Owner: Gyanranjan Hazarika
UFID: 63889154

A Report on Modified Onion Routing and its Proof of Concept

Introduction:

This document briefly describes the architecture, code layout, operation principles and testing covered in the implementation of project Modified Onion Routing and its Proof of Concept. It also briefs on the purpose and the test results.

Background of Onion Routing

Onion Routing is an infrastructure for private communication over a public network. It provides anonymous connections that are strongly resistant to both eavesdropping and traffic analysis. It provides bi-directional anonymous connections. An onion is a data structure that carries relevant cryptographic information for each onion router in the path (in each layer of the onion). Onions themselves appear differently to each onion router as well as to network observers. The same goes for data carried over the connections they establish. An initiator and a responder set up a virtual circuit between themselves by use of an onion. The intermediate nodes of the path know only its immediate next hop neighbor by means a virtual circuit id. Nodes further encrypt multiplexed virtual circuits which make studying traffic patterns really cumbersome.

Purpose:

The purpose of this project is to identify a few vulnerabilities present in a conventional Onion Routing infrastructure and propose a few modifications to overcome these vulnerabilities. The identified vulnerabilities are:

1. Compromised Onion Proxy can nullify anonymity and privacy.
2. Similarly, compromised exit funnel at responder’s proxy can nullify anonymity and privacy.
3. An unsecured connection from responder’s proxy to the ultimate recipient keeps the data exposed to passive adversary.
4. No protection against tampering of data.
5. Finally, passive adversary can link sender and recipient’s data.

Proposed Modifications:

Following are the proposed modifications to overcome these vulnerabilities.

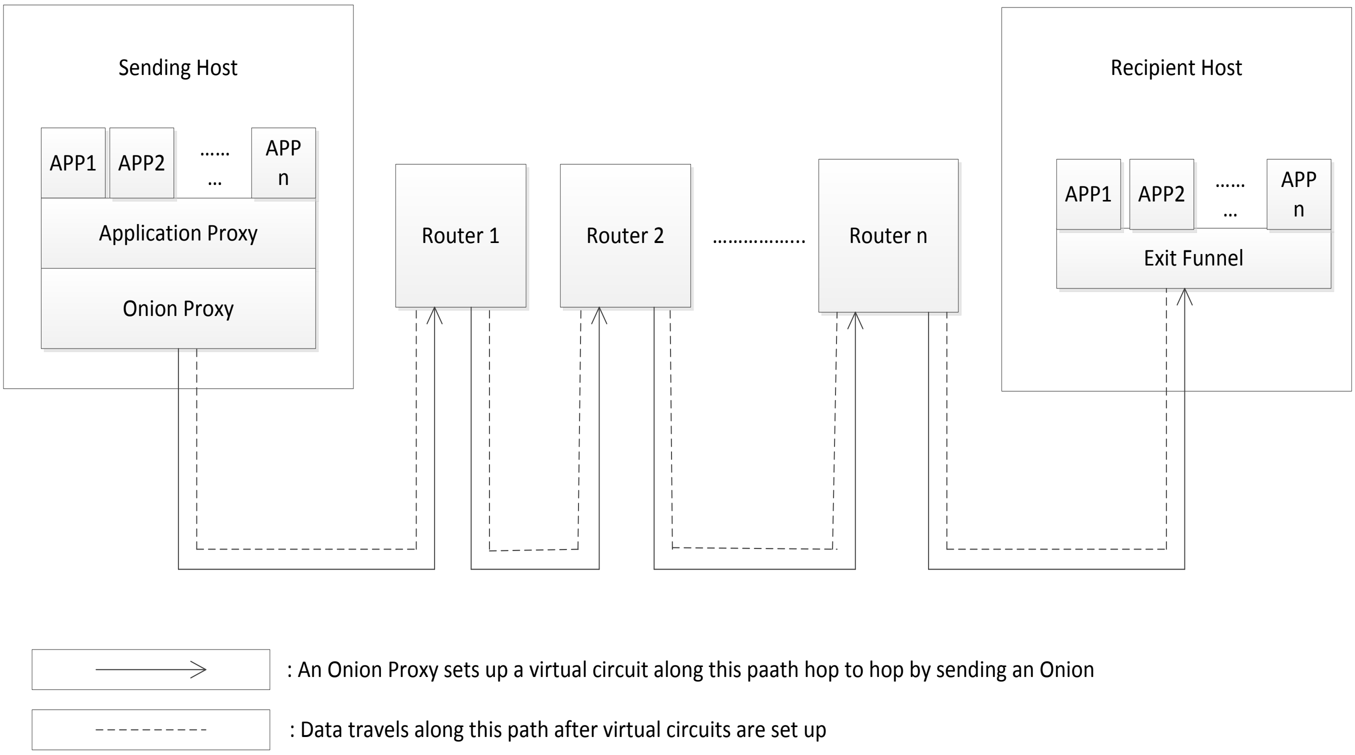
1. Run Onion Proxy at the sender’s host machine.
2. Run Exit Funnel (from responder’s proxy) at recipients’ host machine.
3. Add MAC to frame structure.
4. Introduce MIXes at the intermediate routers/nodes.

Proof of concept:

For providing proof of concept, software based on the following architecture is implemented. The final test result proves the feasibility of the above modifications.

Basic Architecture:

The base architecture of the software is shown in the following diagram. Multiple applications can register against a single application proxy in the sending side. Similarly, multiple applications can register against a single exit funnel in the receiving side. Onion Proxy and exit funnel are designed such a way that they can be decoupled from host and run in dedicated proxy servers. See blow in the relevant section.



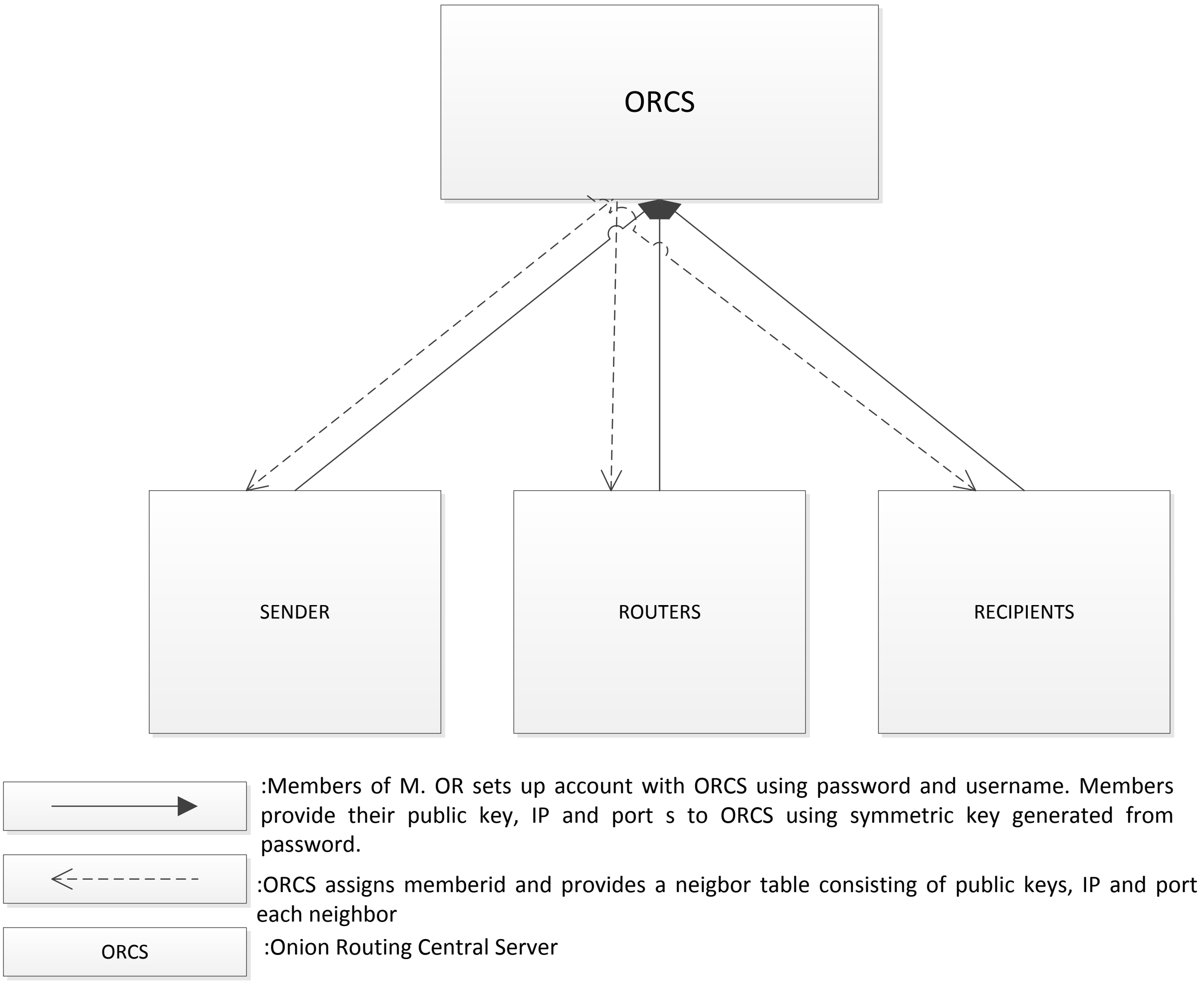
: An Onion Proxy sets up a virtual circuit along this paath hop to hop by sending an Onion

: Data travels along this path after virtual circuits are set up

Figure 1- Basic Architecture Diagram of M. OR

Role of ORCS in Key and Neighbor Information Distribution:

The role of ORCS is described in figure 2 and 3 below. ORCS will maintain a table of registered members. When a new member joins, it will set up an account with the ORCS and will provide its IP, port, device type (router or end device or both) and public key using the public key of the ORCS. Each member will further encrypt the data using the symmetric key generated from the account password. ORCS, in turn, will assign a member id to each new member and update the table it is maintaining. It will then provide this table, signed by its private key and encrypted using the symmetric key. Note that no reconfiguration is required



:Members of M. OR sets up account with ORCS using password and username. Members provide their public key, IP and port s to ORCS using symmetric key generated from password.

:ORCS assigns memberid and provides a neigbor table consisting of public keys, IP and port each neighbor

:Onion Routing Central Server

Figure 2 - Role played by central server in providing keys and neighbor information

as in case of crowds when a new member joins. The following MSC explains the procedure in more detail. How the members avail the public key of the ORCS is considered out of scope of this document.

MSC of ORCS Execution Flow

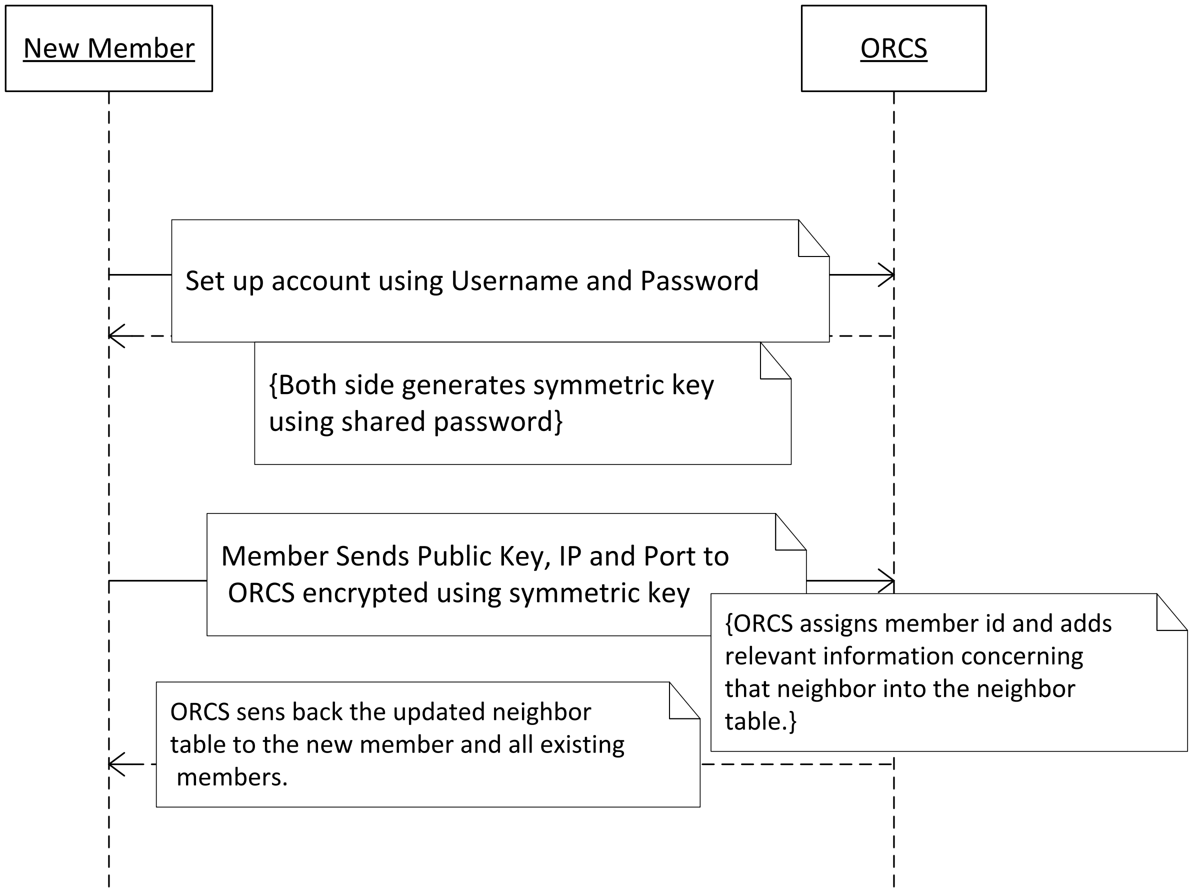

New Member
ORCS
Set up account using Username and Password
{Both side generates symmetric key using shared password}
Member Sends Public Key, IP and Port to ORCS encrypted using symmetric key
{ORCS assigns member id and adds relevant information concerning that neighbor into the neighbor table.}
ORCS sens back the updated neighbor table to the new member and all existing members.

Figure 3 - A new member joins

Flow among Application, Application Proxy and Onion Proxy

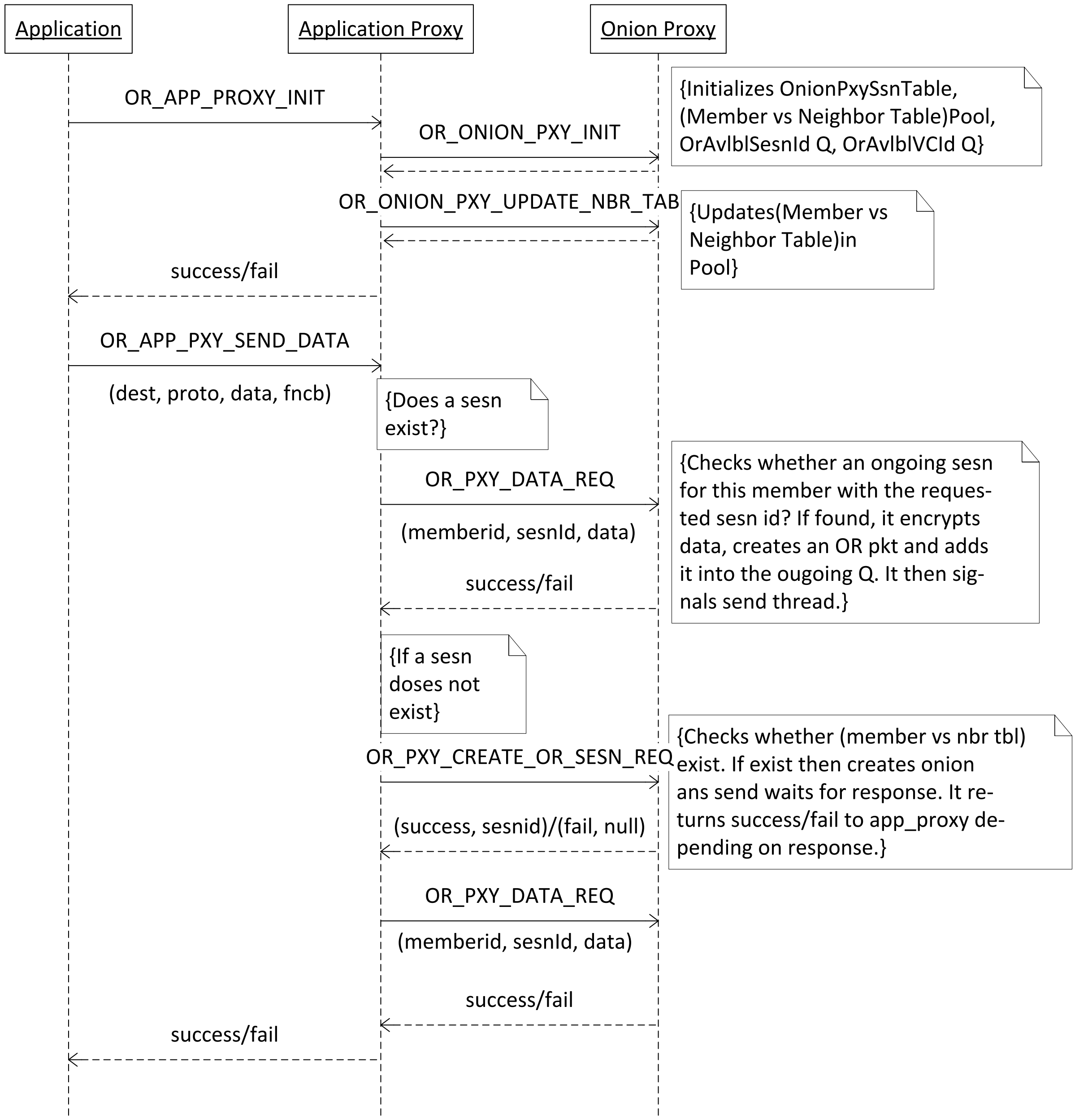

Application
Application Proxy
Onion Proxy
OR_APP_PROXY_INIT
OR_ONION_PXY_INIT
{Initializes OnionPxySsnTable, (Member vs Neighbor Table)Pool, OrAvlblSesnId Q, OrAvlblVCId Q}
OR_ONION_PXY_UPDATE_NBR_TAB
{Updates(Member vs Neighbor Table)in Pool}
success/fail
OR_APP_PXY_SEND_DATA
(dest, proto, data, fncb)
{Does a sesn exist?}
OR_PXY_DATA_REQ
(memberid, sesnId, data)
{Checks whether an ongoing sesn for this member with the requested sesn id? If found, it encrypts data, creates an OR pkt and adds it into the ougoing Q. It then signals send thread.}
success/fail
{If a sesn doses not exist}
OR_PXY_CREATE_OR_SESN_REQ
{Checks whether (member vs nbr tbl) exist. If exist then creates onion ans send waits for response. It returns success/fail to app_proxy depending on response.}
(success, sesnid)/(fail, null)
OR_PXY_DATA_REQ
(memberid, sesnId, data)
success/fail
success/fail

Figure 4 - Execution flow among App, App Proxy and Onion Proxy

Flow between Application/Receiver Proxy and Exit Funnel

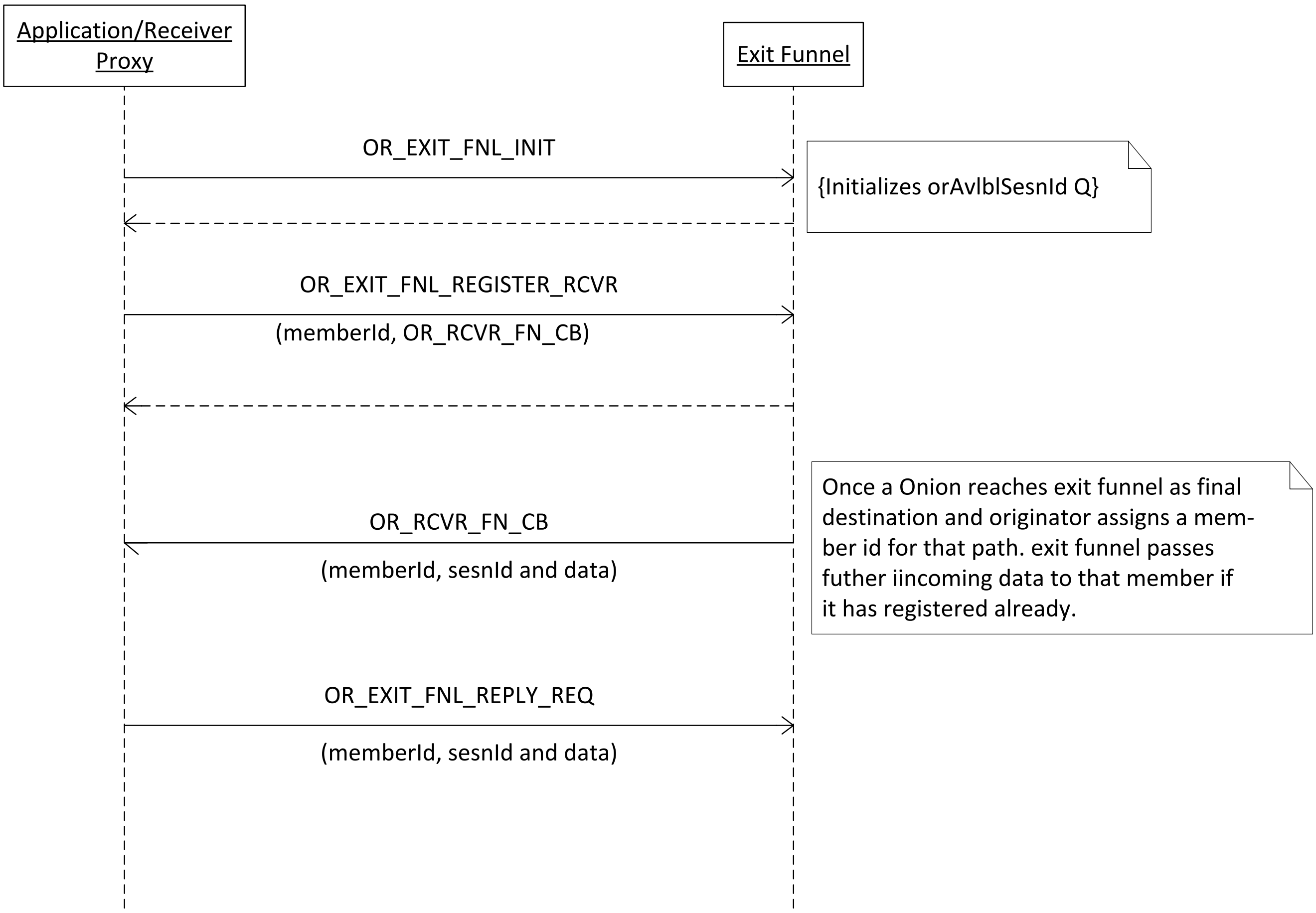

Application/Receiver Proxy
Exit Funnel
OR_EXIT_FNL_INIT
{Initializes orAvlblSesnId Q}
OR_EXIT_FNL_REGISTER_RCVR
(memberId, OR_RCVR_FN_CB)
OR_RCVR_FN_CB
(memberId, sesnId and data)
Once a Onion reaches exit funnel as final destination and originator assigns a mem-ber id for that path. exit funnel passes futher iincoming data to that member if it has registered already.
OR_EXIT_FNL_REPLY_REQ
(memberId, sesnId and data)

Figure 5: Execution Flow among Application/Receiver Proxy and Exit Funnel

Design Flexibility

The software design allows users to decouple Onion Proxy and Exit Funnel from the host machines to dedicated proxy servers.

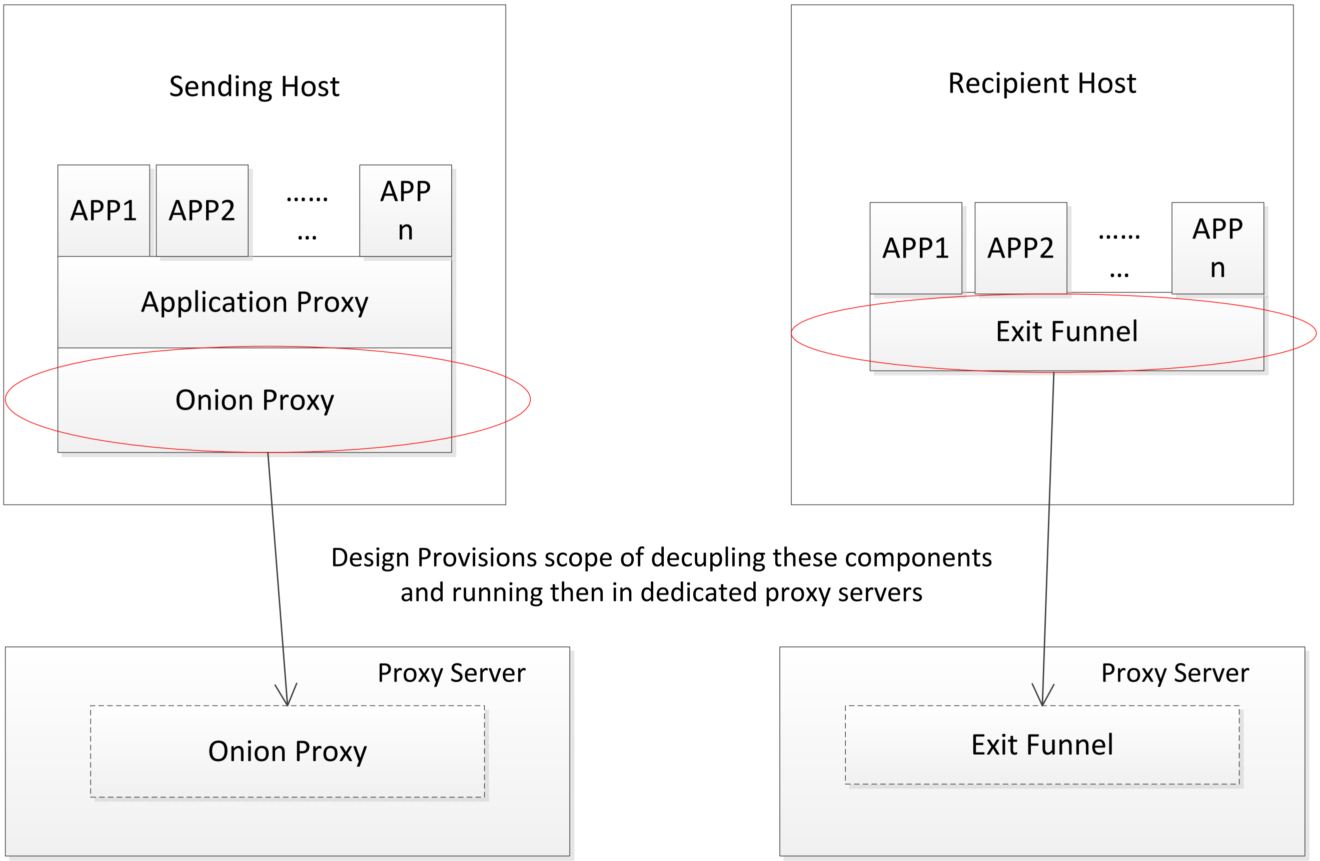


Figure 6 - Design Flexibility

Packet Structure

Each layer of an onion will be of the format shown below. In this version of the implementation, the format shown in figure 7 is used. Each member of the network generates a Diffie-Hellman shared secret using the public key of each such member which may occur in the path involving itself and the other member. In this version of the software, even the onion layer is encrypted using such a shared secret which comes with a flaw. An intermediate node can link the key to the sender of the onion. Therefore, this scheme of encryption can not be used. Instead, conventional PKCS as suggested by Goldschlag et al. is suggested. The proposed packet format for a layer of an onion is as shown in figure 8. An 8 byte MAC will be part of the onion layer.

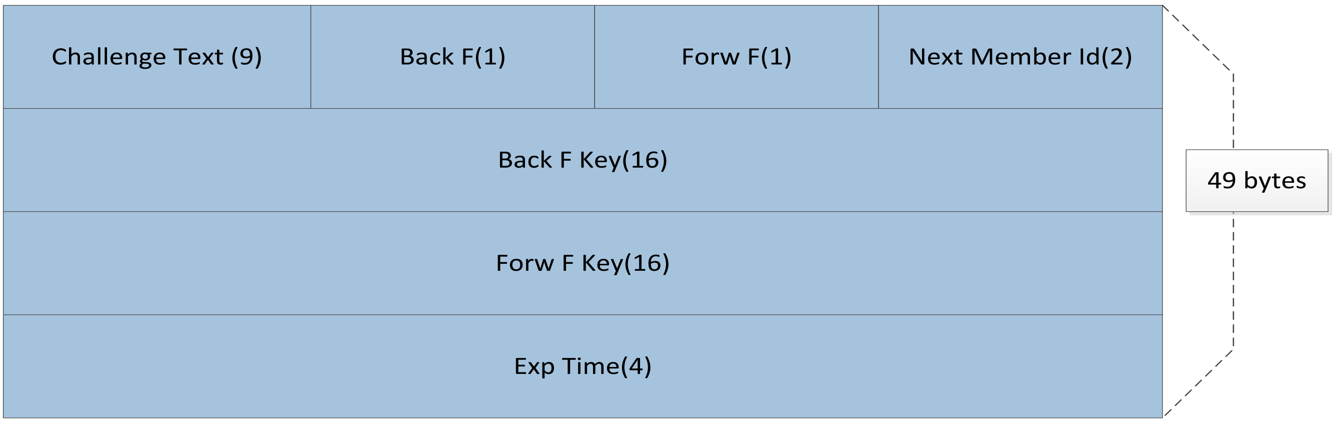


Figure 7 - Packet Format for a Single Onion Layer

Back F: Backward crypting function, Back F key: Key for backward cryption, Forw F: Forward crypting function, Forw F key: Key for forward cryption, Exp Time: Expiration time for the Onion, Next member id: Member id of the next hop in the path, MAC: Message Authentication Code

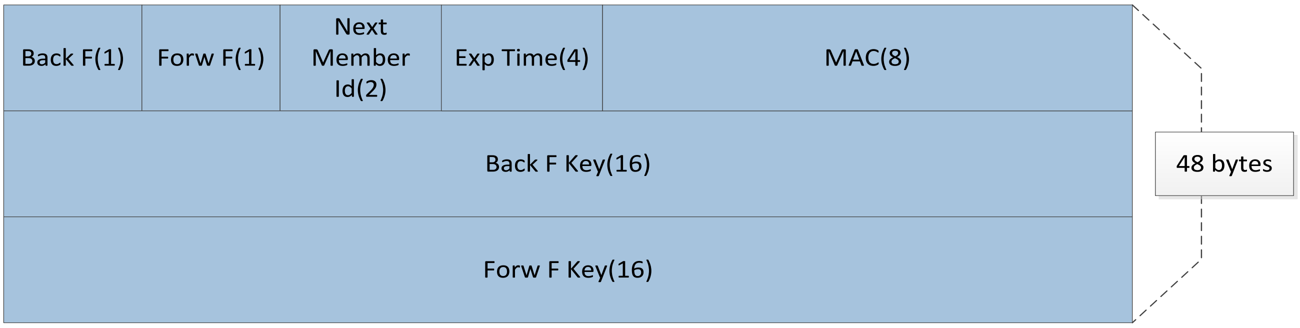


Figure 8 - Proposed Packet Format for a Single Onion Layer for Future Use

A typical packet in this version of the project is of the structure shown in figure 9. But it comes with a flaw. The MAC field is just a place holder. A single MAC can not be used in a OR network as the source does the encryption for each node in between. The challenge text is not desirable because of the same reason as mentioned above. Therefore a modified packet structure as shown in figure 10 is proposed for future use.

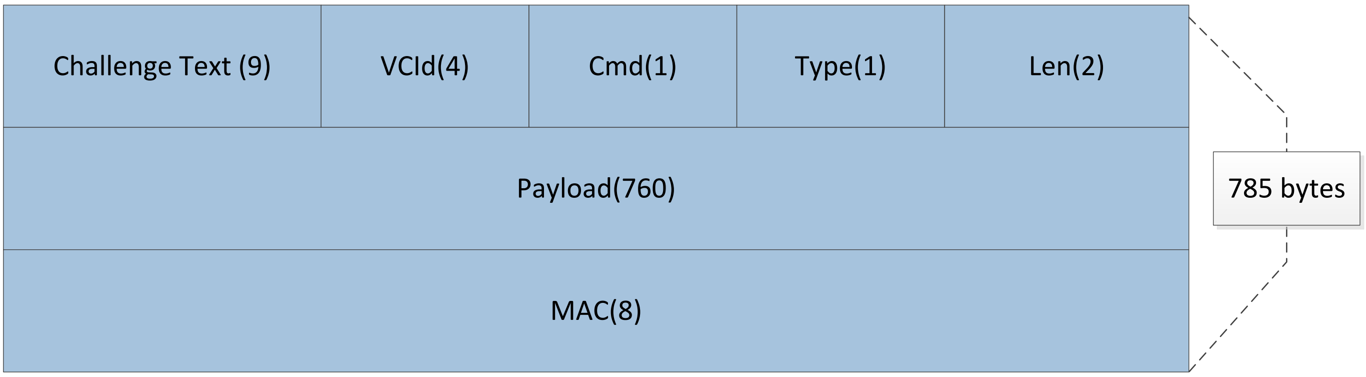


Figure 9 – Packet used in this implementation

The proposed packet structure has a MAC for each node in the route and is valid only for DATA commands. The MAC is generated using (forward function, key) or (backward function key, key) pair. The source will generate the MACs for each intermediate node and place it after the payload field in the order of the intermediate routers of the packet starting with the immediate next router. A router is going to use the first MAC after the payload field, shift the entire 1520 – (760 + 8 + 4 + 1+ 1 + 2) = 744 bytes to the left and add 8 bytes padding at the end.

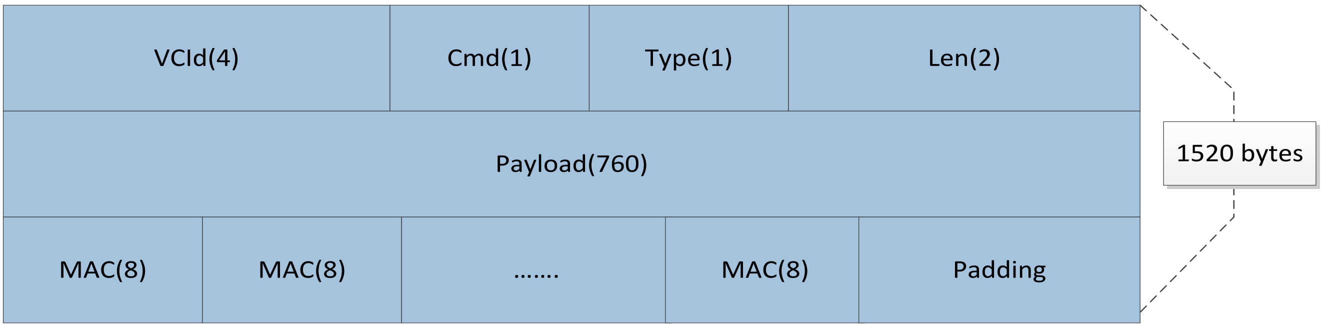


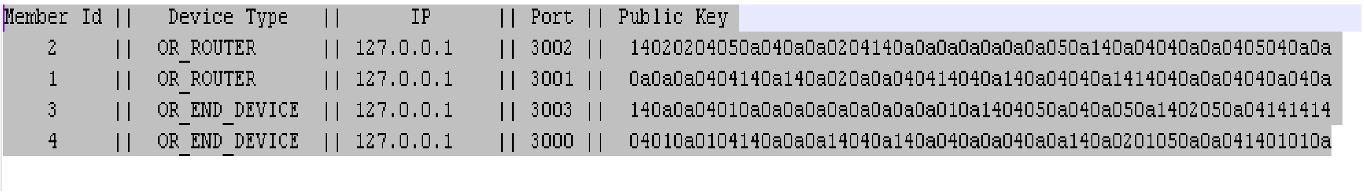
Figure 10 – Proposed Packet for Future Use

VCId: Virtual Circuit Id, Cmd: Command Identifier, Type: Traffic type indicating delay tolerant or not
Len: Payload length, MAC: Message Authentication Code, Padding: random data stream

Owner: Gyanranjan Hazarika
UFID: 63889154

Configuration File (at the absence of an ORCS)

The absence of ORCS generates a requirement of a configuration file from which each member can read memberId, IP, port and public key. The file content look like the following:



```
Member Id ||   Device Type    ||       IP       || Port || Public Key
    2     ||  OR_ROUTER       || 127.0.0.1      || 3002 ||  14020204050a040a0a0204140a0a0a0a0a0a050a140a04040a0a0405040a0a
    1     ||  OR_ROUTER       || 127.0.0.1      || 3001 ||  0a0a0a0404140a140a020a0a040414040a140a04040a1414040a0a04040a040a
    3     ||  OR_END_DEVICE   || 127.0.0.1      || 3003 ||  140a0a04010a0a0a0a0a0a0a0a010a1404050a040a050a1402050a04141414
    4     ||  OR_END_DEVICE   || 127.0.0.1      || 3000 ||  04010a0104140a0a0a14040a140a040a0a040a0a140a0201050a0a041401010a
```

Figure 11 - Configuration file content

Code Layout

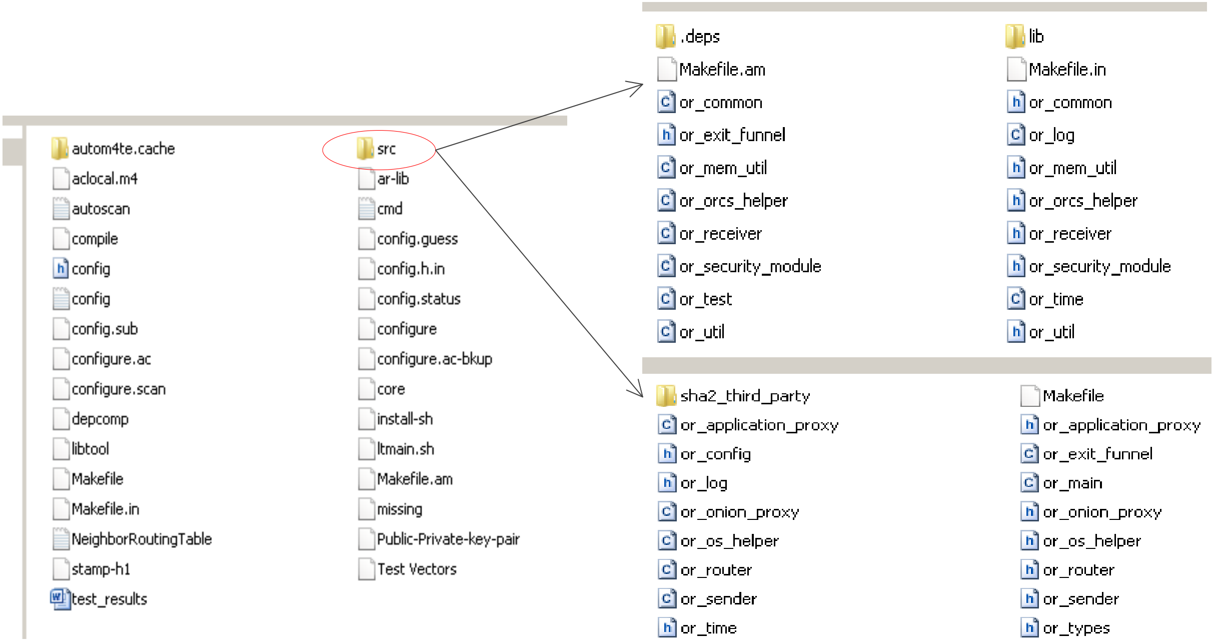


Figure 12 – Code Layout

Steps to Run

1. Go to project root.
2. Do make clean and make.
3. Open four terminals.
4. Start the program by choosing a role. The following commands are relevant when the configuration file shown above is used.
   src/m_onion_routing role=sender port=3000 logfile='DirPath'/or_snd_log.txt nwkLayoutFile='DirPath'/NeighborRoutingTable.txt

   src/m_onion_routing role=router port=3001 logfile='DirPath'/or_router1_log.txt nwkLayoutFile='DirPath'/NeighborRoutingTable.txt

   src/m_onion_routing role=router port=3002 logfile='DirPath'/or_router2_log.txt nwkLayoutFile='DirPath'/NeighborRoutingTable.txt *src/m_onion_routing role=receiver port=3003 logfile='DirPath'/or_rcvr_log.txt nwkLayoutFile='DirPath'/NeighborRoutingTable.txt*

5. The sender needs to be launched at the end as the connections are TCP based.
6. After launching the sender will send an onion, establish a VC based path and send the handshake message:
   "I am ur anonymous friend. Accept Greetings."
   The recipient responds by sending handshake response:
   "Hello anonymous friend. Greetings accepted."
7. After this sender process launces a command line app with the following options:
   Try Someting...
   a. Replay Onion
   b. Send a delay tolerant msg
   c. Send a delay intolerant msg
   d. Inject dummy/padding messages
8. User can try different options from the above list.

LOC and Test Result

Sl. No.	*File name*	*LOC*
1	or_main.c	909
2	or_time.c	162
3	or_util.c	541
4	or_log.c	121
5	or_common.c	765
6	or_os_helper.c	152
7	or_mem_util.c	54
8	or_test.c	129
9	or_sender.c	244
10	or_receiver.c	65
11	or_router.c	1134
12	or_application_proxy.c	331
13	or_onion_proxy.c	1409
14	or_exit_funnel.c	622
15	or_security_module.c	676
16	or_time.h	22
17	or_util.h	30
18	or_log.h	28

19	or_common.h	318
20	or_os_helper.h	11
21	or_mem_util.h	11
22	or_sender.h	9
23	or_receiver.h	11
24	or_router.h	33
25	or_application_proxy.h	16
26	or_onion_proxy.h	20
27	or_exit_funnel.h	44
28	or_security_module.h	27
29	or_config.h	35
30	or_types.h	358
	Total	**8287**
	Effective excluding comments and essential separating lines (not less than)	***7500***

Test Case 1

On an average how many dummy messages are required to inject for a message to reach to the destination through 2 (2-1) threshold pool mix.

Result:

No of sender's message	Threshold pool mix (n[pool]-s[threshold])	Average No. of Dummy messages	No of trials
1	(2-1)	2.3	10 (3, 1, 1, 2, 3, 1, 2, 2, 6, 2)
2	(2-1)	3	5 (6, 3, 2, 3, 1)
1	(3-1)	4	10 (2, 3, 3, 12, 5, 0, 9, 4, 1, 1)
2	(3-1)	5	5(8, 17, 2, 5, 3)
1	(3-2)	6.4	10 (3, 7, 1, 15, 1, 1, 11, 7, 3, 15)
2	(3-2)	7	5(6, 10, 6, 4, 9)

Test Case 2

Replay an Onion.

Result:

Onion rejected by first router.

Test Case 3

Send 50 delay intolerant messages to recipient.

Result:

All messages acknowledged by recipient.

Test Case 4

Send multiple back to back delay tolerant messages to recipient.

Result:

Result covered by results of Test Case 1.

********************END*********************