Interim Report

16 Onions

Josef Stark Charlie Groh

June 20, 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 Process Architecture

Because of the simpler process architecture and better debugging support we will try to implement our module event-driven. This means there will run one main loop waiting for incoming packets from multiple sockets and issuing appropriate actions. As a result the program will not be able to utilize multiple processor cores, but since our application is not compute-bound, that should not induce any problems.

3 Inter-Module Protocol

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. For this it sacrifices some bandwidth and latency, but those two factors aren't of uttermost importance to control messages anyway.

User data messages, i.e. messages containing VoIP 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 data.

Call handling is very simple. If a peer begins sending UDP packets with real instead of random data, this means a call is requested, and the call partner may accept the call by also sending real data. Since everything is encrypted, an attacker can not distinguish cover and real traffic. Neither can he derive the call state from TCP packets, since call handling does not happen over TCP.

To preserve anonymity, all UDP packets are 64 KiB and all TCP messages 512 Bytes. If the actual payload and packet header are smaller than these, the packet/message is padded with random data. There is always UDP data sent, even if there isn't an active call. Thus, an attacker cannot infer from the communication amount and bandwidth if there is an active call between two nodes. We chose 64 KiB as UDP packet size, since the API messages ONION TUNNEL DATA and ONION COVER can have a maximum size of 64 KiB also. Therefore we do not need to take care of packet fragmentation and reconstruction ourselves but can leave this to the IP layer. The downside is, that if only one fragment gets lost or corrupted on the way, the whole UDP packet is discarded silently due to a checksum mismatch. This could be especially problematic, if the actual messages we get from the API tend to be much smaller, so we might discard a packet just because some bit of the irrelevant padding has flipped. Should we at a later point discover that this is actually the case, we might rethink the UDP packet size and the DATA packet. In that case we could handle fragmenting and reassembling ourselves and introduce e.g. fields for total data size, sequence number and message ID.

3.1 Control message flow

When a node A wants to directly connect to another node B, it has to pass the following stages:

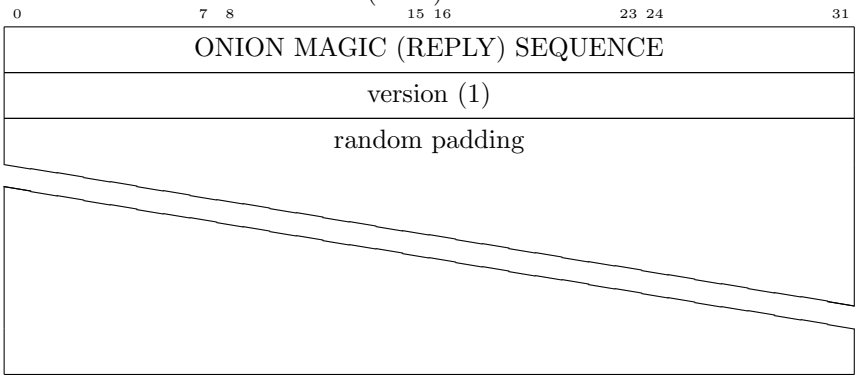
- Establish a simple TCP connection to B.
- Authenticate as onion node to avoid connecting to unrelated services running on the onion port.
- Do the onion handshake using OnionAuth module.

This all happens over TCP. Once the handshake has completed, the two peers can now exchange other control messages (see 3.1.1) as well as user data (see

3.1.2), everything from this point on being encrypted with an ephemeral session key, so no one else can read their communication.

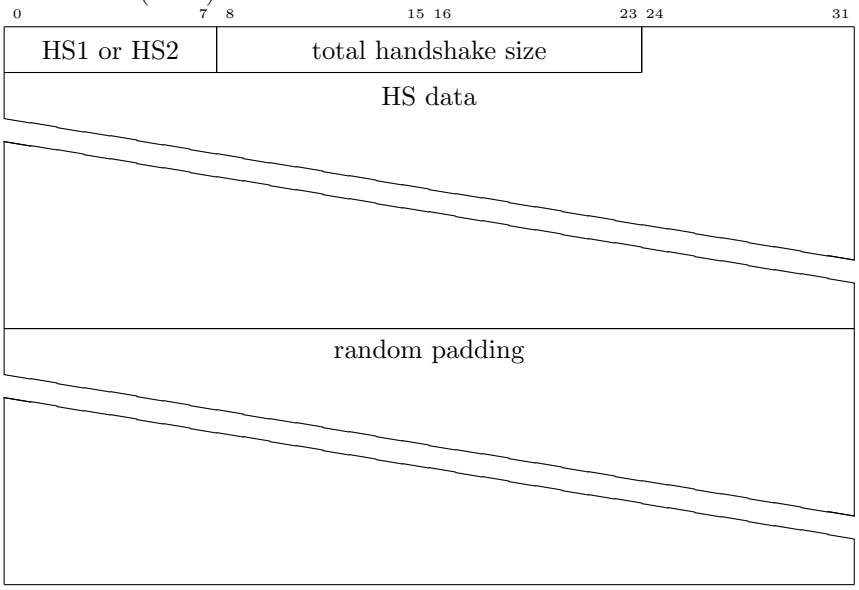
3.1.1 Control message types

- Authentication as onion nodes (TCP):



  After establishing a TCP connection, the connection initiator sends a message with the ONION MAGIC SEQUENCE header to the other node so it knows that it is talking with an onion peer and not some other service that runs on the same port and that their versions are compatible (Only valid version at the time of writing is 1). It then replies with a message with the ONION MAGIC REPLY SEQUENCE header.

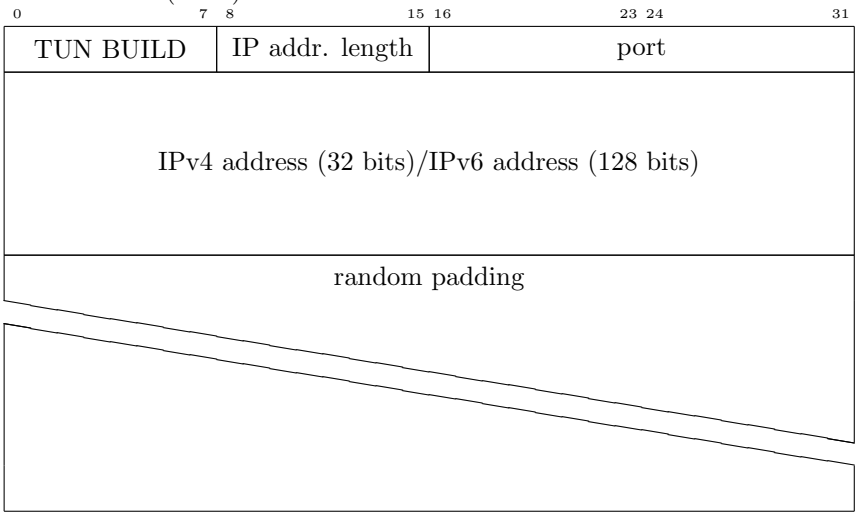
- Handshake (TCP):



  The node initiating an onion connection first sends an HS1 message to the other node. It also includes a size field. If the size of the complete

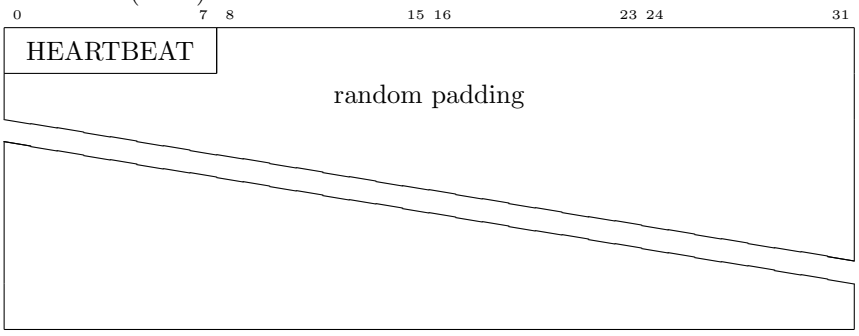
handshake is bigger than one control message (512 Bytes), the node sends then as many follow-up messages as necessary to transmit the whole HS1. Those follow-up messages do not contain a header but only data, and the last one possibly has to be padded to reach the required standard size of 512 bytes. Once the other node has received the complete HS1, it responds with HS2 in the same way.

- Build Tunnel (TCP):



Once A has established an encrypted connection to its first hop H1, it can send this message to it. H1 will then establish an unencrypted TCP connection to the hop H2 specified in this package and from that point on it will forward all TCP and UDP traffic it receives from A to H2 and vice-versa. A can now do the authentication and handshake process with H2 over the encrypted connection to H1. It can iteratively add more hops like this until it reaches the desired target node.

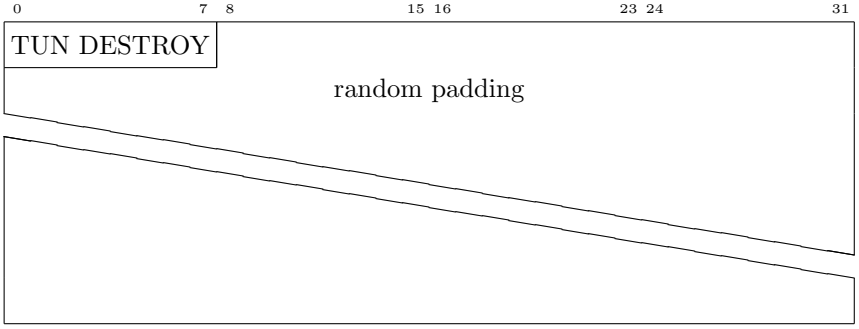
- Heartbeat (TCP):



This is sent if one of the nodes at the end of the tunnel has a suspicion that the tunnel went down. If the originator did not receive the same

message as reply in a certain time interval, it needs to assume that the tunnel is down and it has to do further exception handling (see 3.2.1).

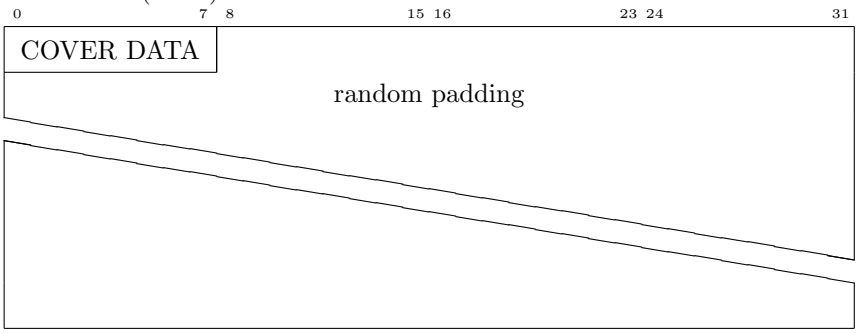
- Tunnel teardown (TCP):



This is used for the controlled destruction of a tunnel. The tunnel initiator has to send this to every hop, starting from the farthest to the closest one.

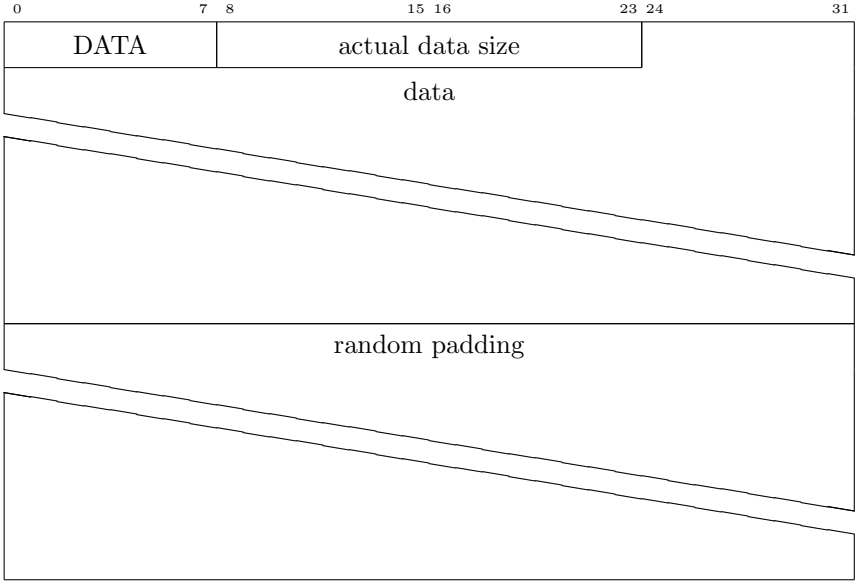
3.1.2 User data message types

- Cover data (UDP):



Used for fake data if there is no active call between two peers. The recipient can just ignore the content.

- Real user data (UDP):



Used for actual VoIP data. If a peer starts receiving these packets, it should interpret it as a call request. Once the user accepts the call, his node should answer with those packets. Both peers then know that the call has started and process the incoming VoIP data.

3.2 Exception handling

In general there are three possible types of exceptions which could occur:

- a packet gets lost
- an unexpected packet arrives
- a packet is malformed

3.2.1 Packet-loss

If the lost packet is an UDP-packet, the loss can not be detected by the Onion module and is therefore ignored. Nevertheless we consider implementing a watchdog-like mechanism to catch abnormal low rates of UDP-packets.

When the module suspects the onion tunnel of being broken it will try to ping the destination peer by sending the heartbeat (see 3.1.1) message. If the suspicion substantiates an appropriate ONION-ERROR-message is delivered.

Since TCP will try to handle lost packets by itself an exception originated by a TCP-socket is considered as critical and will cause an ONION-ERROR-message immediately.

If required, lost connections to certain targets are reconstructed via different hops in the next round. For small enough round-intervals, this enables transition to a different tunnel without too much delay during a call.

3.2.2 Unexpected and malformed packets

Due to the reason that unexpected packets can not be distinguished from malformed packets both exceptions will be handled in the same way. We assume that the Onion Auth module is preventing replay-attacks and is guaranteeing the integrity of every message sent or received. Therefore our module has to consider every received malformed message as an attack against our peer. Such situation will cause an immediate termination of the onion tunnel and an ONION-ERROR-message. Nevertheless an error detected by the OnionAuth module is simply ignored.