Breadcrumb

December 22, 2015

Contents

1 Aim

Breadcrumb is an application which allows users to send text based messages across the internet. The manner in which message transactions to occur will be such that:

- `User Discovery` - Chat initialization occurs in a secure manner, immune to MITM attack
- `End-to-end encrypted` - Message plaintext is only available to the sender and intended recipient
- `Forward secrecy` - Key compromise does not allow previous messages to be decrypted
- `Internal Metadata Leaking` - User metadata is hidden from the Breadcrumb infrastructure
- `External Metadata Leaking` - User metadata is hidden from an adversary capable of observing network traffic

2 Setup

2.1 OS Support

2.1.1 Client

- `GNU/Linux` - x86-64

2.1.2 Node

- `GNU/Linux` - x86-64
- `GNU/Linux` - armhf

2.2 Dependencies

- `OpenSSL` - https://www.openssl.org
- `miniupnpc` - http://miniupnp.tuxfamily.org/

2.3 Getting Started

1. `Install OpenSSL`
    - `Debian` - sudo apt-get install libssl-dev
    - `From source` - https://www.openssl.org/source/
2. `Install Miniupnpnc`

- `Debian` - sudo apt-get install libminiupnpc-dev
- `From source` - http://miniupnp.tuxfamily.org/files/

3. `Clone Breadcrumb Repository`
   (a) git clone https://github.com/nyquist-prompt/breadcrumb
4. `Building Client`
   (a) cd breadcrumb/client/
   (b) make
5. `Building Node`
   (a) cd breadcrumb/node/
   (b) make

2.4 Running a Client

2.4.1 Compilation Switches

The client compilation switches are located at the beginning of client.c and function as follows.

- `ENABLE_LOGGING` - Enable general logging
- `ENABLE_TRANSMIT_RECEIVE_LOGGING` - Enable specific logging of transmit and receive events
- `ENABLE_KEY_HISTORY_LOGGING` - Log all ephemeral breadcrumb keys
- `ENABLE_BANDWIDTH_LOGGING` - Log bandwidth to file (bandwidth.csv)
- `ENABLE_TOTAL_UID_LOGGING` - Log all ephemeral breadcrumb uids (User IDs) to file
- `ENABLE_UID_HISTORY_LOGGING` - Log all ephemeral breadcrumb uid history (User IDs) to file upon SIGINT
- `ENABLE_USER_INPUT_THREAD_LOGGING` - Enable specific logging of user input events
- `ENABLE_PACKET_DATA_LOGGING` - Log all packet (transmit and recieve) data
- `ENABLE_UID_GENERATION_LOGGING` - Log all generated ephemeral breadcrumb uids (User IDs)
- `LOG_TO_FILE_INSTEAD_OF_STDOUT` - Redirect all logging to file (client_<client name>.log)
- `LAN_NETWORKING_MODE` - Use LAN ip instead of WAN ip when NOT in test mode

- `TEST_MODE` - Listen on ETH interface and connect to hard coded relays specified in function 'setup_test_mode_conversation'
- `TEST_MODE_CLIENT_1` - Define user as CLIENT 1 when in test mode, this hardwires the entry node
- `TEST_RR_PACKET` - Force send a 'return route' type packet
- `TEST_UID_CLASH` - Generate UIDs within a small range in order to test potential uid clashes
- `USING_TEST_HARNESS` - Client starts immediately (as opposed to waiting for '/connect' command)

2.4.2 Starting a Client

The start a client enter: ./client USER_ID PORT

2.4.3 Client Commands

- `/connect USER_ID` - Initiate a chat with client with username USER_ID
- `/network` - Perform network test (not implemented yet)
- `/exit` - Exit Breadcrumb
- `/leave` - Leave current conversation
- `/help` - Display command help
- `/testrr` - Force sending of a 'return route' type packet

2.4.4 Test Mode

Currently two clients may successfully communicate via the Breadcrumb infrastructure provided:

1. Both clients have 'TEST_MODE' enabled
2. Only one client has 'TEST_MODE_CLIENT_1' enabled
3. Clients are connected to an ethernet switch
4. Nodes, as defined in 'setup_test_mode_conversation', are connected to the ethernet switch and executing
5. Clients issue the '/connect' command, correctly specifying the other clients USER_ID

2.5 Running a Node

2.5.1 Compilation Switches

- `ENABLE_LOGGING` - Enable general logging
- `ENABLE_LOG_ON_EXIT` - Log all relay data to file (relay_log.csv) upon exit
- `ENABLE_THREAD_LOGGING` - Log thread creationdestruction events
- `ENABLE_PACKET_LOGGING` - Log all packet data
- `ENABLE_UID_LOGGING` - Log all received uid (user ID) data
- `LOG_TO_FILE_INSTEAD_OF_STDOUT` - Redirect all logging to file (<relay name>.log)
- `SOFT_RESET_KEY_STORE` - Upon startup do NOT regenerate key store
- `ENABLE_BANDWIDTH_REPORTING_UI` - Client starts immediately (as opposed to waiting for '/connect' command)

2.5.2 Starting a Node

The start a node enter: ./node NODE_ID PORT [LOGGING INTERVAL]

2.5.3 Important

When starting for the first time, a Node will create a keystore, which (depending upon the amount of free harddrive space) can be quite large. Thus it is advised to only start a Node on the intended machine - don't start one on your local machine 'just for fun'.

3 Concepts

3.1 Client

A Client is a user who connects and performs message transactions with other users across the Breadcrumb infrastructure. Clients are realized by executing the Breadcrumb client application on a users local machine (PC or mobile device) which is connected to the Internet.

3.2 Node

A Node hosts 'breadcrumbs' (UID + Ephemeral key) as well as user messages. Nodes use breadcrumbs (which are implanted by clients) in order to perform packet (onion-style) routing. User messages are stored on the Node until it receives a matching 'return route' packet, which allows the Node to route the message to the intended recipient (see section 'Protocol' - 'Packet Routing'). Although Nodes perform only those two functions, with respect to a user conversation (which must use a finite set of Nodes) they perform different tasks:

- `Entry Node` - The only Node by which a client directly exchanges packets. All packets destined for other Nodes are onion-routed via this Node. The Entry Node must be unique for all participants of a conversation.
- `Server Node` - A Node common to all conversation participants which is used to host user messages.
- `Relay Node` - A Node which may be common amongst participants and only used to relay packets destined for other Nodes.

Nodes are realized by untrusted third parties, primarily hosted on low-power/cost mini PCs (e.g. Raspberry Pi), which are connected to the Internet. Third parties who run Nodes are re-imbursed for their bandwidth expenditure (See 'Spider' and 'Economy' sections).

3.3 Directory

A Directory allows users to create a Breadcrumb IM account and provides users who wish to chat with a 'Chat Cookie'. It also connects to Nodes to enable the user chat (by enabling the storage of messages under that Conversation ID defined by that 'Chat Cookie'). The 'Chat Cookie' defines which Nodes are used by a particular user conversation (e.g. Entry Node, Server Node and Relay Nodes) and are selected based on the current amount of clients using a particular Node with respect to that Nodes maximum capacity (which varies depending upon its specific hardware). Thus Directories ensure that the total traffic passing amongst the Node Network does not exceeded its capability and also that it is evenly spread. Directories are realized by trusted third parties.

3.4 Spider

A Spider is a single client which simply passes traffic throughout the Node Network in order to ascertain Node uptime. The uptime is recorded and used to correctly re-imburse Nodes for their bandwidth. Spiders also ensure a minimum amount of traffic is passing through the Node Network to ensure adequate packet mixing (which obscures packet metadata) is occuring.

4 Networking

4.1 Figures

The figures used in the following section are described as follows:

Figure 1: Directory

Figure 2: Node

Figure 3: Client

4.2 Client Registration

Clients register with a single Directory (Figure 4), encrypting the transfer via HTTPS as well as the Directories 2048-bit RSA key (which is packaged as part of the client application). The registration request is routed via TOR to ensure that the Directory cannot equate a request with an IP address. The Directory then accepts or rejects the registration, depending upon the existence of a client ID clash. Client ID and password is hashed (SHA-2) prior to transfer. If a registration is approved it is synchronized amongst the other Directories (Figure 5).

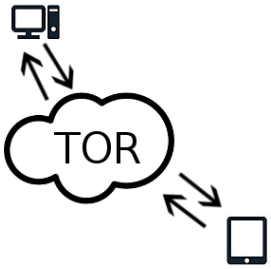

Figure 4: Client Registration

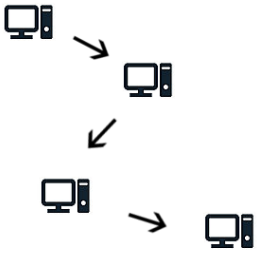

Figure 5: Registration Sync

4.3 Chat Initialization

Chat initialization occurs similarly to Client Registration. A Client that wishes to begin a conversation with another client first contacts the Directory server (via TOR), providing the hashed username of the other member of the conversation. The Directory server creates a 'Chat Cookie' and attempts to synchronize it amongst the other Directories. In the case that another Directory already has a cookie with the same conversation ID (i.e. the other member of the conversation initialized the chat first) the cookie with the more recent timestamp is

discarded and the older cookie is returned to the Client that made the request. The Directory then registers the conversation ID with the conversation Server Node as per Figure 6.

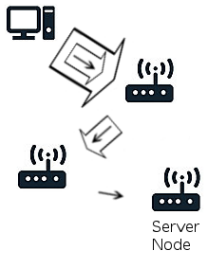

Figure 6: Enabling conversation in Server Node

4.4 Routing

In order to route packets via the Node Network, a Client must first implant 'breadcrumbs' (see section 'Protocol' - 'Breadcrumb'). As per the protocol, Clients may only inject packets into the Node Network via the Entry Node; implanting breadcrumbs is no exception. If a Client wishes to implant breadcrumbs with a non-Entry Node, it must onion route the implant packet via the Entry node as per Figure 7. In order to mask the metadata from an external observer the implant of breadcrumbs must be performed via an encrypted channel. In order to maximize performance, a Client uses DH Eliptical Curve to encrypt the breadcrumb implant packet. Once an initial set of breadcrumbs is implanted, subsequent packets that retrieve that breadcrumb will leave new breadcrumbs in their wake. Thus a breadcrumb is completely ephemeral - after it is retrieved it is discarded by the Node and replaced with the breadcrumb that is defined in the packet that triggered its retrieval.

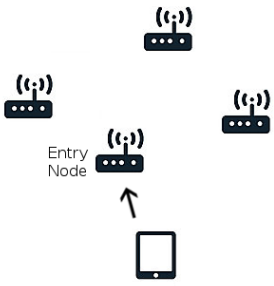

Figure 7: Implanting breadcrumbs with an Entry Node

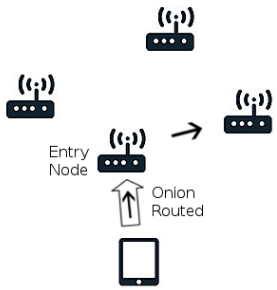

Figure 8: Implanting breadcrumbs with a non-Entry Node

4.5 Sending Messages

A client transmits a message by sending a Message Packet. A Message Packet consists of text that is to be transferred to the receiving client wrapped in an onion route made up of a random non-Entry Node as the first hop and the recieving clients' Entry Node as the second hop. The conversation ID and hash of the Node IDs that make up the route to the receiving client is appended to the packet. This allows the the Server Node to match the message packet to a particular conversation and pair it with a Return Route Packet (see below) which is provided by the receiving client. This is then wrapped in an onion route that consists of the clients Entry Node, a random non-Entry Node and finally the Server Node and transmitted (Figure 9).

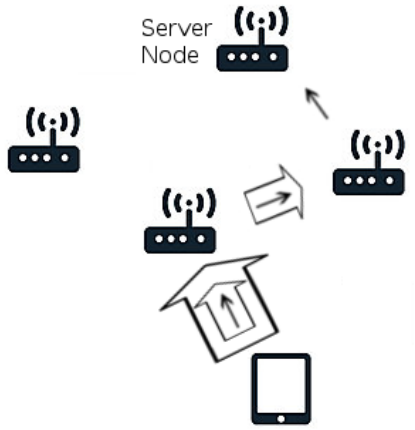

Figure 9: Routing a Message Packet to the Server Node

4.6 Receiving Messages

A return route packet defines an onion route that is capable of being paired to a message packet and the entirety is transmitted back to the client that provided it. A client that wishes to receive a message generates an onion route that consists of a random non-Entry Node as the first hop and its Entry Node as the second. The Node IDs of this route are hashed and appended to allow the Server Node to pair it with a message packet. This is then wrapped in an onion route that consists of the clients Entry Node, a random non-Entry Node and finally the Server Node and transmitted (Figure 10).

4.7 Constant Bandwidth

Dummy packets are primarily used by a client to ensure a constant bandwidth in the case that there are no message packets to create and all possible return

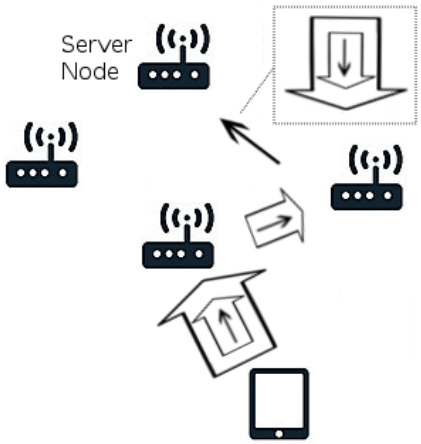

Figure 10: Routing a Return Route Packet to the Server Node

route packets have been defined and transmitted. They also serve to obscure the identity of the Server Node which in turn furthers the obscurity of the other conversation participant. Dummy packets may be singular in direction (Figure 11) and are discarded by the final Node defined in their onion route, or may define a return onion route and are eventually transmitted back to the client. In the latter case they serve to obscure the timing of conversation messages (as they are indistinguishable from the perspective of an external observer) and are also used by the Client to test the Node Network. For example, a Client can ascertain if a Node has gone offline through logical deduction based on the route it defined and whether or not it received a dummy packet.

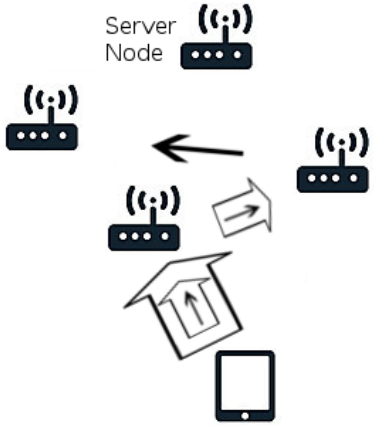

Figure 11: Routing a non-Return Route style Dummy Packet

5 Data Structures

5.1 Breadcrumbs

The datastructure that enables onion routing, a 'breadcrumb', is defined as such:

```
typedef struct breadcrumb
{
      uint8_t key[AES_KEY_SIZE_BYTES];
      uint32_t uid;
} breadcrumb;
```

Where AES_KEY_SIZE_BYTES = 16 (128-bit AES). The breadcrumb UID is used as an index for key storage and retrieval by a Node. UID clashes are currently resolved by using a linked-list at a particular UID index. For example, if a Node receives a packet with UID = X it first attempts to decrypt the current onion layer by applying AES key at index X. If that fails (determined via the veracity of the decrypted checksum value) the Node moves to the next key in the linked-list and again attempts to decrypt the packet. If the packet decryption is successful the key is removed and the new UID / key pair is added to the key storage (see 'Packet Structure' - 'Onion Route Data' section).

5.2 Packet Structure

The core datastructure of Breadcrumb is the packet structure (see Figure 12 - following page). The beginning of the 'Route Onion 1' section is a valid breadcrumb UID (that was registered previously), transmitted in plaintext, which is used by the Node to find the corresponding breadcrumb key and decrypt the rest of the 'Route Onion' section. The same applies to the 'Payload Onion' section. The specifics of the individual 'Onion Route' and 'Payload Onion' are also defined on the following page. Note the 'Next IP' and 'Next Port' fields are simply filled with random data in the case that the packet section is 'Payload Onion'.

```
typedef struct onion_route_data
{
      uint8_t iv[AES_KEY_SIZE_BYTES];
      uint32_t uid;
      uint32_t align_filler1;
      uint64_t align_filler2;
      onion_route_data_encrypted ord_enc;
} onion_route_data;

typedef struct onion_route_data_encrypted
{
      uint64_t next_pkg_ip;
      uint16_t next_pkg_port;
      uint16_t ord_checksum;
      uint32_t new_uid;                    // Next breadcrumb UID
      uint8_t new_key[AES_KEY_SIZE_BYTES]; // Next breadcrumb key
} onion_route_data_encrypted;
```

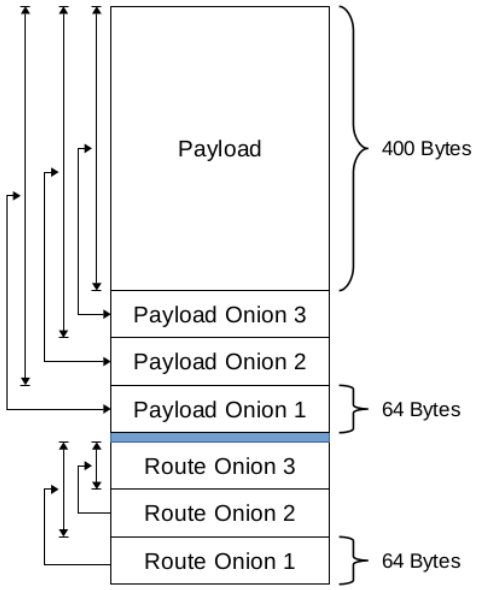

Figure 12: Packet Structure

5.3 Packet Types

There are four packet types, which are described in the following sections. The packet type, as defined in the packet metadata, is included in the 'Payload' of the Breadcrumb Packet. The generic structure of the payload is defined as follows:

```
typedef struct payload_data
{
      uint16_t type;
      uint16_t onion_r1;
      uint16_t onion_r2;
      uint16_t order;
      uint32_t client_id;
      uint32_t conversation_id;
      uint8_t payload[PAYLOAD_SIZE_BYTES];
} payload_data;
```

The fields of the packet metadata are as follows:

- `type` - Enumeration defining the packet type.
- `onion_r1` - If type is 'Dummy Packet /w Return Route' then this is the port for the transmitted packet. If type is 'Message Packet' then this is the XOR of the relay IDs (return route) and conversation ID. If type is 'Return Route' this is the XOR of the relay IDs (return route) and conversation ID of the first defined return route.
- `onion_r2` - If type is 'Return Route' this is the XOR of the relay IDs (return route) and conversation ID of the second defined return route. Otherwise it is unused (random data).
- `order` - If the type is 'Message Packet' this is the conversation packet order otherwise it is unused (random data).
- `client_id` - The client ID of the packet sender unless the packet is 'Dummy Packet /w Return Route' in which it is the most significant two bytes of the IP address for the transmitted packet.
- `conversation_id` - If the type is 'Message Packet' or 'Return Route' this is the conversation ID. If the packet is 'Dummy Packet /w Return Route' in which it is the least significant two bytes of the IP address for the transmitted packet.

5.3.1 Dummy Packet

A dummy packet is essentially the same as Figure 12 (previous pages), except the 'Payload' is simply random data.

5.3.2 Dummy Packet /w Return Route

A dummy packet /w return route is a packet that defines a return route within the 'Payload' section.

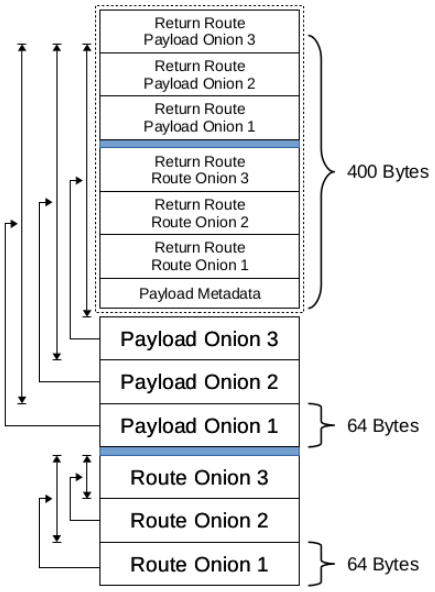

Figure 13: Dummy Packet /w Return Route

5.3.3 Message Packet

A message packet contains a message destined for the other conversation participant within its 'Payload' section. The message itself is wrapped in a onion return route that is comprised of a random non-entry node followed by the entry node (both with respect to the user receiving the message).

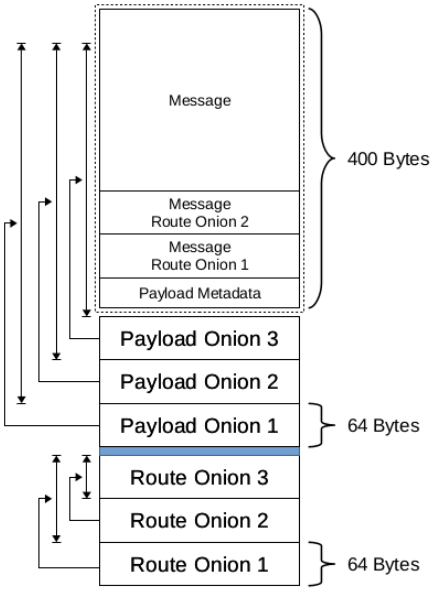

Figure 14: Message Packet

5.3.4 Return Route Packet

A return route packet contains two onion routes that are capable of being paired to a message packet (transmitted by the other conversation participant) to form a packet that may be transmitted back to the client that provided the return route packet. The packet metadata fields, onion_r1 and onion_r2, are the XOR of the relay IDs (and conversation ID) that make up the onion route that is defined. The fields, 'Return Route ID 1' and 'Return Route ID 2' are made up of random data which is cached by the Client that sends the Return Route Packet. In the event that a Client receives a Message Packet, it is these fields that are used to determine which return route the packet took. This is required in order to determine which breadcrumb (key / uid) pairs are now present in the return route Nodes.

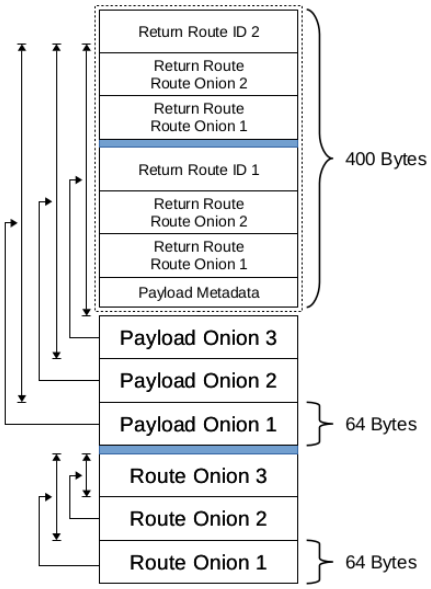

Figure 15: Return Route Packet

6 Processes

6.1 Enabling Routes

Directores enable routes by pushing cookie to Node (part of Chat Cookie). Nodes confirm packet was sent by Directory by checking against list of hard-coded key signatures.

6.2 Enabling Message Exchange

Directory enables message exchange by pushing cookie to Node in particular Message Slot. Nodes confirm packet was sent by Directory by checking against list of hard-coded key signatures.

6.3 Routing Packets

As described in section 'Packet Structure' after receiving a packet the Node decrypts the 'Route Onion' and 'Payload Onion' sections. After decryption, the 'Next IP' address of the 'Route Onion' section is compared with '0.0.0.0', if equal it is a 'Non-routing Packet', otherwise it is a 'Routing' packet. The packet is thus reconfigured as Figure 16, the filler being random data that is appended to preserve the constant packet size and then sent to the 'Next IP'.

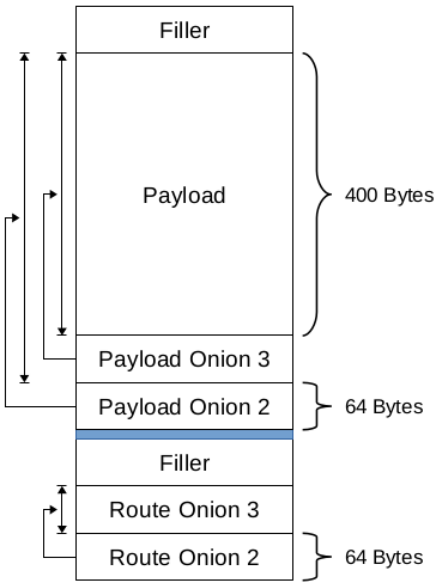

Figure 16: Post Routing Packet

6.4 Non-routing Packets

The manner in which non-routing packets are handled is dependent upon the packet-type, as

- `Dummy Packet` - Nodes simply discard Dummy Packets upon receiving them.
- `Dummy Packet /w Return Route` - Upon receiving a Dummy Packet /w Return Route a node discards the payload metadata and positions the return route data (onion and payload) at the beginning of the packet. The Node then pads the packet with random data such that it is the standard packet size and then transmits the packet.
- `Message Packet` - Upon receiving a Message Packet a Node will cache the packet if the message slot (determined by the conversation ID and user ID as set by the 'Message') has been enabled by a Directory and is currently empty, otherwise the Message Packet is discarded.
- `Return Route` - Upon receiving a Return Route packet a Node checks both 'onion IDs' against the onion ID of any messages in the message slot (determined by the conversation ID). If a match is found, the corresponding return route (first for onion_r1, second for onion_r2) is coupled with the message and the packet is transmitted. Otherwise the return route packet is simply discarded.

6.5 Forward Secrecy

Before message transaction occurs, clients will perform a DH key exchange and derive a session key. This session key will be used (as well as the Chat Cookie session key) as the root key of an Axolotl ratchet. [1, 2] Clients will identify DH key exchange packets by the return route that they occur upon - the 'Chat Cookie' will define a specific, unique return route path that will be used by clients to ONLY transmit DH packets.

7 Scalability

The maximum number of Breadcrumb users is entirely dependent upon the number of Nodes in the Node Network. As a client is only able to use the Node Network after successfully obtaining a 'Chat Cookie' from a Directory, Directories will 'throttle' the number of clients such that it remains within the maximum (and the maximum for each individual Node). Message latency will be only slightly affected by the number of clients using the Node Network as the set of Nodes that are used as part of a conversation is only a small set of the total Nodes, i.e. messages do not transverse all Nodes, only a small subset. The number of clients an individual Node can accommodate will be dependent upon:

- `Memory` - The amount of available RAM determines the size of the key store (i.e. number of breadcrumbs a Node can accommodate) and message store.
- `Performance` - The rate at which the client can perform cryptographic bottlenecks (DH Elliptical Curve).
- `Bandwidth` - The total bandwidth and RX/TX rate.

8 Economy

TODO (Nodes are reimbursed for uptime)

9 Attack Vectors

9.1 Malicious Nodes

As Nodes are run by Untrusted Third Parties they could potentially be malicious and logging/analyzing network traffic. Since traffic is end-to-end encrypted this type of attack could never result in the attacker recovering conversation plaintext, however it could result in the unmasking of conversation participants. However, in order to do so, the malicious actor would have to control the Server Node, as well as both Entry Nodes and one non-Entry node used by participant A and one non-Entry node used by participant B. If any of the aforementioned requirements are not fufilled, either one or both of the conversation participants are obscured. Both scenarios are detailed as follows (red represents malicious Nodes):

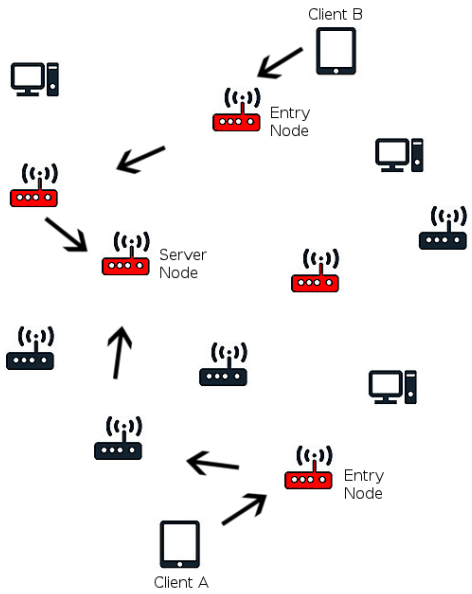

Figure 17: Uncompromised Conversation

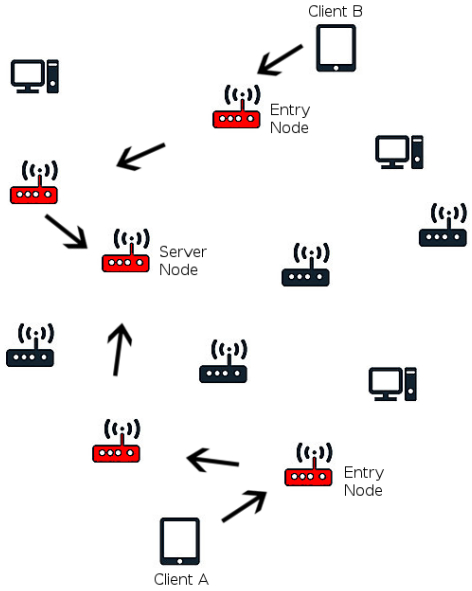

Figure 18: Compromised Conversation

9.2 DDOS

TODO (DDOSing the Node Network)

10 Future Work

Potential improvements include:

- `Pure Hardware Node` - Convert Nodes to be FPGA based (e.g. pure hardware). Improves security (no potential for operating system flaw exploit), performance and reduces cost, physical size and power consumption.

References

[1] Open Whisper Systems, *Advanced cryptographic ratcheting.* https://whispersystems.org/blog/advanced-ratcheting/, 26 Nov 2013.

[2] Axolotl, *Axolotl C Library.* https://github.com/WhisperSystems/libaxolotl-c, 23 Dec 2015.