Group 22: Final Report

Rohit Panda

rohit.panda@tum.de

Benedikt Brandner

ga49xel@mytum.de

Abstract

This is the final report of group 22, working on the onion module. In this report we provide documentation of the finished module, final protocol design, possible future improvements and work distribution details.

1. Preamble

1.1. Team

Teamname: voidphone-onion-qt
Module: onion

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

2. Documentation

This section documents usage and build process of the module. Much of this information can also be found in the module's Readme file in a more condensed form.

2.1. Dependencies

The project is built with the Qt Framework, which is the only dependency required. Qt can be obtained from `https://www.qt.io/download-open-source/`. Make sure to install Qt 5.8 or above, as we use quite recent features in the code.

2.2. Build & Installation

After installing Qt, open the onion.pro project file in Qt creator. The project should build without errors, and create an executable named "onion". The code should be fully functional on most OSes (Windows, Linux, OSX, ...), we only used Windows for our development.

The project file contains a switch for building tests. To build unit tests, open the onion.pro file by double clicking it in the project view. On line 5 a conditional define enables building of unit tests. If this switch is uncommented, the project will build an "oniontest" executable from the tests subdirectory instead. Running that performs the tests, they can also be observed in the application output window.

2.3. Running

If run from within Qt Creator, the run button should be sufficient. The module will need a config file supplied with the -c parameter. Inside Qt Creator, press Ctrl+5 and click on the "Run" configuration on the left, then set command line parameters in the main view.

When running outside Qt Creator under Windows, e.g. for the Marco/Polo test, the application needs to be deployed first. To do this follow these steps:

1. execute the `Qt <version> for Desktop` item from your start menu. It opens a shell setup for the respective toolkit.
2. cd into your build folder
3. execute `windeployqt.exe onion.exe`

If run without parameters, the module prints its help and exits. To run a real-world test, sending messages through tunnels, please refer to the "Marco/Polo" section of Readme.md.

2.4. Limitations

Currently, there is no option to configure the number of intermediate hops that a tunnel uses without recompiling the application. In principle this number is variable by using the setter of the PeerToPeer class, just loading the parameter from a config file or command line parameter is not implemented. All code has been written with a variable number of hops in mind, for implications on messages, see the Protocol section.

With the way that the project is set up, there is no possibility to build both tests and the regular executable, but we consider this a minor limitation, since it can easily be switched by commenting a line in the .pro file and build times are very short for this project.

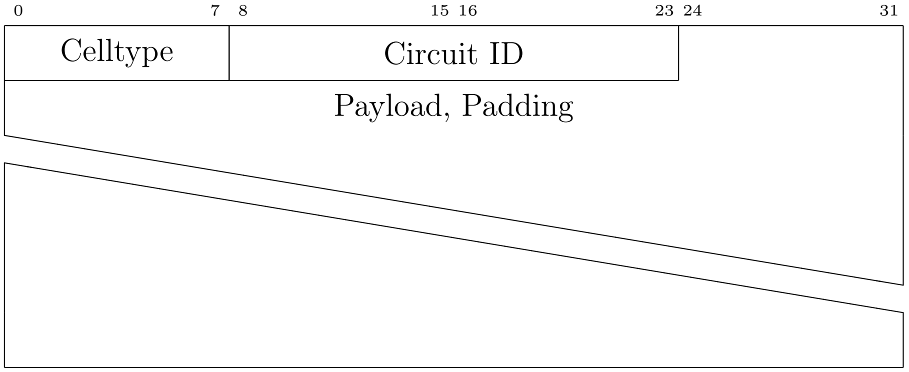
3. Peer To Peer Protocol

This section details the final implementation of the Peer to Peer protocol used by onion. It is still very similar to the interim version, but a number of things have changed to ensure security goals and make the implementation easier.

The protocol is heavily inspired by the tor protocol[1], with some modifications for simplicity and others due to the lack of a public key infrastructure in our setting and the fact that we do not employ link-level encryption on the Peer-to-Peer UDP connection.

3.1. Protocol Messages

The objectives of this module are: Sending messages to an onion module of another peer for establishing the tunnel, relaying actual data, relaying dummy data and tearing down the tunnel. We use a fixed message length of 1027 bytes which consists of a three byte header and a payload.



We differ between three different celltypes, through the Celltype field.

- 01 **Build:** The build message indicates the intention of the sender to establish a connection with the receiver, and contains the sender's half of a key exchange for the given circuit id.

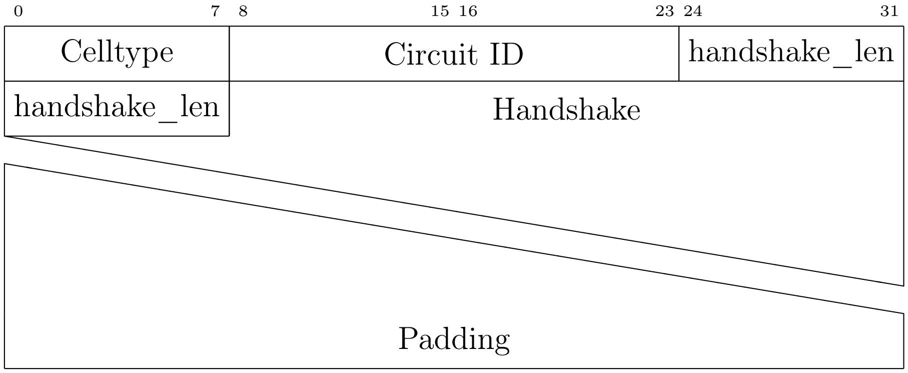
- 02 **Created:** The created message acknowledges the creation of a connection between sender and receiver and contains the other half of the key exchange.

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

- 03 **Encrypted:** The encrypted message is encrypted with the the key specific to this circuit (i.e. sender/receiver address and circuit id). The full payload behind the three-byte header is encrypted. With 1024B payload, no extra padding is needed, as we have a multiple of the most common cipher block sizes of 128 and 256B.

3.1.1. Build & Created

The Build and Created messages have to be sent unencrypted, because there is no way for the receiver of a build message to know the identity of a sender for a build message. In order to do so, every node would have to know the public keys of every other node, which is not the case in the voidphone network. The message layout for build and created cells (with $T = 01|02$) is as below:

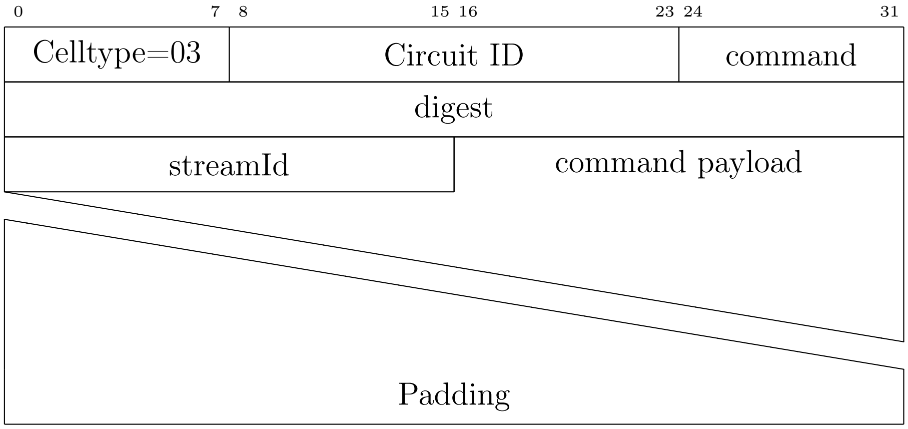


Since both messages are unencrypted, the handshake can be observed by anyone on the line, additionally subsequent messages of this circuit can be distinguished from other circuits between these nodes, by observing the circuit ids of encrypted messages. This leaves the application vulnerable to end to end correlation attacks especially during circuit establishment. Since TOR doesn't protect from that either, we deem this acceptable. It would be good to disguise different streams between hops, but as mentioned above, we don't have public key infrastructure on a hop-to-hop level, and thus we cannot encrypt on a link level, requiring an unencrypted circuit id to determine the correct key for a stream.

With both messages unencrypted, we have to address possible man in the middle attacks. During tunnel establishment, the source knows all public keys of all hops on the path through the gossip and rps modules. Thus a man in the middle would be discovered by the source (i.e. the handshake would produce dysfunctional keys). The receiver of an incoming tunnel has no method to verify the identity of the initiator, since it cannot know all public keys of all nodes in the network. Thus the initiator of a circuit could impersonate another node. We argue that this is not a problem, as all traffic occurring in that circuit is only used to serve requests of the initiator. To be more precise, during the onion routing any hop on the circuit only has an established session key with the initiator, thus there exists no traffic from other nodes that could be manipulated by the initiator impersonating someone else. This leaves traffic from the initiator to us. Since the onion part here behaves as a transparent transport layer, it is up to the API user to prevent sybil attacks on application level.

3.1.2. Encrypted

The fundamental operations of the module are tunnel establishment, teardown, cover traffic and real traffic. To facilitate this we use a combination of build, created and a number of different encrypted messages. Each encrypted message has the header described below:



The command field differentiates the encrypted messages, and determines their command payload. The digest is used to determine whether a message is encrypted or not. It is currently set to zero initially, encryption will scramble the contents and a hop will be able to know the message is still encrypted after it decrypted its onion layer. See Future Work Section for more details.

The stream id field could be used to multiplex different connections of the same circuit in the future. It is ignored for some types of commands, for which it is used as a reserved field.

3.1.3. Destroy Encrypted Command

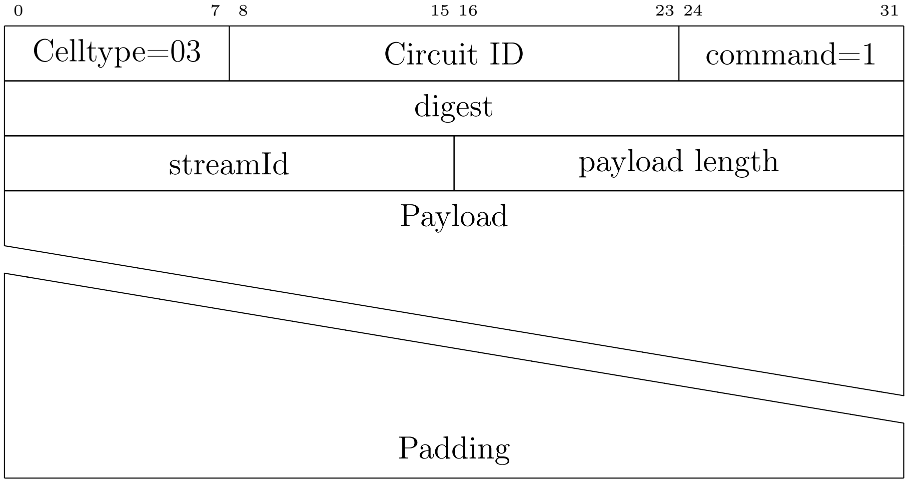
This command indicates circuit teardown from the source and has no further payload. It does not use the stream id. It is in the source's responsibility to sensibly tear down a circuit, i.e. send destroy messages in reverse order of the tunnel. Tunnels cannot be torn iteratively, because there is no secure channel between intermediate hops, only between source and the respective hops. Thus allowing iterative teardown would mean having unauthenticated teardown, which allows for easy denial of service attacks.

3.1.4. Cover Encrypted Command

This command indicates that the message is cover traffic, irrelevant of the stream id. It has no further payload, the cover traffic should be dropped by the destination node.

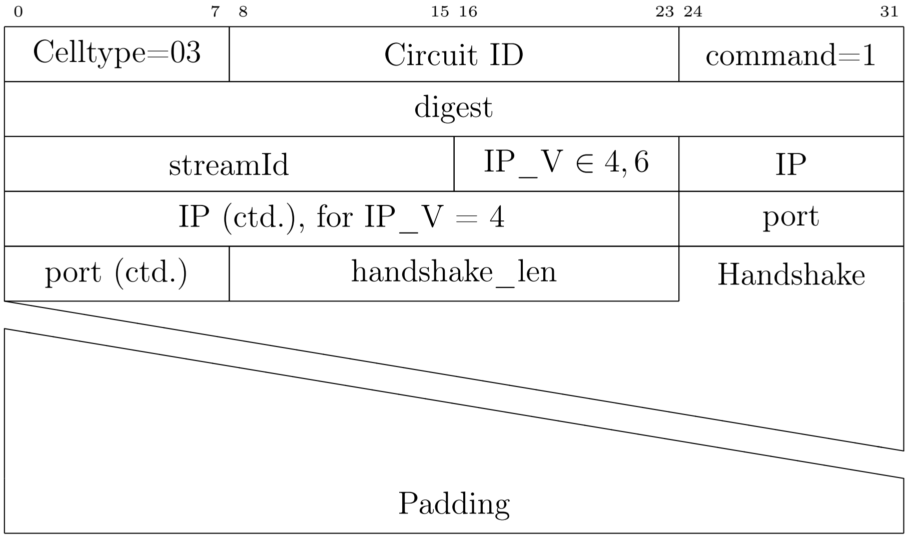
3.1.5. Relay Data Encrypted Command

This command indicates application level data on the tunnel. The data is forwarded to the onion api client. The command payload consists of a 16bit field specifying payload length, followed by the data.



3.1.6. Relay Extend Encrypted Command

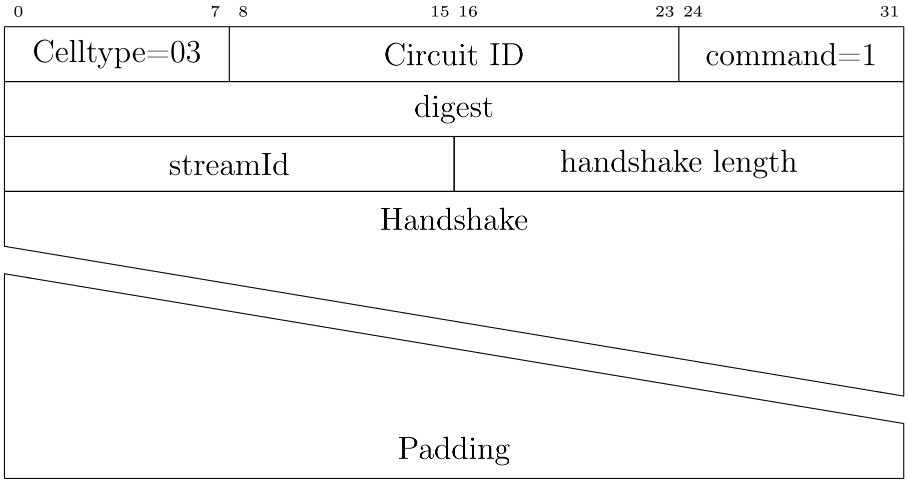
This command indicates that the receiving node should extend this circuit to a new node, specified in the command payload. The receiving node should send a build message to the target, containing the handshake supplied by the source, and upon receiving a created message should reply with a RELAY EXTENDED message, forwarding the other half of the handshake.



The field IP_V specifies the IP version of the peer IP, for IP_V=4 the IP is contained in the following 4B, for IP_V=6 the IP is contained in the following 16B. The packet also contains the first half of the handshake from source to the next hop, which should be used in the subsequent build message.

3.1.7. Relay Extended Encrypted Command

This command is sent in reply to a RELAY EXTEND message, after the tunnel was extended. It contains the handshake of the new peer, which is only interpretable by the source of the circuit.



3.1.8. Relay Truncated Encrypted Command

This command is sent from a intermediate hop towards the source of the circuit. It indicates that the connection to the subsequent node in the connection has gone down. It is in the source's responsibility to deal with the situation, i.e. tear the circuit, or repair it using another node. The command has no further payload, as the source can identify the sender during layered decryption. The stream id field is ignored, as the truncation affects all streams of a circuit.

3.2. Protocol Behavior

We will discuss the behavior of the system from different perspectives:

3.2.1. Circuit Initiator

Upon a TUNNEL_BUILD message from the onion api, the protocol samples intermediate hops from the RPS api, then it creates handshakes for each using the onion auth api. Only after that it sends a build message to the first hop. Upon receiving a created message from that is begins sending relay extend messages, increasing the hop length after each relay extended message. As an example, for the third intermediate hop C, the relay extend message $R_{src,c}$ is encrypted with $K_{src,b}$ and then with $K_{src,a}$. Thus the first intermediate hop A cannot read the message, and will not know which command was sent. Only B will notice and process a relay extend command.

After the tunnel is established, onion api emits TUNNEL_READY and TUNNEL_DATA and TUNNEL_COVER api commands can be processed by onion-encrypting and sending to the first hop.

3.2.2. Intermediate Hop and Destination

A node with an incoming tunnel does not know whether it is the destination node or an intermediate hop. The source has full control over where in the circuit the traffic exits the onion module, by encrypting with the correct amount of keys. As argued above, for an incoming build message, the sender is trusted to be genuine, the module sets up a session using the onion auth api, and replies with a created message. When subsequently receiving a relay extend message, the contained half-handshake is used to establish a new connection with the target node, and upon creation of that a relay extended message is sent back to the source of this tunnel.

3.2.3. Circuit IDs

The circuit ID is specific for each host (IP and port) and circuit from the perspective of a module. That means a circuit ID is valid for one circuit and one link. It is internally mapped onto the 32bit tunnel ids that the onion api uses. The circuit initiator chooses a new circuit id in the build message and saves it together with the recipient IP and port. The recipient always replies with this circuit id. That means that circuit ids are swapped between each hop of the overlay network, much like mac addresses. That also implies that they cannot be used to determine the sender of a message. For that see the following section:

3.2.4. Message Forwarding

Build and Created messages are always only intended for the next hop, establishing a link. When sending an encrypted message, there are two options: Either we are the initiator of the circuit, then we know[2] our destination as well as all the hops on the way. Thus we can onion-encrypt with keys in reverse order and send it to the first hop. Behavior of the hops along the way is described below. The other option is that we are another hop in the circuit, trying to send to the initiator. For that we encrypt the message with our key $K_{us,src}$, and send the message to our previous hop, i.e. the one that sent us a build message initially.

When receiving an encrypted message, we can follow a simple flow chart (see PeerToPeer::handleDatagram):

1. We check if the tunnel id (combination of peer address and circuit id) is a start of any of the circuits we initiated, i.e. the first hops of our circuits for their circuit ids. If so, this message is a reply from a tunnel we made, thus we decrypt in order with the keys of this circuit (starting with the first hop), until the message has a valid digest. Then we know both the sender of the message and the message content and can process it (forward application data, continue building a tunnel, etc.).
2. If the condition above does not apply, the message is from a tunnel where we are not the initiator, thus, depending on the direction of the message, we need to add another layer of encryption (when the message is headed from destination towards initiator) or peel off one

[2] i.e. we have an auth session established with

layer of encryption (when headed to destination). To do so we iterate our incoming tunnel mappings and search the next hops (the ones we built connections with after a relay extend message) for the tunnel id of the message. If found, this message is in transit from somewhere down this tunnel to the originator, thus we add a layer of encryption and send it.

3. If we don't find the tunnel id as above, we look for it in the previous hops of our tunnel mappings. (If not in there we discard the message). This message is in transit from the tunnel source to us or somewhere further down the tunnel. Thus we peel off one layer of encryption (decrypt with $K_{us,src}$) and check the message digest. If it is valid, we process the message, else we forward it to the next hop (known from our mapping) for further processing.

3.2.5. Retry Behavior

In case of packet loss, timeouts or other interference with messages, onion employs a lossy UDP-style best-effort mechanic, i.e. there are no guarantees for data to arrive in order or to arrive at all. However, for building tunnels, we want onion to be a little more robust, dealing with minor hickups. The mechanic we employ is that the initiator of a tunnel retries the current setup step ((layered) relay extend or build) after a certain timeout. That way robustness is achieved with minimal effort. This behavior is also intuitively correct, since an intermediate hop has little incentive to spend much effort in extending a tunnel for someone else, if there is some sort of problem. This also mitigates some of potential amplification attacks.

4. Future Work

As this is only a one-semester project we had to make compromises with respect to feasibility of the project. Below we discuss some next steps that could improve the project.

4.1. Message Layout

For some messages, like the build and created messages or the relay extend encrypted message, the layout of byte fields is not fully aligned with four-byte marks. This not only makes the graphs look more ugly, it also causes problems with respect to performance but can also cause pitfalls with some C++ implementations. One could introduce a one-byte padding for these messages to fix that.

4.2. Digest

The digest is currently simply set to zero, thus the validity check is done by checking for zero. We did this primarily for simplicity while implementing. The error correction of a potential CRC sum can easily be replaced by using the checksum mechanisms built into UDP, which is already widely employed in practice. The problem here is that any packet which encrypts to a digest of zero at any point during encryption will cause errors during processing, as an encrypted packet is erratically processed as unencrypted. The probability of this happening is 1 in $2^3$2, roughly 2.3×10^{-10}.

4.3. Further Testing

Due to restrictions in time and unavailability of functional reference modules we couldn't test the module in the final voidphone setting. However through unit testing all involved api components and the Marco/Polo test that tests the peer to peer behavior of all onion api functions, we are confident that the module is indeed functional. We even simulated a test for the robustness described above through starting some peers late. Still, there are a couple of testing ideas that we could not realize due to a lack of time.

Most prominently we wanted to create an onion api client that essentially tunnels any UDP and/or TCP traffic through onion. Therefore upon launch, a tunnel would be built and ONION_TUNNEL_DATA packets would be transformed to respective UDP/TCP packets. An easy performance test would then be given by e.g. running iperf in UDP mode through onion, either on a localhost setting, or facilitating VMs or a mininet setting.

5. Work Division

- Benedikt Brandner: Initial Report, Implementation Concept, Architecture, Onion Api + Tests, RPS Api + Tests, PeerToPeer module and behavior, message parsing, Marco/Polo test, Final Report
- Rohit Panda: Initial Report, Interim Report, PeerToPeer Protocol concept and initial design, Onion Auth Api + Tests, Final Report

6. Effort Spent

- Benedikt Brandner: Estimated 3 Weeks implementation time on the project. Conceptualization & research time excluded.
- Rohit Panda: ?