Final Report

16 Onions

Charlie Groh Josef Stark

August 22, 2017

1 General

We are the team "16 Onions" consisting of Charlie Groh and Josef Stark, 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.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

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 of the underlay connections and offers a clean interface for message interchange between nodes via logical overlay connections. The core tunnel mechanisms exclusively use this interface. The main tasks of the multiplexer are:

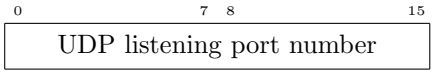
- Establishing and terminating control data channels to other peers using TCP
- Establishing and terminating user data channels to other peers using UDP
- Multiplexing multiple overlay connections over the same underlay connection
- Padding/Unpadding messages to/from a fixed, configurable size while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

The multiplexer differentiate between two types of connections:

Underlay connections between two peers are the respective TCP connection. Between a pair of peers there is at most one underlay connection.

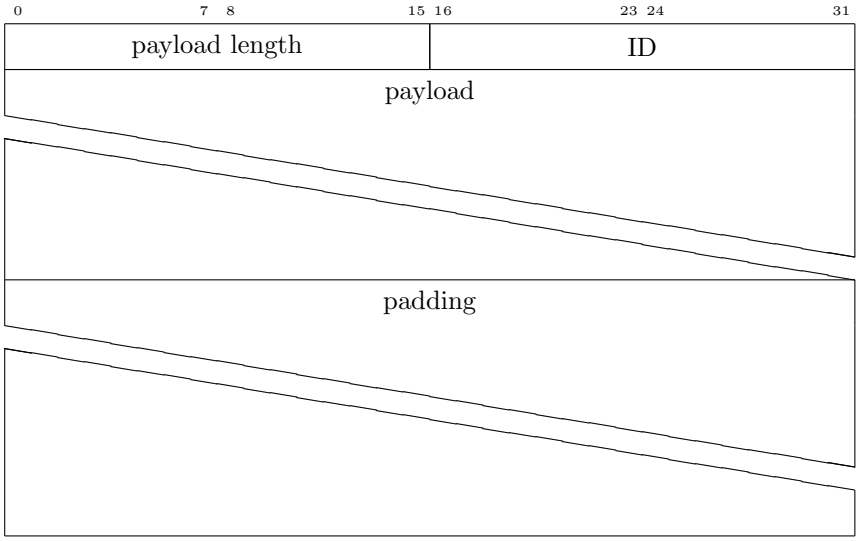
Overlay connections are multiplexed over the underlay connection between the peers. They are identified by the ID field in the packets.

Upon receiving the request to establish an overlay connections to a certain peer, the multiplexer first checks whether the peer is already connected by an underlay connection. If this is not the case, the multiplexer establishes a TCP connection to the peer and immediately sends a packet containing the UDP port on which it is listening for user data:



(TCP message)

All succeeding messages are sent upon request of the core tunnel mechanism. The protocol (TCP or UDP) is selected by the `type` flag in the `OnionMessage` class. The transmitted messages have the following structure:



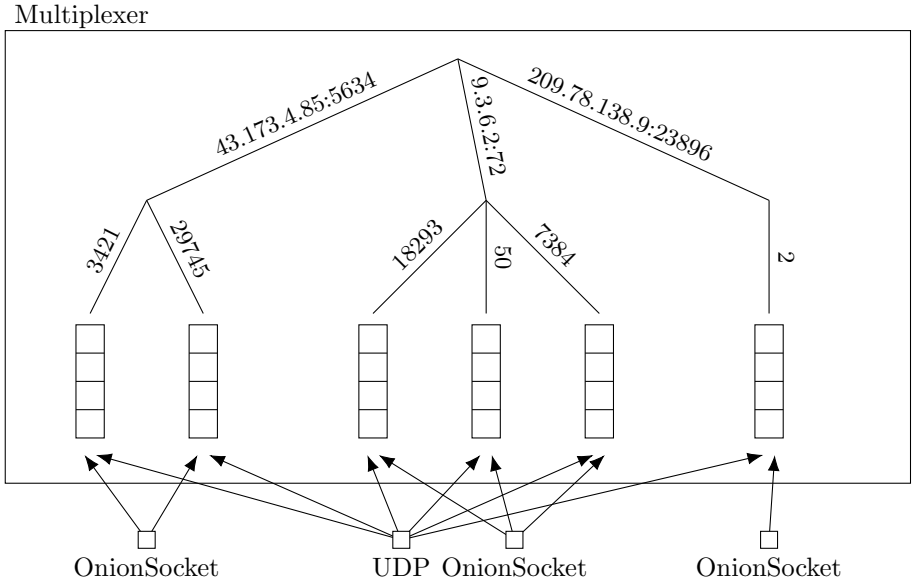
(TCP or UDP message)

It is worth mentioning that everything is transmitted in plaintext. The `padding` is necessary, to guarantee a equal sized packets. It does not provide any kind of anonymity protection mechanism. Furthermore, if the ONION module is configured properly (i.e. `p2p_packetsize` is equal to the packet size of the ONION AUTH module) most packets will not contain a padding.

The multiplexer identifies a overlay connection by the tuple (IP-address:port, ID). This makes it possible to run multiple instances of ONION modules on the same IP-address, although it ease sybil attacks. We decided to take the risk, because it simplifies tests for the module. Nevertheless, the code design makes it simple to change it.

Internally the multiplexer holds for every overlay connection a queue that contains received messages for this connection. Therefore, a queue contains packtes received over UDP and TCP, which is indicated by the `type` field in `OnionMessage`. Incoming UDP packets are processed in the `Main` class; incoming TCP packets by the responsible `OnionSocket` class.

New connections are signaled by calling the `newConnection(...)` method that does almost all higher-level (Layer 1) messaging.

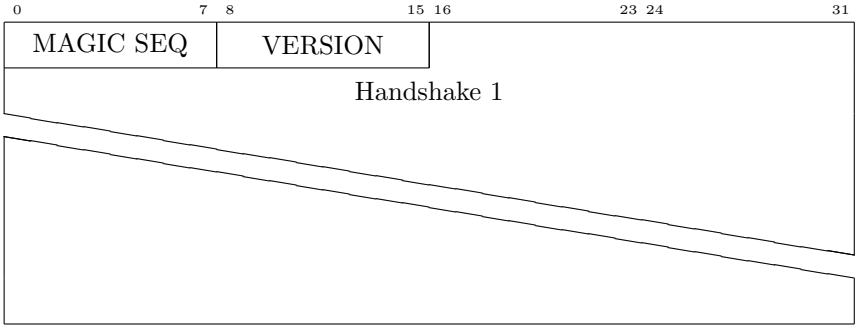


2.1.2 Layer 1: Tunnel Management Layer

The core tunnel mechanisms build upon the network abstraction layer provided by the multiplexer and use its interface to access the logical overlay connections and 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

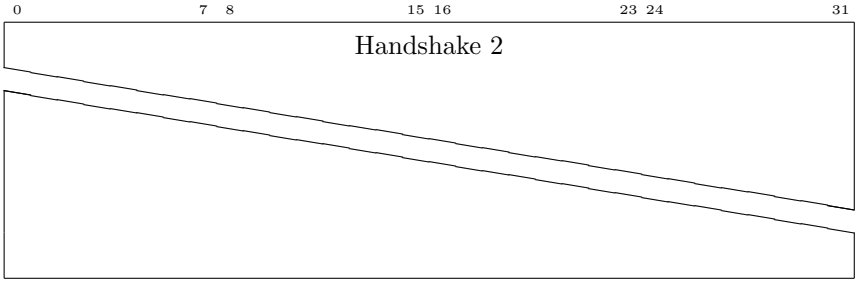


(TCP message)

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.

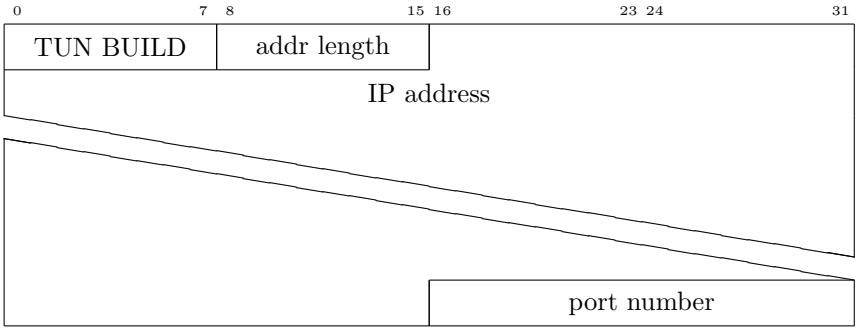
- Handshake 2



(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)

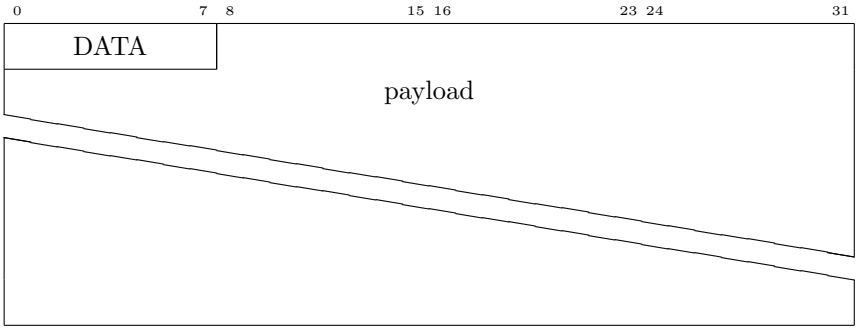
Peer A can send this message to an authenticated peer B to instruct it to connect to another peer C, whose address is specified in this message. Upon receiving it, peer B will now forward all incoming messages between peer A and peer C, so peer A can begin authenticating with peer C over peer B.

- 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.

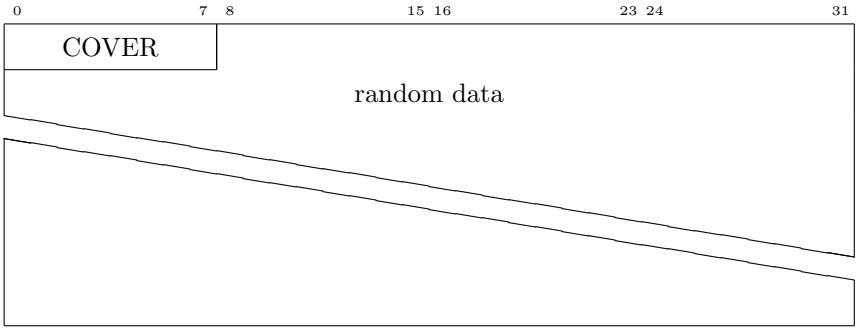
- User data



(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 of a specified size, which was requested to be sent by the local CM/UI module. Since it contains no meaningful data, the message is discarded upon reception.

- Heartbeat

(UDP message)

If a time-out was triggered, meaning that a peer has not received data of any kind from a certain tunnel in a predefined 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. This message (as well as cover traffic and real data) can only be sent by the two end nodes of a tunnel. An intermediate hop also counts the time-outs and disconnects from its predecessor and successor if necessary, but does not send messages on its own, it only forwards them.

2.2 Mode of Operation

After the application has been initialized, it accepts requests from the CM/UI to:

- Build tunnels
- Destroy tunnels
- Send data through tunnels
- Send cover traffic through tunnels

Apart from that, it listens for new incoming tunnels and data arriving through tunnels and notifies the CM/UI for any of those events.
Time in the system is divided into equal intervals called rounds. New tunnels are established at the beginning of the next round and destroyed upon request. If at the beginning of a round there is not any tunnel in existence or pending to be built, the module builds a tunnel to a random destination, which is automatically destroyed at the end of the round. This kind of tunnel can only be used to send cover traffic.

2.2.1 Tunnel Construction Process

The class used for constructing outgoing tunnels is called `OnionConnectingSocket`. For incoming as well as intermediate tunnels (when a peer is only forwarding traffic), `OnionListenerSocket` is used. The process of constructing it is as follows:

1. An array of intermediate hops is constructed by repeatedly asking the RPS module for random peers. The amount of hops is determined in the config file. The last node is the specified target.
2. The first unconnected node in the hop array is selected.
3. Handshake 1 is generated and sent to the selected hop.

4. Handshake 2 is received.

5. Test, if there are any unconnected hops remaining in the array:

   (a) If yes: A tunnel building/expansion request, containing the next hop as target, is sent to the current hop. That hop then connects to the specified target and forwards all traffic coming through that tunnel to it and vice-versa. The next unconnected node in the array is connected and the next loop iteration executed, starting at point 3.

   (b) If not: The tunnel endpoint is reached, so the incoming tunnel notification is sent.

If during this construction process an unexpected packet is received or an answer takes too long, a different intermediate peer is selected or, if the node in question is the tunnel endpoint, the construction is aborted and the failure, if required, reported to the CM/UI module.

Once established, a tunnel can and will be used to transfer data-messages, heartbeats and cover-traffic.

While the CM/UI cannot explicitly send cover data over tunnels whose construction it requested, cover messages are used internally if a node needs to respond to a Heartbeat and has no real user data available to be sent.

2.2.2 Tunnel Destruction Process

Tearing down a tunnel happens in the reverse order of construction: The respective message is sent to all hops, starting with the tunnel node to the node which is directly connected to the initiator. Once a hop receives the message, it closes the connection to its predecessor in that tunnel. Should an error happen while sending the message

3 Usage

In order to use the module, a Java runtime environment (version 7 or later), as well as Apache Ant[1] is needed on the host. The code depends on some 3rd-party libraries, but they are all included in the repository.
For further development of the module, Eclipse[2] (version 3.8 or later) with Lombok[3] (version 1.16.16 or later) is recommended. The respective project file is included. To use the module, the RPS and ONION AUTH modules have to be started. Also, the module has to be compiled and the class `Main` of the package `com.voidphone.onion` must be started. Those last two things can be done either via the respective GUI commands in Eclipse or via command-line interface:

[1] `https://ant.apache.org/`
[2] `http://www.eclipse.org/`
[3] `https://projectlombok.org/`

```
.../16onions$ ant clean
.../16onions$ ant build
.../16onions$ java -classpath bin:\
libs/ini4j-0.5.4.jar:\
libs/commons-cli-1.3.1.jar:\
testing/libs/bcprov-jdk15on-155.jar:\
testing/libs/lombok-1.16.16.jar \
com.voidphone.onion.Main -c <path to configuration file>
```

3.1 Configuration

The module needs a hostkey, which it will use to identify itself to other peers, as well as a configuration file with all the necessary options specified. The hostkey can be generated like this:

```
openssl genrsa -out hostkey.pem 4096
```

The configuration file needs to be in INI format. Here is how an example configuration file could look like:

```
[onion]
hostkey = /home/p2p/hostkey.pem
api_address = 0.0.0.0:2223
listen_address = 0.0.0.0:2512
hopcount = 3
api_timeout = 2000
p2p_timeout = 2000
p2p_packetsize = 20480
roundtime = 15000
[auth]
api_address = 0.0.0.0:2023
[rps]
api_address = 0.0.0.0:2123
```

Explanation of the options:

- `hostkey`: Path to the hostkey
- `api_address` in sections `[auth]` and `[rps]`: The listening address of the API interfaces of ONION AUTH and RPS modules
- `api_address` in section `[onion]`: The listening address of the onion module API, where it will listen and respond to API requests from the CM/UI module
- `listen_address`: The P2P listening address where the onion module will listen for new incoming tunnels from other nodes

- `hopcount`: The amount of intermediate hops used for building outgoing tunnels. This number does not include start and end node, i.e. a hopcount of zero equals a direct connection.
- `api_timeout`: The API timeout in milliseconds. When waiting for a response from one of the API interfaces, the module will give up after not receiving anything in the given amount of time.
- `p2p_timeout`: The P2P timeout in milliseconds. If, for any given tunnel, which the local module instance is a part of, no message has been arrived, measures are taken (i.e. the sending or forwarding of a heartbeat) to ensure that it is still alive. If those measures are unsuccessful, the respective tunnel is presumed dead and torn down.
- `p2p_packetsize`: Every layer 0 packet is padded if necessary. This option defines the size of the `payload+padding` section. It must be larger or equal than the packet size of the ONION AUTH module (for best performance results it should be equal). To achieve unobservability, it must be the same on all peers. Nevertheless, we left this as a config option, because we do not know the packet size of the ONION AUTH module and our module should run with any ONION AUTH implementation.
- `roundtime`: The roundtime in milliseconds

The path to the config file is passed to the module as parameter, as explained before.

3.2 Testsuite

The module contains a testsuite that tests artificial and real world situations. The testsuite is located in the package `com.voidphone.testing`. The class `Testing` is a unit test that tests the module for API conformance and some basic functionality like building a tunnel between two hops. The class `Test` launches multiple hops and tries to build multiple tunnels between them. When the test terminates, it will print the number of successful calls.

In order to avoid stale files, the testsuite will delete generated config files in `tmp/` after executing the test. If you want to keep them, you must set `deleteConfigAfterTest` to `false` in the `Helper` class. To prevent stale open ports, every started process is killed after usage. In addition, every test is terminated after at most two minutes. The suite uses parts of the voidphone testing framework[4]. A keystore has to be set up as described in their README file. Unfortunately, we were not able to add satisfying support for the `SecurityHelper` class in the `testing`-module, so it may be necessary to hardcode the keystore path in the `propertiesPath` variable. The testing module does not need to be provided with hostkeys and configuration files, since it generates them on-the-fly by itself.

[4] `https://gitlab.lrz.de/voidphone/testing`

After having done the mentioned steps, everything has to be built and then the testsuite classes `Testing` and `Test` can be run. This can all be done via the respective Eclipse GUI commands or else via command-line interface:

```
.../16onions$ ant clean
.../16onions$ ant build

.../16onions$ java -classpath bin:\
libs/ini4j-0.5.4.jar:\
libs/commons-cli-1.3.1.jar:\
testing/libs/junit-4.12.jar:\
testing/libs/test/hamcrest-core-1.3.jar:\
testing/libs/bcprov-jdk15on-155.jar:\
testing/libs/lombok-1.16.16.jar \
org.junit.runner.JUnitCore com.voidphone.testing.Testing

.../16onions$ java -classpath bin:\
libs/ini4j-0.5.4.jar:\
libs/commons-cli-1.3.1.jar:\
testing/libs/junit-4.12.jar:\
testing/libs/test/hamcrest-core-1.3.jar:\
testing/libs/bcprov-jdk15on-155.jar:\
testing/libs/lombok-1.16.16.jar \
org.junit.runner.JUnitCore com.voidphone.testing.Test
```

4 Bugs and Limitations

- There remain some errors in the code which lead to sporadic tunnel failures, which are hard to track and reproduce and which we were not able to remedy due to timely constraints. They are probably related to unresolved race conditions. This often affects the node behaviour, when it is an intermediate or end hop of more than 1 tunnel and can result in local exceptions and thus some messages not being forwarded or the tunnel breaking down. It can be observed when executing the real-life test in `Test`, while the artificial tests inside `Testing` do not seem to be affected.

- In the case that at the beginning of a round there is no tunnel existing or to be constructed, a random destination is to be chosen and a tunnel to be constructed to it, which would then be used to exchange cover traffic. The necessary routines for this are in the code, but commented out, since we were not able to make it work in time. It is most probably related to the previously mentioned issue.

5 Project Work

Although there are still some bugs in our module, we consider the project as successful.

5.1 Work Distribution

We tried to split the work equally and to reduce intersections between the two team members. Each member focused on its assigned parts, however, intersections could not always be avoided.

Charlie Groh:

- Multiplexer, layer 1 abstractions
- TestSuite (Includes ConfigFactory)
- API (Onion, Onion Auth and RPS) abstractions

Josef Stark:

- Tunnel building, management, usage and destruction
- Config file parser

The reports were written by both members.

5.2 Spent Effort

Since we mostly worked on the project at the same time, the effort spent by each individual is almost identical. On average, every team member spent around three hours per week for the project.