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 socket 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.

Unfortunately, Tor is detectable. The stream is fingerprintable. Tor developers came up with the Pluggable Transport framework[Developers 2002d]. 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 socket proxy is a proof of concept, and allows for further development into a proper Pluggable Transport

Based on this, we decided that making a obfuscation layer that can be used by Tor along with many other application. We have developed a socket proxy that sits below user applications to provide a level of obfuscation such that the data stream is not an trivially detected as the original stream.

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.

3. DESIGN

HTPT is designed to be a socket 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.

Additionally, since HTTP is a pull-based protocol, HTPT necessarily is pull-based. Clients must poll the server in order to get data.

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 stream. For this reason, the server will mimic an image gallery. Authenticated users will be allowed to send 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 HTPT and non-HTPT 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.

Reasonable performance, as measured by both throughput and latency, is a desired goal. Latency is most important for users of interactive websites, while reasonable throughput 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 by the obfuscating layer. For example, Apache may implement a particular part of HTTP

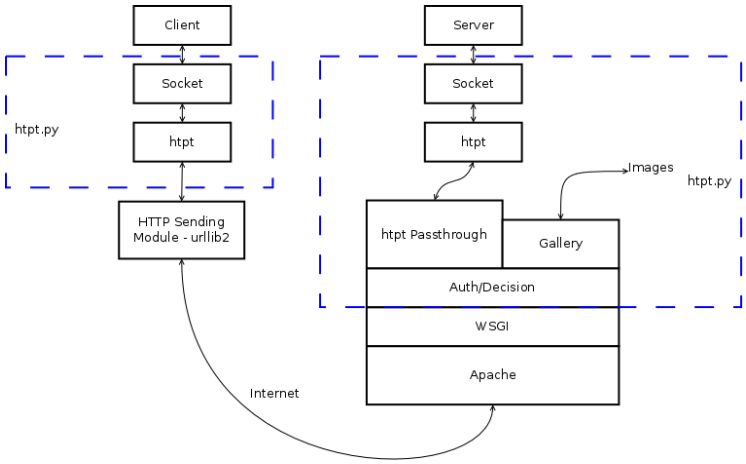

Fig. 1: Overall Architecture

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 socket 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 socket with HTPT. At this point, the design diverges significantly. For this, looking from the bottom up is useful. The client uses urllib2, which is the Python framework for downloading web data, not dissimilar from wget.

The client connects to Apache, which is running the Web Server Gateway Interface (WSGI) 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. Note that Apache can be changed for other web server software, so long as it support WSGI.

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

Within HTPT, there are four main modules, as can be seen in Figure 2. The socket 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. Right now, this is very simplistic, encoding as a URL if a client, as an PNG if a server.

Since each send operation opens a transient TCP connection, reordering of data is vital. To handle this, sequence numbers are assigned by theFraming module. It is responsible for reordering 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 Framing 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 a CSS encoder that generats CSS that encodes data, it could be added to the existing encodering modules.

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 a 4-byte HTPT header, a 1-byte length field, and 35 bytes of 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. This is padded out, if there is less than 35 bytes of actual data with random data.
(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, effectively acting as a hex encoder. These are then placed 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 Base64 encoded. 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 nature, JPEG is a lossy format. In order to avoid loss, Lossless JPEG is used. 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.

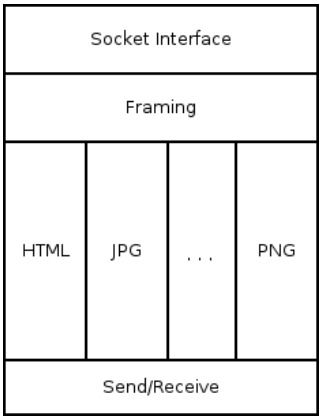

Fig. 2: Modules within HTPT

This can easily be expanded to use other types of encoding. For instance, encoding as a Animated GIF would be possible. Virtually any MIME-type could be used, but the only caveat is that it should be a *valid* version of the filetype. If the files are not valid, this is particularly fingerprintable. An adversary could look for IP addresses that send out invalid Animated GIF and block them as they make be using HTPT.

3.6. Framing and Performance Implications

As noted above, the encoding is an encapsulation scheme. The data from the 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.

4. TESTING AND EVALUATION

Testing was 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

This section will go over the performance evaluation, mostly from a theoretical point of view. There will be discussion about specific areas where performance suffers, and areas that are 'tunable'.

4.2.1. Overhead. This is the second most significant, but not necessarily controllable, factor in performance. Our testing focused on two different encoding schemes, namely the URL encoding scheme and the PNG encoding scheme.

For the *URL encoding* scheme, up to 35 bytes of data is transmitted in each URL. The overhead that exists is fairly consistent, with two factors that can vary the overhead size. First, there are the differences between IPv4 and IPv6 headers. For our testing, we only considered IPv4, as this is still the most common version of IP that is used. Second, there is the length of the website that is hosting the HTPT server. In our testing, the string that existed was "localhost:80", or 12 bytes long. This may not be reasonable in other instances, where the web address may be significantly longer. This does not preclude the usage of an IP address as the server's address, which will be, at most, 15 characters.

In our testing, 265 bytes were transmitted, which encoded 35 bytes of useful data. This is a 13% valid data vs. total data ratio. Clearly, this is not ideal, but it is not suspicious. This can be further improved by using cookies, as the valid data to total data ratio of the cookie itself will be significantly higher, approaching 75% using base64 encoding.

For the *PNG encoding* scheme, significantly more data is transmitted at once, resulting in very low overhead. Unfortunately, this will be less consistent than the URL encoding scheme for a number of reasons. PNG encoding may compress the data, particularly if there is a significant amount of the same data being sent across. There is also the issue of spanning multiple TCP frames, but this may not be a significant issue. If the image was to span multiple TCP frames, in all likelihood, the original data would span *those same frames*. If

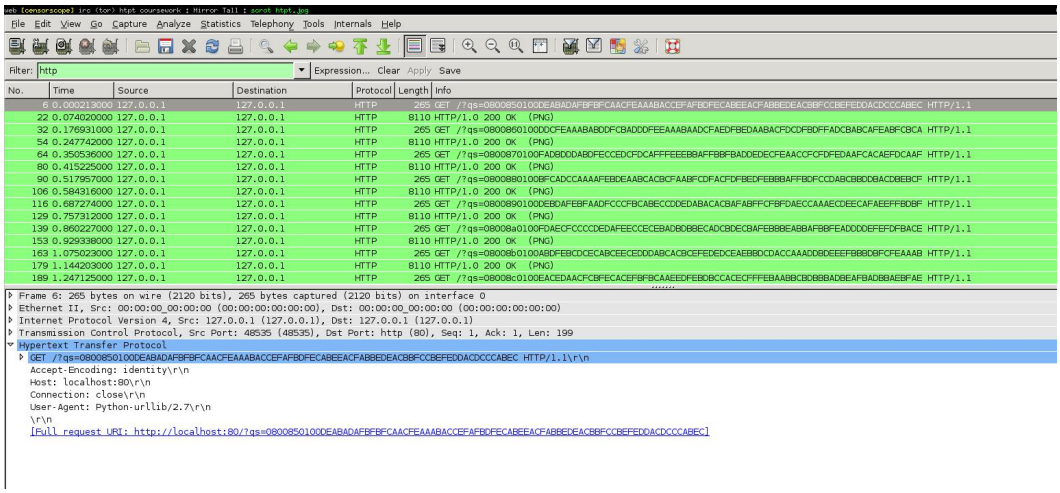

Fig. 3: Wireshark trace showing test stream

there is no compression (or expansion of data) and full sized frames are used (i.e., ¿= 1500 bytes), at most there will be a single extra TCP frame due to the initial HTTP overhead.

In our tests, 8110 bytes were transmitted, which encoded 8192 bytes of useful data. This is due to a small amount of compression, and is not likely a typical case.

Bitmap encoding would be significantly more consistent, as it is similar to the URL encoding in that data is sent more-or-less in the raw.

Note: this is ignoring the TCP connection establishment overhead. This occurs on every piece of transmitted data, but is being glossed over in the overhead section, as it is more affecting of timing rather than of data transmission.

4.2.2. Timing. There are three main pieces of timing information that were looked at during the evaluation phase of this project. The first two are CPU dependent, while the final one was a design decision.

The two encoding schemes' encoding and transmission times were, for the most part, CPU bound. Testing was performed on a single machine, to eliminate network transmission times, so it is safe to say that this was a CPU bound operation. URL encoding and transmission was reasonably quick, at around 2ms per encoded URL. PNG encoding and transmission, on the other hand was rather slow, hovering around 70ms. Transmission time, in the PNG case, was small, in that it was transmitting one frame, so virtually the entire 70ms was spent encoding as a PNG.

As it is CPU bound, a faster processor would be able to shorten these times, but not eliminate them. It is not a quick process, in any case, and would contribute to latency experienced by the end user.

The far more important piece of timing that would affect the user's experience is the polling time that was used. After receiving data, a 100ms gap occurs before requesting any more data. This was simply for the prototype, and could be reevaluated how the polling occurs. Possibilities include polling immediately if a 'MORE' bit is set within the HTPT header, with 100ms polling being the norm otherwise. This is a complex decision, as it would affect how easy it is to fingerprint HTPT traffic.

4.3. Security Aspects

The security considerations that we considered are seemingly satisfied. It would be very difficult to detect that the data that is transiting this server is obfuscated real data, even with deep Deep Packet Inspection. This is due to the design of using HTTP as a carrier

for censored data. The gallery also provides a layer of security in that it prevents active probing attacks. Automated attacks would detect a web gallery of some variety, while an actual person would also see what appears to be a proper image gallery.

5. FUTURE WORK

There are significant possibilities for expanding this work.

The biggest improvement could be made in the framing module. This module could vary the encoding scheme and encoding amount such that the same style of data is not seen consistently. This can be 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.

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.

Replay attacks need to be looked at in greater detail to determine if further mitigation is needed.

The performance of this tool could also be improved by implementingHTTP pipelining and reusing TCP connections. By sending multiple requests without waiting for a response, HTTP pipelining makes more effective use of the available bandwidth. TCP connection reuse also improves performance by avoiding an extra RTT each time data is sent and by allowing the sender to use congestion control and use more bandwidth.

6. CONCLUSION

This project has been a good experience in developing anti-censorship tools. HTPT is not complete enough to be used in an area where censorship is a way of life, but it is a good prototype for further work.

As such, this can be easily expanded upon. The decision to make the encoding schemes easily extensible is important, as it allows for other developers to implement more creative or less frequently used HTTP elements to further mask a stream. HTPT could be turned into a proper Tor Pluggable Transport such that it could be used by a much larger audience.

Additionally, HTPT can be used alone, or in conjunction with an encrypted protocol stream, such as Tor or an SSH tunnel. By providing a scheme that can be used without relying on another protocol, HTPT is valuable alone.

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).