JumpBox – A Seamless Browser Proxy for Tor Pluggable Transports

Jeroen Massar[1], Ian Mason[2], Linda Briesemeister[2], Vinod Yegneswaran[2]

[1] Farsight Security, Inc
{massar}@fsi.io
[2] SRI International
{iam,linda,vinod}@csl.sri.com

Abstract. Anonymity systems such as Tor are being blocked by many countries, as they are increasingly being used to circumvent censorship systems. As a response, several pluggable transport (proxy) systems have been developed that obfuscate the first hop of the Tor circuit (i.e., the connection between the Tor client and the bridge node). In this paper, we tackle a common challenge faced by all web-based pluggable transports – the need to perfectly emulate the complexities of a web-browser and web-server. To that end, we propose a new system called the JumpBox that readily integrates with existing pluggable transports and avoids emulation by forwarding the HTTP/HTTPS requests through a real browser and webserver. We evaluate our system using multiple pluggable transports and demonstrate that it imposes minimal additional overhead.

1 Introduction

Anonymity systems such as Tor are increasingly being used as circumvention systems to bypass Internet filtering and censorship. However, these systems by themselves are ill-suited for this purpose as they serve to obfuscate only underlying use case of the anonymity system (i.e., Tor) and not the use of anonymity system itself. Hence, systems such as Tor are repeatedly and often continuously subject to wholesale blocking attempts, through the use of advanced DPI technologies, by many countries [22, 4, 18, 5, 20].

To address this limitation, the Tor research community has embarked on a collective effort to develop an assortment of pluggable transports that morph Tor traffic to make it resemble some other protocol stream. Examples of such systems include Dust [23], Flash proxy [9], FreeWave [13], Format Transforming Encryption (FTE) proxy [6], Obfsproxy [14], Meek [8], ScrambleSuit [24], SkypeMorph [19] and StegoTorus [21].

A common requirement shared by many of these pluggable transports (e.g., StegoTorus, FTE proxy, Meek) is the need to emulate a browser-based protocol (e.g., HTTP, HTTPS). Prior research has pointed to this as fundamental limitation affecting these systems [11]. The argument is that browsers and web servers are complex systems and the only unobservable way to emulate a browser or a web server is to actually *be the browser or the web server*. We take their suggestion to heart while trying to reconcile the fundamental limitations of developing systems inside a browser-based environment.

Contributions. We describe a new system framework called JumpBox that explicitly addresses the HTTP endpoint emulation problem. The JumpBox framework attempts to strike a balance by implementing two lightweight shims (i.e., a browser plug-in and web server module) that tunnel traffic between existing pluggable transport endpoints. This design choice has three important advantages: (*i*) uses an unmodified browser and web server; (*ii*) flexibility to develop applications outside the constraints of a browser environment and (*iii*) seamless integration with existing pluggable transports. In addition, we implement an HTTPS extension to Chrome that improves HTTPS security with the ability to pin certificates of known HTTP servers.

In the following sections, we first describe related work on pluggable transports and circumvention systems. Then we describe the design and implementation of the JumpBox prototype system and the known hosts verification extension. We then demonstrate system utility by extending three existing pluggable transports: StegoTorus, Meek and FTE proxy. Our system evaluations demonstrate the flexibility of the JumpBox design in supporting diverse use cases while imposing minimal performance overhead. Finally, we conclude by discussing system challenges, limitations and future work.

2 Related Work

Here, we provide background information on pluggable transport research and summarize other related research in the area of blocking resistance.

2.1 Pluggable-Transports Overview.

Obfsproxy [14] was the first implementation of a Tor pluggable transport. Unlike other pluggable transports that attempt to make Tor look like popular benign or unblockable protocols, Obfsproxy transforms Tor to make it look like an unknown high-entropy traffic stream. While Obfsproxy scrubs Tor-related content identifiers, its transformation preserves higher-order statistics such as inter-packet arrival times and packet sizes.

ScrambleSuit [24] is an extension to Obfsproxy that morphs packet lengths and inter-arrival times while also providing a new authentication mechanism that defends against active-probing attacks. However, like Obfsproxy, it also does not attempt to mimic any specific cover protocol.

Flash proxy [9] uses WebSockets to proxy the traffic between a Tor client and a Tor bridge through short-term, frequently changing proxies provided by Internet users who visit volunteer websites helping Flash proxy. The original Flash proxy did not attempt to mimic another protocol, however, it has recently been integrated with Obfsproxy.

SkypeMorph [19] intends to make the traffic between a Tor client and a Tor bridge look like a Skype video call. FreeWave [13] also hides data by modulating a clients Internet traffic into acoustic signals that are carried over Skype connections. However, FreeWave's operation is not bound to a specific VoIP provider and so it its more resilient to blocking attempts that target a specific VoIP service.

StegoTorus [21] transforms a Tor stream into a series of short-lived HTTP connections and implements a client-side request generator and a server-side response generator. The request generator hides data in cookies, URI and upstream JSON POST messages. The response generator hides data in downstream PDF, SWF, JavaScript and JSON content. StegoTorus makes limited attempt to accurately mimic the behavior of browser and web server. Thus it is easily detectable through active probing and man-in-the-middle attacks.

The FTE proxy [6] fools DPI systems into protocol misidentification by ensuring that ciphertexts are formatted to include the telltale protocol fingerprints that DPI systems look for. Protocol formats are specified as regular expressions lifted from system source code or automatically learned from network traces. As we show in our evaluation, HTTP requests generated by FTE proxy are easily distinguishable from that of normal browser requests.

Meek is a new pluggable transport that leverages the Google App engine as an unblockable proxy to relay Tor traffic. It wraps Tor transport with an HTTP header, which is further concealed within a TLS session for obfuscation. Meek is vulnerable to rogue certificate attacks and its TLS requests are acknowledged to be distinguishable from that of the Chrome browser [8].

Dust [23] attempts to define a cryptosystem whose output is wholly indistinguishable from randomness and could be theoretically be blocked by protocol whitelisting techniques. Flash proxies [10] attempt to evade proxy blocking by recruiting thousands of volunteer proxies available from website visitors making it infeasible to block them all. However, it makes no attempt to mask the content of the traffic and is vulnerable to a censor that simply blocks all encrypted connections. Like Meek, Flash proxies could also benefit from browser-based HTTPS tunneling and certificate pinning functionality provided by JumpBox.

2.2 Related Circumvention Systems

Telex [25], Decoy Routing [15], and Cirripede [12] take a different approach to address-filtering resistance: TCP streams are covertly tagged to request that a router somewhere on the path to the overt destination divert the traffic to a covert alternate destination. Telex and Decoy Routing place the tag in the TLS handshake, whereas Cirripede uses the initial sequence numbers of several TCP connections. As all three system rely on the impenetrability of TLS, these clients could also use the browser frontend and certificate pinning functionality provided by JumpBox.

Infranet [7], like StegoTorus and FTE proxy, implements a tunnel protocol for enabling covert communication channel between its clients and servers, modulated over standard HTTP transactions that are intended to resemble innocuous web browsing. Infranet's requestor proxy could be interfaced with the JumpBox daemon for improved HTTP mimicry. Their responder is implemented as an Apache module much like mod_jumpbox. Collage [3] is a scheme for steganographically hiding messages within postings on sites that host user-generated content, such as photos, music, and videos. The sheer number of these sites, their widespread legitimate use, and the variety of types of content that can be posted make it impractical for the censor to block all such messages. However, it is suitable only for small messages that do not need to be delivered quickly. We believe that it could be useful as a rendezvous mechanism for pluggable transports.

3 Background: Goals and Challenges

3.1 Design Goals

We specify below the key design goals of the JumpBox system:

Goal 1. Be the browser – The system should improve the resiliency of the pluggable transport against HTTP mimicry attacks discussed below. We identify ways in which the JumpBox becomes the browser, attacks that we are still vulnerable to and potential ways to address them.

Goal 2. Extensibility – The system should be designed in a way that makes it easy to integrate additional capabilities, i.e., without the constraints of a browser environment.

Goal 3. Seamless integration – The system should be readily integrated, i.e., without any code changes to existing pluggable transports.

Goal 4. Minimal overhead – The system should impose minimal performance overhead to existing pluggable transports.

4 JumpBox System Design

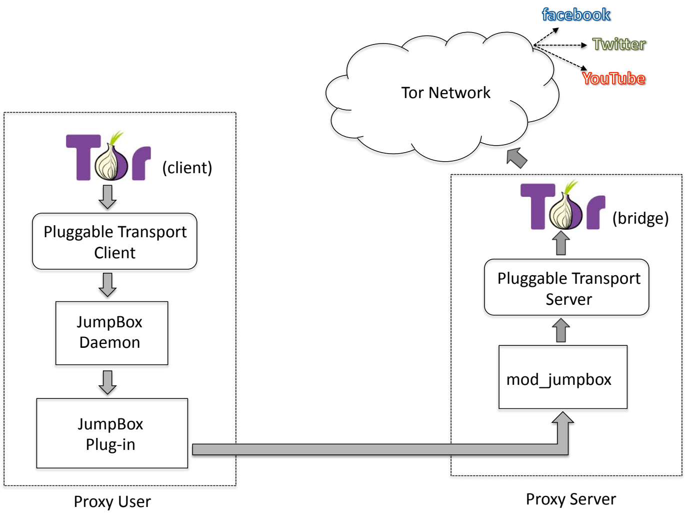

Fig. 1. Overview of the JumpBox architecture. On the left side is the protocol stack running the user host and on the right is the protocol stack running at the remote proxy server. All communications between these two endpoints may be be observed by the censor and uses HTTP or HTTPS. The Tor bridge may be either part of the proxy server or the Tor network.

We illustrate the design and the dataflow of a pluggable transport through the JumpBox architecture in Figure 1. At the client endpoint, our design introduces two lightweight shims: a browser plug-in and a broker module. The former is implemented as a Chrome (or Chromium) browser extension while the latter is a C-based daemon (jbd). At the server endpoint, we design a web server extension (i.e., an Apache module called mod_jumpbox) that forwards connections to the pluggable transport server (PTS). In this section we concentrate on the role of JumpBox as a conduit between a PTC and PTS. In addition, JumpBox serves several purposes, such as enabling rendezvous services that we describe in Section 5.

The JumpBox daemon (jbd) listens on a well-known localhost port for connections from both the PTC and plug-in, which in Chrome can't open a listen port, while acting as a bridge that buffers data between the two components. The plug-in communicates with the jbd using the pull request to GET the next buffered PTC HTTP request to proxy and POSTs back the response through the push request. Thus the JumpBox system, transforms a single request-response round trip sequence, as shown in Figure 2, into a series of three plug-in initiated XMLHttpRequests [1] directed to the jbd, mod_jumpbox, and jbd, respectively, as depicted in Figure 3.

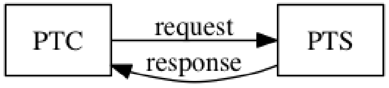

Fig. 2. A single unproxied request/reply

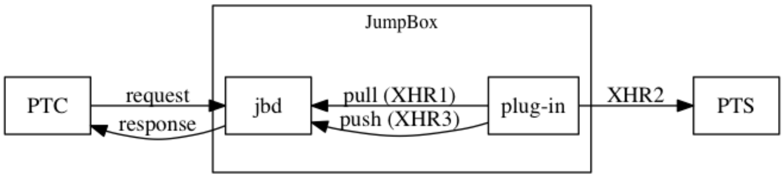

Fig. 3. The JumpBox proxied request/reply

4.1 View from the jbd Daemon

The JumpBox C-based daemon (jbd) provides an HTTP/HTTPS interface on a (local) address and the PTC is simply configured to directs its requests to the PTS as if it were located at this address. The jbd daemon forwards such a request to the plug-in in the form of a response to a plug-in pull request. The response to the PTC request is synthesized from the corresponding plug-in push request to the jbd daemon. The pull-push requests form an integral part of the Jumpbox API which we describe in more detail below.

4.2 View from the Plug-In

From the perspective of the plug-in the communication has three legs:

- [**Leg 1 (XHR1):**] The plug-in requests the next data block (i.e., HTTP request) to forward. This is a "GET /pull" request whose response contains the ordinal PTC request (which can either be an HTTP GET or POST). The URI, method, and cookie from the original PTC request are stored in unique `jbd` header fields. In addition, there is a `jbd` sequence number field that is added by `jbd`. HTTP contents, if any, are forwarded without modification.
- [**Leg 2 (XHR2):**] The plug-in transforms the result of the first leg into the actual request sent to the remote mod_jumpbox front-end. Here, we only preserve the URI, cookie and content from the first leg and rely on the host browser to generate all other aspects of the HTTP request.
- [**Leg 3 (XHR3):**] The response to the request from the mod_jumpbox front-end is forwarded back to `jbd`.

Note that legs (1) and (3) are on localhost while (2) is visible over wide-area network links and so is subject to adversarial scrutiny.

4.3 The JumpBox Plug-in API

The core of the API relevant to the JumpBox's role as a proxy conduit for the PTC is centered around the `pull` and `push` requests. When the PTC makes a request to `jbd`, this request is synthesized into the response to (a presumably pending) `pull` request from the plug-in. The response to the plug-in `pull` request is the contents of the PTC request together with the following five additional `jbd` headers: `JB-URI`, `JB-Method`, `JB-Content-Type`, `JB-Cookie`, and `JB-SeqNo`. A description of these headers is provided in Table 1.

Header Field	Description
`JB-URI`	URI of the original PTC request
`JB-Method`	HTTP method used in the original PTC request
`JB-Content-Type`	Content type of the underlying PTC request
`JB-Cookie`	PTC cookie value
`JB-SeqNo`	Sequence number for `jbd` book-keeping

Table 1. Summary of additional headers introduced by `jbd` when communicating with the plug-in

The plug-in processes such a request by making an XMLHttpRequest to PTS through the mod_jumpbox server, with the method and headers as specified by the above `jbd` headers. The PTS response through mod_jumpbox to this synthesized request is then forwarded back to `jbd` as a `push` POST request, again making use of additional `jbd` headers. In this case the `push` `jbd` headers are `JB-HTTPCode`, `JB-HTTPText`, `JB-SeqNo`, and `JB-Set-Cookie` (as described in Table 2).

Header Field	Description
JB-HTTPCode	Status code of the underlying PTS response
JB-HTTPText	HTTP status text of the underlying PTS response
JB-SeqNo	Book-keeping sequence no, maintained by jbdthat is preserved by the plug-in
JB-Set-Cookie	Value of any Set-Cookie header received by the plug-in in the response from mod_jumpbox

Table 2. Summary of additional headers introduced by the plug-in when responding back to jbd

5 System Implementation

5.1 JumpBox Plug-In Prototype

The JumpBox prototype plug-in is a relatively simple Chrome (or Chromium) plug-in written entirely in JavaScript (approximately 2,100 LoC). Apart from the XMLHttpRequest API that it uses to carry out the underlying HTTP requests, and the usual chromium plug-in infrastructure (chrome.tabs and localStorage) it uses to present a reasonable UI experience. It also makes use of two other APIs: the chrome.WebRequest and chrome.browsingData APIs. The use of chrome.browsingData is to ensure that our browser cache remains empty. We do not want the browser to cache our requests to the PTS, since we have no real knowledge of whether the request will look unique to the hosting browser. The use of the chrome.WebRequest API requires more explanation since it is central to the design.

Two design problems were encountered in developing the prototype. First, the ECMAScript specification [2] provides no mechanism to modify either the Cookie or Set-Cookie headers of an XMLHttpRequest. Hence, we rely on the chrome.cookies API to make the transformation, which introduces few additional complexities. For example, we need to incorporate logic for parsing the cookies, since the chrome.cookies API exposes them as key-value pairs, not as raw headers). We also need to protect against possible race conditions if we ever issued than one XHR to the server at a time, since the browser's cookie store is essentially an unprotected global variable.

Second, in POSTs but not GETs Chrome adds a Origin:chrome-extension header similar to the following:

```
Origin:chrome-extension://mbglkmfnbeigkhacbnmokgfddkecciin
```

which somewhat defeats the whole purpose of the plug-in, as adversaries could use this as a signal for filtering. Hence, we use the chrome.webRequest API to scrub the origin header before the request is sent out to the remote server. We also use the chrome.webRequest API to convert the intangible cookie related headers into more tangible ones. Specifically, we convert JB-Cookie headers into a Cookie header for outbound requests and convert an incoming Set-Cookie into a JB-Set-Cookie header. Finally, we also add a distinguishing header to the Leg (1) & (3) XHRs, so that the chrome.webRequest event does not modify them.

5.2 JumpBox Daemon Prototype

The JumpBox daemon (jbd) is implemented in pure C (approximately 3,600 LoC) and exposes a JSON/HTTP-based interface through which both HTTP clients (PTC) and our JumpBox plug-in communicate. The core functionality is built around a generic Functions and Utilities library libfutil (approximately 10,000 LoC) which supports a broad range of network functionality including an HTTP Server engine, a generic network sockets framework and list functions.

Our implementation is optimized for performance and scalability. The HTTP Server engine is event based and has several worker threads to handle multiple requests in parallel. When a network read from a client would block the request is moved back to it's queue and the read is retried when data becomes available on the socket.

Internally jbd has three request queues: proxy_new, proxy_out and api_pull. All API requests are matched to responses using the JB-SeqNo field. The JB-SeqNo is generated by jbd, and preserved by the plug-in, though of course it does not appear in the headers sent over the wire to the remote server.

jbd differentiates between a client or Plug-in request by looking for an HTTP Host: header of localhost:<listen_port>. The presence of such a header field indicates that the request was originated by the Browser plug-in. For jbd an incoming request from a normal client is a 'proxy request', these are stored in the proxy_new list, where they await for the browser plug-in to retrieve them with the API /pull/ request at which point these requests are moved to the proxy_out list, awaiting an API /push/ which contains the response to the proxied HTTP request.

The client proxy request is blocking, i.e., the answer only comes back when the Browser plug-in has performed an API /push/ to return the answer. In an API request, jbd causes HTTP JB-Set-Cookie: headers to be translated to a standard Set-Cookie header, this as Chrome/Chromium does not allow setting of the Set-Cookie header in AJAX requests.

Similarly, to prevent the browser from using and caching the Cookie, we send out the cookie header as JB-Cookie. The various API requests are either in jbd main (/pull/, /push/, /acs/, /shutdown/, /launch/ and /) or in their specific modules (/acs/, /rendezvous/, and /preferences/). This is ordered this way to allow new code to easily extend jbd. Requesting http://localhost:<listen_port>/ returns a simple HTML page with status details, showing the request queue status inside jbd and a variety of statistics.

Finally, the API /launch/ URI allows launching of either Tor or the PTC along with the parameters that jbd retrieved using rendezvous and ACS (see below). Processes launched through this API are tracked inside jbd.

The current JumpBox prototype serves several related functions in addition to its role as an HTTP proxy between the PTC and the PTS. Specifically, it provides: (*i*) an implementation of rendezvous based on mod_freedom [16]; (*ii*) an implementation of Address Change Signaling (ACS) [16]. Both of these features provide representative examples of how the JumpBox can provide *binary services* that are unavailable in the JavaScript environment provided by the hosting browser. One example is for instance steganographic or other crypto-related functions that would be slow/hard to implement securely inside a browser, especially as one can host the key material outside the browser and thus outside the reach of potentially harmful code.

5.3 mod_jumpbox Prototype

We use a custom Apache2 module (mod_jumpbox) to intercept requests to the PTS process. Our objective is to make the client-facing server as similar to a normal web server as possible. mod_jumpbox installs itself at the head of the Apache internal module handler list. If mod_jumpbox sees an anticipated request, it forwards details of the request to the appropriate pluggable transport server.

6 HTTPS Known Hosts Verification

In this section, we describe an extension to the JumpBox plug-in that improves security of HTTPS communications by adding the ability to pin the certificates of known HTTPS servers.

The JumpBox plug-in uses Asynchronous JavaScript and XML (AJAX) requests for communication. A normal HTTPS AJAX request is made using the following Javascript code:

```
ajax = new XMLHttpRequest();
ajax.onreadystatechange = function () { ... };
ajax.open('GET', 'https://www.example.org/ajax/');
ajax.send(null);
```

This contacts the webserver www.example.org using TLS and makes a `GET /ajax/ HTTP/1.1` request over the TLS connection. The web browser uses the X.509 certificate chain to verify, based on the root certificates, that the certificate of the server is valid and that it really is the site we expect to be talking to. In case an adversary is able to control or otherwise issue a valid certificate for this site, they would be able to perform a man-in-the-middle (MiTM) attack and eavesdrop on communications unbeknownst to the browser.

One solution to this attack is to verify the fingerprints of the TLS certificates being used. Modern browsers (Chrome, Firefox) do not provide any built-in mechanisms for performing this check [3].

We therefore propose a modification to the XMLHttpRequest object that allows us to specify a callback that allows us to verify the fingerprints of the certificates when the certificates are being verified. Additionally we propose that the HTTP request URL and optional body can be replaced with a 'innocuous' request, e.g., `GET / HTTP/1.1` that would not be uncommon to be executed by a standard browser.

The combination of these two modifications allows us to verify the fingerprint of the certificates involved and if we detect an inconsistency warn the caller of this code. They can then decide to change the request to an innocuous request. This allows one to use HTTPS as a covert channel with extra verification, while not compromising or demonstrating to the attacker that you noticed that the certificate had an issue by merely dropping the connection without making an actual request even though one did perform a TLS handshake and certificate exchange.

Our proposed modification looks as follows:

```
ajax = new XMLHttpRequest();
ajax.onreadystatechange = function () { ... };
ajax.open('GET', 'https://www.example.org/ajax/');
```

[3] While Chrome provides limited certificate-pinning ability for selected Google properties, it is insufficient for our needs as it does not extend to all sites and also does not have the innocuous request generation capability described below.

```
    ajax.oncertificatecheck = function () { ... };
    ajax.send(null);
```

The `oncertificatecheck()` callback has one argument which is a Javascript object containing the following structure:

```
{
  "cn":  "*.example.net",
  "serial": "00 A1 A4 94 40 B8 CC E4 29 0E 71 01 2C 40 E0 52 9E",
  "valid-from": "2013-11-23 00:00:00 UTC",
  "valid-till": "2016-11-23 00:59:59 UTC",
  "fingerprint":
  {
    "sha1": "46 B8 FC C4 4D 9F 8D E8 3F 89 D2 42 12 CF 58 7F BF 61 02 D8",
    "md5": "D5 5F E7 FF 78 05 13 43 83 88 57 23 61 C3 12 A3"
  },
  "signed-by":
  {
    "cn":  "ca.example.com",
    "serial": "1",
    "valid-from": "2000-05-30 10:48:38 UTC",
    "valid-till": "2020-05-30 10:48:38 UTC",
    "fingerprint":
    {
      "02 FA F3 E2 91 43 54 68 60 78 57 69 4D F5 E4 5B 68 85 18 68",
      "1D 35 54 04 85 78 B0 3F 42 42 4D BF 20 73 0A 3F",
    }
   }
 }
```

The structure illustrated above illustrates a certificate with the root certificate that signed for it. The caller follows the `signed-by` chain effectively up the chain verifying if it has the same fingerprints for those certificates. Note that SSL certificate chains are normally shown from the root down to the signed certificate. As most users of this modification will know the fingerprint of the signed certificate and not those of the root certificates, the object is effectively listed in reverse order making it quicker to find at which level the compromise likely happened.

For JumpBox, the ACS Bridge list returned by the ACS Redirect Contact contains the fingerprints of the SSL certificates that will be used for the communications in the Relay phase. This allows JumpBox to discern between a valid, but falsely issued SSL certificate. In case of a fingerprint mismatch it will send an innocuous `GET / HTTP/1.1` request and possibly a few follow-up requests. When the fingerprints matches with the details provided by the ACS protocol, JumpBox will make the real requests that proxies data. As we have a trusted HTTPS channel we can opt to not use a transforming PTC, and thus maximize performance.

In addition to this known hosts validation extension being important for JumpBox, it is also useful for the anonymous browser (operating without the JumpBox) in the case that the adversary performs a MiTM attack between the Tor exit node and the real website. Our modification allows the browser to detect this attack, send an innocuous request in a similar fashion as JumpBox would and notify the user that the communications are being tampered with.

7 JumpBox System Evaluation

We conduct a performance evaluation of the JumpBox system against three Tor pluggable transports: Meek, StegoTorus and FTE proxy. For performance testing we run 20000 requests using ApacheBench over the following two scenarios: (*i*) communication between the WebClient and WebServer through the PTC and PTS (*ii*) communication between the WebClient and WebServer through the PTC, JumpBox and PTS. All tests are performed locally, to exclude any side-effects of the network. The two scenarios are illustrated below in Figures 4, and 5.

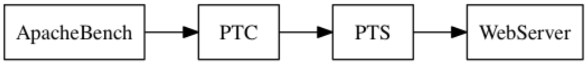

Fig. 4. Scenario: Testing direct PTC

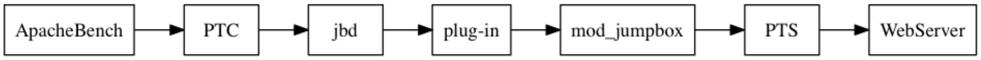

Fig. 5. Scenario: Testing PTC and JumpBox

Note that the typical interface of a PTC and Tor is a SOCKS port, as ApacheBench does not support SOCKS we additionally use socat to interface them together. However, most web browsers do support native proxies and thus provide for SOCKS connections.

Meek normally runs on Google App Engine so that the SSL certificate presented is that as provided by Google, all traffic is thus SSL verified. The Google App Engine acts as a very lightweight HTTP Proxy. In our test, the web server forwards the traffic directly to the Meek Server which allows for simpler and reproducible testing. Running Meek through JumpBox with the Known Hosts HTTPS Fingerprint verification allows for a significant advantage in the case an adversary is able to present a forged certificate (that is fake, but validatable) e.g., when they control a root certificate of an authorized CA on the user's computer. Using JumpBox enables one to verify the fingerprint of the certificate and send innocuous HTTP traffic instead of the HTTP-wrapped Tor traffic forwarded by Meek, which is easily detectable as such.

FTE proxy attempts to hide Tor from standard filters by subjecting them to protocol misidentification attacks. One of the protocols it mimics is HTTP, but unfortunately the data it sends and receives does not comply with the HTTP protocol (missing Host header in the request and missing Content-Length in the replies are two basic examples of such problems). A standard transparent HTTP Proxy would thus break the FTE proxy communication. As JumpBox expects well-formed HTTP, FTE proxy in its current form would not run through JumpBox. For the tests we have thus added an additional mini proxy that detects response boundaries based on the "HTTP/1.1 200 OK" which is fixed as output and inserts Content-Length headers with the correct byte count. In addition, we correct the Content-Type to application/octet-stream instead of ”H” so that we are sufficiently HTTP compatible to work through the JumpBox. Note that these are minimal changes, the remaining complexities of the full HTTP protocol support are handled by JumpBox, though more importantly by the browser and WebServer module that it uses.

Below we provide an illustration of basic FTE proxy on the wire:

```
C: GET /GPcoEIlMxXBh...<base64-encoded-bytes>...LoQas HTTP/1.1
S: HTTP/1.1 200 OK
S: Content-Type: H
S:
S: ....<binary bytes>...
```

Next, we illustrate FTE proxy on the wire through Jumpbox (’...’ indicates ommited data):

```
C: GET /Id5UdpnNYFB...<base64-encoded-bytes>...160VG HTTP/1.1
C: Host: example.com
C: User-Agent:
C: Connection: keep-alive
C: Accept: text/html,...,application/xml;q=0.9,image/webp,*/*;q=0.8
C: User-Agent: Mozilla/5.0 ... Chrome/34.0.1833.5 Safari/537.36
C: Referer: http://www.example.com/
C: Accept-Encoding: gzip,deflate,sdch
C: Accept-Language: en-US;q=0.8,en;q=0.2,de;q=0.2
S: HTTP/1.1 200 OK
S: Date: Thu, 01 Feb 2013 09:01:28 GMT
S: Server: Apache
S: Accept-Ranges: bytes
S: Content-Length: 2529
S: Keep-Alive: timeout=5, max=100
S: Connection: Keep-Alive
S: Content-Type: application/octets
S: Content-Language: en-GB
S:
S: ....<binary bytes>...
```

In both experimental setups an adversary will have to do content-analysis to detect the traffic as non-standard, and hence classify it as either Meek, StegoTorus or FTE proxy. Jumpbox connections are 100% HTTP compliant as they originate from a real browser while the server side is a standard web server. As such fingerprinting based on protocols becomes as good as impossible. Using a standard browser like Chromium/Chrome means that all these HTTP Pluggable Transports also gain support for SPDY, QUIC and other new protocols and methods that the used browser supports.

In Figures 6, 7 and 8, we respectively illustrate the response time variation when directly connecting using the three pluggable transports: Meek, StegoTorus and FTE proxy. In each case, the graph on the left corresponds to response time variation when making a single connection at a time and the graph on the right corresponds to making 10 parallel connections at a time. Next, in Figures 9, 10 and 11, we

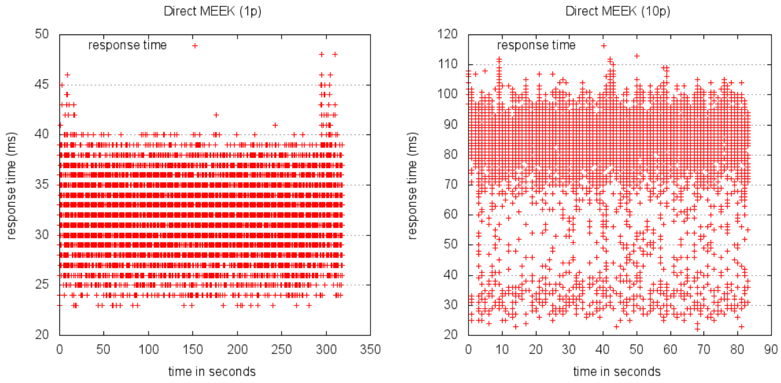

Fig. 6. Response time variation when connecting directly with Meek using 1 (left) and 10 (right) parallel connections. X-axis is absolute time.

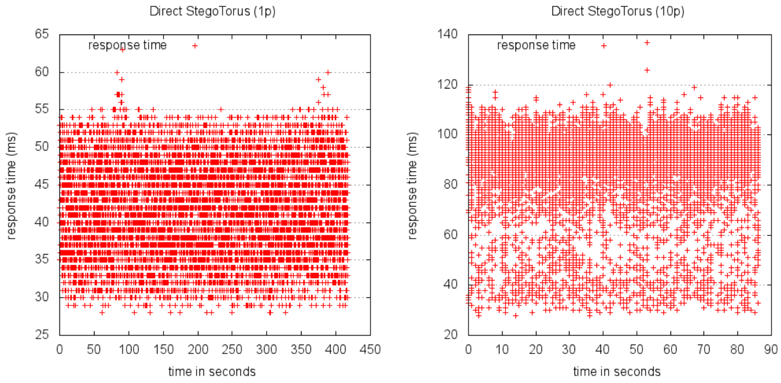

Fig. 7. Response time variation when connecting directly with StegoTorus using 1 (left) and 10 (right) parallel connections. X-axis is absolute time.

respectively illustrate the response time variation when connecting with the three pluggable transports and JumpBox.

Our results are quite encouraging. We find that for Meek and StegoTorus the per-connection overhead is under 15% (5 ms) and for FTE proxy the per-connection overhead is about 80% (20 ms) when we operate with one active connection. Note that the higher overhead for FTE proxy could likely be due to the extra proxying added by our mini-proxy to make its requests more HTTP conformant. Interestingly, when we consider the case with 10 parallel connections, these overheads are further reduced, although the average latency of each connection increases. Here, the additional overhead for StegoTorus and Meek is about 10% (10 ms) while FTE proxy the added overhead is roughly 35% (30 ms). Overall, we find these to be quite reasonable and worth the trade-off in terms of reducing complexity in the pluggable transports and improved indistinguishability.

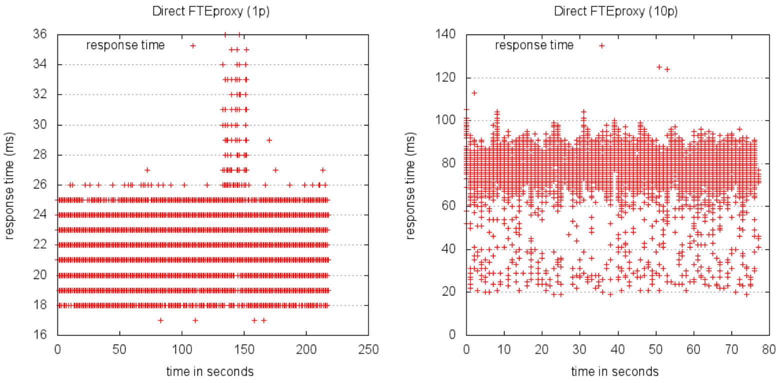

Fig. 8. Response time variation when connecting directly with FTE proxy using 1 (left) and 10 (right) parallel connections. X-axis is absolute time.

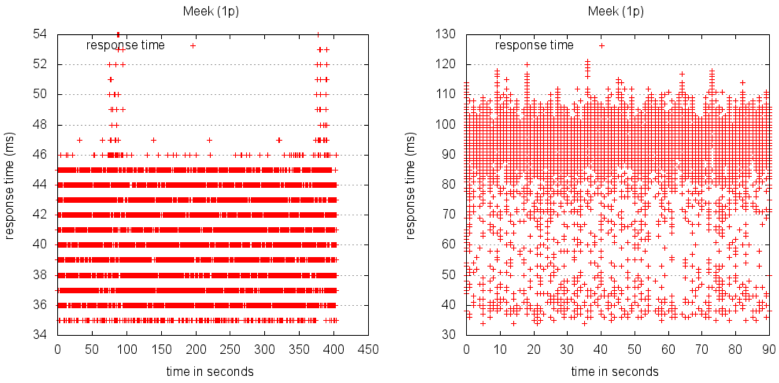

Fig. 9. Response time variation when connecting through the JumpBox and Meek using 1 (left) and 10 (right) parallel connections. X-axis is absolute time.

8 Challenges, Limitations, and Future Work

There are several challenges involved in emulating browser-based HTTP communications which adversaries could exploit to distinguish between real clients. JumpBox addresses some of these detections and relies on the pluggable transport to handle others. In this section, we make explicit the division of responsibility between the two components.

8.1 Passive and Active Man-in-the-Middle Attacks

HTTP header inconsistencies. The header fields in the HTTP protocol are colon-separated name value pairs, each terminated by a carriage return and line feed (`\r\n`), that are transmitted immediately following the request or response line. It is impor-

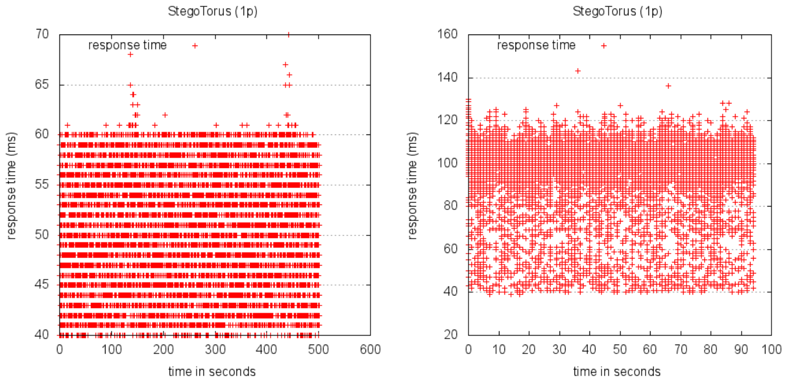

Fig. 10. Response time variation when connecting through the JumpBox and StegoTorus using 1 (left) and 10 (right) parallel connections. X-axis is absolute time.

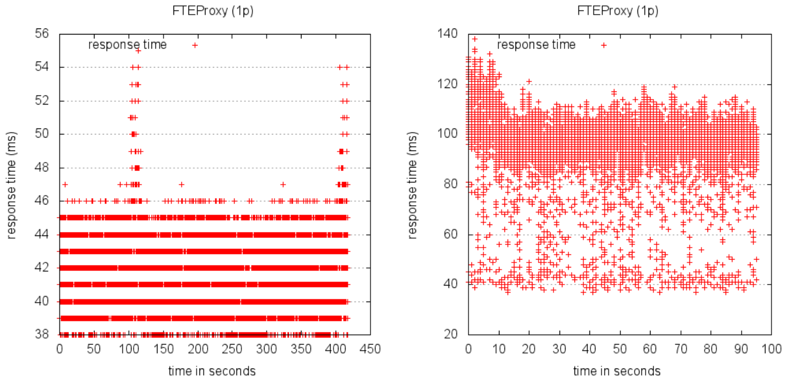

Fig. 11. Response time variation when connecting through the JumpBox and FTE proxy using 1 (left) and 10 (right) parallel connections. X-axis is absolute time.

tant that the ordering and types of various fields in the HTTP header (e.g., Content-Type, Accept, Content-Length, Host, User-Agent) be consistent with the browser or web server that one imitates.

Adversaries may use both active and passive techniques to detect inconsistencies in header parsing between and browser and mimicing agent. By using a real web server and browser, JumpBox is resilient to such attacks.

HTTP URI encodings. There are several popular encodings for the URI field in the HTTP header (e.g., hex encoding, double hex encoding, %u encoding). Adversaries could employ active man-in-the-middle attacks that leverage these encoding techniques to disrupt the pluggable transport communications. By using an Apache web server at the receiver end, JumpBox is able to normalize such transformations, allowing the pluggable transport server to be agnostic to such encodings.

HTTP content encodings. In addition HTTP specifies various content encodings to improve performance of web downloads. Common encoding techniques include gzip, chunked-encoding etc. If a pluggable transport client, that mimics HTTP, fails to support one of these encodings that is supported by the browser it is claiming to be (through the User-Agent string), this could be detected through an active man-in-the-middle attack by the adversary that essentially encodes responses back from the web server. By using a real web server and browser, JumpBox is able to support all encoding agents claimed by the underlying browser.

Replay attacks. Adversaries could replay HTTP requests and use variability in response as a potential means to detect pluggable transports. The current JumpBox system does not explicitly address this attack and relies on the pluggable transport to handle such scenarios. Note, pluggable transports such as StegoTorus are currently stateless and do not handle this. An alternate solution would be to place a caching HTTP proxy in front of the web server running the `mod_jumpbox`.

Content-Injection attacks. Adversaries could insert new content into web pages and identify JumpBox users from the way it reacts to data tampering. JumpBox relies on the pluggable transport to detect such corruptions and react in a manner that is non-fingerprintable. For example, StegoTorus has the ability to detect and recover from data corruption in specific content-types.

Content-Rendering attacks. The JumpBox systems does not actually render the HTML content within a browser. Hence, it does not follow links on a page and does not execute JavaScript content within the browser. There are benign reasons for both of these scenarios, caching and disabling JavaScript. Furthermore, not following links is an explicit design choice that was made for performance reasons. This capability could be included in the JumpBox with additional performance cost or more optimally introduced into pluggable transports. The advantage of the latter approach is that each of the link traversals could actually transport steganographic data.

Timing attacks. Adversaries could also attempt to distinguish JumpBox using timing signatures that fingerprint the extra delay introduced by JumpBox at both ends due to proxying. However, building effective timing attacks at layer 7 is complicated as similar delays could also be introduced due to webserver or database load, web proxies, load balancers etc., which are commonplace on the Internet. Studying the vulnerability of the JumpBox to such attacks is future work.

HTTPS attacks. Similarly HTTPS is vulnerable to a range of attacks including rogue certificate attacks, fingerprinting TLS handshake differences [8, 5] and implementation bugs such as HeartBleed [17]. Certificate-pinning and browser emulation through the JumpBox are attempts to address the first two classes of attacks. Protocol implementation bugs are out of scope.

8.2 Active Probing Attacks

Unsupported methods. Adversaries could pro-actively send malformed requests (e.g., unsupported method types) to Pluggable Transport servers and distinguish them through differences in the way in which they respond from a legitimate web server. In the JumpBox scenario, `mod_jumpbox` only forwards GET, HEAD and POST requests to the pluggable transport. Other method types include unknown methods that are handled by the default Apache handler, which typically cause a 404 or 405 error.

9 Conclusion

We propose JumpBox as a new HTTP forwarding system for making detection of HTTP based Tor pluggable transports much harder. JumpBox is implemented as a set of three components that interpose the communications between the Tor Pluggable Transport Client (PTC) and the Pluggable Transport Server (PTS): `jbd`, JumpBox browser plug-in and the mod_jumpbox webserver extension. Together, these components facilitate a browser-based and webserver-based interface to pluggable transports that improves their resilience against many MiTM attacks that exploit differences in the HTTP implementations of browsers and pluggable transports. We implement support for HTTP as well as HTTPS transport through the JumpBox and evaluate its integration with a range of pluggable transports including StegoTorus, FTE proxy and Meek. Our performance measurements indicate that our prototype system introduces minimal additional overhead.

10 Acknowledgements

We acknowledge Drew Dean, Roger Dingledine, Mike Lynn, Dodge Mumford, Paul Vixie and Michael Walker for various discussions that led to the design and improvement of the JumpBox system. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4022. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Project Agency or Space and Naval Warfare Systems Center Pacific. Distribution Statement A: Approved for Public Release, Distribution Unlimited.

References

1. XMLHttpRequest. W3C Working Draft 6, 2012.
2. ECMAScript, 2014. https://www.ecmascript.org.
3. S. Burnett, N. Feamster, and S. Vempala. Chipping Away at Censorship Firewalls with User-Generated Content. In *Proceedings of the 19th USENIX Security Symposium*, pages 453–468, 2010.
4. R. Clayton, S. J. Murdoch, and R. N. M. Watson. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies*, volume 4258 of *Lecture Notes in Computer Science*, pages 20–35, 2006.
5. R. Dingledine. Iran blocks Tor; Tor releases same-day fix. Tor Project official blog, 2011.
6. K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, CCS '13, 2013.
7. N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of the 11th USENIX Security Symposium*, pages 247–262, 2002.

8. D. Fifield. Meek: A simple HTTP transport. Tor Wiki, 2014.
9. D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras. Evading censorship with browser-based proxies. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, PETS'12, 2012.
10. D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, R. Dingledine, P. Porras, and D. Boneh. Evading Censorship with Browser-Based Proxies. In *Privacy Enhancing Technologies*, volume 7384 of *Lecture Notes on Computer Science*, pages 239–258, 2012.
11. A. Houmansadr, C. Brubaker, and V. Shmatikov. The Parrot is Dead: Observing Unobservable Network Communications. In *The* 34^{th} *IEEE Symposium on Security and Privacy (Oakland)*, 2013.
12. A. Houmansadr, G. T. Nguyen, M. Caesar, and N. Borisov. Cirripede: Circumvention Infrastructure using Router Redirection with Plausible Deniability. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 187–200, 2011.
13. A. Houmansadr, T. J. Riedl, N. Borisov, and A. C. Singer. Ip over voice-over-ip for censorship circumvention. 2013.
14. G. Kadianakis and N. Mathewson. Obfsproxy, 2012.
15. J. Karlin, D. Ellard, A. Jackson, C. E. Jones, G. Lauer, D. P. Makins, and W. T. Strayer. Decoy Routing: Toward Unblockable Internet Communication. In *USENIX Workshop on Free and Open Communications on the Internet*, 2011.
16. P. Lincoln, I. Mason, P. Porras, V. Yegneswaran, Z. Weinberg, J. Massar, W. A. Simpson, P. Vixie, and D. Boneh. Bootstrapping communications into an anti-censorship system. In *2nd USENIX Workshop on Free and Open Communications on the Internet*, 2012.
17. Mashable. The Heartbleed Hit List: The Passwords You Need to Change Right Now.
18. N. Mathewson. Tor and Circumvention: Lessons Learned. Invited talk at the 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), 2011.
19. H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. Skypemorph: protocol obfuscation for tor bridges. In *ACM Conference on Computer and Communications Security*, 2012.
20. M. Price, M. Enayat, et al. Persian cyberspace report: internet blackouts across Iran. Iran Media Program news bulletin, 2012.
21. Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. Stegotorus: A camouflage proxy for the tor anonymity system. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.
22. T. Wilde. Knock Knock Knockin' on Bridges' Doors. Tor Project official blog, 2012.
23. B. Wiley. Dust: A Blocking-Resistant Internet Transport Protocol, 2010.
24. P. Winter, T. Pulls, and J. Fuss. Scramblesuit: A polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, 2013.
25. E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the Network Infrastructure. In *Proceedings of the 20th USENIX Security Symposium*, pages 459–473, 2011.