session with *Alice*, with email adres *info@hethelejaarherfst.nl*. We used some domain names that we owned but where not in use. Lines starting with `$` and `>` are commands typed by the user, the former being a system prompt, in which the user types command to the operating system, the latter being the SEEPost prompt, in which the user types commands to the SEEPost demo program.

```
$ ./client -v -f keyfiles/info_tagware_nl.skf
Jun 11 15:22:09 Hello Tagware Corp.
You have 1 new message
> retr
You have 1 new messages
```

Here the demo program is started with the configuration file as second argument. The client informs the user there is a new message waiting, so the user types the command `retr` to retrieve the new message.

Next the user uses the `list` command to get an overview of all the messages. To read the new message from *Alice* the command `read 2` is used.

```
> list
 # ST From                                Subject             Time
 2  U Alice <info@hethelejaarherfst.nl>  Hello world         Mon, 11 Jun 2012 13:21:23 +00
 1  R Alice <info@hethelejaarherfst.nl>  Testje              Mon, 11 Jun 2012 10:04:38 +00
 0  R Admin <admin@mail.tagware.nl>      Welcome to SEEPost! Mon, 11 Jun 2012 08:24:07 +00

3 from 3 messages listed
> read 2
From: Alice <info@hethelejaarherfst.nl>
Signature: OK, verified at Mon, 11 Jun 2012 13:22:12 +0000
Date: Mon, 11 Jun 2012 13:21:23 +0000
Status: U, Priority: 0
Subject: Hello world

Hello Tagware Corp.,

This is a test message.

Greetings,
Alice
```

Now the user wants to send a message back to Alice. An actual reply command in not available in the demo program, but the *write* command does the same with a manual copy-paste of the subject line.

```
> write
To: info@hethelejaarherfst.nl
```

```
Subject: RE: Hello world
Message: (end with EOF)
Hello Alice,

Message recieved.

Bye,
Tagware Corp.
EOF
Message sent!
> quit
QUIT bye
```

After the `EOF` the message is directly sent and the user quits the demo program.

Attacks

In this session we show how the demo program responds to attacks. In the first example the user receives a message with a nonsense signature.

```
$ ./client -v -f keyfiles/info_tagware_nl.skf
Jul 12 11:34:21 Hello Tagware Corp.
You have 1 new message
> retr
Signature did NOT verify, discarding message
You have 0 new messages
> quit
QUIT bye
```

In this second example, a man-in-the-middle attack is simulated.

```
$ ./client -v -f keyfiles/info_tagware_nl.skf
Jul 12 13:29:15 Hello Tagware Corp.
An error occurred: cannot establish connection.
Bye!
```