I2NP Polling HTTP Transport

Revision 0.9, 28 August, 2003

http://www.InvisibleNet.net/ info@invisiblenet.net

jrandom@invisiblenet.net

Table of Contents

1.Transport Overview....2
- Goals....2
- Assumptions....2
- Components....2
- Transport Scope....2
- I2NP Integration....2
- Threat Model....3

2.Communication Security....3
3.Bridge Usage....4
4.Requests and Responses....4

1.Transport Overview

Goals

The goal of the Polling HTTP Transport is to allow bidirectional communication between applications, even if one or both of the parties are behind firewalls or even HTTP proxies, as long as there is a location on the Internet that they can both reach. In addition, the bridge component must be trivial to install and make available to other people on the Internet. To meet the needs of I2NP, this transport can protect both the integrity and the confidentiality of all messages passed through it.

Assumptions

The Polling HTTP Transport requires that clients are able to initiate HTTP/1.0 requests to arbitrary sites on the Internet.

Components

The transport is made up out of two components:

- **Clients**: the applications that want to communicate with each other are clients – these components send messages to other clients by bouncing the messages through bridges, and they also pick up messages destined for them from their own bridges.
- **Bridges**: the application serving as a way-point for clients to communicate, storing and forwarding messages, deterring abuse, and building its own network of bridges.

Transport Scope

Given a RouterAddress and a set of bytes to be delivered, the Polling HTTP Transport communicates with the appropriate bridge, establishes a private session with the remote address, delivers the data, and optionally waits until the data is confirmed received. The transport protocol is required to make sure that the data is passed to the bridge successfully, and that the client can know for certain that the recipient received the data fully and correctly.

I2NP Integration

I2NP requires certain pieces of information to be specified up front beyond the transport's API, and this data is included here:

Unique Identifier:

HTTP.Polling.1

Options:

url: full URL that messages should be sent to

clientId: identifier assigned by that bridge for the client (e.g. “a43fae32”)

expiration: date the address is no longer reachable, or “none” for never

(format is *yyyy/MM/dd.hh:mm:ss* in GMT)

Threat Model

The I2NP Polling HTTP Transport aims to ensure the confidentiality of messages sent between locations even if all data passed over the network is recorded. In addition, the transport will operate such that even if the bridge is controlled by an attacker, the attacker will not know the contents of the messages being sent, and if the client uses an anonymous HTTP proxy, the bridge will not even know where the client is located. Finally, the transport will not use a central bridge discovery point so that an attacker cannot take that out – instead it will allow bridges to maintain, discover, and verify their own links to other bridges.

2. Communication Security

To secure the data passed over HTTP between the client and the bridge, data in encrypted with 256bit AES in CBC mode based on a session key determined by a Diffie-Hellman key exchange. Since anonymizing proxies may be used to communicate with the bridge, HTTP sessions must be maintained through secure alternate means – by using random one time use session tags prefixed to the encrypted data.

In addition, the client to bridge to client communication is encrypted client to client so the bridge has no idea what is being said. This operates in a similar fashion – since clients already know other client's public keys, a client initially creates a set of session tags and an associated session key, encrypted to the receiving client's public key, and then delivers that encrypted package to the bridge. The bridge then passes that package on to the client who opens it up and associates a set of session tags with the enclosed session key. Actual inter-client messages are encrypted with that session key, prefixed with one of the unused session tags, delivered to the bridge where it is queued for delivery to the destination client at its next poll.

Session keys can be rotated whenever a new set of session tags are generated and delivered for a client or a new DH exchange takes place. In addition, new session tags can be attached to any of the messages.

Finally, to allow clients to know for certain whether messages are actually being delivered to the client instead of simply swallowed by the bridge, client to client messages include an acknowledgment code that, once received by the receiving client, it sends back to the bridge which then offers it to the sending client as proof of successful delivery.

3. Bridge Usage

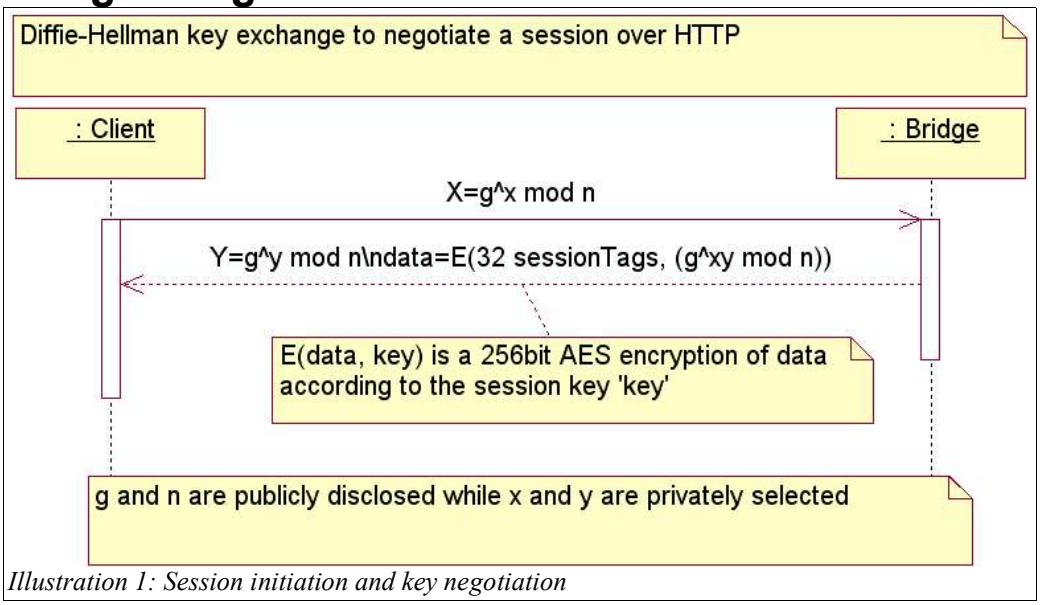

Illustration 1: Session initiation and key negotiation

The above message is the first message sent from the client to the bridge negotiating a private session key and a set of initial session tags (32 byte random numbers used to identify a request as being encrypted with the associated session key). This message, like all other requests in the Polling HTTP Transport Protocol, is sent via HTTP POST. (*g^x mod n*, *g^y mod n*, and *g^xy mod n* are the formulas for calculating the value – the calculated unsigned integers in network byte order are put in their place)

The content of all messages from the client to the bridge begins with one of the unused session tags, and the remainder of the HTTP POST is AES encrypted using the negotiated session key. The content encrypted in the POST begins with a 1 byte unsigned integer in network byte order specifying how many session tags follow, and then that many 32 byte random session tags are produced. After that comes a 1 byte unsigned integer in network byte order specifying the type of request being made of the bridge, and then the remainder of the HTTP POST is dedicated to the contents of that request specific data (all, again, encrypted with AES).

4. Requests and Responses

Within the encrypted HTTP data sent to the bridge is the type of request being made, which determines what goes into the rest of the request and what the bridge should respond with.

Request:	1 (Register client)
Request Data:	**hc=***hashcash(currentSessionTag, k bits)*
Response:	**status=ok&clientid=***id***&exp=***yyyy/MM/dd.hh:mm:ss***&passkey=***passkey* **status=badhashcash** **status=overloaded[&althost=***host***&altport=***portnum***]**

Notes:	*hashcash(data, n)* runs a hashcash calculation against the first n bits in data On overload, the server may optionally return an alternate http bridge *passkey* is the client's private pass key to poll for messages. *yyyy/MM/dd.hh:mm:ss* is the date and time (GMT) the client id expires

Request:	2 (Client Session Negotiate)
Request Data:	**clientid=***id***&hc=***hashcash(currentSessionTag, k bits)***&data=***data*
Response:	**status=ok** **status=badid** **status=badhashcash** **status=overloaded**
Notes:	*hashcash(data, n)* runs a hashcash calculation against the first n bits in data The contents of *data* is a 32 byte session key, followed by 7 32 byte session tags, all of which are encrypted by ElGamal against the receiving client's 2048bit public key.

Request:	3 (Send message)
Request Data:	**clientid=***id***&hc=***hashcash(currentSessionTag, k bits)***&hash=***h(data)* **&data=***data*
Response:	**status=ok&messageid=***id* **status=badid** **status=badhashcash** **status=badhash** **status=overloaded**
Notes:	*hashcash(data, n)* runs a hashcash calculation against the first n bits in data *h(data)* runs a SHA256 calculation against the data *data* begins with a session tag (as negotiated with a Client Session Negotiate message), with the remaining data being encrypted with the session key from the associated client session negotiation with AES256 in CBC mode. The contents of this data is a 32 byte authentication code, then a 1 byte unsigned integer specifying how many additional session tags follow, then that many 32 byte session tags, then the SHA256 of the payload of the message, followed by the actual payload.

Request:	4 (Poll for messages)
Request Data:	**clientid=***id***&passkey=***passkey*
Response:	*#negotiations (clientNegotiationData)* #messages* *(messageid sizeof(messageData) h(messageData) messageData)**

Notes:	The body of the response begins with a 1 byte unsigned integer specifying how many client negotiation structures follow, then those structures (as sent by previous clients via Client Session Negotiate messages), followed by a 1 byte unsigned integer specifying how many messages follow, then for each of those messages there is a 4 byte message id, followed by a 4 byte unsigned integer specifying the size of the message itself, then the SHA256 of the message, and finally the message in question. The message payload referred to is the data encoded and encrypted according to the associated Send Message messages, not the unencrypted and decoded payload.

Request:	5 (Acknowledge Receipt)
Request Data:	**clientid=***id***&passkey=***passkey***&messageid=***msgid***&code=***code*
Response:	**status=ok** **status=badmessageid**
Notes:	*code* is the 32 byte authentication code from the unencrypted payload sent to it in the associated message id.

Request:	6 (Verify Receipt)
Request Data:	**clientid=***id***&messageid=***msgid*
Response:	**status=pending** **status=***code*
Notes:	*code* is the 32 byte authentication code from the unencrypted payload sent to it in the associated message id, delivered only if the message has been acknowledged by the recipient.

Request:	7 (Register Bridge)
Request Data:	**host=***host***&port=***port*
Response:	**status=ok&(&host***n***=***host***&port***n***=***port*)*
Notes:	After submitting a host/port pair, the response may provide 0 or more host/port pairs (n=0..k).