Hiding Data in HTTP: A Network Steganography Project

Sean Donovan, Sathyanarayanan Gunasekaran, Sarthak Grover, Karim Habak, and Ben Jones

1. ABSTRACT

To prevent a censor from being able to trivially recognize a Tor data stream, the Pluggable Transport framework was established to allow developers to mask Tor data by masquerading as something innocuous.

HyperText Pluggable Transport (HTPT) is a SOCKS proxy that is designed to be the precursor of a Tor Pluggable Transport. HTPT is designed to mask a data stream by converting the data stream into a series of HTTP requests, responses, and image data. It is extensible in that further encoding schemes can be used to encode stream data into HTTP-related resources. For instance, a developer could create a new encoding for Scalable Vector Graphics (SVG) to hide data within the XML elements of the filetype. HTPT provides resistance to Deep Packet Inspection (DPI) by appearing to be an HTTP stream, which are generally not blocked by even the most repressive of governments, due to the innocuous nature of the majority of HTTP traffic.

2. INTRODUCTION

Internet censorship is a fact of life in many locations and circumvention tools are the only access some citizens have to unfiltered information. The censor restricts access to these circumvention utilities to maintain information control and a cat and mouse game is born as the circumvention tools try to evade detection. Tor is a useful tool for circumventing Internet censorship that encrypts data and provides an untraceable method for accessing data.

For this project, we have developed a SOCKS proxy that sits below Tor to provide a level of obfuscation such that the data stream is not an trivially detected Tor stream. A Pluggable Transport is an obfuscation layer between a Tor client and a Tor bridge to prevent detection of otherwise well known Tor data streams. Pluggable Transports consist of two parts: a client and a server. Typically these are slightly different obfuscate/de-obfuscate layers, however they can be very different depending on what the obfuscation mechanism is. This obfuscate/de-obfuscate layers operates below Tor and relays Tor traffic in a form that is difficult for DPI device to detect in a computationally cost effective manner. In some cases, the Pluggable Transport mechanism mimics or tunnels through an existing protocol[Houmansadr et al. 2013a; Mohajeri Moghaddam et al. 2012] and in other cases, the transport mechanism tries to appear random[Winter et al. 2013]. Implementing the obfuscate/de-obfuscate layers as a SOCKS proxy is a proof of concept, and allows for further development into a proper Pluggable Transport

For our proxy, the obfuscation layer will hide data within HTTP resources in HTTP requests and responses. More specifically, data will be encoded to look like images or more general HTTP traffic and either uploaded to the server or downloaded to the client. By allowing for both upload and download, we are not limiting the application of this pluggable transport to primarily unidirectional traffic. This can be used by not *just* Tor, but also by web browsers and any other SOCKS client.

3. DESIGN

HTPT is designed to be a SOCKS proxy. Since client and server HTTP traffic is different, the encoding in each direction is asymmetric. That is, the client-side and server-side encoding is different, and the decoding is similarly asymmetric such that the client can decode server

data and vice versa. In what follows, the threat model will be discussed, followed by the goals, HTPT's architecture, and finally the implemented encoding schemes.

3.1. Threat Model

HTPT's threat model includes the following:

(1) Lightweight Deep Packet Inspection (DPI) is used. The DPI device may look at the header and contents of packets, but may not reassemble streams or otherwise retain state.
(2) Stateful DPI is not used frequently. This is due to the cost of remembering state information about a particular connection or series of connections.
(3) HTTP traffic is not widely blocked. Specific sites may be blocked but through an alternate mechanism (e.g., DNS redirection).
(4) Connection to a generic web server for a long period of time may be considered unusual, but connecting to an image server for a long period of time is less unusual.
(5) Image files transferred over HTTP can be verified to be valid images. Manual checking of image data will not happen due to cost.

3.2. Design Goals

With the threat model in mind, the main goal is to increase the cost for a censor to discover the underlying Tor stream. For this reason, the Tor bridge will mimic an image gallery. Authenticated users will be allowed to send Tor traffic through the web server, while non-authenticated users will be returned a simple image gallery website. This means that the website must be able to correctly identify Tor and non-Tor traffic, and direct the traffic to the correct handlers.

An additional goal is to provide plausible traffic going from the client to the server and vice versa. This effectively means that multiple encoding schemes must be created, as client traffic looks significantly different from server traffic in HTTP.

In many situations, users will want to use HTTP to upload large amounts of data. However, HTTP and our transport are client-server protocols, which means clients typically send small requests to the server and the server returns large responses. HTPT mimics this structure, but also allows the user to achieve significant upload throughput. By structuring the upload traffic to look like images being uploaded to an image gallery, the user can achieve reasonable upload throughput while still maintaining an innocuous connection.

Preventing obvious fingerprintable behaviors is also a desired goal. This means that the server should not always send 128KB JPEGs, but rather it should vary the size and data type regularly.

Reasonable performance, as measured by both 'goodput' and latency, is a desired goal. Latency is most important for users of interactive websites, while reasonable goodput is required for uploading and downloading bulk data.

3.3. Motivations

Though it seems odd to add yet another layer to an already tall stack, HTPT is similar to other protocols like IPSEC. It seems like the additional layer would introduce excessive overhead, but the purpose of this layer is to provide security first and performance second. This layer can be removed or inserted as necessary and is not required to exist. Despite the focus of this protocol on security, the size of images allow large amounts of data to be sent with limited headers. The limited overhead from headers should allow HTPT to achieve high throughput.

[Houmansadr et al. 2013a] discusses the need to use the applications themselves and not just the protocol to obfuscate traffic. Otherwise adversaries can easily identify obfuscated traffic by finding differences in implementations of the protocols by the application and

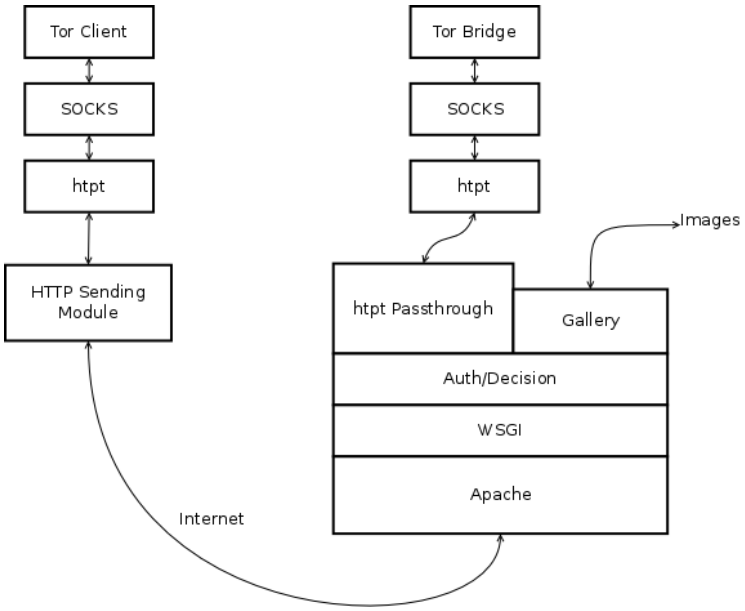

Fig. 1: Overall Architecture

by the obfuscating layer. For example, Apache may implement a particular part of HTTP different from specification, while the obfuscating layer may not. This would make it susceptible to fingerprinting. As such, using actual applications for generating HTTP requests and responses will be necessary.

3.4. Architecture

The overall architecture is represented in Figure 1. It is a basic client-server model, but with several important additional layers.

On the client-side, there is a SOCKS interface that HTPT presents to its user, typically Tor as is represented here. HTPT is then connected to an HTTP sending module that interacts with the server-side.

The server-side from the top down is very similar: Tor connects via SOCKS with HTPT. At this point, the design diverges significantly. For this, looking from the bottom up is useful. The client uses Headless Webkit, which is the Webkit browser engine that has been modified to not need user interaction to properly simulate web traffic and handle errors. The client connects to Apache, which is running the Web Server Gateway Interface module. This is a specification to allow web servers to communicate with web applications [Developers 2013]. This is used to interface with the Authorization/Decision module that is part of the HTPT package. It is used to decide whether to send the traffic to the Image Gallery (for non-authorized traffic), or to the HTPT Passthrough (for authorized traffic). This is what it sounds: a simple passthrough. Traffic is then handed off the HTPT.

The reverse is also true: traffic from the Tor Bridge to the Tor client goes through the same layers, just in reverse.

Within HTPT, there are four main modules, as can be seen in Figure 2. The SOCKS interface is the primary interface for communicating with Tor. This can also be used for other clients in a similar fashion, but in a much more limited manner.

The Encoding Decision module is the main location where anti-fingerprinting measures are taken. This module varies the encoding scheme and encoding amount such that the same style of data is not seen consistently. This is done by using a timer to buffer data up

to a random amount of data to be sent. If that random amount is seen before the timer expires, it is sent, and a new random time and amount of data is chosen. This should make inter-arrival times more random than Tor normally is.

Which encoding scheme is chosen depends on whether HTPT is acting in the client or server mode, and the data rate. This is measured based on the pervious transmission. That is, if 1000 bytes was received in the previous 10 milliseconds, the data rate can be calculated as being 100,000 bytes per second. This assumes that the data rate stays constant, which is why the time and buffer sizes are used in conjunction with each other.

This also acts as a framing module. Since each send operation opens a transient connection, reordering of data is vital. To handle this, sequence numbers are assigned by the Encoding Decision module. It is responsible for reorder of received data, for similar reasons.

The Send/Receive module is an interface module. It is different for both the client and the server, as the lower interface is different. The client-side Send/Receive module will interface with an HTTP Sending Module, while the server-side will interface with the HTPT Passthrough.

Between the Send/Receive module and the Encoding Decision module lies the encoding modules themselves. They have a standard interface and are designed in such a way that new encoding modules can be added. For instance, if a developer wanted to create CSS encoder that generated CSS that encodes data, it could be added to the existing encoders.

Of note: this is a pull protocol. The client can push data to the server, but the server must be polled for more data. This is due to the nature of HTTP, and such mechanisms for polling for more data already exist: see how AJAX systems work.

3.5. Encoding Schemes

There are currently six developed encoding modules. Each takes in a common header, optionally appends its own header, and a quantity of data. This is then encoded and sent out. The header is retrieved from the Encoding Decision module, such that consistent sequence numbers can be retrieved. Local headers can consist of padding information, if the particular item sends fixed-width data chunks.

(1) *Advertising URL Encoder* — This encodes 39 bytes of data (plus 1 byte for a header telling the length of the data) at a time by mimicking an advertising header. These frequently have 80-character hex-encoded identifiers contained within them, and are useful for encoding small amounts of data from the client to the server.
(2) *Google Search String Encoder* — This encodes variable, albeit small, amounts of data. Initial implementation uses 16 common English words to represent 4-bits of data, effec-

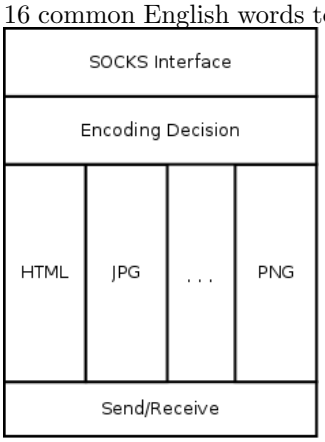

Fig. 2: Modules within HTPT

tively acting as a hex encoder. These are then places in what looks like a Google search string that can be decoded on the receiving end.
(3) *Baidu Search String Encoder* — This is a similar scheme as the Google Search String Encoder, except that it is designed to create Baidu-like search strings.
(4) *Cookie Encoder Encoder* — When data is encoded with one of 3 URL encoding schemes listed above, the URL can only be used to store 40 bytes of data. Every byte sent to the URL encoder beyond 40 bytes will be encoded as a series of cookies. Essentially, the data will be split into a key and a value, then compressed with Base64 encoding. When a GET request is sent to the server, each cookie is added as a field in the HTTP header.
(5) *BMP Encoder* — This is the most basic image encoder. A valid bitmap header is created, then the raw data makes up the image data.
(6) *JPEG Encoder* — By using lossless JPEGs, data can be sent across without loss. This uses the BMP Encoder, and converts it into a lossless JPEG to be sent by either the client or server.
(7) *PNG Encoder* — This is a similar scheme as the JPEG Encoder, with PNGs as output.

3.6. Framing and Performance Implications

As noted above, the encoding is an encapsulation scheme. The data from Tor, or any other SOCKS enabled client, is treated as a byte-stream, much like data that is sent via TCP. A 4-byte header is attached on each segment sent. The segment is then encoded according to the selected scheme which is where the overhead comes into play. As development completes, the performance of the protocol will be evaluated experimentally.

4. TESTING

Testing is an ongoing process. During development unit tests are used, and after development, performance tests are used.

4.1. Unit Tests

Each individual module is being tested during development. These tests are tailored to the type of module being tested

For instance, each encoding module is connected to itself to verify the what is encoded can be decoded correctly, without loss. This prevents accidental corruption of data by the encoding modules, which would cause the Tor connection to be taken down.

Integration tests between modules also occur, to verify proper functionality. For instance, testing of the image gallery application along with Apache is necessary independent of HTPT development.

4.2. Performance Characterization

As one of the goals is reasonable performance, various performance assesments are necessary. Since the goal was about relative performance, three iterations for each test is used: one with just a browser, one with the browser and Tor, and one with the browser, Tor, and HTPT. This will give relative performance for comparison.

The following test configurations are being used for performance characterizations:

(1) Loading twenty popular web pages — This will be a demonstration of latency. By finding the total time it takes to perform this operation, additional latency can be seen. This test will not work directly with the browser and HTPT, so it is not being performed directly.
(2) Downloading 1 gigabyte file over HTTP — This is an example of a bulk download that will allow measurement of goodput.

(3) Uploading 1 gigabyte file over HTTP — Since this is an assymetric encoding scheme, upload goodput testing is required.

5. PROGRESS

In general, steady progress is being made. Most of the lower level modules are written and tested, and there is significant progress on the modules that are not finished yet. Integration of all the modules, along with the associated testing, will be challenging, but certainly doable in the few weeks that are left.

Below is a checklist from a high level that shows the current progress. All coding activities are at the top, followed by integration and testing. Further details follow the checklist.

- ☐ Framing of data
- ☐ Reassembly of stream (part of framing)
- ☑ Generation of headers
- ☑ Sequencing
- ☑ Encoding as URL
- ☑ Encoding as Cookie
- ☑ Encoding as BMP
- ☑ Encoding as JPG
- ☑ Encoding as PNG
- ☑ Image gallery website pieces
- ☐ Integrating with Apache
- ☐ Integrating with Headless Webkit
- ☐ Unit testing of framing
- ☐ Unit testing of reassembly
- ☑ Unit testing of headers
- ☑ Unit testing of sequencing
- ☑ Unit testing of encoding schemes
- ☐ End-to-end testing with regular browser with packet capture
- ☐ End-to-end testing with Tor
- ☐ Performance testing design
- ☐ Performance testing

How the framing is going to work is being discussed and overall decisions have been made. There are some subtleties that need to be worked out prior to integration.

Connection establishment needs to be finalized. There is discussion of details related to authentication and keeping track of multiple connections ongoing, but nearly finalized. Actual integration on the client-side and server-side needs to be done as well.

Since additional encoding schemes can be added easily by writing a new encoder, further schemes are being considered. SVG in particular is being looked at as something that could be added.

Overall tests need to be created as well as performed. This will provide performance information, to characterize the overhead introduced by this layer within the stack.

6. FUTURE WORK

There are significant possibilities for expanding this work. Integrating this into the obfsproxy framework and bundling with the Tor Browser Bundle would be good to do in the future.

Another further enhancement is to enhance authentication. Current authentication is basic HTTP's basic access authentication. This is very simplistic, and any information is sent in plaintext. Clearly, that is a security problem, so a more secure authentication scheme is justified.

REFERENCES

Tor Developers. 2002a. Blocking resistant protocol testing framework. https://gitweb.torproject.org/user/blanu/blockingtest. git. (2002).

Tor Developers. 2002b. Censorship Environment Simulator. https://github.com/TheTorProject/EvilGenius. (2002).

Tor Developers. 2002c. Tor Performance Measurement Tool. https://github.com/gsathya/torperf2. (2002).

Tor Developers. 2002d. Tor: Pluggable Transports. https://www.torproject.org/docs/pluggabletransports.html.en. (2002).

Tor Developers. 2002e. Tor Pluggable Transports. https://www.torproject.org/projects/obfsproxy.html.en. (2002).

WSGI Developers. 2013. What is WSGI? http://wsgi.readthedocs.org/en/latest/what.html. (2013).

Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David R Karger. 2002. Infranet: Circumventing Web Censorship and Surveillance.. In *USENIX Security Symposium*. 247–262.

David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. 2012. Evading censorship with browser-based proxies. In *Privacy Enhancing Technologies*. Springer, 239–258.

Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. 2013a. The Parrot is Dead: Observing unobservable network communications. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*.

Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. 2013b. I want my voice to be heard: IP over Voice-over-IP for unobservable censorship circumvention. In *Network and Distributed System Security Symposium (NDSS)*.

Amir Houmansadr, Wenxuan Zhou, Matthew Caesar, and Nikita Borisov. 2012. SWEET: Serving the Web by Exploiting Email Tunnels. *arXiv preprint arXiv:1211.3191* (2012).

Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. 2013. Towards efficient traffic-analysis resistant anonymity networks. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 303–314.

Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. 2012. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 97–108.

Philipp Winter. 2013. Towards a Censorship Analyser for Tor. (2013).

Philipp Winter, Tobias Pulls, and Juergen Fuss. 2013. ScrambleSuit: A Polymorph Network Protocol to Circumvent Censorship. *arXiv preprint arXiv:1305.3199* (2013).