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

Langer, Valentin
valentin.langer@tum.de

Müller, Till
till.mueller@tum.de

1. July 2020

1 Changes in Assumptions

All assumptions from the initial report are still valid except for the following:

- To implement a Diffie-Hellman handshake, we required cryptographic functionality beyond the scope of Go's own `crypto` module and therefore added an unofficial OpenSSL wrapper library [1]
- To improve testing and our ability to find errors in our implementation we used Go's official logger module to implement our own logging system. It records erroneous behaviour and significant changes in the state of our program in an easy-to-use way. This runtime information then aids us in finding logic and implementation issues and fixing them quickly

2 Architecture

2.1 Logical Structure

We have chosen to divide our project into different packages corresponding to different jobs that our module performs. Our general approach is to separate our implementation into a lower layer, called "Hop Layer" in charge of maintaining encrypted connections between nodes of our network, and a higher "Onion Layer" in charge of building onion bridges on top of these connections. We encapsulated additional functionality in the packages below and follow with a more detailed explanation of the Hop Layer's inner workings. We have not started development on the Onion Layer so it will not be part of this section.

dh	Holds functions for generating a prime256v1 elliptic curve public-private key pair and for deriving a shared secret using two Diffie-Hellman parameters
encryption	Provides Encrypt/Decrypt functions using a shared secret
logger	Initializes logger for writing errors and status changes to a log file

Table 1: Additional packages we have implemented

2.1.1 Hop Layer

This is the lower layer of our two-layer architecture, contained in its own package called `hopLayer`. It provides a function for sending packets and another that waits for a message and checks as well as decrypts it before passing it on for higher level processing. Internally, when trying to send a packet, it performs the following steps:

- Throw out packet if it is too large
- Check for existing shared secret, otherwise perform new Diffie-Hellman key exchange with destination peer to derive a common shared secret
- Fill out *size* field in Hop Layer header to specify offset at which content has ended and buffer bytes start
- Fill out *sequence number* field in Hop Layer header. The value is the previous one incremented by one each time a packet is sent using the same DH key, with a value of 0 for new connections. The aim of this is to guarantee a different encrypted message when resending the same bytes, avoiding replay attacks
- Encrypt message (excluding parts of the header, see 3.1) to send using the negotiated shared secret
- Pad packet with random data until a uniform packet length of 1232 bytes is reached
- Send message to destination using UDP

The *SetPacketReceiver* function is used for binding a listening address to a callback function that handles an incoming packet from said address. When a message is received the following steps are performed internally before handing the content off to the higher level callback function. Whenever the packet is discarded, the error is logged and the subsequent steps are not performed. The steps taken are the following:

- If message is not exactly 1232 bytes long, discard message

- Check first bit for a set *handshake flag*. If it is set, perform the second part of the Diffie-Hellman handshake and record the shared secret before returning
- If the *handshake flag* is not set, look up shared secret for this connection and use key to decrypt everything after the initial *reserved* field. If no key is found, discard packet
- Confirm that sequence number comes after sequence number of previous message and increment locally recorded sequence number, otherwise discard packet
- Call the callback function with the extracted *data* of length *size*

2.2 Process architecture

For multithreading we employ Go's `goroutine` functionality which offers lightweight threading. While this does require some synchronization to take place, for example to ensure that only one thread is writing to certain data structures at the same time, this approach still fulfills all of our threading requirements while not creating too much implementation overhead.

At the moment we aim to make our module as concurrent as possible to avoid bottlenecks. This means that we intend to create a thread for the following tasks:

- Listening on a socket for incoming packets
- Decrypting an incoming packet and passing it's data to higher layers
- Performing a Diffie-Hellman key exchange
- Handling an API request
- Building / tearing down a tunnel

2.3 Networking

The underlying protocol for our networking stack is UDP, due to its simplicity advantage over TCP. It offers good compatibility to our message-based tunnel protocol, while TCP's stream based approach would require manually recreating message boundaries. Additionally, since the specification defines that we do not need to guarantee further reliability than what UDP can already provide, we were also not required to implement our own mechanism for dealing with packet loss, congestion- or flow control.

We do, however, have a way to recognize when packets have not been transmitted correctly, due to our protocol containing sequence numbers. While these currently serve only a security purpose, their use could be extended in a future implementation.

2.4 Security Measures

We have added several security measures to our architecture in order to ensure that no unwanted behavior can be achieved by a potentially malicious actor. This includes:

- Preventing replay attacks by using sequence numbers and dropping messages in case they contain a sequence number which has already been seen
- Encrypting messages between hops in our Hop Layer protocol as well as encrypting in the normal onion fashion where encryption layers are all added at the initial sending peer and then *peeled off* one by one by the subsequent hops
- Not reusing any tunnel details (see Tunnel Port numbers in 3.1.2) in a way where they can be read by multiple hops. This ensures that most correlation attacks are prevented and correlation would only be possible via time-based traffic analysis. However, due to the VoIP systems round-based model we do not expect these to enable meaningful insights into communication paths either
- To ensure that a malicious peer cannot pretend to be another peer, the Diffie-Hellman nonce we send out to establish the ephemeral symmetric key, which is then used in communication with that peer, is encrypted using the already known public key of the intended recipient. Therefore, only the network member we are trying to add to the tunnel can derive the ephemeral key and read our messages

3 Protocol Documentation

3.1 Messages and formats

In this section we will list the formats of all messages which have been finalized and implemented. While there are additional messages explained below, we have not decided on a final format for them, yet.

3.1.1 Hop Layer messages

- The Diffie-Hellman exchange message format can be found in Figure 1. This message is required by the Hop Layer to create a ephemeral symmetric encryption key with a peer (Diffie-Hellman key exchange)
- The data message format can be found in Figure 2. It is used for transmission of Onion Layer (see 3.1.2) data, after the ephemeral key has been created

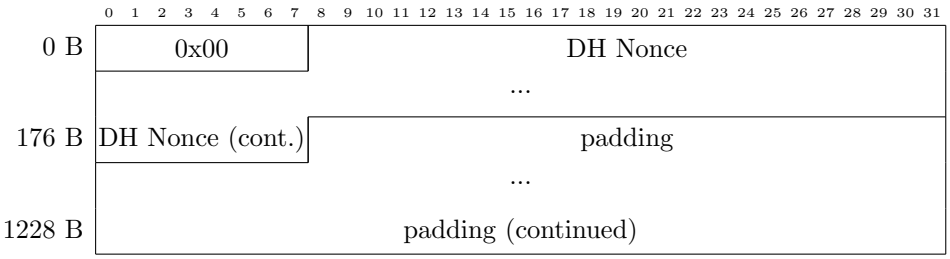

Figure 1: Hop Layer Diffie-Hellman key exchange message.
Padded size: 1232 bytes

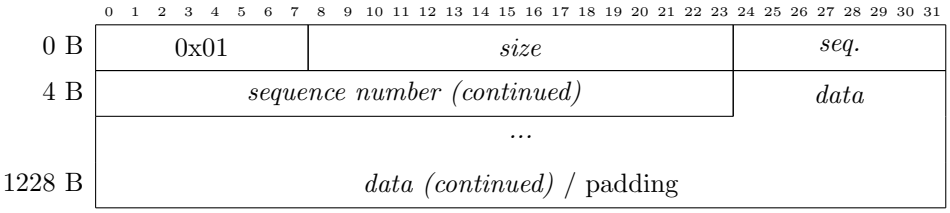

Figure 2: Bit diagram of a Hop Layer data message.
Padded size: 1232 bytes
Encrypted fields are emphasized

3.1.2 Onion Layer messages and general concept

Since the implementation of the Onion Layer is not yet complete, all these details are still subject to change and only an initial idea of how the system could work. Our plan is to have an Onion Layer header which contains at least a `TPort` (Tunnel Port), sequence number and an action field that indicates how the rest of the message is to be handled. The TPort identifies the tunnel uniquely from a peer's point of view and is only ever the same between two peers to prevent correlation attacks. A tunnel with `n` peers is therefore referenced by `n - 1` TPorts in total with each peer keeping a record of the TPort used for communication in either direction. The sequence number, unlike TCP sequence numbers, is only intended to eliminate the possibility of replay attacks by requiring incoming packets to have a strictly increasing sequence number. This means that every communication between two peers has two sequence numbers in the Onion Layer, one for each direction, and a peer needs to translate between them when forwarding a message. We could use the sequence numbers to detect lost packets, however, since we are not required to guarantee delivery of a given packet we do not intend to do this.
Additional Onion Layer messages we will implement are (fields that are encrypted on the Onion layer are *emphasized*):

- `EXTEND` (Fields: IP version, IP next hop, *Last Hop Flag, Diffie-Hellman Nonce*): This message indicates to a peer that the tunnel creator intends

for a tunnel to be created or extended further. For this purpose, the IP address (IPv4 or IPv6) of the next hop is included in the message. The Diffie-Hellman nonce is intended as the first part of the ephemeral key creation with the new hop and is therefore encrypted with this new hop's public key. This ensures that only the peer we intend to add to the tunnel can derive the ephemeral key between the tunnel initiator and itself. Finally, the last hop flag indicates that a tunnel will not be extended further by the initiating peer

- `EXTENDED` (Fields: Diffie-Hellman Nonce Response): This messages indicates that the `EXTEND` command was successful and contains the nonce from the new tunnel hop so that the tunnel creator can derive the ephemeral key to use for onion layered encryption.

- `FORWARD` (Fields: *Data*): This message tells a peer to forward a message to the next peer as indicated by the TPort number from the onion header. A peer receiving this message should be able to forward based on an internal data structure mapping TPorts to IPs. Different actions are taken depending on the type of forward received:

  - A `FORWARD_RIGHT` message indicates a forwarding in the direction of the tunnel endpoint (as opposed to the tunnel creator). A peer receiving this message would then translate and replace the TPort and sequence number with the one it shares with the next hop (and then increment the latter in internal memory after sending).

    This is followed by decrypting the SDU [2] of the Onion Layer and sending a concatenation of the new TPort, new sequence number and new SDU [2] to the next hop as given by the internal TPort/IP map. The peer also has to increment the sequence number for the peer it received the message from in its memory

  - A `FORWARD_LEFT` message indicates a forwarding in the direction of the tunnel creator. A peer receiving this message would check its internal memory to determine if it is the last hop or if a next TPort in the appropriate direction is stored. In the former case the peer can decrypt the SDU [2] using its ephemeral keys for this tunnel. Otherwise, it would fetch the new TPort and sequence number and add one more layer of encryption to the SDU [2] using its ephemeral key for this tunnel and use these fields to send a new `FORWARD_LEFT` message to the next hop in the direction of the tunnel creator as given by the internal TPort/IP mapping

3.2 Exception handling

We identified the following possible exceptions as requiring to be addressed explicitly:

- When a node which is part of the tunnel leaves the network or loses its connection, we need to detect this change and rebuild the tunnel. For this purpose, we will implement regular keep-alive messages sent from the tunnel initiator to all nodes within the tunnel which prompt the node they are addressed to to answer with a confirmation that the message was received. This allows us to quickly rebuild a tunnel, should it stop passing messages. Additionally, a node can assume that a tunnel it is part of is not used anymore when these keep-alive messages are not received for some time. To make sure that the tunnel is not rebuilt unnecessarily, we will require multiple messages to remain unanswered before assuming that the tunnel is not functional anymore

- Since our system provides voice over IP, a slightly corrupted data packet does not need to be corrected. We will therefore not checksum or resend these packets, to simplify our implementation and increase the performance of the tunnel. However, we will implement checksumming for control messages since they rely on the correct data being received. Since these control messages need to be able to deal with packet loss by sending unanswered messages again, our implementation will just discard any incoming control packet that does not provide a correct checksum

- Should a peer send an invalid message (either malformed or not applicable in the current application state) we will answer with a reset control message, requesting the peer to initiate a new Diffie-Hellman handshake. This will make sure that a node which has crashed but then restarted can still take part in the network, even when did not retain any information about active connections it had before the crash

4 Future Work

Over the next weeks we will implement the Onion Layer as described above, including refinements to its architecture. Our efforts will also include exhaustive testing and the creation of automatic tests for as many parts of our system as possible. As a part of this, an automatic build and deployment system will be created. Depending on its complexity, this will either be a Makefile which builds our code using Go, or be dockerized for improved reproducibility.

5 Workload Distribution and Effort

So far, we have done all development and planning work in remote pair-programming sessions, which enabled us to quickly exchange ideas about the implementation and review changes as we go along. Therefore, we both have knowledge of everything we have implemented and cannot separately assess our work. Our schedule consists of weekly meetings which take up the entire day. We expect

to finish the implementation on time at the current pace, however, we are also prepared to increase the meeting frequency, should this be necessary.

Furthermore, we have spent a considerable amount of time on the architectural design of the system, taking as many security considerations into account as possible. This delayed our implementation phase but in turn made it possible to catch possible security holes before writing any code.

References

[1] https://github.com/spacemonkeygo/openssl

[2] https://grnvs.net.in.tum.de/slides_chap0.pdf
Slide 45