Final Report

16 Onions

Josef Stark Charlie Groh

August 17, 2017

1 General

We are the team "16 Onions" consisting of Josef Stark and Charlie Groh, and our goal is to develop a prototype implementation of the Onion module.

2 Application internals

In this section we will describe the general structure of the application, the network protocol we implemented, as well as an overview over how the source code is organized.

2.1 Network protocols

For the communication between distinct onion instances we decided to use both TCP and UDP as underlying protocols in order to avoid reinventing the wheel, since they both fulfill the respective requirements perfectly.

Control messages, i.e. messages for authentication, tunnel construction, tunnel destruction and heartbeat are transferred and forwarded over TCP, because for those messages it is very important that they actually arrive or that we get feedback if one of those messages could not be delivered to the target, so we can react in an appropriate manner—e.g. assume that the corresponding node went down and construct an alternative route. TCP satisfies these requirements as it acknowledges the reception of messages, resends messages if necessary and reports a failure if a message still didn't provoke an acknowledgement after a few retries. If we used UDP instead, we would have had to re-implement all of these features by hand.

User data messages, i.e. messages containing VoIP and cover data, are transferred and forwarded over UDP, since for those a short delay is a requirement which UDP can satisfy. UDP is packet based and does not acknowledge the arrival of packets at all, so the lower delay that this causes comes at the price of possibly losing some packets which are not resent and therefore never reach their target, without the sender being informed about the loss. This is acceptable for VoIP and cover data.

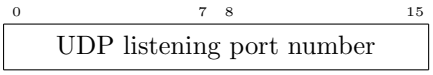
Other modules can use the onion module only to send user data messages (UDP), while control messages are exclusively used internally for tunnel management tasks.

2.1.1 Layer 0: Network abstraction layer

At the lowest level of our application lies the so-called multiplexer, which abstracts away low-level protocol details and offers a clean interface for message interchange between nodes. The core tunnel mechanisms exclusively use this interface. The main tasks of the multiplexer are:

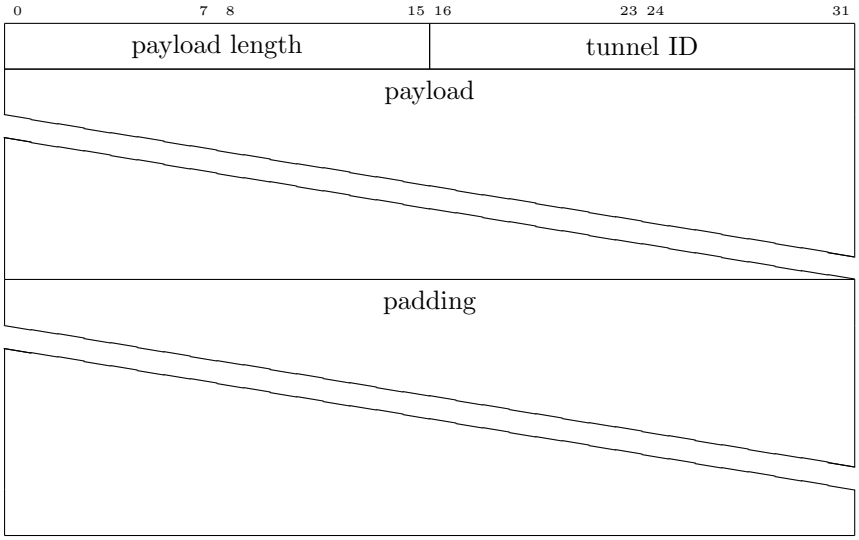
- Establishing and terminating control data channels to other peers using TCP
- Establishing and terminating user data channels to other peers using UDP
- Padding/Unpadding messages to/from a fixed, configurable size while offering transport functionality of variable-sized payload
- Correctly assigning and relating incoming and outgoing messages to the tunnel they belong to

Upon receiving the request to connect to a certain peer, the multiplexer first establishes a TCP connection to that peer for transporting control data and then immediately sends a packet to that peer containing the UDP port on which it is listening for user data:



(TCP message)

This is the only message type that is exchanged on its own. After it has been sent, messages are only sent upon request of the core tunnel mechanism. They can be transmitted over TCP or UDP, which is controlled by a flag of the corresponding function. They have the following structure:



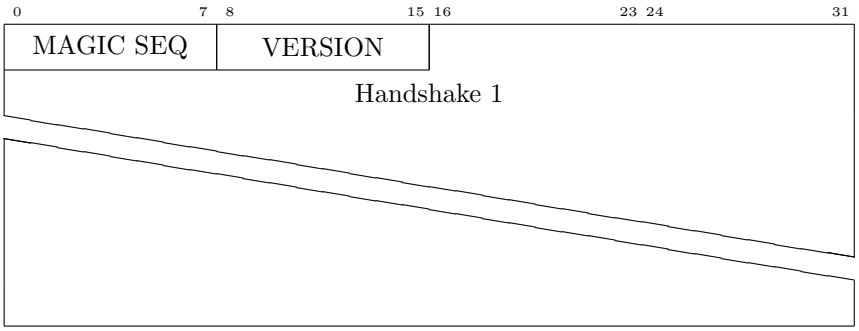
(TCP or UDP message)

2.1.2 Layer 1: Tunnel management layer

The core tunnel mechanisms build upon the network abstraction layer provided by the multiplexer and use its interface to exchange messages. Those messages, which will be described in this section are thus free of metadata and padding from Layer 0. This information is added/removed and generally handled transparently by the multiplexer.

This section lists each message type and their exact structure used during all the phases of a tunnel lifespan. For the sake of readability, we will refer to the initiator of a connection as "peer A" and to the other node as "peer B".

- Handshake 1

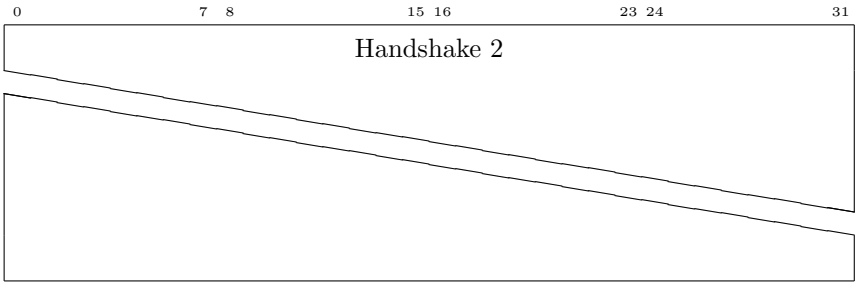


(TCP message)

This message is sent from peer A to initiate the authentication process. MAGIC SEQ is a bit sequence identifying the connection initiator as onion peer; VERSION contains the P2P protocol version. Handshake 1 is obtained from the ONION AUTH module of peer A and forwarded to peer B

via this message, where it is passed to the local ONION AUTH module—given that it is actually a onion node, MAGIC SEQ matches and VERSION is compatible.

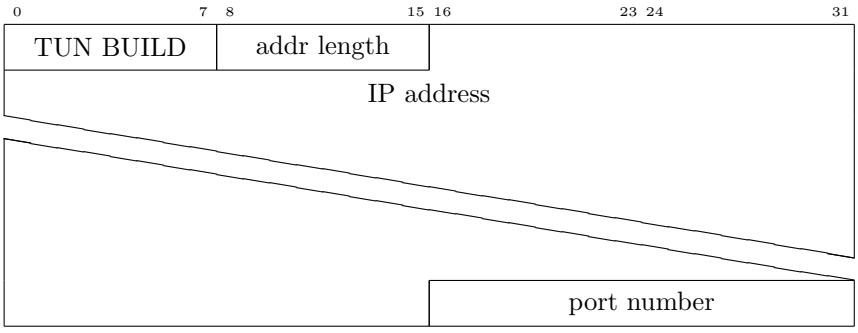
- Handshake 2



(TCP message)

After having processed Handshake 1, peer B obtains Handshake 2 from its ONION AUTH module—given that the Handshake was valid. It then sends this Handshake 2 back to peer A. ONION AUTH of peer A now checks the received Handshake for validity. If positive, the handshake is further processed and an encrypted connection between peer A and peer B has been established. In any other case—peer A received an invalid handshake or no answer at all and timed out—the authentication is aborted and adequate measures are to be taken, e.g. selecting a different node and starting over the procedure.

- Tunnel building/expansion request



(TCP message)

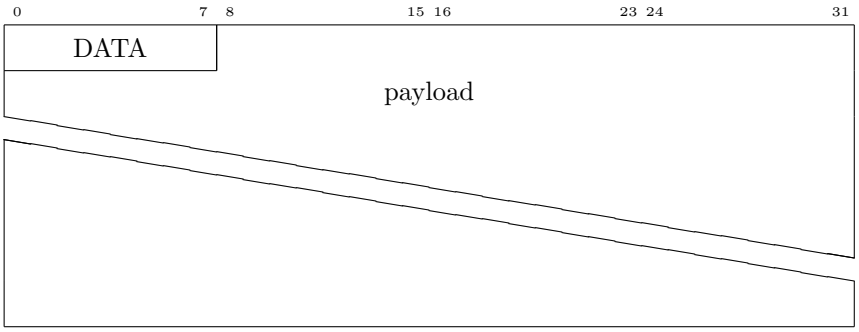
Peer A can send this message to an authenticated peer B to instruct it to connect to another peer C, whose address is specified in this message. Upon receiving it, peer B will now forward all incoming messages between peer A and peer C, so peer A can begin authenticating with peer C over peer B.

- Incoming tunnel notification

(TCP message)

This message is sent from peer A to the peer D at the end of the tunnel, so that peer D knows that the established tunnel ends at peer D and that the further incoming data is destined to him and must not be forwarded. After receiving this message, no more tunnel building/expansion requests can be sent to peer D.

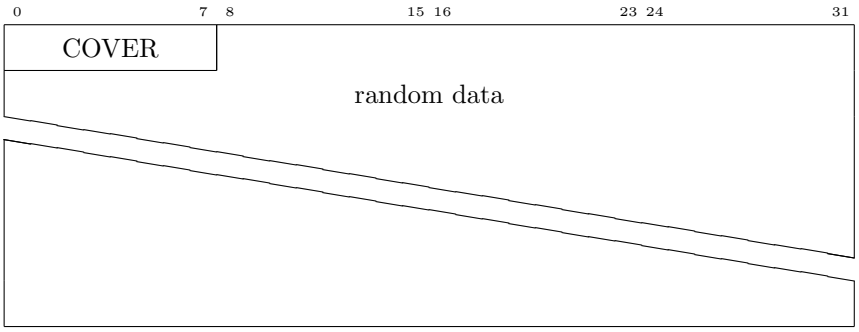
- User data



(UDP message)

Contains real user (i.e. VoIP) data, which was requested to be sent by another local module. Requires the previous reception of an incoming tunnel notification in order to be interpreted.

- User data



(UDP message)

Contains cover traffic, which was requested to be sent by another local module. The message is discarded upon reception.

- Heartbeat

(TCP message)

If a time-out was triggered, meaning that a peer has not received data of any kind from a certain tunnel in a certain amount of time, it sends this message to request a sign of live from the node at the other end of the tunnel. This sign of live can be any data, but if there is currently no real user data available for being sent, it is simply cover traffic. Each time-out triggers the sending of such a heartbeat message and the incrementation of a retries-counter. Once a response has been received, that counter is reset to zero; if the counter exceeds five, the tunnel is presumed dead.