The cipher mail transport protocol (CMTP)

Jonathan Moroney
University of Hawaii at Manoa
jmoroney@hawaii.edu

July 28, 2016
ABSTRACT

The Cipher Mail Transport Protocol (CMTP) is a new mail transport system designed to make encryption the default option for email. This thesis lays out what a modern email system is, how it works, the shortcomings of the current system and will propose CMTP as a solution. CMTP provides a solution to these problems in an effort to secure email transmission and storage. Inspired by the Pretty Good Privacy (PGP) email security system, CMTP implements public key cryptography, hidden from users, while maintaining the broad usability of email. This is in contrast to secure protocols that pre-date CMTP and generally provide good security, but reduce usability. After an initial, unauthenticated key exchange, CMTP encrypts and authenticates all messages. In this way, CMTP provides security without compromising usability.
# TABLE OF CONTENTS

Abstract .................................................................................................................. ii

List of Figures .......................................................................................................... vii

1 Introduction ............................................................................................................ 1
   1.1 Modern Email ............................................................................................... 1
   1.2 Motivation .................................................................................................... 2

2 Related work .......................................................................................................... 3
   2.1 Transport Layer Security (TLS) ................................................................. 3
   2.2 OpenPGP ..................................................................................................... 3
   2.3 S/MIME ...................................................................................................... 4
   2.4 Darkmail ..................................................................................................... 4
   2.5 XMPP and Instant Messaging .................................................................... 4

3 Message passing on the internet ......................................................................... 6
   3.1 The current email ecosystem ...................................................................... 6
   3.2 The SMTP network .................................................................................... 7
   3.3 NIST on email ........................................................................................... 7

4 Cipher mail transport protocol ........................................................................... 9
   4.1 High level design ....................................................................................... 9
   4.2 CMTP interface ......................................................................................... 10
   4.3 Message structure ..................................................................................... 13
   4.4 Field Definitions ....................................................................................... 13
   4.5 Log Structure ............................................................................................ 14
LIST OF FIGURES

4.1 Initial Server key request ........................................ 9
4.2 Mail passing ....................................................... 10
4.3 Example CMTP exchange ........................................ 12
4.5 Cipher Mail Log Format .......................................... 14
4.4 CMTP message structure. ......................................... 16

7.1 cmtp server main loop ........................................... 22
7.2 cmtp client send ................................................... 23

B.1 Message Diagram .................................................. 33
B.2 Log Diagram ....................................................... 35
B.3 CMTP Message Internals .......................................... 35
B.4 CMTP Message with no carbon copies ......................... 36
B.5 CMTP Attachment Internals ..................................... 36
B.6 CMTP Xzibit Internals ............................................ 37
CHAPTER 1
INTRODUCTION

Email is an integral part of modern communications and daily life. It’s used for nearly all formal communications and it’s a broken system. Email has evolved through the years to provide an asynchronous messaging system where users are tied to servers and where servers facilitate email transmission, delivery and storage. This relationship between the user and their server allows email to work, but it is also imperfect. This client-server relationship means that a user must trust their server to work on their behalf. For example; a generic user will trust their server to (attempt to) deliver any email they send to it, they will trust it to receive email on their behalf and to occasionally forward their email to another server. To the user a email server is a diligent butler. However, as has become evidently clear in the past few years, users are trusting their infrastructure far more than is reasonable.

The infrastructure in question dates back to 1982 when SMTP was first defined and stretches to the present day with ESMTP, POP, IMAP, OpenPGP, and webmail[17]. This infrastructure includes many disparate servers across the globe speaking the same language and making use of other internet infrastructure such as the DNS system. This classical email infrastructure works as a decentralized, federated network and is a natural design for a email system which mimics a postal transport system. There are no barriers to entry if one wants to become a new email provider and this has lead to a great deal of choice in email providers. There is however an issue in that currently email providers have no regulations or restrictions set upon them in the way a classical postal system might. As a result an evil email provider or a disgruntled employee has full access to a client’s email.

1.1 Modern Email

Throughout the life of email a certain level of service has come to be expected. For the purposes of this paper “modern email” will refer to SMTP used in conjunction with IMAP or a webmail client and is meant to represent the quality of service that is expected of an email system. POP, while historically important, is not considered to be modern. Modern email is a system with which users can send and receive email from many devices/programs, can change programs/devices without administrative approval, and can deal with email in an asynchronous fashion. It’s important to understand that email is a generic term for a number of different systems and that by modern email; I mean none of them specifically, but all of them generally. That is, all of their “good” parts generally. For instance, SMTP originally required every user to have an SMTP server of their own and this I qualify as bad. Conversely, gmail allows a user to travel the globe and to have full access
to their communications from anywhere so long as they know some login information; this I qualify as good.

1.2 Motivation

The motivation for building a new email protocol comes simple. Email as it is currently is broken and all prior proposed solutions have failed to fix it. Further, email is the de-facto standard in official communications. Banks, Hospitals, Schools, Government Agencies, and just about every other organization use email as their preferred or at least their default messaging system. Email is clearly important and yet it relies on the ancient SMTP system. SMTP is a plaintext protocol and as such is messages are transmitted and stored in the clear. With all messages being transmitted and stored in the clear it’s trivial for an attacker to monitor a user either through observing message transmission or by gaining access to the user’s email server. In the latter, the email server is a tempting target as one exploit on a misconfigured email server could provide an attacker with the contents of thousands of users mail. Beyond the defense from attacks, securing email would allow users to exchange things like social security numbers through email which could simplify a good deal of bureaucracy.
CHAPTER 2
RELATED WORK

Securing communication has been a topic of active research for about as long as communications have been written. Similarly, securing email has been a topic of active research for about as long as email has been around. Email security comes in two broad types; transport security and message security. Transport Layer Security (TLS) has been added to most email implementations and is largely invisible to users while message security is used only by a small group of the technically literate. TLS is the most widely deployed email security system. The most prominent message security systems are the OpenPGP and the S/MIME standards. To the best of my knowledge both standards provide good security, but neither has been able to make security a standard feature of email. A more comparable system to the Cipher Mail Transport Protocol (CMTP) can be seen in Darkmail. Darkmail is a creation of the Darkmail Alliance and is a completely new set of protocols. The Darkmail Alliance has introduced the Dark Mail Transfer protocol (DMTP) and the Dark Mail Access Protocol (DMAP) which aim to partially anonymize conversations, create a trust chain for key rotation and more[15]

2.1 Transport Layer Security (TLS)

Transport security in the form of the TLS protocol is easily the most widely deployed email security measure. The high level idea is to create a cryptographic tunnel through which otherwise unsecured communications can be passed. In concept this is similar to the functionality provided by the Secure Shell (SSH). In theory the idea of adding a security layer to an application seems like a practical way to secure an existing system. In practice adding TLS to SMTP communications has had problems. In particular there is a need to negotiate a cipher and an opportunity for a man in the middle attacker to prevent or degrade the transport security. In the real world this seems to be a serious problem[13].

2.2 OpenPGP

The open pretty good privacy (OpenPGP) standard serves as the primary inspiration for CMTP and that’s because the author of this document thinks OpenPGP is a brilliant system. OpenPGP works by having users exchange independently created public keys which are then used to encrypt mail and pass it in the normal SMTP way. OpenPGP is a decentralized system that doesn’t require users to trust an arbitrary authority, but does allow for the development of a web of trust through shared contacts[11]. The fundamental problem with OpenPGP is simply that it has never reached universal adoption and so users have always had to be aware of who they are talking to
(on a technical level). As a result of non-universal adoption are a number of usability problems, including, but not limited to, requiring users to manage their own keys, requiring users to manage their own key exchanges, support for broken ciphers, and a lack of clean mail client integration.

### 2.3 S/MIME

If OpenPGP is the bazaar then the Secure/Multipurpose Internet Mail Extensions (S/MIME) is the corresponding cathedral with a top down command and control key exchange system [19]. In an S/MIME environment users define a trusted key source and then software will sort out all the key exchange and usage[18]. Problems with S/MIME come from the same lack of universality as with OpenPGP and include cross client compatibility, inter-domain key exchange, and communication with non-S/MIME users.

### 2.4 Darkmail

The Dark Internet Mail Environment (DIME or Darkmail) is a system developed by Ladar Levison in response to the Lavabit incident[2]. Darkmail aims to prevent mail operators from being in the position where they become informants on their users. Ladar has designed Darkmail to be anonymous at the domain level in addition to the encrypted content of the messages. Additionally the idea of a ‘signet’ is introduced as a sort of super key which includes an amount of plaintext information (Name, Address, Phone Number, Motto, and many more) along with a cryptographic key. The idea of the signet is that the addition of the plaintext information make keys easier for a human to accept. The problem with Darkmail is that it aims to solve every security problem that email faces and as such Darkmail has yet to make a meaningful impact.

### 2.5 XMPP and Instant Messaging

Instant messaging fills a different usage model than email, but has been included here for an alternative perspective on message passing. Instant messengers aim to provide real time text communication over data networks like the internet. Common traits include user presence display and control, file transfer capabilities, group chats, and of course ‘instant’ message passing. To facilitate the ‘instant’ message passing users must be online in order to receive or to send messages. Here we see a feature set very similar to what we see in email, but with the constraint that users need to be connected and the benefit of lower latency messaging. One major difference between instant messaging and email is the incompatibility between different networks which is a non-issue in the realm of email. This inter-network incompatibility has been addressed by the XMPP standard which has become the defacto instant messaging protocol as far as this author is concerned. In 2004 instant messengers were graced with the Off the Record encryption system which raised the security
level of instant messaging while also maintaining the usability of the medium [10]. The Authors of the Off the Record Paper noted that instant messaging and email are used in fundamentally different ways which makes different cryptosystems applicable.
CHAPTER 3
MESSAGE PASSING ON THE INTERNET

The internet has evolved significantly since the early ARPANET and with it message passing techniques have evolved as well. The design of email has the embedded idea that you already know with whom you are talking to (or at least with whom you would like to talk to). This prior knowledge is the User@domain syntax; in order to start a conversation a user need to know the identifier associated with their desired recipient. There are other methods of message passing such as chat rooms and instant messaging which are far less formal and often used as a medium to connect previously unconnected individuals[16]. Instant messengers have gone through a number of design revisions in the last twenty years alone. Encrypted messaging has been available through a single click since at least 2004 and has now become a standard default rather than an option[10]. Why is it that the state of email, the most formal message passing system on the internet, has been so stagnant? Instant messaging is much like email with the main difference being the instantaneousness of the former. If we observe the XMPP message passing protocol we see a decentralized, federated system of message passing servers and clients. A user can register an account on one XMPP server and talk with users on other XMPP servers; very much like SMTP based email[20].

3.1 The current email ecosystem

Currently email is transported between servers using the simple mail transport protocol (SMTP) and retrieved for viewing through the post office protocol (POP) or the internet mail access protocol (IMAP). SMTP is an antiquated protocol that only supports attachments and languages other than English through a hack called MIME[14]. MIME has the effect of increasing the complexity of SMTP and decreasing its efficiency; for these reasons CMTP has been designed to use unicode by default and to support arbitrary extensions. SMTP server discovery is done through domain name system (DNS) queries which return a mail exchange (MX) record for a given domain. Using these DNS queries systems are able to discover the machine is in charge of handling mail for any valid domain and further they are able to quickly ignore invalid domains. CMTP also uses the DNS system and MX records for host discovery. Using the same discovery mechanisms as SMTP has led CMTP to also use the same port as SMTP and has led to CMTP using a similar command syntax in order to coexist with SMTP. Ideally an SMTP client connecting to a CMTP server will fail gracefully as will a CMTP client connecting to an SMTP server. The CMTP commands discussed in the section 4 have been chosen to not overlap with SMTP commands in hopes that a combined server can be created.
3.2 The SMTP network

First it’s worth noting that SMTP without PGP ‘just works’. That is, SMTP does a fine job of transporting a piece of mail from one place to another. More often than not mail arrives in a usable state and without delay. SMTP defines an open network which anyone can join and which no one controls. I believe that the open design and ease of implementation are critical elements of SMTP’s success as they ensure low barriers to entry, foster deployment, and encourage development. Further, when considering a replacement for SMTP one must also consider the resources that have been deployed which enable SMTP and the users who have become accustomed to the usage model of SMTP. The main reusable resource of SMTP is the DNS system which has a query-able mail exchange (MX) record. Traditionally the MX record has held the IP address of an SMTP server, but there’s no reason CMTP cannot coexist in the same space. However, traditional SMTP clients will eventually try to talk to a CMTP server which constrains the CMTP interface and in particular led to the decision to use a text based interface described in section 4.2 and in particular to use the four character initiator ‘OHAI’. SMTP clients that connect to a CMTP server will issue a ‘HELO’ or ‘ELHO’ and will fail in a defined way while CMTP clients that connect to an SMTP server will issue an ‘OHAI’ and will also fail in a defined way.

3.3 NIST on email

The United States National Institute of Standards and Technology (NIST) has published a large amount on security and messaging standards. As a standards body NIST is a source worth being aware of when developing a standard and as this document is proposing a secure messaging standard the NIST Guidelines on Electronic Mail Security has been a necessary reference[23]. The guidelines on email cover far more than message encryption, but the existing SMIME and openPGP standards are considered. In section 3.4 a number of issues with encryption are identified and they are

1. Scanning for viruses and other malware and filtering email content at the firewall and mail server is made significantly more complicated by encryption.

2. Encryption and decryption require processor time.

3. Organization-wide use of encryption can require significant ongoing administrative overhead.

4. Email encryption can complicate the review of email messages by law enforcement and other investigative parties.

5. Encrypted emails sent to or received from other organizations may be insufficiently protected if those organizations do not support the use of strong encryption algorithms and key sizes.
Issues one and two speak to a lack of compute power on the client and is an issue that is primarily a concern for legacy machines. Modern computers and phones are more than capable of doing their own virus scanning, many do on a constant basis. Spam filtering at a network firewall is an unfortunate casualty of securing email, but it is the author’s opinion that it is a cost worth paying. Encryption and decryption work is also a trivial amount of work for modern machines.

Issues number three and five are true of existing systems and are probably the biggest issues with implementing secure mail as a default. The Author hopes that CMTP is found to fix these issue. Issue number four is a tricky issue to comment on as it brings into question the morality of enabling malicious users. I won’t comment here except to mention that the United Nations has issued a report on the necessity of encryption and anonymity in the exercise of the freedoms of opinion and expression [12].
CHAPTER 4
CIPHER MAIL TRANSPORT PROTOCOL

The cipher mail transport protocol (henceforth CMTP) has two goals:

First
To maintain the usage model of modern email

Second
To maximize security while not violating the first goal

4.1 High level design

CMTP works with the same federated network model that SMTP uses and host discovery is done with the same Domain Name System (DNS) querying which SMTP uses. In order to send a message a CMTP client must first do a query to the DNS system with a request for the MX record associated with a given domain. This lookup allows the initial key transfer of the CMTP server public key as shown in Figure 4.1.

![Diagram of initial server key request]

Figure 4.1: Initial Server key request

The steps here are numbered and we can see the ordering of events. After this initial key request the client can verify signatures on all subsequent communications. The next logical step is that the user will want to send mail with this mail transfer protocol. The next communication path can be seen in Figure 4.2.
Here we can see the steps numbered again and that for any message a key request is done before any email handling. This is the critical step that allows all email to exist in ciphertext form.

At the high level CMTP works by ensuring that all users have public-private key pairs and by tasking the infrastructure with transporting public keys as well as messages. Servers have their own keys which allows them to trust and keep track of each other in a manner similar to how SSH instances keep track of each other. Users generate their keys client side, encrypt their keys with a symmetric cipher and task the server with holding the keypair. For convenience the public key is stored both in an encrypted form and in the clear on each user’s email server. In a webmail scenario this should be done in javascript or some other client side language so that the server never sees an unencrypted private key. Having the server hold both keys allows a user the freedom of modern email where they can have multiple devices, add a new device without talking to a server administrator, and can expect to get any public key for any user at any time.

4.2 CMTP interface

Inspired by SMTP a CMTP connection is driven by a text interface which readies client and server for a binary message transfer. The commands for CMTP have been designed to make stateless implementation easy. All commands are ASCII strings terminated by a null character (\0). Commands are capitalized ascii strings terminated with the new line character (0x0a) and after the OHAI command there is no ordering to the commands; each can be thought of as an atomic operation. ASCII is used over utf-8 for commands as each ASCII character corresponds to one byte. All commands should be acknowledged before the client sends anything more. This text interface is the set of CMTP commands which are

```
OHAI
```
The OHAI command is in place to prevent SMTP clients from seeing a CMTP server as a valid SMTP server. Conveniently OHAI is four characters long which should mean even the oldest SMTP client should fail in a defined way when attempting to connect to a CMTP server.

**MAIL**

The MAIL command is a stand alone command which serves only to tell the CMTP server that a message is to follow. The CMTP message is self describing in length so that a server knows when it has received the entire thing. Subsequently no state is needed in order to pass a message.

**KEYREQUEST**

The KEYREQUEST command takes at least one parameter and at most two. The user parameter is required while the domain parameter is not. Each parameter is null terminated. In the case that the domain parameter is not present the CMTP server should assume that the user is local to it.

**NOOP**

This command does nothing but prompt a reply from the CMTP server.

**LOGIN**

The LOGIN command takes two parameters in the form of a user name and a domain. The correct return is an encrypted copy of the users public and private keys or an error message if no keys are found. The container for these keys is called the xzibit.

**OBAI**

The OBAI command is in place to terminate a connection. Connections may terminate for other reasons (time outs, tcp connection breaking, etc...), but OBAI allows for a graceful exit.

A complete list of the CMTP commands, their parameters and their responses is available in Appendix A. An example time-line of an client and server is shown below
Figure 4.3: Example CMTP exchange
4.3 Message structure

The structure of the CMTP message can be broken down into two sections, the plaintext “envelope information” and the cipherbytes which constitute the body and the attachments. The log structure is plaintext and is used to track forwarding, acceptance, and rejection by CMTP servers.

The fields that begin the CMTP message are the fixed length fields and their position in the file structure is for programming convenience. The variable length fields described below are UTF-8 encoded strings that are null terminated, that is, the first null byte (8 bits of zero) defines the end of that field. The fields where this is not true are the log, the message body and the attachment(s) which have their own structure. Since the number of attachments is variable each attachment field is preceded by an attachment length field. The fields are defined in the next section.

4.4 Field Definitions

Version
A four byte integer used to mark the version of the message structure. The type of cryptography in use is also defined by this number. This document covers version 1.

Attachment Count
A four byte integer used to mark how many attachment fields follow.

Log Length
An eight byte integer used to mark how many bytes are in the log field.

Body Length
An eight byte integer used to mark how many bytes are in the message body field.

Destination Account
A null terminated utf-8 string used to denote the recipient account of the message. Max length is 255 bytes.

Destination Domain
A null terminated utf-8 string used to denote the recipient domain of the message. Max length is 255 bytes.

Source Account
A null terminated utf-8 string used to denote the sender account of the message. Max length is 255 bytes.
Source Domain
A null terminated utf-8 string used to denote the sender domain of the message. Max length is 255 bytes.

Log
A collection of UTF-8, null terminated routing messages.

Message Body
The actual email being transmitted with length in bytes defined by the message length field.

Attachment Length
An eight byte integer used to mark how many bytes are in next attachment body.

Attachment Body
An arbitrary attachment with length defined by the attachment length field preceeding.

4.5 Log Structure
The log is simply defined as a collection of null-terminated utf-8 messages. The log is to be used by any entity processing the specific piece of email and can be used for general status messages as well as for more critical issues.

<table>
<thead>
<tr>
<th>Log Message 1</th>
<th>\0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Message 2</td>
<td>\0</td>
</tr>
</tbody>
</table>

Figure 4.5: Cipher Mail Log Format

4.6 Message Body Internal Header
The internal header of a cipher mail message needs to contain a variable amount of information; A Subject line/field, a Reply To address, Carbon Copy addresses, and a Creation Date. As such the header itself is a variable length structure with each line item being a utf-8 string terminated by a null character. The ordering of the fields is Reply-To, Subject, Date, Carbon Copies. The carbon copies are username\0domain\0 and are terminated by a double null. ie. \0\0.
4.7 Message Body Internal Body

This section is mostly a block of utf-8 encoded text. Any non-text data should be transmitted through attachments.

4.8 Attachment Structure

The Attachment is a simple structure. The first 256 bytes of the structure is reserved for the utf-8 file name with null terminator and the rest of the structure is for actual file data. The file name is null terminated and the non-null portion can be at most 255 bytes long. Similarly this data structure is covered in detail in the appendix.
Figure 4.4: CMTP message structure.
The cipher mail transport protocol (CMTP) aims to provide end to end encryption and message security regardless of network security. However, unlike many crypto systems CMTP does not try to provide identity verification and this is done to reduce the complexity of the protocol. Identity verification is not a goal of CMTP, however it can be added at a later date through a system like TLS with certificate verification or through a verified Domain lookup system like DNSsec. The basic design of CMTP aims to allow for an open network and thus does not concern itself with identity verification. Even though CMTP does not try to verify who you are talking to it does concern itself with making sure that you are talking to that same person for the entire conversation. Some inspiration for this was taken from a reverse engineering of the Nintendo 3DS system. The motto 'Everything that can be encrypted is and everything that can be signed is' is followed. This security motto comes from the Chaos Communication Congress talk by Smealum et al. 3DS [22].

5.1 Threat model

It would be easy to say that for a given message the attackers are simply everyone who is not either the sender or the receiver, but this ignores the fact that Alice and Bob are often incapable of using complex systems and can’t be trusted to properly manage their keys or to reason about security. Our users’ errors can be seen as either system vulnerabilities or as resources for the attackers. So, from the point of view of the CMTP design we aim to minimize what we need from our Alices and Bobs in order to use our system and we should treat user error as a weakness in the security system. The crypto system being used for encryption mode one is assumed to be secure[8] which leaves operational security left to be considered in this protocol. The primary known weakness in CMTP comes from the key requests which can be intercepted by an attacker who has become a man in the middle of two CMTP servers. The attacker is able return any key she desires to the requesting party and thus can own the communication channel. This stems from the design decision to not consider identity verification in CMTP.

Moving on to key storage; in general we need to maintain key pairs for users over a long period of time. One weakness of openPGP is that losing a key is very easy to do and by design there is no key retrieval mechanism. A simple scenario is a system crash. Users generally are as bad about backups as they are about security. This leads to scenarios in which users can communicate with each other, but must be willing to accept new keys for contacts they already know. This is a problem as it either makes the man in the middle a trivial attack or leads to socially awkward situations. We must consider these cases as they happen daily. To avoid this problem CMTP tasks
the CMTP server with being a key server for both public and private keys. Ideally the user is completely unaware of what their private key even is. To facilitate this ease of use the Xzibit key system is introduced.

5.2 Xzibit Key System

The idea of the Xzibit key system is simply to encipher the cryptographic keys before storing them with a third party. This system is named after an internet meme known for recursion. With the private key encrypted a user can task an untrustworthy server to hold their private key and will have a reasonable assurance that their key is recoverable and safe. This has two major benefits to the user:

1. It lowers the cognitive demand on the user. They only needs to be able to decrypt their private key which can be done with a single password and a symmetric cipher.

2. It maintains the usage model of modern email. A user can retrieve their keypair from any machine, anywhere in the world.

During account creation a CMTP client randomly generates a public-private keypair and uses user input to encrypt the pair for storage. CMTP uses a password for implementation convenience, but any repeatable user input is acceptable. At this point the CMTP client can send both the Xzibit and the plaintext public key to the CMTP server for storage without fear of leaking secrets. It’s worth noting that for political reasons users should not trust their servers any more than they have to as service providers may occasionally be their enemies[1]. Storing user keys in a publicly available data structure does make it prone to offline attacks such as brute force and dictionary key space searches. This offline attack vulnerability is a real issue, but is somewhat mitigated by the choice of password hash [9].

5.3 Usable security

The Xzibit system allows users to outsource their system reliability concerns without leaking secret information. This plus the automation of key discovery and message encryption allow a user to ignore security and thus to use it without the cognitive load normally associated with security systems. As mentioned above trusting any key that gets returned after a key request is insecure. When accepting keys it’s possible that an attacker could intercept the key request and return a key of their choice. This key interception could lead to a man in the middling of a conversation and is clearly a security concern. Implementations of CMTP clients should route all key requests through the user’s own mail server. Servers are generally more reliable than clients and with multiple users a server will gather the keys of other servers quicker than any one user will leading to an SSH-like known hosts situation.
The Cipher Mail Transport Protocol (CMTP) has been designed to easily adapt to new crypto-systems as the need arises. Each communication from a server has a version field which denotes the crypto-system in use. There are two currently defined crypto-systems:

- 0 - Plaintext. That is, no encryption.
- 1 - XSalsa20\[7\] with 256 bit keys. Further details below.

As the state of cryptography evolves additional crypto systems can be added.

6.1 Crypto System 0

As the name implies this is the ‘no encryption’ encryption system. There’s not much to say about this system other than it should be avoided where possible. It has been included for a few reasons.

1. Someone somewhere is eventually going to want a plaintext pathway, so defining it at the birth of the protocol will prevent fracturing of the protocol.

2. Backward compatibility with SMTP based system necessitates going plain and so having this system will allow for that compatibility to be added.

3. Usage of the 0 version simplifies some error messages in the protocol.

6.2 Crypto System 1

The choice for crypto-system 1 to use the XSalsa20 stream cipher along with the Poly1305 signing algorithm\[6\] was a fundamentally pragmatic one. In particular messages are enciphered with the LibSodium crypto_box_seal() [3] function. This function does the job of encrypting and signing the message as well as throwing errors should something not match up. The signing function is the crypto_sign_detached() function [4]. Many cryptographic messaging systems on the internet are using this library and the algorithms it contains which helped make this decision easier. Two high profile projects are Tox and Discord [5]. However, this does end up meaning that the “correct” implementation of encryption is whatever LibSodium does. That said, outsourcing to LibSodium allows for this project to focus on passing messages rather than being stuck in the woods developing crypto system 1.
6.3 The Xzibit

The Xzibit format is defined on a per-version basis. Version zero is unsupported for the xzibit as version zero corresponds to plaintext communication where there are no keys. Version one is defined as

[Version][CipherSalt][Length][Ciphertext]

Where the

- Version is a 4 byte big endian integer.
- CipherSalt is a 32 byte salt used in the AES decryption. The first 12 bytes are also used as a nonce.
- Length is an 8 byte big endian integer.
- Ciphertext is the payload of the xzibit with length in bytes defined by the length field.

Currently the internal data in the ciphertext is

[PublicKey][PrivateKey]

Where the public key is 32 bytes long and the private key is 64 bytes long. The cipher text may eventually hold things like an address book as well. The user Xzibit is encrypted using the hash of a user password. The hash function chosen for crypto system one is the Argon2 password hash [9].
CHAPTER 7
REFERENCE IMPLEMENTATION

Reference code is provided at http://www.github.com/darakian/cmtp and is in two parts.

- cmtpd - The reference cmtp mail server
- cmtp_client - The reference cmtp mail client

All reference code is has been implemented in POSIX C99 and tested on Linux systems. The current development system uses the 4.4.7 Linux kernel.

7.1 cmtpd

The server component cmtpd has been created with the aim to be easy to administer. To this aim a config file can be used, but is not necessary for normal operation. If no config file is provided cmtpd will use a variety of sane defaults which include

- Getting local domain from the host system.
- Jail the running process into /var/cmtp
- Use ext3 safe directory and file names for users and mail.
- Write logs to /dev/log via the syslog interface.

On a first run cmtpd will generate new keys for itself. These should be backed up as the server will simply generate new keys in the event that the old ones are not found and this will causes a number of authority issues. Because cmtp works on port 25 and because cmtpd needs to setup a jail for itself it must be started with root privilege. Root privilege is quickly dropped and cmtpd lowers itself to the ‘nobody’ user and the ‘nogroup’ group for the purposes of operation. Great effort has been made to make cmtpd fail responsibly should it fail and to log as much useful information as is possible so that recovery is as easy as possible. To facilitate general usage the jail file system mimics a standard POSIX filesystem structure, but is mostly empty. For example.

/var/cmtp/
|--dev
 ||--log
|--etc
 ||--resolv.conf
|--mail
 ||--jon
 ||--public.key

21
We can see that the /dev/log and /etc/resolv.conf have been linked into the cmtp jail. This is done to allow usage of the linux log device and for allowing DNS queries to resolve.

### 7.1.1 Design

cmtpd has been designed around a single infinite loop.

```c
while(1)
{
    if ((temp_connection=accept(server_socket, (struct sockaddr *)&
                              temp_connection_sockaddr, &temp_addr_length))>=-1)
    {
        //Create thread
        pthread_t connection_thread;
        //Thread_arg is freed within connection_manager as one of the first
        //operations.
        int * thread_arg = calloc(1, sizeof(int));
        *thread_arg = temp_connection;
        temp_connection = -1;
        if (pthread_create(&connection_thread, NULL, connection_manager, (void *)
                thread_arg)!=0)
        {
            print_to_log("Thread error! We're all doomed!", LOG_EMERG);
            perror("pthread");
        }
        #ifdef DEBUG
        printf("Returned from connection manager.\n");
        #endif /*DEBUG*/
    }
}
```

Figure 7.1: cmtp server main loop

The general idea of the loop is to wait for a connection, accept it, spawn a thread and pass the connection to the thread, reset and wait for a new connection. This design gives a mapping of one connection to one thread and this was done in order to isolate concurrent connections from each other.

### 7.2 cmtp_client

The CMTP client is mostly two functions; send and read mail.
int32_t send_message(uint32_t socket, char * header_buffer, uint32_t header_buffer_length, unsigned char * message_buffer, uint32_t message_buffer_length)
{
    //Send MAIL\0 command
    #ifdef DEBUG
    printf("send_message called with header of size %d and message of size %d\n", header_buffer_length, message_buffer_length);
    #endif /*DEBUG*/
    if (write(socket, cmtp_command_MAIL, sizeof(cmtp_command_MAIL))<0)
    {
        perror("Write");
        print_to_log("Sending mail command failed.", LOG_ERR);
        return -1;
    }
    if (write(socket, header_buffer, header_buffer_length)<0)
    {
        perror("Write");
        print_to_log("Sending message failed.", LOG_ERR);
        return -1;
    }
    if (write(socket, message_buffer, message_buffer_length)<0)
    {
        perror("Write");
        print_to_log("Sending message failed.", LOG_ERR);
        return -1;
    }
    return 0;
}

Figure 7.2: cmtp client send

The send function shown above is aided by helper functions which construct the header and message buffers. Those are detailed in the appendix and simply prepare the header and body buffers for the send function. The reading function asks a user to select a file and attempts to decrypt it. The code for this read function is also found in Appendix C.
CHAPTER 8
EVALUATION AND LIMITATIONS

CMTP is hopefully an incremental and implementable improvement to the state of message passing on the internet, but it is not without its limitations. CMTP is designed to function in a way that is highly similar to SMTP and has adopted some of the baggage that SMTP is known for. This chapter evaluates the CMTP reference implementation as well as covers the limitations in both the protocol and the reference implementation.

8.1 Evaluation

As has been mentioned CMTP is currently working in a proof of concept state. Proof of concept code is available and does the job of passing messages from clients to servers which are not the same machine, but where both are connected to the internet. There are no requirement for cmtpd to work is that the MX record be set for the domain. Setting an MX record should be very familiar to anyone who deals with mail servers. For the systems administrator the cmtpd code compiles and runs with zero intervention. Currently starting and stopping of cmtpd needs to be done manually and the program only runs on linux systems. Two users other than the author have used the cmtp client and have been left underwhelmed by its normalcy; this is perfect. The whole point of CMTP is to be invisible to the user and to automate the mundane house keeping required by PGP. OpenPGP by contrast runs on nearly everything and has been extensively tested in the field. CMTP needs to address this lack of testing by building to the point where an ecosystem can arise around it.

8.2 Protocol Limitations

CMTP has no concept of authentication by design. As such CMTP should not be used as a tool to determine with whom you are talking. CMTP does not try to decentralize mail or to in any way re-imagine what a mail network could or should be; it is simply a refinement of an existing idea. One area where CMTP may be seen as a regression on the SMTP idea rather than an advancement is in spam detection and prevention. By encrypting the messages users send, normal spam prevention heuristics have been made unusable. Spam filtering in CMTP will be limited to what can be derived from envelope information and from network information. The envelope and network information lend themselves to black, white and grey listing techniques which are generally considered to be less effective than heuristic based spam classification approaches [21].
8.3 Implementation Limitations

Formalized in the CMTP specification is encryption mode zero which is to be used if plaintext communication is desired. Encryption mode zero is not implemented in the reference code. This is a feature that will most likely be implemented to allow communication with the SMTP network rather than for its own sake. Further cmtpd has only had the benefit of development testing. None of the email handled by cmtpd has yet been anything more than a test. Thankfully cmtpd has been attacked in its short lifetime and has stood up to it. The attack was essentially a brute force of the protocol; a fuzzing attempt similar to what might be seen in a black box test. This was conducted by a colleague of the author. Details of the attack are limited, but the attack took place over a long weekend, involved creating hundreds of simultaneous connections and serving each of them without noticeable impact to the rest of the system. That said, no formal testing of the software has been conducted and no user experience testing has been done.
CHAPTER 9  
FUTURE WORK

9.1 Near term work

The most immediate need for the CMTP system is a mail access protocol to enable access to mail by systems which are not running an instance of a CMTP server. Existing mail delivery systems are not applicable to CMTP messages as they require access to the message plaintext in order to properly function. How this is to be done is an open question at the moment and will be explored in coming months and years. Second, a method of key replacement is necessary. As it stands CMTP simply accepts whatever key it has. Should keys become compromised a new key can be used, but a prior key cannot be declared bad. Beyond this there are a number of bugs that exist in the code base which have not yet made themselves known.

9.2 Long term work

Some long term goals for CMTP include

**Implementing encryption mode zero**

Supporting plaintext communications might seem counter to the whole idea of cmtp, but there are scenarios where users may want to send a message in the clear and the infrastructure should allow for that.

**Adding SMTP command support to cmtpd**

SMTP is an entrenched standard and it would be ideal to allow some level of interoperability with the existing email system. Enabling encrypted communications to SMTP users is not a goal, but extending encryption mode 0 to the SMTP network is. This would take some format translation at the CMTP server, but should be within the realm of possibility.

**Solving the key request insecurity**

This one is a longer term goal, but research needs to be done on secure key distribution.

**Implement authorized sending**

When a user receives their xzibit with the LOGIN command some information could be passed along with it. A time stamp could be given to the user. The server could then expect a signature of the timestamp as proof that the user was able to decrypt the xzibit and refuse to send mail on behalf of the user until then.

On the keyrequest issue I have two major ideas
1. Server keys in DNS records

2. Blockchain based key storage

In the first idea the systems administrator would register their server’s public key in the DNS record under one of the many unused (or underused) record types. This would add some administrative overhead and could still be attacked though it may increase the difficulty of the attack. The second idea is not yet fully formed, but the broad concept is to use a blockchain structure to store signed keys and to make the chain a publicly available item. Where the chain gets stored, how it gets created and how it grows are details that are still undecided, but the idea of a self verifiable ledger to record keys is enticing to say the least.
CHAPTER 10
CONCLUSION

Perfect is the enemy of good; CMTP is not perfect. CMTP is a good protocol which raises the security level of a communication system used by all. The current state of email has user data being mined by ad companies, intercepted by criminals across the globe and read by anyone that can get access to it at rest or in motion. Email as a system is the de facto standard for formal electronic communications and is woefully insecure in its current state. My work on CMTP shows that the high cognitive load associated with PGP is not necessary to employ a public key cryptosystem with email and that upgrading the infrastructure of email can be done transparently. There are other systems working on solutions to these problems; darkmail promises more than CMTP, but changes everything that email is and doesn’t yet work. There are problems with CMTP and there is still work to be done in order to grow CMTP into a complete email ecosystem, but CMTP provides a familiar interface, a familiar network, and a familiar mode of operation all with the benefits of end to end encryption. CMTP works today and only needs a mail delivery protocol in order to be at feature parity with the SMTP/IMAP combo. All of this has been done with minimal administrative needs and using pre-existing internet infrastructure. This work and CMTP demonstrate that there is no reason that email needs to be as insecure as it is.
APPENDIX A
INTERFACE OVERVIEW

The Cipher Mail Transport Protocol (CMTP) uses ASCII text for commands and replies. UTF-8 is used for user strings and domain strings where applicable. The commands used were chosen to be similar to SMTP commands without overlapping. Further the design of the CMTP commands enables stateless server design. Null bytes are used as terminators for commands and parameters. The null byte will be denoted with a $\langle 0 \rangle$.

A.1 OHAI

OHAI is the ‘HELO’ of CMTP. It was chosen to conform with the 4 character limit of early SMTP implementations. By using a 4 character string which is unknown to SMTP, CMTP should be able to gracefully coexist on the same ports as SMTP.

A.1.1 Parameters

None.

A.1.2 Replies

A reply to OHAI can either be success or failure. In the success case the respondent should reply with ‘OHAI’ followed by their CMTP server version string. Anything else is considered a failure.

A.2 MAIL

The MAIL command is used to pass a message and is followed by the CMTP message data structure. This data structure is self describing.

A.2.1 Parameters

The only parameter MAIL handles is a CMTP message.

A.2.2 Replies

On success a CMTP server replies with

SUCCESS[Sig]$\langle 0 \rangle$

Where ‘Sig’ is the server signature of the message ‘SUCCESS’. The message is followed by a null character.
On failure a CMTP server replies with
FAILURE[Sig][\0]
Where ‘Sig’ is the server signature of the message ‘FAILURE’. The message is followed by a null character.

A.3 KEYREQUEST

The KEYREQUEST command is used to transfer user and server public keys. This command is what allows for the transparent encryption model of CMTP.

A.3.1 Parameters

There are two parameters for KEYREQUEST; USER and DOMAIN. A null user refers to the server and a null domain refers to the local domain. Thus, a server receiving a keyrequest for null null would return its own public key. Keyrequest always replies with the most recent key version available.

A.3.2 Replies

Success is when a key is available and can be returned to the requester. In this event the key is passed back in the following form.
[Version][PublicKey][Sig][\0]
Where ‘Sig’ is the server signature of the public key. The version number is four bytes and defines the length of both the public key and the signature. The server signature is always present even in the case when the server is distributing its own public key.
Failure cases are handled with the generic failure message.
FAILURE[Sig][\0]
Where ‘Sig’ is the server signature of the message ‘FAILURE’. The message is followed by a null character.

A.4 LOGIN

The LOGIN command facilitates private key distribution through the xzibit data structure.

A.4.1 Parameters

LOGIN takes two parameters which are null terminated UTF-8 string. This strings are the user who wishes to retrieve their xzibit and the domain for the user.
LOGIN[\0]USER[\0]DOMAIN[\0]
A.4.2 Replies

On success the CMTP server replies with

\[\text{UserXzibit}[\text{Sig}][\backslash 0]\]

Where \text{Sig} is the server signature of the users’ xzibit.
If the user has no xzibit then the user has not been added. In this case the CMTP server passes a
generic failure message.

\[\text{FAILURE}[\text{Sig}][\backslash 0]\]

A.5 NOOP

The NOOP command is used as a way to test an active connection.

A.5.1 Parameters

NOOP takes no parameters.

A.5.2 Replies

A CMTP server should reply with its version string in response to the NOOP command.

A.6 OBAI

OBAI is the inverse of the OHAI command and is used to terminate connections.

A.6.1 Parameters

OBAI takes no parameters.

A.6.2 Replies

There is no reply to the OBAI command. The CMTP server should simply kill the active
connection.
B.1 CMTP Message

At the high level a Cipher Mail Message is broken into two sections. Envelope Data and Message Data. Envelope data is the plaintext used for routing a message and determining how to read it. Message data is the encrypted payload. Envelope information includes the sender and receiver, the version number, the log length and log body, the attachment count and attachment lengths, and the message length. All data is in network order on the wire, that is Big Endian. It is not strictly required that CMTP messages be stored on disk in Big Endian format, but it is highly advised. All reference code preserves the network data structure on disk. A high level view of the CMTP message can be seen in figure 1 with a scale in bytes.
B.1.1 Envelope Data

The envelope data is everything other than the message body and the attachment body. These fields are plaintext and used for routing and general message management. The fields in the message are

Version
A four byte integer used to mark the version of the message structure. The type of cryptography in use is also defined by this number. This document covers version 1.
Attachment Count
A four byte integer used to mark how many attachment fields follow.

Log Length
An eight byte integer (long) used to mark how many bytes are in the log field.

Body Length
An eight byte integer (long) used to mark how many bytes are in the message body field.

Destination Account
A null terminated utf-8 string used to denote the recipient account of the message. Max length is 256 bytes including the null terminator.

Destination Domain
A null terminated utf-8 string used to denote the recipient domain of the message. Max length is 256 bytes including the null terminator.

Source Account
A null terminated utf-8 string used to denote the sender account of the message. Max length is 256 bytes including the null terminator.

Source Domain
A null terminated utf-8 string used to denote the sender domain of the message. Max length is 256 bytes including the null terminator.

Log
Discussed in subsection 2.

Message Body
Discussed in subsection 3.

Attachment Length
An eight byte integer (long) used to mark how many bytes are in next attachment body.

Attachment Body
Discussed in subsection 3.

B.1.2 The Log
The Log is a simple structure of null terminated UTF-8 strings. The smallest possible log has no length at all.
B.1.3 The Message Body

The internals of the encrypted message body were designed to be feature compatible with current email implementations, i.e. carbon copy, reply-to, and subject headers are all present. Additionally a 8 byte unix time-stamp is placed inside to provide a creation date record. The message body diagram follows.

The only intricacy of the message body is the variable number of carbon copies. The last carbon copy is double null terminated while all other carbon copies are single null terminated. It is
perfectly valid to have no carbon copies at all and in this case there are simply two null characters following the reply-to null character. For clarity a diagram of the no carbon copy case is included.

```
+-------------------+----+
| Time Stamp        |   -|
+-------------------+----+
| Subject           |   -|
+-------------------+----+
| Reply To Address  | \0|\0|\0|
+-------------------+----+
| Message Body      |   -|
```

Figure B.4: CMTP Message with no carbon copies

### B.1.4 The Attachment Body

The attachment body is two parts. The first is the null terminated UTF-8 file name and the second is the attachment data itself.

```
+-------------------+----+
| File Name         |   -|
+-------------------+----+
| Attachment Data   |   -|
```

Figure B.5: CMTP Attachment Internals

### B.2 Xzibit key store

The Xzibit continues the trend of simple data structures with the following.
In this diagram we see a version number which is an unencrypted integer and the two user keys which are encrypted using a symmetric cipher keyed from user input.
Listing C.1: Build Header Function

```c
int32_t build_header(char *recipient, uint32_t recipient_length, uint32_t version, uint32_t attachment_count, uint64_t log_length, uint64_t message_length, char *return_buffer)
{
    #ifdef DEBUG
    printf("Building message header destined for: \n");
    for (uint32_t i =0; i<recipient_length; i++)
    {
        printf("%c / %x\n", recipient[i], recipient[i]);
    }
    printf("Variables are: recipient_length = %d, attachment_count = %d, log_length = %ld\n", recipient_length, attachment_count, log_length);
    #endif /*DEBUG*/
    //Builds the CMTP message header
    uint32_t target = 0;
    uint32_t net_version = htonl(version);
    uint32_t net_attachment_count = htonl(attachment_count);
    uint64_t net_log_length = htonl(log_length);
    uint64_t net_message_length = htonl(message_length);
    char maximal_header[MAX_HEADER] = {0};
    memcpy(maximal_header+target, &net_version, 4);
    target += 4;
    memcpy(maximal_header+target, &net_attachment_count, 4);
    target += 4;
    memcpy(maximal_header+target, &net_log_length, 8);
    target += 8;
    memcpy(maximal_header+target, &net_message_length, 8);
    target+=8;
    memcpy(maximal_header+target, recipient, recipient_length);
    target += recipient_length;
    memcpy(maximal_header+target, local_account, local_account_length);
    target += local_account_length;
    memcpy(maximal_header+target, local_domain, local_domain_length);
    target += local_domain_length;
    memcpy(return_buffer, maximal_header, target);
    //Return -1 if error
    #ifdef DEBUG
    printf("Complete buffer contents: \n");
    for (uint32_t i =0; i<target; i++)
    {
```
Listing C.2: Build Message Body Function

```c
int32_t build_message(unsigned char * body, long body_length, unsigned char * recipient_key,
                      char * attachments, long attachments_length, unsigned char * cipher_buffer)
{
    //Step 1: Generate curve25519 key from ed25519 key for use with crypto_box_seal
    unsigned char recipient_curve25519_key[crypto_scalarmult_curve25519_BYTES];
    crypto_sign_ed25519_pk_to_curve25519(recipient_curve25519_key, recipient_key);
    //Step 2: Encipher body and attachments
    #ifdef DEBUG
    printf("Building message with body_length = %ld and attachments_length = %ld. 
           crypto_box_SEALBYTES = %d\n", body_length, attachments_length, crypto_box_SEALBYTES);
    #endif /*DEBUG*/
    uint64_t cipher_text_length = body_length+crypto_box_SEALBYTES;
    char * crypto_buffer = calloc(1, cipher_text_length + attachments_length);
    unsigned char ciphered_body[cipher_text_length];
    memset(ciphered_body, 0, cipher_text_length);
    //memset ciphered_body to zero here
    print_buffer(body, body_length, "Prepacked plaintext", body_length, 1);
    crypto_box_seal(ciphered_body, body, body_length, recipient_curve25519_key);
    //Step 2: copy encrypted contents to the buffer working
    memcpy(crypto_buffer, ciphered_body, cipher_text_length);
    memcpy(crypto_buffer+cipher_text_length, attachments, attachments_length);
    //Step 3: Return everything as cipher_buffer
    #ifdef DEBUG
    printf("Message size is %ld\n", (cipher_text_length+attachments_length));
    #endif /*DEBUG*/
    memcpy(cipher_buffer, crypto_buffer, (cipher_text_length+attachments_length));
    free(crypto_buffer);
    // print_buffer(cipher_buffer, cipher_text_length, NULL, sizeof(cipher_buffer), 1);
    return (cipher_text_length+attachments_length);
}
```

Listing C.3: Display Message Function

```c
int32_t display_message(char * message_path, char * private_key_buffer, char *
                         public_key_buffer, uint32_t key_version)
```
int32_t mail_file_descriptor = 0;
if((mail_file_descriptor = open(message_path, O_RDONLY))<0)
{
    perror("open");
    print_to_log("Opening message to display has failed", LOG_ERR);
    return -1;
}
if (key_version!=1)
{
    print_to_log("Incorrect key key_version", LOG_ERR);
    return -1;
}

//Generate crypto_box_seal keys from ed25519 keys
#ifdef DEBUG
print_buffer(public_key_buffer, 32, "public Key", 32, 1);
print_buffer(private_key_buffer, 64, "private key", 64, 1);
#endif /*DEBUG*/
unsigned char box_public_key[crypto_scalarmult_curve25519_BYTES];
unsigned char box_secret_key[crypto_scalarmult_curve25519_BYTES];
crypto_sign_ed25519_pk_to_curve25519(box_public_key, public_key_buffer);
crypto_sign_ed25519_sk_to_curve25519(box_secret_key, private_key_buffer);

//Header buffers
uint32_t message_version = 0;
uint32_t attachment_count = 0;
uint64_t log_length = 0;
uint64_t message_length = 0;
char recipient[255] = {0};
char recipient_domain[255] = {0};
char sender[255] = {0};
char sender_domain[255] = {0};
if (read(mail_file_descriptor, &message_version, 4)<0)
{
    perror("read message_version");
    print_to_log("Failed to read first 4 bytes from mail_file_descriptor", LOG_ERR);
    return -1;
}
message_version = be32toh(message_version);
if(message_version!=key_version)
{
    perror("Version mismatch");
    print_to_log("Version difference between message beign read and key provided", LOG_ERR);
    return -1;
}
if (read(mail_file_descriptor, &attachment_count, 4)<0)
{
    perror("read attachment_count");
    print_to_log("Failed to read second 4 bytes from mail_file_descriptor", LOG_ERR);
    return -1;
}
attachment_count = be32toh(attachment_count);
if (read(mail_file_descriptor, &log_length, 8)<0)
{
    perror("read log_length");
    print_to_log("Failed to read 8 bytes from mail_file_descriptor", LOG_ERR);
    return -1;
}
log_length = be64toh(log_length);
if (read(mail_file_descriptor, &message_length, 8)<0)
{
    perror("read message_length");
    print_to_log("Failed to read 8 bytes from mail_file_descriptor", LOG_ERR);
    return -1;
}
message_length = be64toh(message_length);
if (read_until(mail_file_descriptor, recipient, 255, '\0')<0)
{
    perror("read_until");
    print_to_log("read_until failed to read message recipient", LOG_ERR);
    return -1;
}
if (read_until(mail_file_descriptor, recipient_domain, 255, '\0')<0)
{
    perror("read_until");
    print_to_log("read_until failed to read message recipient_domain", LOG_ERR);
    return -1;
}
if (read_until(mail_file_descriptor, sender, 255, '\0')<0)
{
    perror("read_until");
    print_to_log("read_until failed to read message sender", LOG_ERR);
    return -1;
}
if (read_until(mail_file_descriptor, sender_domain, 255, '\0')<0)
{
    perror("read_until");
    print_to_log("read_until failed to read message sender_domain", LOG_ERR);
    return -1;
}
unsigned char * encrypted_message_body = calloc(1, message_length);
unsigned char * plain_message_body = calloc(1, message_length-crypto_box_SEALBYTES);
int32_t bytes_read = 0;
if((bytes_read = read_n_bytes(mail_file_descriptor, encrypted_message_body, message_length)) < 0)
{
    perror("read_n_bytes");
    print_to_log("Reading ciphertext from mail_file_descriptor has failed", LOG_ERR);
    return -1;
}
#if debug
printf("Message length = %ld, bytes_read = %d\n", message_length, bytes_read);
print_buffer(encrypted_message_body, bytes_read, "ciphertext", bytes_read, 1);
#endif /*DEBUG*/
if (crypto_box_seal_open(plain_message_body, encrypted_message_body, message_length,
        box_public_key, box_secret_key) != 0)
{
    perror("crypto_box_seal_open");
    print_to_log("crypto_box_seal_open failed to decrypt message", LOG_ERR);
    return -1;
}
print_buffer(plain_message_body, (message_length-crypto_box_SEALBYTES), "plaintext", (message_length-crypto_box_SEALBYTES), 1);
for (uint64_t i = 0; i < (message_length-crypto_box_SEALBYTES); i++)
{
    printf("%c", plain_message_body[i]);
}
printf("\n");
return 0;
BIBLIOGRAPHY


