EOF
Eris Onion Forwarding
The secure, peer-to-peer, decentralised anonymous chat network[1]

Nico -telmich- Schottelius

June 10, 2010

[1]Version 0.6.0

Contents

Chapter 1

Introduction

1.1 Copying

Copy it as you like - send corrections to me.

1.2 Abstract

EOF is a chat protocol that supports secure chatting. Secure chatting consists of the following features:

1. Nobody, but the intended receiver(s) know(s) *that* you wrote a message.
2. Nobody, but the intended receiver(s) can view the *message content*.
3. Nobody, but the intended receiver(s) can *verify* the source of the message being you.
4. Nobody, but the intended receiver(s) know(s) *who* you sent a message to.
5. The network must survive attacks of a single attacker.
6. Hard (if not practilcally impossible) to block chatting.

Additionally, for practical reasons, EOF must support the following chat features:

1. Direct chat (”‘message is only seen by one person”’)
2. Group chat (”‘message is sent to specific group, which may consist of more than one person”’)

1.2.1 Hide message sending

We don't think it's possible to hide that you are part of the chat network, because some heuristics will be developed to detect the chat packets. So we use a different idea: Every participant of an EOF network will constantly send chat packets with a pre-defined frequency (for instance every 250 ms). If you don't chat, *noise* is sent.[1] The noise is also used to defend against timing analysis. In case you are sending out a message, the message packet will be added to the queue and sent within the next free time slot.

From outside it can easily be seen, that you are part of the network, but not, if you sent a message.

1.2.2 Hide message content

We encrypt every message via public-key cryptography[1], so that only the receiver can decrypt and view the message content.

1.2.3 Verify sender

Before the encrypt the packet, it is signed via public-key cryptography[1]. Thus only the receiver can verify the message sender.

1.2.4 Hide message receiver

The message packets are always sent indirectly via onion routing[2]. The idea is taken from the Tor project[3], though EOF uses an enhanced version: EOF does not know about entry or exit nodes. If you are the intended receiver you may or may not continue to forward the message, which is defined by the sender of the message. That said, EOF must use source routing[5].

To support onion routing, the sender of a packet needs to encrypt the packet multiple times, once for each host that receives the packet. This may look as follows:

1. Create message (from noise or user input)
2. Create source path
3. Create packet for last peer ("'pkg-last"')
4. Create packet for last-1 peer including *pkg-last*
5. Continue until first peer is reached

[1] Noise is just random data, see below for a more detailled description of noise.

6. Sent packet to first peer

Thus every peer only knows the previous and the next peer.

1.2.5 Reliable against single user attacks

Traditional chat networks depend on one or more central organised servers. An attacker can stop all communication, if she runs a successful denial of service ("'DoS"') attack against the central systems. To protect against this, EOF uses a dynamic peer-to-peer network, which works as long as the minimun number of peers and the destination peer is available. It has no dependency on a central server.

1.2.6 Hide packets in network stream

As said before, we don't think it's possible to hide the participation in the chat network. To be able to send packets, although an attacker *knows* about the participation, EOF embeds all chat packets into other (well known) protocols (which is knows as steganography[12]). EOF does not implement nor specify *transport protocols* itself. The EOF community is urged to implement them in a creative way: Usage of well-known protocols like TCP[6], HTTP[7], SMTP[8] or even transmission of packets on avian carriers[9] are encouraged. The tunneling of EOF packets through those protocols (also know as obfuscation) makes it harder to detect and *block* EOF traffic. If an attacker wants you to stop sending messages, she has to completly remove you from the network, because any open protocol may be (ab)used to encapsulate EOF packets into it.

1.2.7 Real world usability

To be able to be compete with other chat protocols, EOF needs to support *direct* and *group chat*, which is implemented by two different chat destinations:

1. *Peers*

2. *Groups of peers*

A peer is just another person (direct chat), a group of peers is the EOF equivalent of the IRC channel[4]. As there is no central server, groups of peers are managed by each client, and thus the compositions of group members may be different on different peers.

1.3 Motivation

1.3.1 Current implementations are not secure

There are already many different chat protocols available like

- talk
- IRC
- SILC
- Jabber
- ICQ
- Skype

Only two of those protocols contains mandority encryption (SILC, Skype), which still lacks many features for secure chatting.[2] Skype and SILC still depend on a central server architecture. The Skype architecture is not publicly documentated, the executing binary is encrypted, and the system depends on a single company, which excluded it from being a secure chat system.

Because SILC depends on a central server architecture, it was also excluded.

1.3.2 Create more crypto traffic

We also want to convey usage of PGP: currently PGP is quite seldom. Thus if you use PGP, you may be conspicuous. We try to make PGP encrypted packets part of the regular internet traffic, like webtraffic is today.

1.4 The project

The project started in 2007 as an idea of !eof[11] (a friendly hacker community). After several meetings it was clear that we need to do some experiements and create phases to structure the development.

[2] Most of them can be enhanced to use TLS/SSL, but this is not enforced.

1.4.1 The three phases

So we divided the project into the three phases

1. *EOF-1*,
2. *EOF-2*,
3. *EOF-3*.

EOF-1: Finding ideas

The first phase was the so called "'finding ideas"' phase. We did some tests, measured packet sizes and did some theorethic calculations on intervals. There was also some discussion about implementing EOF in a ring structure, like token ring[**?**]. The first try to do the first implementation as a complet modularised version began and stalled after some months of decentral developemnt.

EOF-2: Proof of concept

We are currently working on a prototype in EOF-2. The idea is to get the basic features working and to attract testers and developers.

EOF-3: The final destination

The idea of EOf-3 is to cleanup all parts of EOF-2 and create a "'ready to be used"' software package, that may be used by end users. This includes "'good documentation"' (this document).

1.4.2 Further directions

After the deployment of EOF-3, there may be more work necessary, like

- Deep analysis of security (by us and foreigners),
- enhance performance,
- port to other systems,
- adapt to changed environment,
- ...

1.5 Conventions

1.5.1 Writing convention

There are not many things to take care about, when reading this document:

- *$NAME* is used to mark an environment variable.
- Parameters enclosed in < and > must be specified to the command.
- Parameters enclosed in < and ... > must be specified at least one, but may be repeated. This is only valid for the last parameter.
- Parameters enclosed in [and] can be specified optionally.
- *EOF_L_NAME* is used to specify a certain length

1.5.2 Abbreviations

As some terms are quite often used, we created some abbreviations:

Table 1.1: List of abbreviations

Abbreviation	**Meaning**
EOF	Eris onion forwarding, name of this project
EOFi	An EOF implementation
rEOFi	The reference EOF implementation (also called "'ceofhack"')
EOFs	An EOF subsystem
EOFsdt	EOF simple data type
FLS	Fixed length string

Chapter 2

The EOF standard

This section defines the complete EOF standard. All required operations, features, protocol specifications, paths, interfaces and environment definition, etc. are described.

2.1 Introduction

2.1.1 Version

As soon as this standard is usable and the first version of *rEOFi* is released, it will version number **01** of the standard. Further changes will be documented in this section. All version numbers are two ASCII digits (see below).

2.1.2 Modular setup

As there are many parts needed to realise such a chat system, the EOF standard defines a modular setup to be used:

- EOFi is the central EOFi implementation
- EOFi is the central connection point for all other EOFs

This modular design allows different developers to code each EOFs in a different way (and probably different programming language). A complete implementation consists of at least:

- EOFi - central communication daemon
- UI - at least one user interface
- TP - at least one transport protocol

2.2 Interface description (”‘eofi2any”’)

This section defines which interfaces EOFi offers to communicate with subsystems in general. Each subsystem may use a subset or the full set of interfaces, which is defined in their respective sections.

2.2.1 Stdin, stdout and stderr (via pipes)

EOFi is capable of executing a program and connecting its STDIN, STDOUT and STDERR to EOFi. The rEOFi uses pipes, but other EOFis may use different methods.[1] The three filehandles are used as follows:

Table 2.1: Stdin, stdout, stderr

Name	Used for
stdin	Commands sent from EOFi
stdout	Commands sent from EOFs
stderr	Diagnostic messages sent from EOFs

2.2.2 Unix Sockets

Unix sockets are provided by EOFi to allow external applications to connect to EOFi.

2.2.3 Environment variables

The following environment variable are either used or set by EOFi and are available for all EOFs.

$HOME

The home directory of the user. It is required by EOFi to locate the configuration directory.

$EOF_HOME

The EOFi configuration directory. If set by before EOFi starts, it will skip the autodetection of the configuration directory. This variable is exported to all EOFs.

[1] See pipe(2) and forkexecpipe() in rEOFi.

$EOF_UI_SOCKET

This variable contains the absolute path to the socket, which should be used by user interfaces (UIs).

2.2.4 Asynchronous bidirectional communication

Every connection between any EOFs and EOFi is asynchronous (both sides can start sending on their own) and bidirectional (both sides can send at the same time.

Thus every EOFs and EOFi have to implement one (or more) queues to keep the state of their packet.

To identify an answer to a packet, each packet contains an identification number, which is chosen by the sender (for more information have a look at section 2.6.2 on page 18).

2.3 Paths

2.3.1 EOFi configuration directory

Default case

Normally, $HOME is set and $EOF_HOME is not set. In that case the configuration directory defaults to *$HOME/.ceof*.

$HOME is unset

If the environment variable "'*$HOME*'" is not set, the directory named *.ceof* in the current directory will be used.

$EOF_HOME is set

If the environment variable "'*$EOF_DIR*'" is set, its content will be used to refer to the configuration directory.

2.3.2 User interface (unix socket)

If the environment variable *$EOF_UI_SOCKET* is set, the user interface should connect to this unix socket. Otherwise it should connect to the socket "'ui/socket"' below the EOFi configuration directory (see above for rules how to locate that).

2.4 Configuration

The configuration of EOFi is stored in the cconfig[**?**] format.

2.5 Basic data types (”‘EOFbdt”’)

This section specifies the basic datatypes used in EOF.

2.5.1 The zero byte

The zero byte is a byte with the value 0.

2.5.2 Line feed

The line feed, ”‘\n”’, was used to terminate data sections, but is *DEPRECATED* now.

2.5.3 ASCII numbers

ASCII numbers use the decimal string representation of a number (versus binary representation, which is *never* used between EOFi and EOFs). ASCII numbers are often used in a packet header. ASCII numbers are used to specify the length of the packet (excluding itself).

2.5.4 Strings in general

Strings are transmitted without termination (i.e. no new line, no 0 byte), but are padded with zero bytes, if shorter than the specified length.

2.5.5 Fixed length strings

Fixed length strings contain exactly the specified number of bytes: A 128-byte fixed length string consists of at most 128 bytes of text, which is then not zero terminated! If the text it contains is shorter than the specified length, it must be padded with zero bytes.

2.5.6 Variable length strings

The EOF protocol currently does not specify any variable length strings. All strings are fixed length (see above).

2.5.7 Noise

There are many situations in which an EOFi sends out data to the network, although you did not write a message: In fact, as EOFi **always** sends packets in a fixed interval, it needs to have data to encrypt and send.

Noise can be any type of random data. As the current random number generators are quite expensive, it is recommend to use a huge dictionary, old messages, logfiles, public emails, etc. for noise input.

2.5.8 Unused

To make life harder for attackers we try to make packets always be more or less the same size. That results in fields being present in a packet, which are unsued.

Unused fields should be filled up with noise.

2.6 EOF simple data types ("'EOFsdt"')

The following sections define the datatypes used in EOF related applications. The recommened name for use in source code is added in parentheses after the human understandable name.

2.6.1 EOF commands and command fields (mapping table)

An EOF command is exactly *EOF_L_CMD* bytes long (fixed length string) and contains an ASCII number.

EOF commands are the main method of communication between EOFs and EOFi.

The command field 0 indicates the direction. The command field 1 indicates the EOF subsystem.

Table 2.2: Command fields

Value	Subsystem / Description	Ref
1***	Message is coming from the EOF implementation	
10**	**eofi2tp**: Transport protocols	p20
11**	**eofi2ui**: User interface	p26
12**	**eofi2crypto**: Crypto engine	p33
13**	**eofi2noise**: Noise generator	p??
2***	Message is coming from EOF subsystem (internally)	

20**	**eofi2tp**: Transport protocols	p20
21**	**eofi2ui**: User interface	p26
22**	**eofi2crypto**: Crypto engine	p33
23**	**eofi2noise**: Noise generator	p**??**
3***	Message is coming from outside ("'onion packet"'))	

The command fields 2 and 3 are defined by the respective subsystem.

2.6.2 Identification string (id)

To identify a packet, each packet contains an identification string, which is *EOF_L_ID* bytes long. It may contain only the following characters:

- A-Z (alphabet in upper case)
- a-z (alphabet in lower case)
- 0-9 (the digits)
- ! (exclamation mark)
- - (minus)

The EOFs or EOFi may chose freely any of the *68719476736* possibilities.[2] The characters are limited to those characters to allow easy debugging and to keep the non-binary command layout.

2.6.3 Size (size)

All sizes used in this document are "'symbolic sizes"': The real size is defined in the attached file "'*eof.h*"'. Developers are advised to use the symbolic name in their programs.

A size is is always represented as an ASCII number found in a fixed length string of *EOF_L_SIZE* bytes.

2.6.4 Nick name (nick)

The peer name is a *EOF_L_NICKNAME* byte fixed length string. It is only used internally to give a peer a rememberable name ("'a nick'"). It is never transmitted over the network.

[2] $(26 + 26 + 10 + 2)^6$, as long as EOF_L_ID is 6.

2.6.5 Group name (group)

The group name is a *EOF_L_GROUP* byte fixed length string.

2.6.6 Message text (msgtext)

The message text is a *EOF_L_MESSAGE* byte fixed length string.

2.6.7 Peer address (addr)

The address of a peer, which is is a *EOF_L_ADDRESS* byte fixed length string. Peer addresses are specified as URLs as defined in RFC3986[10]. For more information have a look at section **??** on page **??**.

2.6.8 Keyid, the fingerprint (keyid)

A (PGP) fingerprint[3] is a *EOF_L_KEYID* byte fixed length string. It does not contain any spaces. It can be retrieved by issuing the following gpg-command:

```
LC_ALL=C gpg --fingerprint  | \
   grep "Key fingerprint =" | \
   sed -e 's/.*=//' -e 's/ //g'
```

2.7 EOF packets ("'EOFpkg"')

2.7.1 Introduction

No packet (including everything) may exceed the size of *EOF_L_PKG_MAX*.
EOF knows about

- commands: internal plaintext packets.
- onions: multiple times encrypted packets including routing information
- postcards: packets containing transport protocol dependent header

Commands are the innermost packet type and only seen within EOFi and EOFs. Commands are then bundled into a multi layer onion. Each layer contains commands after decryption. Onions are put onto a postcard afterwards and are sent out on the network. **Only encrypted packets are sent out on the network.**

[3] See RFC 2440, 11.2. Key IDs and Fingerprints

2.7.2 Commands

Command packets are used for the communication inside of EOFi and EOFs and are described in detail in the following sections:

- eofi2tp, 2.8, page 20
- eofi2ui, 2.10, page 26
- eofi2crypto, 2.11, page 33

2.7.3 Onions

Onions are sent out on the network and are multiple times encrypted. They are building the base for the EOF protocol. Onions are described in detail in chapter **??**, page **??**.

2.7.4 Postcards

Onions are afterwards encapsulated into the transport protocol specific packet type and sent out onto the network. Postcards are described in detail in chapter 2.13, page 35.

2.8 Transport protocols (”‘eofi2tp”’)

Environment variables: CWD for listener URLs are used as defined in RFC3986[10].
Transport protocols use EOF command ID 0.

Stateless: source routing, usage of acks.

Unterscheiden zwischen listener und send.

2.8.1 1000: Send packet

Table 2.3: Command 1000 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Destination	EOFsdt: addr	complete URL (with ”‘*scheme:*”’)	tcp:127.0.0.3:42
Size	EOFsdt: size	Size of message, excluding this header	424242
Message	Binary data	The message	BLOB

1000: Example

Added linebreak after some
0 for readability, which are **not** in the real packet!

```
1000abfudh127.0.0.3:42\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\010\0\0\0\0HEREISDATA
```

2.8.2 1001: Enable listening

When the listening transport protocol starts up, EOFi sends this command and waits for an acknowledge (*2003*, before it marks the listening transport protocol as enabled.

Table 2.4: Command 1001 parameters

ID	EOFsdt: id	packet id	afdb12
Parameter	**Type**	**Description**	**Example**
Destination	EOFsdt: addr	URL without "‘*scheme:*"’	127.0.0.3:42

1001: Example

```
1001afdb12127.0.0.3:42\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```

2.8.3 1002: Stop listening

This command requests a listening transport protocol to shutdown. It should free all ressources and exit. After a grace time (maybe seconds, not yet defined), EOFi will kill the badly behaving transport protocol. listening transport protocol as enabled.

Table 2.5: Command 1001 parameters

Parameter	**Type**	**Description**	**Example**

1002: Example

```
1002
```

2.8.4 2000: Packet successfully sent

This code is returned by the transport protocol subsystem to EOFi on success. After this return code, the transport protocol exits.

Example

```
2000
```

2.8.5 2001: Packet not sent

This code is returned by the transport protocol subsystem to the EOF implementation on failure. After this return code, the transport protocol exits.

Example

```
2001
```

2.8.6 2002: Received packet

The listening transport protocol received a packet and notifies EOFi.

Table 2.6: Command 2002 parameters

Parameter	Type	Description	Example
Size	EOFsdt:size	Excluding the 2002 and this field	42
Message	Binary data	The message	BLOB

Example

```
200212
RECEIVEDDATA
```

2.8.7 2003: Listening

This code is returned by the listening transport protocol, as soon as the listening process is ready to receive data. After that, EOFi can announce the listening URL to other EOFi.

Example

```
2003
```

2.9 The user interface ("'ui2user"')

This section specifies the appereance of a user interface to the user.

2.9.1 Minimal philosophy

All EOF compliant user interfaces must support the named commands, so the user can change the UI, but can be sure that this minimal amount of commands is always available.

Every user interface *may* add additional input methods or commands.

2.9.2 Commands in general

All commands begin with a "'/"' as first character (adopted from IRC).

2.9.3 Command length

The user interfaces need only to accept commands up to a length of *EOF_L_UI_INPUT*. If the user inputs longer commands, they should be truncated to *EOF_L_UI_INPUT* bytes.

2.9.4 Send text

If the entered text does not begin with a "'/"', it should be treated as a message to the current selected destination (either a *peer* or *group* of peers.

2.9.5 /peer add <nick> <initial addr> <keyid>

Add the peer as *name* to the list of known peers.

Table 2.7: /peer add parameters

Parameter	Type	Description	Example
Nick	EOFsdt	Name you identify the peer with	telmich
Initial addr	EOFsdt	Where we can make the first contact	tcp:10.0.42.42:4242
Keyid	EOFsdt	The PGP fingerprint of the peers public key	F27987E34E66...

Example

```
/peer add telmich tcp:10.0.42.42:4242 F27987E34E7866B2BA39C2FD793EB8FC325251FE
```

2.9.6 /peer send <nick> <msgtext...>

Send message *msgtext* to peer *nick*.

Table 2.8: /peer send parameters

Parameter	Type	Description	Example
Nick	EOFsdt	Name you identify the peer with	telmich
Msgtext	EOFsdt	The message itself	Hallo, wie geht es Dir?

Example

```
/peer send telmich Hallo, wie geht es Dir?
```

2.9.7 /peer rename <oldnick> <newnick>

Renames the peer.

Table 2.9: /peer rename parameters

Parameter	Type	Description	Example
Oldnick	EOFsdt	Old nick name	susi
Newnick	EOFsdt	New nick name	heinz

Example

```
/peer rename susi heinz
```

2.9.8 /peer show <nick>

Display detailled information about peer.

Table 2.10: /peer rename parameters

Parameter	Type	Description	Example
Nick name	EOFsdt	Nick name as known by EOFi	karl-otto

Example

```
/peer show karl-otto
```

2.9.9 /peer list

List currently known peers. This command does not accept any parameters.

Example

```
/peer list
```

2.9.10 /exit

Request the user interface to exit. It will deregister from EOFi, but EOFi will continue to run (even with no user interface attached).

Example

```
/exit
```

2.9.11 /quit

The UI tells EOFi to quit. The EOFi will tell all EOFs to quit and quits. Afterwards the UI will also quit.

Example

```
/quit
```

2.9.12 Aliases

Aliases may optionally be provided by the UI. If an UI provides support for aliases, it must implement the "'*/alias*"' command.

The following aliases should be provided by default, to aid new users using EOF.

2.9.13 /alias < *aliasname* > < *command...* >

This command should be used to setup aliases.

2.9.14 /msg < *nick* > < *msgtext* >

Should be an alias for */peer send <nick> <msgtext>*

2.9.15 /whois < *nick* >

Should be an alias for */peer show <nick>*

2.10 Interface to the user interface (”‘eofi2ui”’)

This section specifies the commands used between the EOFs ”‘user interface”’ and EOFi. As the user interface has to translate the user commands (”‘ui2user”’) to *eofi2ui* commands, the section titles refer to the *ui2user* and *eofi2ui* commands.

2.10.1 UI startup

What todo, when the UI starts. Connect to EOFi. Find out about

- joined group,
- connected marktscheier
- and open queries.

It thus issues the following commands: register, list joined groups, list open queries, list marktschreier.

2.10.2 Connection

All user interfaces connect to the user interface socket, as specified on page 15, section 2.3.2.

2.10.3 1100: Acknowledge

The is a general acknowledge answer. The request with the same *ID* as the packet was successful.

Parameters

Table 2.11: Command 1100 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12

Example

```
1100abfudh
```

2.10.4 1101: Failure

The is a general failure answer. The request with the same *ID* as the packet failed. Details are specified in the reason message.

Parameters

Table 2.12: Command 1101 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Reason	EOFsdt: msgtxt	Why the connection was refused	Too many UIs connected.

If the failed command was "'2100"', EOFi will close the socket afterwards.

Example

```
1101abfudhSome error\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0
```

2.10.5 1102: Exit requested

This is a shutdown request to the UI. After this message EOFi will exit and there is no communication possible.

Parameters

Table 2.13: Command 1102 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12

Example

```
1102abf93a
```

2.10.6 1103: Recieved message

This message is issued by EOFi, if a message is received.

Parameters

Table 2.14: Command 1103 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
nick	EOFsdt	The sender	telmich
msgtext	EOFsdt	The message	Hallo, mein Freund!

Example

```
1103abfudhtelmich\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0Hallo!\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```

Possible answers

- None: EOFi does not resend this packet if the UI lost it.

2.10.7 1104: List of peers

This is the answer to command *2106* and thus contains the same ID, as the *2106* request command.

Parameters

Table 2.15: Command 1104 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Number of peers (nop)	EOFsdt: size	How many peers follow	20
$nop * Peer$	EOFsdt: nick	The nickname	telmich

The last field is repeated as many times as specified in number of peers.

2.10.8 1105: Peer information

This is the answer to command *2105* and thus contains the same ID, as the *2105* request command.

Parameters

Table 2.16: Command 1105 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Keyid	EOFsdt:keyid	This peers pgp-keyid	389E5481065EAA253...
Number of addresses (noa)	EOFsdt: size		1
$noa * address$	EOFsdt: addr	Adress of peer	tcp:127.0.0.1:4243

The last field is repeated as often, as specified in the number of addresses field.

2.10.9 1106: Peer renamed

This is the answer to command *2104* and thus contains the same ID, as the *2104* request command. It is sent out to **all** connected user interfaces.

Parameters

Table 2.17: Command 1106 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Oldnick	EOFsdt	Old nick name	susi
Newnick	EOFsdt	New nick name	heinz

Possible answers

- None

2.10.10 2100: Register user interface

This must be the *first* message sent by the UI. If the answer is not *1100*, the UI should close the socket afterwards.

Parameters

Table 2.18: Command 2100 parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12

Possible answers

- 1100
- 1101

2.10.11 2101: Deregister user interface

Parameters

- none

Possible answers

- none

EOFi will close the connection to the UI after receiving this message.

2.10.12 2102: /peer add

The UI adds a peer to the list of known peers.

Parameters

Table 2.19: 2102: /peer add parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Nick	EOFsdt: nick	Name you identify the peer with	telmich
Address	EOFsdt: addr	Where we can make the first contact	tcp:10.0.42.42:4242
Keyid	EOFsdt: keyid	PGP fingerprint of the peers key	F27987E34E66...

Possible answers

- 1100
- 1101

2.10.13 2103: /peer send

The UI wants to submit a message to a peer.

Parameters

Table 2.20: 2103: /peer send parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Nick	EOFsdt: nick	Name you identify the peer with	telmich
Message	EOFsdt: msgtxt	The message itself	Hallo, wie geht es Dir?

Possible answers

- 1100
- 1101

2.10.14 2104: /peer rename

The UI wants to rename a peer.

Parameters

Table 2.21: /peer rename parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Oldnick	EOFsdt	Old nick name	susi
Newnick	EOFsdt	New nick name	heinz

Possible answers

- 1106
- 1101

2.10.15 2105: /peer show

The UI requests details about a peer.

Parameters

Table 2.22: 2105: /peer show parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12
Nick name	EOFsdt	Nick name, as known by EOFi	karl-otto

Possible answers

- 1101
- 1105

2.10.16 2106: /peer list

The UI requests the list of known peers.

Parameters

Table 2.23: 2106: /peer list parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12

Possible answers

- 1101
- 1104

2.10.17 2199: /quit

The user interface requests EOFi and all other EOFs to exit. The EOFi will not answer, but send an exit request to all other EOFs.

Parameters

Table 2.24: 2199: /quit parameters

Parameter	Type	Description	Example
ID	EOFsdt: id	packet id	afdb12

Possible answers

- none

2.11 Interface to the encryption engine ("'eofi2crypto"')

2.11.1 Connection

The crypto engine is started by EOFi at startup and communicates with EOFi through stdin and stdout.

2.11.2 1200: Encrypt packet

Passes the following information to the crypto:

- GPG-Fingerprint of the peer (40 Bytes char array) (*fpr*)
- Adress of the peer (128 Bytes, 0 padded, 0 terminated) (*address*)
- The length of the message (uint32_t) (*msg_len*)
- The message (*msg*)

2.11.3 1201: Decrypt packet

2.11.4 2200: Encrypted a packet

- The length of the packet (uint32_t) (*pck_len*)
- The packet (*pck*)

2.11.5 2200: Decrypted a packet

2.12 EOF crypto packets ("'3***: onions"')

Onions are the result of decrypting an incoming packet (respective vice versa when sending out).

2.12.1 Introduction

The following types are defined:

- 3000: Drop packet
- 3001: Forward packet
- 3002: Message / drop packet
- 3003: Message / forward packet
- 3004: Acknowledge

2.12.2 Parameters

All **3*** ** packets have the same length and contain the same fields:

Table 2.25: Command 3*** parameters

Command	**id**	**addr**	**group**	**msgtext**
3000	-	-	-	-
3001	-	x	-	-
3002	x	-	x	x
3003	x	x	x	x
3004	x	-	-	-

- -: Not used
- x: Used

Table 2.26: Command 3*** parameters

Parameter	**Type**	**Description**	**Example**
id	EOFsdt	Packet id	alg4f!
addr	EOFsdt	Adress of next peer	tcp:123.123.123.132:8080
group	EOFsdt	The destination group	!eof
msgtext	EOFsdt	The message	Hallo, mein Freund!

2.12.3 3000: Drop packet

You are the last recipient and there's nothing interesting left. Just drop the packet and continue work.

2.12.4 3001: Forward packet

If a peer receives a packet with the command 3001, it simply forwards the message to the peer specified in the **addr** field. All data contained in the message is noise. After the message has been forwarded to the next peer, it should be dropped. If the peer is unreachable, the message should also be dropped.

2.12.5 3002: Message / drop packet

This packet contains a messages to be read and does not need to be forwarded anymore: You are the last peer in the chain.

- If the first byte of the group is the zero byte, the message is a private message (i.e. only sent to you).
- If the first byte of the group field is **non-zero** the message is addressed to the specified group.

2.12.6 3003: Message / forward packet

The command 3002 is a combination of command 3001 and 3002.

2.12.7 3004: Acknowledge

Acknowledge the receipt of a received message. The ID must be the same as the one specified in the original messages packet. Every message packet must be acknowledged.

2.13 EOF network packets ("'postcards"')

All data that is transferred over the network must be encrypted. The EOF packets described in the previous section are multiple times encrypted and assembled according to the calculated source route. These packets are code-named "'postcards"', as it is assumed they can be read by an attacker.

2.13.1 Routing

This version of EOF does not know how to create a route. All packages are transferred directly to the final peer (which is an incredible big huge bug) in this version of EOF. Source routing will be described and defined in future versions.

2.13.2 Onion packets

An onion packet is a (multiple times) encrypted packet. An onion packet contains at least one plaintext packet, but can also contain already encrypted packets. It may look like as follows:

Example onion packet

-

2.13.3 Postcard packets

A postcard "'packet"' contains one onion packet plus the transport protocol shell. Postcard packets are the only packet type that is seen by a possible attacker. The name postcard was choosen to reflect the fact, that anyone passing the postcard can read what is written on it.

All packets must be signed by the sender and encrypted for the receiver. The different datatypes are just concatenated in the order described. The following description of the content describes the pakckets in their unencrypted form.

Chapter 3

Transport protocols

3.1 Introduction

Only lower-case names are allowed (to prevent problems with broken filesystems).

3.2 Examples

This sections shows some theoretic (theoretic because nobody implemented them yet) transport protocols.

3.2.1 tcp

3.2.2 tcps

3.2.3 udp

3.2.4 http

3.2.5 https

3.2.6 smtp

connect to smtp server

3.2.7 smtps

3.2.8 mediawiki

url, user, password

3.2.9 smb

write on windows shares

3.2.10 mailto

Write an email to some address. Needs smtp-server set in configuration. May have different methods for retrieval (like connecting to imap,pop, read from mbox/maildir directly).

Example URLs

`mailto:nico-eof@eof.eof.name`

3.2.11 dns

Use dns traffic to transport EOF protocols.

3.2.12 media

like images, videos, sounds, real noise, spam, ...

3.3 Embedding into EOF

This section explains how to integrate a transport protocol into EOF.

3.3.1 Adding a transport protocol

If you want to add the transport protocol *tptest*:

- Create the directories
  - **$HOME/.ceof/tp/available/*tptest***.
- Create (link or copy) the executables to the filenames **listen** and **send**.

The EOF implementation will register the transport protocol automatically at the next start. You may register only the **listen** or **send** part of a protocol.

3.3.2 Using a transport protocol

So far the EOF implementation knows about the implementation, and may also already use it for *sending* packets. But there is no listener configured yet. To enable a listener for the protocol *tptest* at the URL *tptest:somewhere@protocol-specific*

- create a directory below
  - **$HOME/.ceof/tp/enabled/**
- with a name of your choice (f.i. *tptest-somwhere* or *http-80*).
- Then add the URL to the file named < **url** >.
- If the transport protocol needs or allows additional configuration files, you need to create them in that directory.

The EOF implementation will parse the URL and check whether a supporting listener application is available.

Will do chdir() to the directory! tp can open config files in current dir. what about url?

same for listen and send? listenurl, sendurl, config in curdir Yes!

The maximum length of the URL is defined in eof.h (EOF_L_ADDRESS). If it is longer, it will be truncated after **EOF_L_ADDRESS** bytes.

IMPORTANT!

3.4 Implementations

The following sections cover existent protocol implementations. The name of the section is the name of the registered implementation. Every section must at least contain:

- Supported scheme
- Author contact information
- URL of website
- Programming language
- EOF-Version that introduces support for the protocol

You should add your implementation to the directory "'tp"' within the repository. Sort sections by alphabet.

3.4.1 dummy/c

- dummy
- Nico Schottelius [nico-eof-tp-dummy-c =at= schottelius.org]
- https://www.eof.name
- C
- Version: 1

3.4.2 tcp-apic

- tcp
- A. Pic
- ?
- C
- Version: 1

Appendix A

Sources

Bibliography

[1] http://en.wikipedia.org/wiki/Public-key_cryptography

[2] http://en.wikipedia.org/wiki/Onion_routing

[3] https://wiki.torproject.org/noreply/TheOnionRouter

[4] RFC 1459: http://www.irchelp.org/irchelp/rfc/rfc.html

[5] http://en.wikipedia.org/wiki/Source_routing

[6] RFC 793: http://www.faqs.org/rfcs/rfc793.html

[7] RFC 2616: http://www.faqs.org/rfcs/rfc2616.html

[8] RFC 2821: http://www.faqs.org/rfcs/rfc2821.html

[9] RFC 1149: http://www.faqs.org/rfcs/rfc1149.html

[10] RFC 3986: http://www.faqs.org/rfcs/rfc3986.html

[11] https://www.eof.name

[12] http://en.wikipedia.org/wiki/Steganography