P2P Systems and Security
Final Report
Team Onion-16
Team name: GoCutYourOwnOnions
Module: Onion (39943)

Müller, Till
till.mueller@tum.de

12. September 2020

1 System architecture

The architecture of the system has mostly stayed as described in the mid-term report. Since most of the work has been focused on implementing the `Onion Layer`, some changes were introduced in its design. The tunnel building process is described in Section 1.2 while all other operations are relatively simple and resemble the standard operating principles of onion routing. Apart from that the storage module was added, providing functions for storing and retrieving the data each peer has to handle. It also offers a simple implementation of a notification system, enabling use cases similar to Java's `Thread.notify` since Go's own functions for synchronization did not fully provide the required functionality.

1.1 Onion Layer

The core module of the system is the Onion module, which implements all functions required for building, using and tearing down tunnels. It communicates with most other modules to retrieve information or provide functionality, e.g. the API module passes incoming API messages to the Onion module for processing.

Some additions and changes were made to the message types the Onion module uses, an overview of the types is available in Tables 1 and 2.

Message type	Description
MSG_KEYXCHG	Indicates that the message contains a Diffie-Hellman nonce encrypted with the receiving peer's public key. It is expected that this message is answered with a MSG_KEYXCHGRESP
MSG_KEYXCHGRESP	May only be sent in response to a MSG_KEYXCHG and contains the response nonce to complete the Diffie-Hellman key exchange
MSG_EXTEND	After the key exchange was successful, this message indicates to a peer that the tunnel should be extended to another peer. The message contains the next peer's IP address, onion port and the encrypted Diffie-Hellman nonce for the key exchange between the tunnel initiator and the next peer. When this message is received, the message type is replaced by MSG_KEYXCHG and the nonce is sent to the new peer. It is then expected that the new peer answers with MSG_KEYXCHGRESP which is forwarded backwards through the tunnel to the initiator which can then derive the shared secret with this new peer
MSG_COMPLETE	As an alternative to MSG_EXTEND the initiator can also send MSG_COMPLETE through the tunnel to a peer with which it just completed the Diffie-Hellman key exchange. This indicates to this peer that it is the destination of the tunnel and that it can now send and receive data through it
MSG_COMPLETED	Send by a peer who received a MSG_COMPLETE message. When the initiator of a tunnel receives the MSG_COMPLETED message it knows that the tunnel has been built successfully and that data can be sent through it in both directions

Table 1: Onion layer message types used during tunnel creation

Message type	Description
MSG_FORWARD	May only be sent to peers that have already received a MSG_EXTEND message. Indicates that the data should be forwarded to the next peer in the tunnel
MSG_DATA	May only be sent to the destination or initiator of a tunnel. Contains payload data to be processed by the receiver
MSG_DESTROY	May only be sent to the destination or initiator of a tunnel. Indicates that the peer at the other end of the tunnel does not require this tunnel anymore and wishes for it to be destroyed
MSG_KEEPALIVE	May only be sent to the destination of a tunnel. This message does not contain any data and is only sent in order to signal to every peer that the tunnel is still being used. It is expected that this message is answered with MSG_KEEPALIVERESP
MSG_KEEPALIVERESP	May only be sent to the initiator of a tunnel. Indicates to the initiator that the tunnel is working and forwarding data correctly
MSG_COVER	May only be sent to the tunnel destination. This message contains random data which is expected to be returned to create the illusion of an actively used tunnel
MSG_COVERRESP	May only be sent to the tunnel initiator. Contains the random data received by the tunnel destination in a MSG_COVER

Table 2: Onion layer message types used after a tunnel has been established

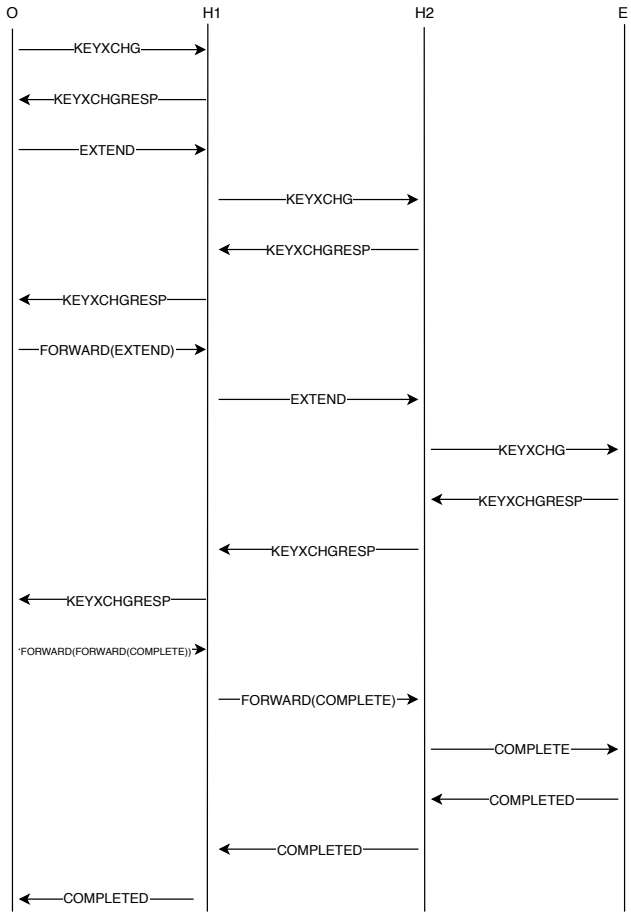

Figure 1: Simplified depiction of the tunnel build process with two intermediate peers. Onion layers and encryption are not shown.

1.2 Building the tunnel

The tunnel building process is relatively straight forward, establishing a shared secret with each peer, extending the tunnel peer by peer and finally sending a `MSG_COMPLETE` to the last peer to indicate that the tunnel is now fully built. This process is shown for two intermediate peers in Figure 1. API communication and other parts of the system that are not directly involved in the module's onion communication have been left out to simplify the depiction.

1.3 Timeouts and Keepalives

To ensure that a tunnel which stopped working does not use resources on the peers unnecessarily each peer monitors all their active tunnels and removes the

tunnel if no data has been received for a set amount of time (currently set to five seconds in the variable `TUNNEL_INACTIVITY_TIMEOUT`). To ensure that a working tunnel is not removed just because no data transfers were requested for a short amount of time, the tunnel initiator sends keepalives which are answered by the tunnel destination. Therefore, as long as the tunnel is working and in use, the keepalive messages ensure that no peer removes the tunnel from its memory, however, if the tunnel stops working (e.g. because one of the peers goes offline) the missing keepalives quickly show to the tunnel peers that the tunnel does not transfer data anymore and needs to be rebuilt.

Similarly, when a tunnel is destroyed on purpose, both the tunnel initiator and destination are notified immediately of this (because one initiated the destruction of the tunnel and sends a `MSG_DESTROY` to the other) while the intermediate peers do not receive a dedicated message informing them that the tunnel has been removed. Instead, after a few seconds of tunnel inactivity they assume that the tunnel is not used anymore and remove it. This reduces the number of message required to be sent when a tunnel is destroyed.

2 Running the software

Due to Go's relatively simple build process using Docker for building the software was deemed unnecessary and a step-by-step guide on how to build it is provided instead. This guide has only been tested on a fully updated Ubuntu 20.04, however, as long as the dependencies are available it should also work on other distributions. A guide containing specific commands for each step can be found in the projects readme file under `implementation/src/README.md`.

1. Install `gcc`, `libssl-dev` and `Go 1.14.8`
2. Switch to the projects `src` folder at `implementation/src`
3. Execute `go build` to build the binary
4. Run the program (`./onion [-c path_to_configuration]`)

When the parameter for the configuration file is not given the program searches for a file called `config.ini` in the current folder and tries to use it as its configuration.

2.1 Testing

To simplify testing the implementation, a testing harness was created which simulates a multi-peer setup on a single machine. The testing setup runs the peers in different sub-processes to fully separate them from each other and enables easy debugging since each peer writes its own log file. The configuration, log and key files for each peer are located in the `src/testing_setup/peer*` folders. Three tests are included with the program, implemented in `main_test.go`:

- `TestBuildTunnel` builds a single tunnel, sends data through it and then tears it down
- `TestMultipleTunnels` creates three tunnels (two from `peer0` to `peer4` and one the other way around) and sends data through them, thereby testing whether simultaneous tunnels are working properly
- `TestCoverTraffic` sends an `ONION_COVER` command to the first peer's API which then creates a tunnel and sends cover traffic through it

To run all tests it is sufficient to execute `go test -v` in the `src` folder. A single test can be run by executing `go test -v -run [TEST]`.

3 Limitations

The system has some limitations with regards to reliability and the amount of data that can be transmitted. Firstly, the system does not make any guarantees about the reliability of the data delivery, all message are sent on a best-effort basis. For some delivery issues an error is returned, although the absence of an error does not necessarily mean that the delivery was successful.

Additionally, the amount of data per packet that the system can handle is limited. All packets are fixed to a size of 1232 bytes and for each hop there is an additional overhead of 21 bytes per packet (16 bytes for cryptography data, 4 bytes sequence number and 1 byte message type). Furthermore, 23 bytes of data are required between each hop for the Hop Layer header and 4 bytes for the `TPort`. Therefore, the maximum size a packet can be when three intermediate peers are used is: `1232 - 23 [Hop Layer header] - 4 * 21 [Onion Layer header] - 4 [TPort] = 1121 bytes`. If more data is required per packet, a fragmentation and reassembly system would need to implemented on top of the Onion module.

Lastly, increasing the number of intermediate peers will decrease the amount of data that can be sent per packet and the reliability will decrease due to the nature of sending more UDP packets that might be lost in transmission.

4 Future work

In the future the system could be improved mainly with regards to privacy, reliability of the data delivery and configurability. To ensure that not even peers which are part of the tunnel can make any guesses about the contents of the packets they forward, padding could be utilized not only on the Hop Layer but also on the Onion Layer. While this would introduce some overhead and require a size field in the Onion header, peers would be able to gather even less information about a tunnel of which the are a part but neither the initiator nor the destination. However, the peers could still count the number of packets and measure the time in which these packets were sent to make guesses about the bandwidth usage of a tunnel.

To improve the reliability of the data stream a TCP-like layer could be introduced over the top of the Onion Layer, ensuring that lost packets are resent and that the messages are delivered to the API in order. While this was explicitly not required by the specification it would improve the applicability of this implementation to general purpose programs.

Lastly, some parameters of the program are defined as constants within the code. While the open-source nature of the system still allows users to change these values, it would be easier if they were defined in the configuration file. A possible issue here is that this might enable the user to create unsafe configurations where tunnels are still built but do not fulfill the projects requirements with regards to security.

All in all, these changes can generally be considered improvements, but they also come with downsides that potentially make them less desirable.

5 Workload distribution

Table 3 describes the workload distribution by implemented package to give an overview over how the work was split up.

5.1 Time spent on the project

During the planning and initial implementation phase, meetings between the two team partners were conducted over one to two days per week. The major part of the implementation took place in the last four weeks of the project. In this time Till Müller worked alone on the project for about three days per week.

While the architecture and design phase was a collaborative effort with many discussions, especially the implementation of the core functionality of the system was performed in the aforementioned last month of the project.

Package	Functionality	Implemented by
`Main`	Initializes and starts the program	Till Müller
`API`	Enables sending and receiving of API messages, keeps track of API connections	Till Müller
`Config`	Loads and parses the configuration and hostkey files	Valentin Langer and Till Müller
`DH`	Implements functionality required for Diffie-Hellman keyexchanges like generating keypairs and deriving the shared secret	Valentin Langer and Till Müller
`Encryption`	Offers functions for asymmetric and symmetric encryption	Valentin Langer and Till Müller
`HopLayer`	Enables secure communications between two peers by performing a Diffie-Hellman handshake and encrypting all subsequent communication	Valentin Langer and Till Müller
`Logger`	Implements logging to simplify debugging and error reporting	Valentin Langer and Till Müller
`OnionLayer`	Implements the main logic and functionality of the system: Builds tunnels, handles the forwarding of data through tunnels, receives tunnel messages and implements the handler for incoming API commands	Till Müller
`Storage`	Handles almost all other modules' storage requirements by offering thread-safe temporary data storage capabilities. Also implements thread signalling functionality	Till Müller
`Testing Setup`	Runs and tests the whole system in different scenarios; implements a mock RPS module and a minimal API client for this purpose	Till Müller

Table 3: Package functionality and workload distribution