Group 22: Interim Report

Rohit Panda

rohit.panda@tum.de

Benedikt Brandner

ga49xel@mytum.de

Abstract

This is the interim report of group 22, working on the onion module.In this report we provide an explanation of our design choices regarding the architecture of the module and the P2P protocol used to communicate with onion module on another system.

1. Preamble

1.1. Team

Teamname: voidphone-onion-qt
Module: onion

- Bendikt Brandner, ga49xel@mytum.de
- Rohit Panda, rohit.panda@tum.de

2. Process Architecture

We plan to use C++ with the Qt framework. Qt being an event-driven toolkit, events and event delivery play a central role in Qt architecture. Events can be generated from both inside and outside the application. They're instead queued up in an event queue and sent sometime later. The dispatcher itself loops around the event queue and sends queued events to their target objects, and therefore it is called the event loop. We enter Qt's main event loop by running QCoreApplication::exec(); this call blocks until QCoreApplication::exit() or QCoreApplication::quit() are called, terminating the loop.The event loop can be woken up by sockets activity (there's some data available to read, or a socket is writable without blocking, there's a new incoming connection, etc.). All low-level Qt networking classes (QTcpSocket, QUdpSocket, QTcpServer, etc.) are asynchronous by design. When you call read(), they just return already available data; when you call write(), they schedule the writing for later. It's only when you return to the event loop the actual reading/writing takes place. Notice that they do offer synchronous methods (the waitFor* family of methods), but their use is discouraged because they block the event loop while waiting. High-level classes, like QNetworkAccessManager, simply do not offer any synchronous API and require an event loop. Network programming model works well with a state machine that reacts on inputs of some sort and acts consequently. Now, there are several ways to build a state machine (and Qt even offers a class for that: QStateMachine), the simplest one being an enum (i.e. an integer) used to remember the current state. [1]

3. Inter-module Protocol

Onion module deals with onion routing which involves encapsulating data in layers of encryption analogous to layers of an onion. The layers are to be decrypted by the intermediary hops before arriving at its destination.The original message remains hidden as it is transferred from one node to the next, and no intermediary knows both the origin and final destination of the data, allowing the sender to remain anonymous.[2]

In this project onion module is divided into two parts: Onion Forwarding and Onion Authentication module. Onion Forwarding is concerned with handling the P2P connections and API connections. Onion Authentication is concerned with the authentication of intermediate and destination hops and, ephemeral session key establishment.[3]

[1] https://wiki.qt.io/Threads_Events_QObjects

[2] https://en.wikipedia.org/wiki/Onion_routing

[3] https://www.moodle.tum.de/pluginfile.php/1178989/mod_resource/content/6/specification-2017-2.0.pdf

3.1. P2P Protocol

The objectives of this module are: Sending messages to onion module of another peer for establishing the tunnel, relaying actual data, relaying dummy data and tearing down the tunnel.[4]

We use a fixed message length of 512 bytes which consists of a header and a payload.

Assumption: According to specification we need two intermediate hops between source and destination.

We need two kinds of messages in our protocol:

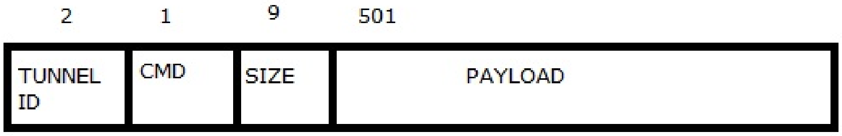
1. Command Messages which are meant to be interpreted by the node that receives it.

2. Relay Messages which are meant to be relayed to other peers. These messages may carry other command messages or end-to-end data.

3.2. Command Messages

The different commands to be used in Command Messages are:

1. BUILD
2. CREATED
3. DESTROY
4. COVER



2	1	9	501
TUNNEL ID	CMD	SIZE	PAYLOAD

Figure 1: COMMAND message format

Command Messages have fields:

1. TUNNEL ID

[4] https://svn.torproject.org/svn/projects/design-paper/tor-design.pdf

Tunnel ID is a unique number denoting the tunnel between two peers. All messages between the two peers will have the same Tunnel ID.

2. COMMAND
Can be any of the above commands.

3. PAYLOAD SIZE

4. PAYLOAD

3.3. Relay Messages

The different commands to be used in Relay Messages are:

1. RELAY DATA
2. RELAY EXTEND
3. RELAY EXTENDED
4. RELAY TRUNCATED

Relay Messages have fields

1. TUNNEL ID
Same as above.

2. STREAM ID
May be used for multiplexing if allowed.

3. COMMAND
Can be any of the above commands.

4. CHECKSUM
end-to-end checksum for integrity checking

5. PAYLOAD SIZE

6. PAYLOAD

3.4. Circuit Establishment

When the onion module of source 'SRC' wants to establish a tunnel with first peer say 'A' it sends a BUILD message with the handshake payload provided by the onion authentication module.'A' uses a tunnelID($\mathrm{t_{SRCA}}$) at its end to signify this tunnel.

Then 'A' responds with CREATED message with the negotiated session key.

To establish connection with the second peer 'B' in the chain it sends a RELAY EXTEND message to 'A' with the address of 'B' and handshake payload for 'B'. 'A' on receiving this message creates a CREATE command message with payload from SRC and a local tunnelID with 'B'($\mathrm{t_{AB}}$). The 'SRC' is never aware of this tunnelID. 'A' associates this tunnelID($\mathrm{t_{AB}}$) with the tunnelID($\mathrm{t_{SRCA}}$) When 'B' responds with a CREATED message 'A' copies the payload into a RELAY EXTENDED message and sends it to 'SRC'.

To establish connection with the destination 'DEST', 'SRC' sends a RELAY DATA message to 'A' with encrypted(session key between 'SRC' and 'A') payload. 'A' will see the tunnelID($\mathrm{t_{SRCA}}$) of this message from 'SRC' and knows it has to decrypt and relay this message to 'B'. The payload of this message is a RELAY EXTEND message to 'B' with the address of 'DEST'.'B' on receiving this message creates a CREATE command message with handshake payload from SRC and a local tunnelID with 'DEST'($\mathrm{t_{BDEST}}$). When 'DEST' responds with a CREATED message 'B' copies the payload into a RELAY DATA message with a RELAY EXTENDED message payload and sends it to 'A'. 'A' will encrypt this message(session key between 'SRC' and 'A') and send it to 'SRC'.

3.5. Relay Data

Once 'SRC' has established the circuit/chain he can send RELAY messages.

On receiving a message, the peer looks up the corresponding tunnelID, and decrypts the relay message header and payload with the session key for that tunnel. If the cell originates from 'SRC' the peer then checks if decrypted message has a valid digest. If so it knows that the message was meant for it otherwise it looks up the tunnelID and finds the next peer for the message, changes the tunnelID and sends the decrypted message to the next peer. To construct a message for a given node ’SRC’ encrypts the message iteratively with shared session key of all nodes in the chain upto that node. When a node replies to ’SRC’ it encrypts the message with shared key with ’SRC’ and sends it to the next node in the circuit towards ’SRC’. The ’SRC’ on receiving a message decrypts the message using the keys from all nodes in the path

3.6. Tear Down

To tear down a circuit, ’SRC’ sends a DESTROY command message. When ’A’ receives the DESTROY message it closes its connection with ’SRC’ and forwards the message to ’B’. ’B’ does the same.The circuits are torn down incrementally. ’SRC’ can send a relay truncate cell to a single node on the circuit. That node then sends a destroy cell forward, and acknowledges with a relay truncated cell.

3.7. Cover

To mimic the characteristics of real traffic a node can send dummy data with COVER command message. On receiving such a message a node has to do nothing.

3.8. Authentication

When the onion module starts to build a tunnel it sends the first message BUILD with the first half of the Diffie-Hellman handshake.The onion module running on the node answers with CREATED message and completes the handshake. With this handshake we can achieve entity authentication and key authentication. We also achieve forward secrecy and key freshness.

3.9. Error Handling

If the node at the end or beginning of the chain receives an unrecognised message or the digest doesn’t match the payload then the circuit needs to be torn down. The ’SRC’ can send a DESTROY command or a RELAY TRUNCATED command to tear down the circuit. Also if a node in the circuit goes down the adjacent node can send a RELAY TRUNCATED message to ’SRC’.

3.10. Message Types

COMMAND MESSAGE

1. BUILD 0x01
2. CREATED 0x02
3. DESTROY 0x03
4. COVER 0x04

RELAY MESSAGE

1. RELAY DATA 0x01
2. RELAY EXTEND 0x02
3. RELAY EXTENDED 0x03
4. RELAY TRUNCATED 0x04