UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

IMAP proxy to sanitize attachments

Dissertation presented by
Xavier SCHUL

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Ramin SADRE

Reader(s)
Alexandre DULAUNOY, Gorby KABASELE NDONDA , Etienne RIVIERE

Academic year 2017-2018

Abstract

As it will be explained in this document, current anti-viruses are still not able to spot most of 0-day malwares. Additional tools are then required to complement these anti-viruses. This is the reason for the implementation of PyCIRCLeanMail, an email sanitizer based on active content detection and blacklist of extensions known as malicious.

PyCIRCLeanMail required a support based on the Internet Message Access Protocol (IMAP) and transparent to users who do not have access to their mail servers. This kind of support corresponds to an IMAP transparent proxy but this is not freely available. Therefore, this thesis aims at multiple objectives: Support the PyCIRCLeanMail implementation and provide a generic and scalable implementation of an IMAP transparent proxy for the open-source community.

More precisely, in this paper, the concepts of IMAP and proxy have been explained. After explaining the current situation with anti-virus, PyCIRCLeanMail has been presented. Then, the implementation of the IMAP transparent proxy and modules have been described and their performance has been measured to evaluate their effectiveness.

Acknowledgements

I would like to thank Professor Ramin Sadre, who provided me wise advice and who has always been available to answer my questions. Then, from the Computer Incident Response Center Luxembourg (CIRCL), I would like to thank Raphaël Vinot and Alexandre Dulaunoy for supporting me, giving me the opportunity to work on this project, for their expertise and with whom it was a pleasure to share ideas and implement them.

I would also like to thank Delphine Mottet and Bernard Schul, who gave me good advice for the correction of the written work.

Finally, I am grateful to my parents and my girlfriend for their support and encouragement during these academical years.

Contents

Chapter 1

Introduction

EMAIL is the most frequently used delivery mechanism for malware. Attackers find new ways for viruses to spread and become less detectable every day. It has been demonstrated in several experiments that anti-viruses are ineffective against most 0-day malwares.

This is the reason why PyCIRCLeanMail[1] was implemented by the CIRCL[2], a government-driven initiative designed to provide a systematic response to computer security threats and incidents. PyCIRCLeanMail analyzes the files statically and without running their content in order to label as dangerous each file containing or likely to contain active content and rejects particular extensions. Thanks to this methodology, this tool is very effective to spot emails intended for an attack.

Once the tool was implemented, it needed a transparent support. The target users were small businesses that do not have access to their mail servers and do not have the financial means for sophisticated sanitizing tools. It was, therefore, necessary to integrate this tool into an IMAP transparent proxy.

However, since there is no open-source IMAP transparent proxy, this proposition of thesis has been released. In collaboration with the CIRCL, the initial project was to implement a proxy with PyCIRCLeanMail as a base component for a standalone email sanitizer. Later, another module related to Malware Information Sharing Platform (MISP) [1], an open-source threat intelligence platform and open standards for threat information sharing, has been implemented.

The requirements of this thesis were therefore to implement a reliable, generic and scalable IMAP transparent proxy containing PyCIRCLeanMail and MISP modules.

This document is intended to be read by anyone with a minimum of knowledge of the Transmission Control Protocol (TCP) and the Transport Layer Security (TLS) protocol. The source code of this thesis is available at `https://github.com/xschul/IMAProxy`.

Before going into the details of the implementation, a summary of all the concepts necessary to understand the implementation is given. These notions are about IMAP and transparent proxies. Then, the problem of anti-viruses, of which a foretaste was given at the beginning of this chapter, will be exposed in order to introduce the PyCIRCLeanMail tool. Next, we will dive into the implementation of the proxy and its modules. Finally, we will finish with some performance tests before summarizing and giving some indications of use as a conclusion.

[1] `https://github.com/CIRCL/PyCIRCLeanMail`
[2] `http://circl.lu/`

Chapter 2

Theoretical base

This chapter presents the main concepts that will be used throughout this thesis, namely, the Internet Message Access Protocol (IMAP) and the transparent proxies.

The first section explains the purpose of the IMAP and the way it should be used. Then, the last section will give some definitions about proxy servers.

2.1 IMAP

The next information has been taken from the Request For Comments (RFC) 3501 [2] with the aim of providing a synthesis of this protocol for the good understanding of this thesis.

2.1.1 Protocol overview

The last version (4rev1) of the IMAP, also called IMAP4rev1, is an Internet standard protocol used by e-mail clients to retrieve e-mail messages from a mail server over a TCP connection. It is defined by the RFC 3501 [2]:

This protocol allows a client to access and manipulate electronic mail messages on a server. IMAP4rev1 permits manipulation of mailboxes (remote message folders) in a way that is functionally equivalent to local folders. IMAP4rev1 also provides the capability for an offline client to resynchronize with the server.

An IMAP server typically listens on port number 143 and on port number 993 over an Secure Sockets Layer (SSL)/TLS connection. This protocol handles a wide range of commands. A client can request a server to permanently remove, append, search, parse RFC 2822 [3] and RFC 2045 [4] or fetch messages, set and clear flags. Operations on mailboxes like their creation, deletion or renaming are also supported.

IMAP should not be confused with the Post Office Protocol (POP) which allows clients to get access to their email from a remote server as well, but simply downloads email on one device and does not always delete the email from the remote server. This is thus impractical when using other devices because emails might not have been deleted or correctly flagged. Moreover, folders created and organized on one device will not be replicated on the other devices.

IMAP should also not be confused with the Simple Mail Transfer Protocol (SMTP) which is used to send messages to a mail server. On the other hand, IMAP retrieves the emails on the server.

2.1.2 Commands and responses syntax

Each interaction transmitted by the client and the server are in the form of a string ended with a CRLF (`\r\n`) which means Carriage Return, Line Feed. Carriage Return (`\r`) is used to send the cursor to the beginning of the line. Line Feed (`\n`) is used to advance the terminal of one line. In Unix, unlike Windows programs, CR is implied by LF.

Client commands

Each client command is prefixed with an identifier. This identifier, called "tag", is an alphanumeric string chosen by the client. For example, the Python library *imaplib* uses four characters followed by an integer (e.g., ZRTF2, JHTY43, etc.), Thunderbird only uses digits (e.g., 1, 2, 3, etc.) and, for each command, the letters remain the same but the integer of the tag is incremented. However, the email program Outlook generates a completely different tag for each command.

Server responses

From a client command, the last response of the server is always a *command completion response* indicating success or failure of the operation. The tag used in the response is the same as in the initial client command. There are three possible *server completion responses*: `OK` (success), `NO` (failure), or `BAD` (unrecognized command or command syntax error).

The next example shows the interaction between a client (**C:**) and a server (**S:**). The `CLOSE` command permanently removes all messages that are `\Deleted` flagged in the current mailbox. After the server executed this command, it directly returns a *command completion response*.

```
C:   10 CLOSE
S:   10 OK CLOSE completed
```

When the server has to transmit data to the client, it uses *untagged responses* that are prefixed with the token "*".

In the following example, the `EXPUNGE` command has the same purpose as the `CLOSE` command. However, before returning a *command completion response* to the client, an *untagged* `EXPUNGE` *response* is sent for each message removed.

```
C:   5 EXPUNGE
S:   * 7 EXPUNGE
S:   * 9 EXPUNGE
S:   5 OK EXPUNGE completed
```

When the server is ready to accept the continuation of a command from the client, it uses the *command continuation request response* that is prefixed by a "+" token. This token can be followed by a line of text.

When the server reads an `APPEND` command, it expects the client to send the content of the email to append. Once the client has sent the email, the server ends the interaction with a *command completion response*.

```
C:   4 APPEND "INBOX"
S:   + Ready for email content
C:   From: Will Imap
C:   Subject: Example
```

```
C:
C:   Body of the email
C:
S:   4 OK APPEND completed
```

2.1.3 States

In IMAP4rev1, commands are only valid in certain states. The IMAP commands and their corresponding valid states are shown in table C.1. Once the TCP connection between client and server is established and until it is closed, an IMAP connection is in one of four different states. These states are *Not Authenticated*, *Authenticated*, *Selected* and *Logout* represented by the figure 2.1 in blue. If the client attempts a command in an inappropriate state, the server will respond with a `BAD` or `NO` *command completion response*.

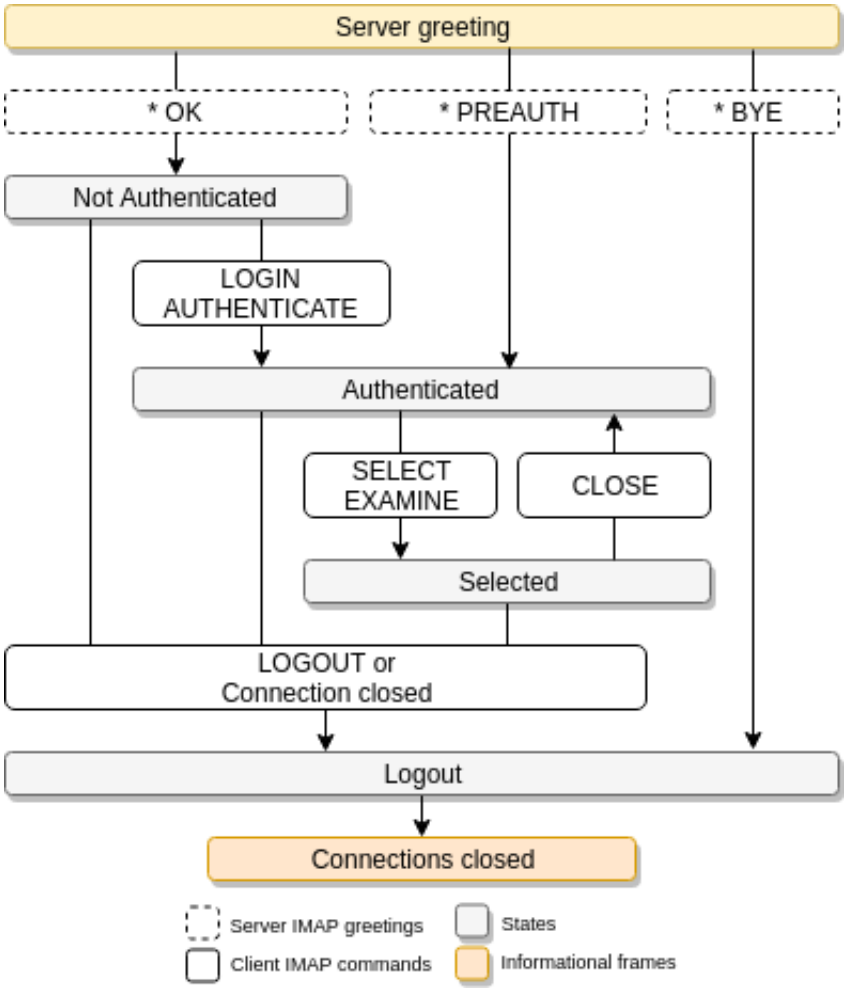

Figure 2.1: The IMAP4rev1 state and flow diagram

Once the TCP connection is complete between the client and the server, the server sends its greeting:

- `OK` greeting: The state of the client becomes *Non Authenticated*.
- `PREAUTH` greeting: It indicates that the connection has already been authenticated by external means. Therefore, no `LOGIN` or `AUTHENTICATE` command is needed and the state of the client becomes *Authenticated*.
- `BYE` greeting: The server rejects the connection and the client state is *Logout*.

Not Authenticated

In the most used mail client like Thunderbird and libraries, this state has two main purposes: negotiate the capabilities of the server and, depending on these capabilities, authenticate the client to the server.

When a `CAPABILITY` command is received, the server responds with the list of capabilities it supports. These capabilities refer to authentication mechanism, extensions or revisions. For example, the server might support the `MOVE` extension defined by the RFC6851 [5] and the `PLAIN` authentication mechanism.

```
C:   1 CAPABILITY
S:   * CAPABILITY IMAP4rev1 MOVE AUTH=PLAIN
S:   1 OK CAPABILITY completed
```

Once the capabilities are sent, the client authenticates to the server using one of two commands:

- `LOGIN`: The client directly sends his credentials in plaintext.

  ```
  C:   2 LOGIN username password
  S:   2 OK LOGIN completed
  ```

- `AUTHENTICATE`: The client sends his credentials using an authentication mechanism given the capabilities of the server. In our example, the mechanism is `PLAIN` and the client response will be parse as defined in the RFC4616 [6].

  ```
  C:   2 AUTHENTICATE PLAIN
  S:   +
  C:   dXNlcm5hbWUicGFzc3dvcmQi
  S:   2 OK AUTHENTICATE completed
  ```

Logout

This state can be reached via the client or the server:

- The client closes the connection by sending a `LOGOUT` command and the server responds with an untagged `BYE` response and a command completion response. The client must read these responses before closing the connection.

  ```
  C:   12 LOGOUT
  S:   * BYE IMAP4rev1 Server logging out
  S:   12 OK LOGOUT completed
  ```

- The server closes the connection by sending an untagged BYE response that contains the reason for having done so.

  ```
  S:   * BYE IMAP4rev1 Server shutdown
  ```

However, if the server detects that the client has unilaterally closed the connection, the server would simply close its connection.

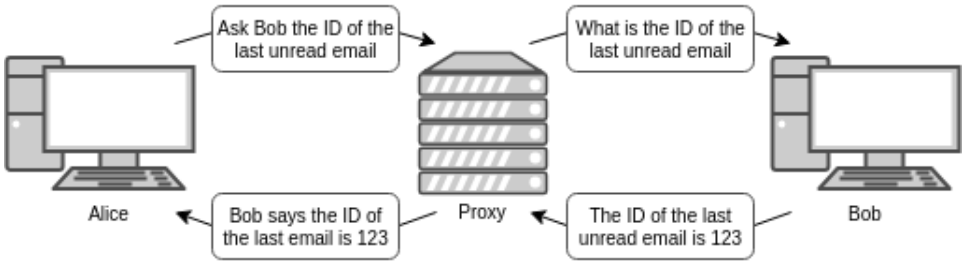

Figure 2.2: Communication between two computers connected through a third computer acting as a proxy.

2.2 Transparent proxy

As illustrated in the figure 2.2, a proxy server is an intermediary between a client and a server. It listens to requests from the client, transmits them to the server and communicates the responses back to the client.

According to the RFC2616 [7], *a "transparent proxy"*, also known as an intercepting proxy, inline proxy, or forced proxy, *is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. A "non-transparent proxy" is a proxy that modifies the request or response in order to provide some added service to the user agent, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering.* Transparent proxies also do not need any configuration on the client side.

Nowadays, the most common proxies are the web proxies. People use them to keep their searches private, to hide their online identity or to bypass IP address blocking. As proxies work as an intermediary, the client is able to hide any trace of his presence. In our case, it will allow the customer to receive a sanitized version of his emails without modifying his requests.

In the next sections, IMAP proxies should not be confused with transparent IMAP proxies. Open source IMAP proxies are already numerous like *Imapproxy*[1] and *Perdition*[2]. In a company, they are used to distribute users between different IMAP servers for load balancing or to maintain persistent connections to an IMAP server using cached connections. In our case, the goal is to have access to the content of queries.

[1] `http://www.imapproxy.org`
[2] `http://horms.net/projects/perdition/`

Chapter 3

Detect malicious attachments

MALICIOUS attachments have been a problem since creation of e-mail messages. Many means have been put in place to detect them in order to sanitize them. However, as anti-virus become more efficient, attackers find new ways for viruses to spread and become less detectable.

In this chapter, we will see how most of current anti-viruses work and analyze their respective advantages and disadvantages based on theory and some experiments. Then, the sanitizer tool used in the thesis implementation, PyCIRCLeanMail, will be presented.

3.1 Current situation

Email is the most frequently used delivery mechanism for malware, according to Symantec [8]. A definition of a malware is given by Eric Filiol, director of *Laboratoire de Virologie et de Cryptologie Opérationnelles* [**?**]:

A Computer Infection or Malware (short for **mal**icious soft**ware**) *is any simple or self-reproducing program which has offensive features and/or purposes and which is without the users' awareness and consent, and whose aim is to affect the confidentiality, integrity and the availability of the system, or which is able to wrongly incriminate the system's owner/user in the realization of a crime or an offense (either in the digital or real world).* This should not be confused with a *virus* that is a malware that has the particularity to spread itself once it is run. However, an anti-malware software is also known as an anti-virus.

In June 2017, one out every 7 users has received a malicious email and this statistic tends to rise over the time [8]. The reason for such a high percentage is the same as the popularity of this distribution channel: it requires low-barrier of entry and is really easy to use. This way, the attacker does not have to put in a lot of effort to gain access to a company's network or a user's credentials. Opening an attachment can be very expensive and users do not always have the financial and technical means to guard against a computer attack.

Moreover, the fraud is not always easy to spot. A trusted but infected contact could send a malicious email without his knowledge. However, downloading and reading the email will not do any harm. As long as the attachment is not opened, the virus should not spread. It has not always been the case when Outlook users could be infected by just opening or previewing a message [9].

3.2 Anti-malware problem

As stated in the previous section, malicious attachments can be received from trusted sources. That is the reason why, in this section, we will focus on malware detection instead of e-mail spam filtering as Spam Assassin.

3.2.1 Methodologies

There are two major ways to protect against malwares, using *signature-based* or *behaviour-based* methodologies. Most organizations rely on the *signature-based* analysis. Yet, this kind of detection only identifies "known" malwares and does not recognize new versions of malicious code that massively emerge every day (e.g., Panda Security reports that on average 200,000 new malware programs appeared every day in 2016 [10]). Newly released forms of malware can only be distinguished by the behavioral analysis [11].

Signature-based methodology

When a new malware is detected, most of the anti-virus solutions analyze it and generate a special handcrafted signature that is added to the signatures database of the anti-virus software and released as an update to their clients [12].

This technique has the main advantage of providing a good protection against millions of threats in circulation today. However, the following paragraphs mention its limitations.

Because of the complexity of some malware, their analysis can take a while [13] [14] [15] [16]. Researchers have to analyze and understand the virus which requires both high skills and time. By the time the virus is first detected and then analyzed, it has already spread vastly. The time needed to analyze is all the more important given the number of new malware samples available daily. Furthermore, there is also a risk of human error during this analysis [17].

An attacker can get around anti-virus by writing encrypted, *polymorphic*, *oligomorphic* and *metamorphic* versions of a virus [18] [19]. A Symantec enterprise paper [20] describes an encrypted virus as a type of virus in which the body is encrypted. The virus itself contains the key for decryption and a decryption engine. The encryption key changes every time the virus is replicated, causing the encrypted body to be different. A *polymorphic* virus has a mutation engine that generates randomized decryption routines changing each time a virus infects a new program [20]. *Oligomorphic* viruses possess a set of decryption engine which allows using a different decryptor after each replication [18]. Other transformation techniques like code permutation, register renaming, expanding and shrinking code, as well as the insertion of garbage code are used [11] to bypass anti-virus detection.

Behaviour-based methodology

As mentioned in an introduction to malware document [21], this detection technique watches the behavior of a running program to determine if it is malicious. It is recommended to scan the program in a virtual environment to avoid any trouble before the potential malware is recognized as malicious. Therefore, the strength of this methodology is to detect unreported malware.

On the other side, compared to the *signature-based methodology*, this one identifies much more programs as false-positives [22]. It means that it labels a file as dangerous when it is not. Another drawback is the fact that some malwares can behave as a normal applications and appear legitimate during the detection.

3.2.2 Efficiency experiments

According to *AV-Comparatives*, the malaware detection rate of well-known commercial anti-virus system ranges from 99.0% to 99.8%. However, malwares used for these statistics have been in circulation for at least four weeks before testing. Results on more recent malicious software samples are exposed in the following paragraphs.

First, a 2016 cloud-based energy efficient system for enhancing the detection and prevention of modern malware [23] estimated the detection rate of 10 best-known anti-viruses between 40% and 80%. A comparable analysis conducted in 2017 [22] reported similar results.

In 2014, Lastline Labs malware researchers studied hundreds of thousands malware samples over a full 365 days [24], testing new malware against 47 anti-virus vendors signatures to determine which, and how quickly, caught the malware samples. Here are the results:

- Only 51% of anti-virus scanners detected 0-day malwares.
- It took an average of two days for at least one anti-virus scanner to detect new malware sample.
- Over one year, no single anti-virus scanner caught every new malware sample.
- After a year, there are samples that 10% of the scanners still do not detect.

3.3 PyCIRCLeanMail

As seen in the previous section, anti-viruses are very effective against what they know. However, against new viruses (less than two or one weeks old), their solutions are very vulnerable. These technologies need to be complemented by other approaches that provide an additional signal for detection.

One of the approaches is PyCIRCLeanMail. Developed and maintained by the CIRCL, this project is a versatile and open-source Python framework[1] designed to sanitize emails.

The tool analyzes the files statically and without running their content in order to label as dangerous each file containing or likely to contain active content. An active content is an additional functionality in a file or program, such as a macro, an add-in or a data connection. During the analysis, archives are unpacked and their content is processed. The following items are labeled as dangerous by the implementation:

- Commonly used malicious extensions [25] present in table C.2. These extensions are known to have active content. An entry also contains the banned extensions from LimeWire[2] that was a P2P file-sharing application.
- Files that do not have the excepted extension.
- Extensions that do not correspond to the mimetype.
- Files without extension.
- Office (Libreoffice and Windows Office) and PDF documents with active content.

[1] `https://github.com/CIRCL/PyCIRCLeanMail`

[2] `https://github.com/wiregit/wirecode/blob/master/components/core-settings/src/main/java/org/limewire/core/settings/FilterSettings.java`

When sanitizing, PyCIRCLeanMail also provides useful logs for the user. For each analyzed file, a log file is generated with informations about the file. Here is an example with a PDF containing active content:

```
{'origFilename': 'contract.pdf', 'maintype': 'application',
'subtype': 'pdf', 'extension': '.pdf', 'processing_type': 'pdf',
'openaction': True, 'dangerous': True}
```

This implementation has the advantages of being entirely in Python, open-source, fast, reliable and being adjustable according to the needs of the user. The user may decide to accept only a few types of files. However, it does not block a 0-day malwares in a non-active content and generates a medium level of false-positive. Indeed, some non-malicious attachments with active content can be seen as dangerous.

Chapter 4

Proxy implementation

At this time, no open-source IMAP transparent proxy exists. It was, therefore, necessary to think of a reliable strategy to connect to a client and its server in order to intercept and transmit requests. In this chapter, each implementation choice will be presented and justified.

The source code of this implementation is available at `https://github.com/xschul/IMAProxy/blob/master/proxy/proxy.py`. The language used is Python to easily integrate the PyCIR-CLeanMail code.

4.1 State and flow

The goals to reach was to make a simple, modular and scalable solution. Figure 4.1 shows the implementation's working and includes the states seen in figure 2.1. A larger figure containing both figures is available in appendix B.1. Each of these steps and methods will be explained in the next sections.

4.2 Implementation of the proxy

4.2.1 Initialization

The initialization of the proxy takes four optional arguments:

- `port` on which the proxy listens. By default, the port is 143 (IMAP port) or 993 (IMAPS port) if `certfile` is specified.
- `certfile` is the path to a certificate. Certificates are electronic documents used to prove the ownership of a public key. They are requirements for SSL/TLS connections. For this implementation, a certificate is also required for a secure connection. When `certfile` is provided, the connection between the client and the proxy is secure. Otherwise, it is not. A secure connection is not always necessary, for example, when the client and proxy are running on the same machine. However, the connection between the proxy and the server will always be secure because, as the proxy acts like a client, it does not have to provide any certificate. In Python, certificates are contained in files. They should be formatted as *PEM* [26], which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
(certificate in base64 PEM encoding)
-----END CERTIFICATE-----
```

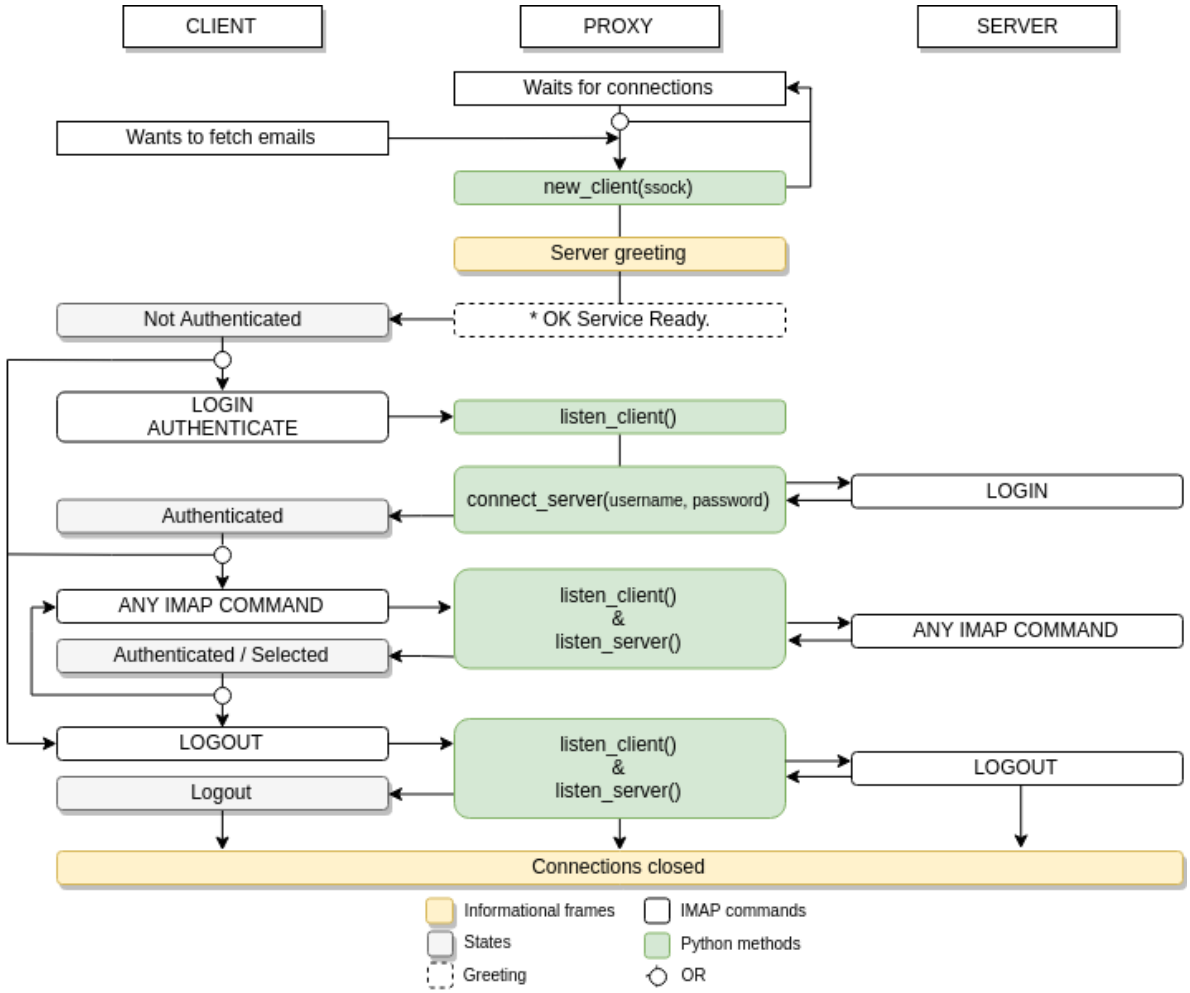

Figure 4.1: Simplified IMAP proxy state and flow diagram

- `max_client` is the maximum number of connection supported by the proxy. It must be taken into account that each time a mail application as Thunderbird or Outlook opens a folder, a new connection is required. Thereby, fetching emails from *Inbox* and *Archive* folders will require two connections.

- `verbose` is a boolean variable that, when *True*, displays in the console the IMAP requests received and sent by the proxy. Enabling this feature helps to understand the protocol and makes debugging easier.

  The next example shows what is displayed on the console when this feature is enabled. The request sent to the proxy is transmitted to the server and the responses are transmitted back to the client.

```
[-->]:  7 UID fetch 1412:* (FLAGS)
  [-->]:  INHL6 UID fetch 1412:* (FLAGS)
  [<--]:  * 16 FETCH (FLAGS (\\Seen) UID 1411)
[<--]:  * 16 FETCH (FLAGS (\\Seen) UID 1411)
  [<--]:  INHL6 OK FETCH completed.
[<--]:  7 OK FETCH completed.
```

  Acting as a transparent proxy, content of the queries remains unchanged. Only the tag is modified.

4.2.2 Connection with the client

As long as the proxy is running, it listens for new connections over the port previously initialized. As shown in listing 4.1, each new connection runs in a different thread. As a reminder, the connection between the user and the proxy is secured if a valid `certfile` was assigned during the proxy initialization.

```
def new_client(ssock):
    if not self.certfile:
        IMAP_Client(ssock, self.verbose)
    else:
        IMAP_Client_SSL(ssock, self.certfile, self.verbose)

while True:
    ssock, addr = self.sock.accept()
    threading.Thread(target = new_client, args = (ssock,)).start()
```

Listing 4.1: Proxy listening for new clients routine

Then, the proxy listens for requests in the `listen_client` function (listing 4.2). It should not happen that the request received does not respect the pattern. The request should be composed by an alphanumerical tag (`[A-Za-z0-9]+`), an optional `UID` keyword (`(UID)?`), an alphabetical command (`[A-Za-z]*`) and some optional flags (`(.*)?`), each separated by a whitespace (`\s`). If the request does not match with this pattern, a `BAD` response is sent to the client and an exception is raised which closes the connection.

The function `getattr(object, name[, default])` calls the method `name` in argument. For example, `getattr(self, 'logout')` is equivalent to `self.logout()`. This way, each command has its own method and is easily called without using `if` and `else` conditions.

```
Request = re.compile(r'(?P<tag>[A-Z0-9]+)'
    r'(\s(UID))?'
    r'\s(?P<command>[A-Z]*)'
    r'(\s(?P<flags>.*))?', flags=re.IGNORECASE)

def listen_client(self):
  while self.listen_client:
    for request in self.recv_from_client().split('\r\n'):
        match = Request.match(request)
        if not match:
            self.error('Incorrect request')
            raise ValueError('Error while listening the client: '
                        + request + ' contains no tag and/or no command')

        self.client_tag = match.group('tag')
        self.client_command = match.group('command')
        self.client_flags = match.group('flags')
        self.request = request

        if self.client_command in Commands:
            getattr(self, self.client_command)()
        else:
```

```
23                 self.transmit()
```

Listing 4.2: listen_client method

The proxy listens to the client as long as the client does not disconnect with the `LOGOUT` command:

```
1 def logout(self):
2     self.listen_client = False
3     self.transmit()
```

Listing 4.3: logout method

Capabilities

Using the `CAPABILITY` command, a server sends the list of supported capabilities. Capability names must either begin with "X" or be standard IMAP4 extensions or revisions. Based on the needs of normal users and inspired by modern IMAP servers, the next items are the capabilities supported by the proxy:

- `AUTH=PLAIN`. It indicates that the proxy supports `PLAIN` authentication mechanism.
- `UIDPLUS` [27]. It adds additional commands for the `UID` requests defined in the IMAP4rev1 [2]. The purpose of these commands is to provide equivalent functionality in a more efficiently way that require less communication between the client and the server.
- `MOVE` [5]. It defines an IMAP extension consisting of two new commands, `MOVE` and `UID MOVE`, that are used to move messages from one mailbox to another.
- `ID` [28] is a command that provides a facility to advertise information on what programs are being used along with contact information. An `ID` exchange between a client mail like Thunderbird and a Microsoft IMAP server will be:

  ```
  C:  4 ID ("name" "Thunderbird" "version" "52.8.0")
  S:  * ID ("name" "Microsoft.Exchange.Imap4Server" "version" "15.20")
  S:  4 OK ID completed
  ```

- `UNSELECT` [29]. As defined in the IMAP4rev1 [2], the `CLOSE` command permanently removes all messages that have the `\Deleted` flag set from the currently selected mailbox, and returns to the authenticated state from the selected state. The `UNSELECT` command simply returns to the authenticated state from the selected state without removing any messages.
- `CHILDREN` [30]. It provides a mechanism for a client to efficiently determine if a particular mailbox has children (or submailbox), without issuing the `LIST` command on each mailbox.
- `NAMESPACE` [31] allows clients to know where located are mailboxes and whether they are private, shared or public.

Much more extensions exist, and adding new capabilities might require modifying the source code of the proxy.

Authentication

At the current state of implementation, the proxy only supports one authentication mechanism, in addition to the `LOGIN` command. It did not seem necessary to implement additional ones because the `PLAIN` mechanism, specified in RFC4616 [6], already allows connecting any user. This mechanism is a simple password authentication command and should be associated with a data confidentiality service provided by a lower layer as SSL/TLS.

However, it is very easy to add a new authentication method to the proxy. For example, if this authentication mechanism is called `PSW`, just add a new method `authenticate_psw()` that uses `connect_server()` and insert in the list of capabilities `AUTH = PWD`. These next pieces of code (4.4 and 4.5) testify the simplicity of the process.

```
1 def authenticate(self):
2     auth_type = self.client_flags.split(' ')[0].lower()
3     getattr(self, self.client_command+"_"+auth_type)()
```

Listing 4.4: authenticate method

```
1 def authenticate_plain(self):
2     self.send_to_client('+')
3     request = self.recv_from_client()
4     # Parse the request to get the username and password
5     (username, password) = [...]
6     self.connect_server(username, password)
```

Listing 4.5: authenticate_plain method

4.2.3 Connection with the server

Once the `AUTHENTICATE` or `LOGIN` command received and parsed to retrieve the username and the password of the client, the hostname is found thanks to a hardcoded dictionary:

```
1 HOSTS = {
2     'hotmail': 'imap-mail.outlook.com',
3     [...]
4     'gmail': 'imap.gmail.com'
5 }
```

Use of this dictionary is unavoidable because there is no way to figure out which IMAP server is related to a user of a domain, and that for several reasons. For example, many organizations have their IMAP servers in their intranet and other organizations map their users to different servers (e.g., users starting with letter A-P use imap-server1.com and users starting with letter Q-Z use imap-server2.com). That is why Thunderbird set up an *ISP database*[1] to enable automatic configuration of the users account to their mail servers.

Once the hostname discovered, the proxy will connect to the real IMAP server thanks to the *imaplib* Python library[2] in the `connect_server` method (listing 4.6). The connection to the server via the `IMAP4_SSL` object is one of the strengths of the implementation. By relying on this

[1] `https://support.mozilla.org/en-US/kb/isp-database`
[2] `https://docs.python.org/3/library/imaplib.html`

actively updated library, we have access to a multitude of IMAP methods already implemented. Thanks to this library, we can make high-level calls but also directly send raw requests to the server.

Login to the server may raise exceptions if, for example, the username or the password is incorrect. In this case, user is warned by a `NO` *command response completion*.

```
def connect_server(self, username, password):
    [...]
    domain = [...] # depends on the username
    hostname = HOSTS[domain]
    self.conn_server = imaplib.IMAP4_SSL(hostname)
    try:
        self.conn_server.login(username, password)
    except imaplib.IMAP4.error:
        self.send_to_client(self.failure()) # NO

    self.send_to_client(self.success()) # OK
```

Listing 4.6: connect_server method

4.2.4 Transmission

Now that the proxy is connected to the client and the server, it can forward requests from one to the other via the `transmit` method (listing 4.7). The only element to change in the query is the tag. The *imaplib* library generates the next tag to send to the server. Once the tag is replaced, the request is sent to the server.

```
def transmit(self):
    server_tag = self.conn_server._new_tag()
    new_request= self.request.replace(self.client_tag, server_tag, 1)
    self.send_to_server(new_request)
    self.listen_server(server_tag)
```

Listing 4.7: transmit method

In the `listen_server` method (listing 4.8), the response of the server is treated differently according to its syntax:

- If the response matches with the *command completion response* pattern, prefixed by the tag of the initial request and a `OK` keyword, it means that client request has been fully processed. In this case, server response is sent to the client and the proxy listens to the client again.

  However, an attacker could exploit a proxy flaw by sending a *command completion response* in the body of an email. Because of the injected *command completion response*, the proxy thinks that the server has finished transmitting and it will stop fetching the email:

```
[-->]:  7 fetch 54 (BODY.PEEK[])
  [-->]:  INHL6 fetch 54 (BODY.PEEK[])
  [<--]:  * 54 fetch (FLAGS (\\Seen) UID 1411)
[<--]:  * 54 fetch (FLAGS (\\Seen) UID 1411)
  [<--]:  From: Attacker
[<--]:  From: Attacker
  [<--]:  Subject: Trying brute-force DoS
```

```
[<--]:  Subject: Trying brute-force DoS
  [<--]:  To: Victim
[<--]:  To: Victim
  [<--]:
[<--]:
  [<--]:  ABCD1 OK fetch proxy is dead because of me
[<--]:  ABCD1 OK fetch proxy is dead because of me
        [.. Command completion response brute-force attempts ..]
  [<--]:  INHL6 OK fetch proxy is dead because of me
[<--]:  INHL6 OK fetch proxy is dead because of me
[ The proxy is now listening to the client but FETCH is not complete ]
```

However, as *imaplib* uses a four letters string randomly generated at the beginning of each connection followed by a digit as a tag, an attacker should generate at least 26^4 (for the alphabetical part of the tag) multiplied by 50 (arbitrarily chosen number for the numerical part of the tag) combinations. This implies that the attacker must send an email with approximately 23 million lines in the body to be sure to accomplish a Denial-of-service (DoS) attack on the proxy. It will not be possible as it would exceed maximum email size limit of 25Mb for Gmail and 10Mb for Outlook.

- The *Untagged response*, prefixed by the '*' keyword, means that the server has additional data to deliver. Each data is transmitted to the client until a *command completion response* is received.

- The *Continuation response*, prefixed by a '+' keyword, means that the server is ready to accept continuation of a command from the client.

  An additional condition has been added so that the proxy does not stop listening to the server when it is fetching an email containing a '+' at the beginning of a line in the body and avoids this case:

```
[ The body of an email is being fetched ]
  [<--]:  For my birthday, there will be you
[<--]:  For my birthday, there will be you
  [<--]:  + my family
[<--]:  + my family
[ The proxy is now listening to the client but FETCH is not complete ]
```

```python
def listen_server(self, server_tag):
    while True:
        response = self.recv_from_server()
        response_match = Tagged_Response.match(response)

        ##   Command completion response
        if response_match:
            server_response_tag = response_match.group('tag')
            if server_tag == server_response_tag:
                new_response = response.replace(server_response_tag,
                    self.client_tag, 1)
                self.send_to_client(new_response)
                return

        ##   Untagged or continuation response or data messages
```

```
16          self.send_to_client(response)
17
18          if response.startswith('+') and self.client_command != 'FETCH':
19              ##   Continuation response
20              client_sequence = self.recv_from_client()
21              while client_sequence != '': # Empty request
22                  self.send_to_server(client_sequence)
23                  client_sequence = self.recv_from_client()
24                  self.send_to_server(client_sequence)
```

Listing 4.8: listen_server method

Chapter 5

Modules

In order to keep the proxy as generic as possible so that other developers use it, the tool to sanitize attachments is in a separate file called module. This way, another user can easily remove modules by commenting the line of code that calls this module. In this section, the two implemented modules will be presented.

The implementation of these modules is available at `https://github.com/xschul/IMAProxy/tree/master/proxy/modules`.

5.1 PyCIRCLeanMail

This module is the integration of the tool presented in section 3.3 and its state and flow is represented by the figure 5.1. The idea of this module is to sanitize emails and keep a copy of the originals in a quarantine folder. This process occurs each time the client wants to retrieve its emails using the `FETCH` command:

```
1 def fetch(self):
2     pycircleanmail.process(self)
3     self.transmit()
```

As shown in figure 5.1, when the client fetches an email, we first verify that this email is not already sanitized by fetching the header part containing the signature `X-CIRCL-Sanitizer`. This first step is fast. Recovering the entirety of an email would have been counter-productive if this email had already been sanitized. If signature is present and value is correct, the email has already been disinfected and the client request is sent to the server. Otherwise, the disinfection process starts.

First, the entire email is retrieved and passed as an argument to PyCIRCLeanMail. The output will be the sanitized email. The `X-CIRCL-Sanitizer` and its value (more information about this value in subsection 5.1.1) are added in the headers and the email is appended on the server using the `APPEND` command in the current mailbox. Then, a copy of the original email is created with the `X-CIRCL-Sanitizer` signature and another value is appended to the server. Finally, the original email is deleted. When this process is complete, the client `FETCH` request is sent to the server. However, the server will respond that the email does not exist, since it has just been deleted. A new `FETCH` request will be required to get the sanitized version of the email. If an exception is raised during the process, an error is notified in the console and the `X-CIRCL-Sanitizer` signature will be associated with an error value.

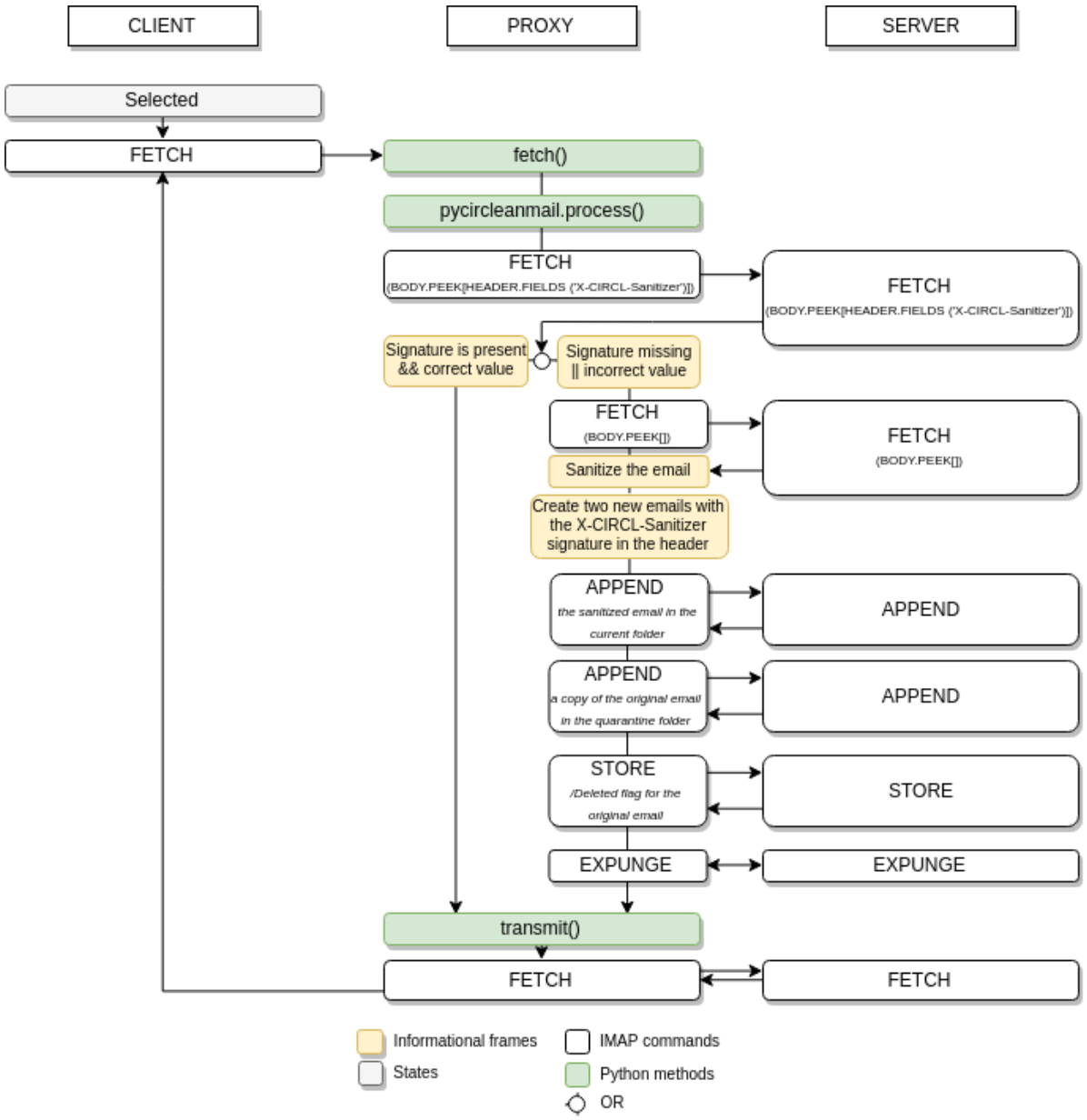

Figure 5.1: Simplified PyCIRCLeanMail module state flow diagram

5.1.1 Security consideration

This module becomes useless if a malicious person sends an email with the `X-CIRCL-Sanitizer` signature, its default value and some logs. To prevent this case, a user can change the value of the `X-CIRCL-Sanitizer` signature by a secret password. The user just has to change the values in the code:

```
VALUE_ORIGINAL = 'my_secret_passw'
VALUE_SANITIZED= 'my_other_secret_passw'
```

The only way for an attacker to find this password is to have access to the mailbox. A forwarded email does not contain the `X-CIRCL-Sanitizer` signature so the user is not likely to share this password.

5.2 MISP

MISP [1] is *a threat intelligence platform for sharing, storing and correlating Indicators of Compromise of targeted attacks, threat intelligence, financial fraud information, vulnerability information or even counter-terrorism information*[1].

This module departs slightly from the main purpose of this thesis but remains just as important in the field of fight against malware. The idea of this module is to easily send to MISP an email to analyze. A user who wishes to share a dangerous email only has to drag the email and drop it to the "MISP" folder. This email will be fetched and attached to a new email that will be sent to the SMTP MISP server.

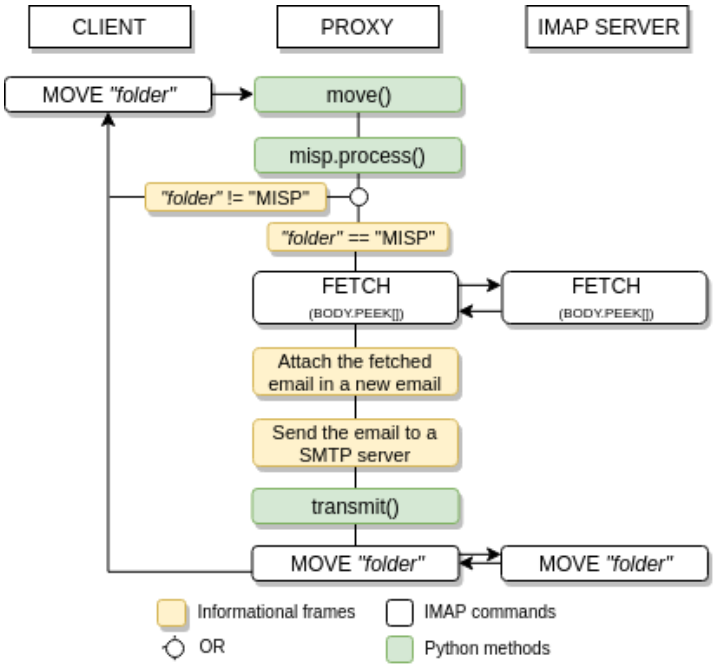

Figure 5.2: Simplified MISP module state flow diagram

The module is, again, really easy to integrate in the implementation:

```
1 def move(self):
2     misp.process(self)
3     self.transmit()
```

[1] `http://www.misp-project.org/features.html`

5.3 New modules

As seen in the two previous modules, adding a new one is very simple. Moreover, by having the `IMAP_Client` object as an argument, the module benefits from an access to the connection to the client, to the server, to the last request received from the client and to the folder in which client is located. More information about the integration is available at the appendix A.3.

Chapter 6

Performance

In this section, implementation of the proxy and its modules is tested and measured to assess whether this solution is sustainable for a large number of customers and for large attachments.

6.1 Reliability

In order to test the proper functioning of the proxy, a series of tests `test_proxy` corresponding to a typical use of a mailbox has been implemented. Inspired by the tests of the Python library *imaplib*, it tests all the commands marked as *Tested* (✓) in the table C.1 and others like `NAMESPACE`. It seemed not necessary to test the other commands because of their similar operation to those already tested as well as their low use.

Another series of tests has been implemented to challenge the proper functioning of the sanitizer. First, a new email is appended and fetched. Then, we verify that the last email in the current mailbox and the last email in the quarantine mailbox have the X-CIRCL-Sanitizer signature.

The implementation of the proxy with PyCIRCLeanMail and MISP modules has also been manually and regularly tested using Thunderbird.

6.2 Time

Two performance tests were done to evaluate the time complexity of implementation. The results can be found in the figure 6.1. All tests were performed on a single virtual machine running on the same computer as the client. The results may be different from one computer to another and from an Internet connection to another.

The first test compares time spent to fetch the last 10 emails from 1 to 13 simultaneous clients using threads. The modules have been deactivated beforehand. These clients were initiated from the same email address and the IMAP server (`imap-mail.outlook.com`) does not seem to accept more than 13 simultaneous connections from the same user.

The goal was to simulate a likely use of the proxy. We observe that, up to 3 simultaneous clients, the time to process the requests is quite good. Then, time seems to increase exponentially which means that the proxy is not suitable for a large-scale use. However, it is quite reasonable to

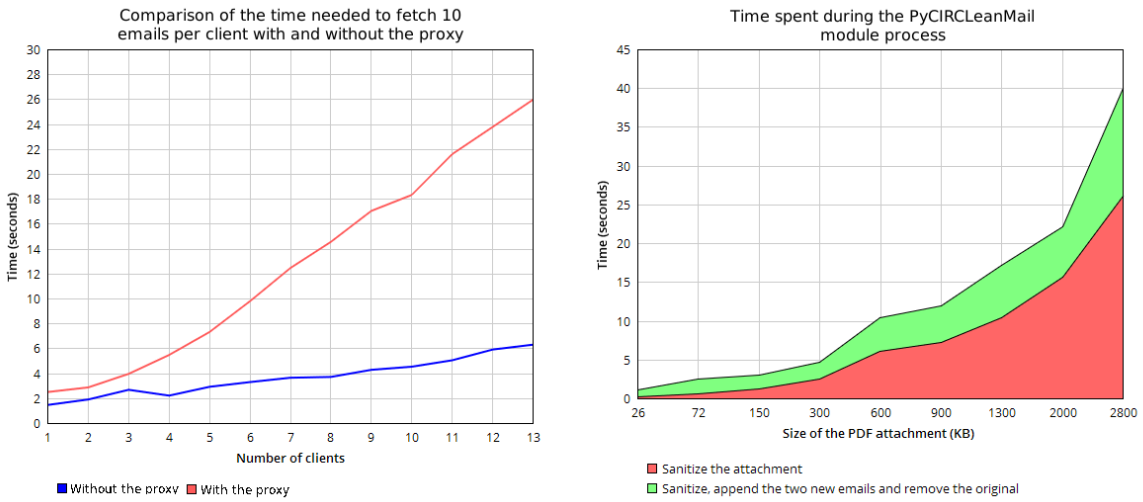

Figure 6.1: Time performance charts

use it for a small company of a dozen people since, in normal use, these users will not continuously fetch a large number of emails.

A second test seemed useful in order to estimate the time spent by the PyCIRCLeanMail module to sanitize PDF attachments. At first glance, time spent to sanitize attachments is exponential compared to the file size. Compared to the analysis scan time of an anti-virus, the time to process a 2800KB (kilobytes) file does not seem disproportionate. The process can be much shorter for other types of files like images and almost instantaneous for banned extensions (see appendix C.2). An email without attachments takes about 1 second to be processed (this includes appending of the sanitized, copy of the email and deletion of the original email).

In conclusion, additional time generated by the proxy and its modules on a single virtual machine may seem insignificant for a normal use. In this thesis, a usage is considered as normal when a user receives a hundred emails a day with a large majority of emails without attachments, an average amount of emails with attachments lighter than 1MB (megabytes) and a small number of emails with attachments heavier than 1MB.

Chapter 7

Conclusion and Future work

THROUGHOUT this thesis, the importance of PyCIRCLeanMail, a complementary tool to anti-viruses based on active content detection and extensions blacklist, has been justified by the inefficiency of most anti-viruses to detect 0-day attacks. The initial goal of this thesis was to support this tool but the project has evolved. The final requirements were to implement a reliable, generic and scalable IMAP transparent proxy supporting PyCIRCLeanMail and MISP, an open-source threat intelligence platform, modules.

It has been shown, in the implementation part, how the proxy works and how messages from clients and servers are handled and how much the proxy. Series of tests have been implemented to prove the proper functioning of the implementation.

Following the performance tests, it was concluded that the proxy is not suitable to support a large number of simultaneous connections. With the PyCIRCLeanMail module, the time taken to sanitize a PDF document seems exponential to its size.

As stated throughout this thesis, the implementation has been designed in a modular and scalable way. A very large number of modules could be integrated in order to protect or collect information. Depending on the use of the proxy, some additional capabilities could also be added. Current capabilities were chosen based on capabilities supported on current mail servers as *Outlook* and *Gmail*. In future, we need to be attentive to the evolution of these capabilities.

As future work, it might also be possible to deploy this solution on the cloud for remote manipulation. For the moment, the proxy has only been tested in a virtual machine on the same computer.

Glossary

CRLF means Carriage Return, Line Feed. Escape sequence is `\r\n`. 7

Outlook Free email program available at `https://outlook.live.com/owa/`. 7, 11, 16

Spam Assassin is an Open Source anti-spam platform giving system administrators a filter to classify email and block spam (unsolicited bulk email). Available at `https://spamassassin.apache.org/`.. 12

Thunderbird Free email program available at `https://www.thunderbird.net/`. 7, 9, 16, 18, 19, 27

Acronyms

CIRCL Computer Incident Response Center Luxembourg. 2, 5, 13

DoS Denial-of-service. 21

IMAP Internet Message Access Protocol. 6, 8, 15, 18, 20, 32

MISP Malware Information Sharing Platform. 5, 25

POP Post Office Protocol. 6, 32

RFC Request For Comments. 6

SMTP Simple Mail Transfer Protocol. 6, 25

SSL Secure Sockets Layer. 6, 15, 19

TCP Transmission Control Protocol. 5, 6, 8

TLS Transport Layer Security. 5, 6, 15, 19

Appendix A

Installation

A.1 Installing and running the proxy in a virtual machine

The virtual machine must support the different requirements[1] and have `git` and `pip3` installed. To communicate with the virtual machine, modify the **Network** settings to enable an additional **Bridged Adapter** on the interface of your choice.

Then, simply clone the repository, install the requirements and start the proxy:

```
git clone https://github.com/xschul/IMAProxy.git
cd IMAProxy
pip3 install -r requirements.txt
python3 proxy/proxy.py -h
```

A.1.1 Run the tests

- Test the proxy:

  ```
  python3 tests/test_proxy.py $username $password $ip_proxy
  ```

- Test the proxy and the PyCIRCLeanMail sanitizer:

  ```
  python3 tests/test_sanitizer.py $username $password $ip_proxy
  ```

A.2 Connect a client to the proxy

When connecting a client to the proxy, the usual maneuver is to replace the **Server Name** by the IP address of the proxy and the authentication method should be **Normal password** (that corresponds to the `PLAIN` authentication mechanism). The port number and the connection security (None or SSL/TLS) can also be modified depending on the initialization of the proxy (see section 4.2.1).

A.2.1 Thunderbird

When adding a new account

Configure the new account until the choice between IMAP and POP. Click on **Manual config** and replace the **Server hostname** of IMAP by the IP address of the virtual machine as shown in figure A.1.

[1] https://github.com/xschul/IMAProxy/blob/master/requirements.txt

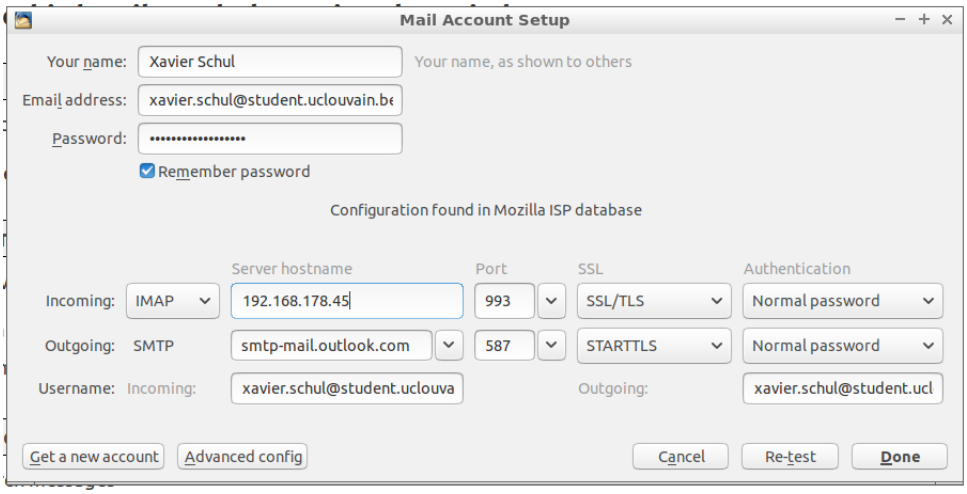

Figure A.1: Link a new account to the proxy

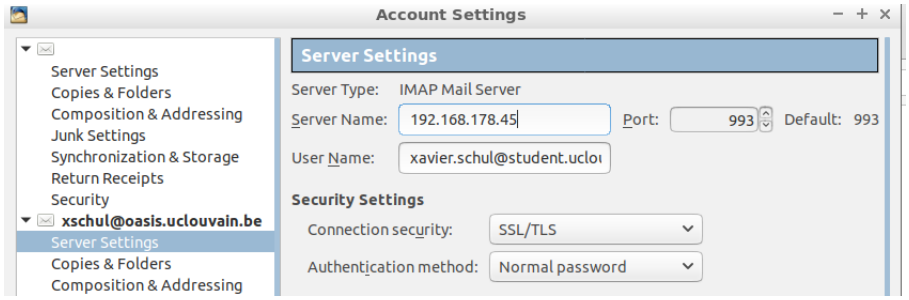

Figure A.2: Link an already configured account to the proxy

Already configured account

Go to **Account Setting** and select the **Server Settings** of the email address of your choice. Then, simply replace the **Server Name** with the IP address of the proxy as shown in figure A.2.

A.2.2 Outlook

When adding a new account

Click on **Advanced options** and set up your account manually. Choose **IMAP** type account and insert the proxy IP address into the **Server** field for **Incoming mail**.

Already configured account

Unfortunately, you must remove the existing account and re-add it.

A.3 Support a new command and add a module

If the command is not present in the `COMMANDS` global variable, add it to the list and create the corresponding method. For example, if you want to support the command `APPEND`, add it to the `COMMANDS` list and create a new `append` method:

```
COMMANDS = (
    'authenticate',
    'append',
    [...]
    'fetch'
)
```

```
1 def append(self):
2     # Add your module here
3     self.transmit()
```

The call to the `transmit` method is necessary if you want the request to be transmitted to the server.

Appendix B

Large figures

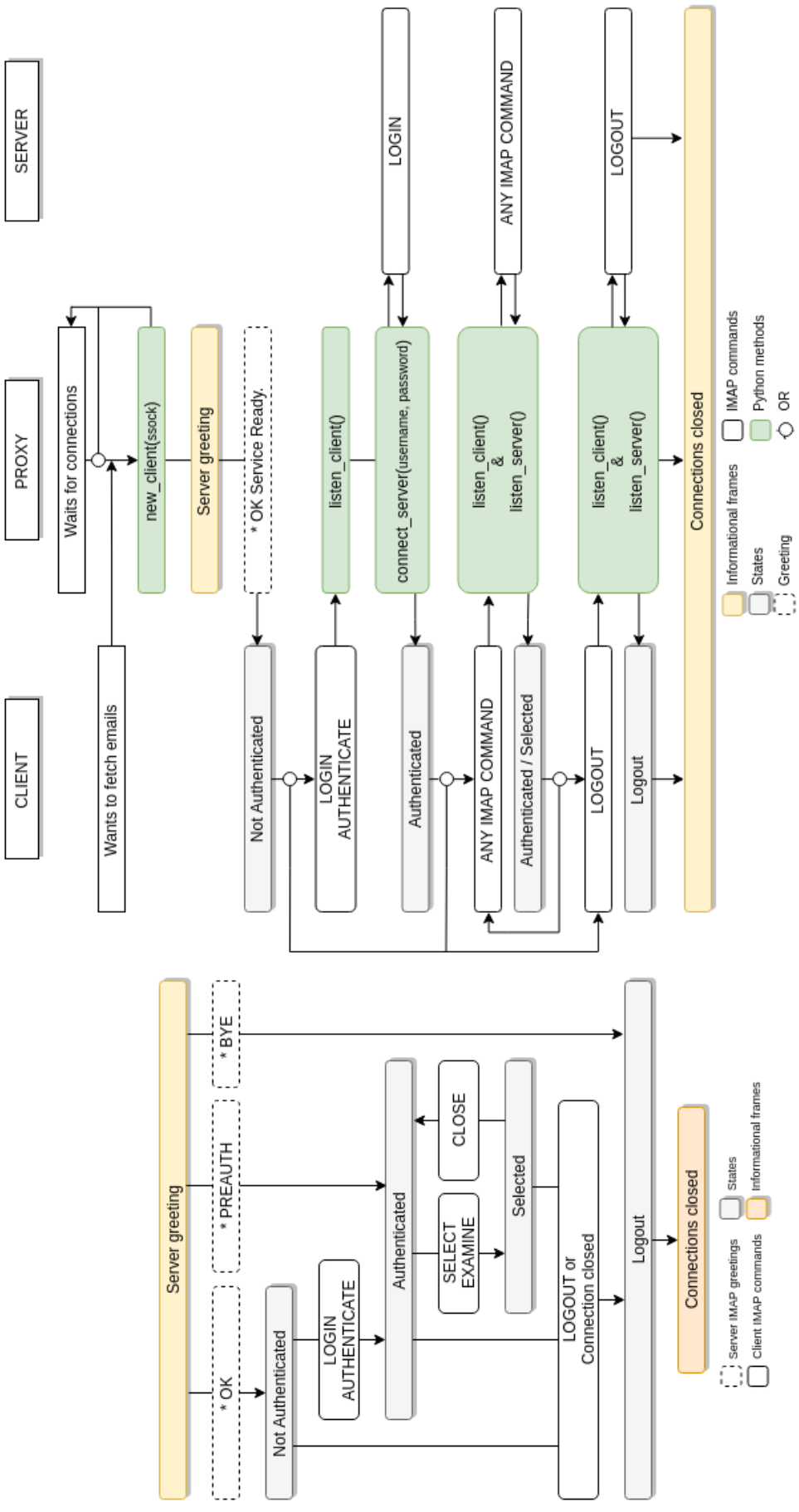

Figure B.1: IMAP client and IMAP Proxy stateflows

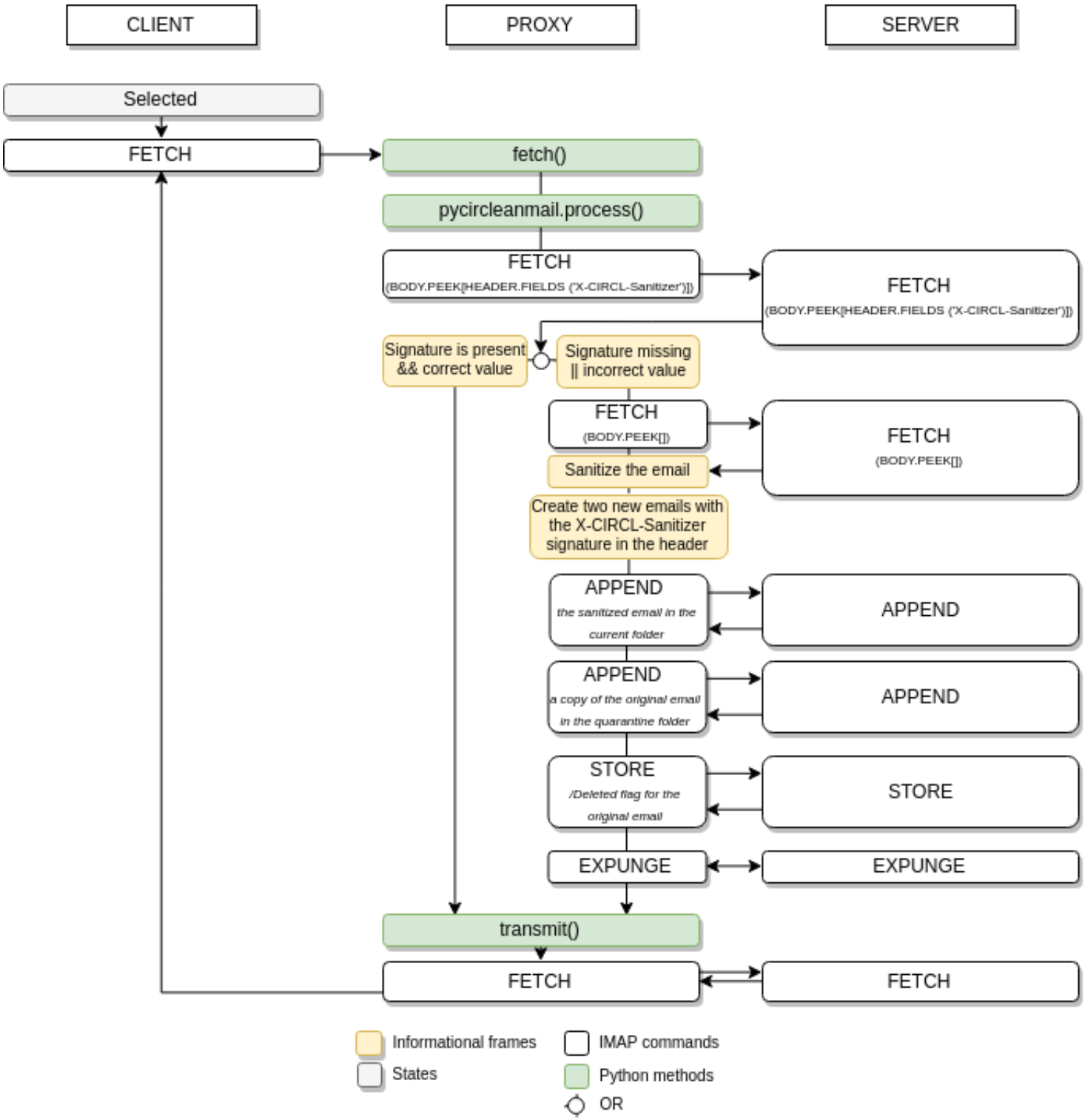

Figure B.2: Simplified PyCIRCLeanMail module state flow diagram

Appendix C

Large arrays

Table C.1: RFC3501 IMAP client commands, their corresponding valid states and if these commands are tested in the proxy implementation or not

Command name	**Valid state**	**Tested**
APPEND	AUTH, SELECTED	✓
AUTHENTICATE	NONAUTH	
CAPABILITY	NONAUTH, AUTH, SELECTED, LOGOUT	✓
CHECK	SELECTED	
CLOSE	SELECTED	✓
COPY	SELECTED	
CREATE	AUTH, SELECTED	✓
DELETE	AUTH, SELECTED	✓
EXAMINE	AUTH, SELECTED	
EXPUNGE	SELECTED	✓
FETCH	SELECTED	✓
LIST	AUTH, SELECTED	✓
LOGIN	NONAUTH	✓
LOGOUT	NONAUTH, AUTH, SELECTED, LOGOUT	✓
LSUB	AUTH, SELECTED	
NOOP	NONAUTH, AUTH, SELECTED, LOGOUT	
RENAME	AUTH, SELECTED	✓
SEARCH	SELECTED	✓
SELECT	AUTH, SELECTED	✓
STARTTLS	NONAUTH	
STATUS	AUTH, SELECTED	
STORE	SELECTED	✓
SUBSCRIBE	AUTH, SELECTED	
UID	SELECTED	✓
UNSUBSCRIBE	AUTH, SELECTED	

Table C.2: Commonly used malicious extensions

Types of file	Extensions
Application	.exe, .pif, .application, .gadget, .msi, .msp, .com, .scr, .hta, .cpl, .msc, .jar
Scripts	.bat, .cmd, .vb, .vbs, .vbe, .js, .jse, .ws, .wsf, .wsc, .wsh, .ps1, .ps1xml, .ps2, .ps2xml, .psc1, .psc2, .msh, .msh1, .msh2, .mshxml, .msh1xml, .msh2xml,
Shortcut	.scf, .lnk, .inf
Other	.reg, .dll
Office macro	.docm, .dotm, .xlsm, .xltm, .xlam, .pptm, .potm, .ppam, .ppsm, .sldm
Banned from LimeWire	.asf, .asx, .au, .htm, .html, .mht, .vbs, .wax, .wm, .wma, .wmd, .wmv, .wmx, .wmz, .wvx

Bibliography

[1] C. Wagner, A. Dulaunoy, G. Wagener, and A. Iklody, "Misp: The design and implementation of a collaborative threat intelligence sharing platform," in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, pp. 49–56, ACM, 2016.

[2] M. Crispin, "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1." RFC 3501, Mar. 2003.

[3] P. Resnick, "Internet Message Format." RFC 2822, Apr. 2001.

[4] N. Freed and D. N. S. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies." RFC 2045, Nov. 1996.

[5] A. Gulbrandsen and N. Freed, "Internet Message Access Protocol (IMAP) - MOVE Extension." RFC 6851, Jan. 2013.

[6] K. Zeilenga, "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism." RFC 4616, Aug. 2006.

[7] H. F. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616, June 1999.

[8] B. Nahorney, *Internet Security Threat Report: Email Threats 2017*, October 2017.

[9] S. Systems, *Protecting Microsoft Outlook against Viruses*, February 2009.

[10] PandaLabs, "Cybercrime reaches new heights in the third quarter," October 2016.

[11] J. Cloonan, "Advanced malware detection - signatures vs. behavior analysis," April 2017.

[12] E. David and N. Netanyahu, "Deepsign: Deep learning for automatic malware signature generation and classification," pp. 1–8, 07 2015.

[13] N. D. Grosso, "It's time to rethink your corporate malware strategy," February 2002.

[14] U. Mishra, "Methods of virus detection and their limitations," 08 2010.

[15] A. A. Al Amro S, "A comparative study of virus detection techniques," vol. 9, p. 1566–1573, 2015.

[16] H. Singh, "Computer viruses – analysis of detection techniques and their limitations," vol. 1, pp. 498–504, 2015.

[17] A. A. Al Amro S, "Behavior-based features model for malware detection," vol. 12, p. 59–67, May 2016.

[18] P. Szor, *The Art of Computer Virus Research and Defense*. February 2005.

[19] D. M. A. G. Anita Thengade, Aishwarya Khaire vol. 5, October 2014.

[20] C. Nachenberg, "Understanding and managing polymorphic viruses."

[21] Kalpa, *Introduction to Malware*, January 2011.

[22] A. Zarghoon, I. Awan, J. P. Disso, and R. Dennis, "Evaluation of av systems against modern malware," pp. 269–273, Dec 2017.

[23] Q. K. A. Mirza, G. Mohi-Ud-Din, and I. Awan, "A cloud-based energy efficient system for enhancing the detection and prevention of modern malware," pp. 754–761, March 2016.

[24] G. Vigna, "Antivirus isn't dead, it just can't keep up," May 2014.

[25] C. Hoffman, "50+ file extensions that are potentially dangerous on windows," February 2013.

[26] S. Kent, "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management." RFC 1422, Feb. 1993.

[27] M. Crispin, "Internet Message Access Protocol (IMAP) - UIDPLUS extension." RFC 4315, Dec. 2005.

[28] T. Showalter, "IMAP4 ID extension." RFC 2971, Oct. 2000.

[29] A. Melnikov, "Internet Message Access Protocol (IMAP) UNSELECT command." RFC 3691, Feb. 2004.

[30] R. Cheng and M. Gahrns, "The Internet Message Action Protocol (IMAP4) Child Mailbox Extension." RFC 3348, July 2002.

[31] M. Gahrns and C. Newman, "IMAP4 Namespace." RFC 2342, May 1998.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**