Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2020

B. Wiley
Operator Foundation
D. Oliver
Guardian Project

July 08, 2019

Enabling Network Traffic Obfuscation - Pluggable Transports
draft-oliver-pluggable-transports-00

Abstract

Pluggable Transports (PTs) are a mechanism enabling the rapid development and deployment of network traffic obfuscation techniques used to circumvent surveillance and censorship. This specification does not define or limit the techniques themselves, but rather focuses on the startup, shutdown, and inter-process communication mechanisms required to make these technologies interoperable with applications.

This document is based heavily on [PT2.1].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The increased interest in network traffic obfuscation technologies mirrors the increase in usage of Deep Packet Inspection (DPI) to actively monitor the content of application data in addition to that data's routing information. Deep Packet Inspection inspects each packet based on the header of its request and the data it carries. It can identify the type of protocol the connection is using even if it was encrypted. DPI is not a mechanism to decrypt what is inside packets but to identify the 'protocol' or the application it represents.

Deep packet inspection has become the prime tool of censors and surveillance entities who block, log, and/or traffic-shape access to sites and services they deem undesirable.
As deep packet inspection has become more routine, the sophistication of monitoring has increased to include active probing that fingerprints and classifies application protocols. Thus, even as conventional care in application design has improved (via encryption

and other protocol design features that encourage privacy), network traffic is still under attack.

The techniques of network monitoring are changing and improving day by day. The development of traffic obfuscation techniques that foil these efforts is slowed by the lack of common agreement on how these techniques are invoked, made easily interoperable with applications, and deployed quickly. This specification addresses those issues.

This specification describes a method for decoupling protocol-level obfuscation from an application's client/server code, in a manner that promotes rapid development of obfuscation/circumvention tools and promotes reuse across privacy tools such as VPNs and secure proxies.

This decoupling is accomplished by utilizing helper code, either in-process through a language-specific API or in a separate sub-processes, that implements the necessary forward/reverse proxy services that handle the censorship circumvention, with a well defined and standardized configuration and management interface. Any application code that implements the interfaces as specified in this document will be able to use all specification-compliant Pluggable Transports.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Background

We define an Internet censor as any network intermediary that seeks to block, divert or traffic-manage Internet network connections for the purpose of eliminating, frustrating and/or logging access to Internet resources that have been deemed (by the censor) to be undesirable (either on a temporary or permanent basis). A variety of techniques are commonly applied by Internet censors to block such traffic. These include:

1. DNS Blocking

2. IP Blocking

3. Port Blocking

These techniques are applicable to a connection's metadata (IP routing information) and do not require inspecting the connection's datastream.

DPI, in contrast, actually looks at the connection's datastream - often, specifically, the initial data elements in the stream (or within blocks of the stream). These elements of the stream can contain clues as to the application-level protocol employed, even when the data itself is encrypted. Through observation over time, these clues ("fingerprints") can be learned by the censor and (along with the routing information) used to block or divert targeted traffic.

A defense against this type of active probing is traffic obfuscation - disguising the application data itself in a manner that is less-easily fingerprinted. However, in early experiments it quickly became clear that repeated use of the same obfuscation technique would, itself, be learned. Methods were developed by which a single obfuscation technique could transform on its own TODO: cite FTE proxy, ScrambleSuit, Dust. This approach proved expensive in terms of computational load. Interest gathered in solving this problem and as more ideas arose so to did the need for a mechanism supporting rapid deploying of new obfuscation techniques.

While intense work on network traffic obfuscation commenced initially and continues within the Tor Project (and across a wider set of external parties using Tor as a vehicle for research), vendors of other privacy-enhancing software (such as VPNs) quickly found their products also foiled by DPI. Thus, it becomes important to see transport pluggability as a mechanism implemented in a manner independent of a specific product or service. The notion of "Pluggable Transports" (PT) was born from these requirements.

4. Architecture Overview

The PT Server software exposes a public proxy that accepts connections from PT Clients. The PT Client transforms the traffic before it hits the public Internet and the PT Server reverses this transformation before passing the traffic on to its next destination. The PT Server directly forwards this data to the Server App, but the Server App itself may itself be a proxy server and expect the forwarded traffic it receives to conform to a proxy communication protocol such as SOCKS or TURN. There is also an optional lightweight protocol to facilitate communicating connection metadata that would otherwise be lost such as the source IP address and port EXTORPORT.

When using an in-process, language-specific API ("Transport API Interface") to integrate PTs into an application on both client and server, the PT Client Library is integrated directly into the Client App and the PT Server Library is integrated directly into the Server App. The Client App and Server App communicate through socket-like APIs, with all communication between them going through the PT library, which only sends transformed traffic over the public Internet.

```
     +------------+                     +---------------------------+
     | Client App +-- Socket-like API --+ PT Client (Library)       +--+
     +------------+                     +---------------------------+  |
                                                                       |
                  Public Internet (Obfuscated/Transformed traffic) ==> |
                                                                       |
     +------------+                       +-------------------------+  |
     | Server App +--  Socket-like API  --+ PT Server (Library)     +--+
     +------------+                       +-------------------------+
```

Figure 1. API Architecture Overview

When using the transports as a separate process on both client and server, the Dispatcher IPC Interface is used. On the client device, the PT Client software exposes a local proxy to the client application, and transforms traffic before forwarding it to the PT Server. The PT Dispatcher can be configured to provide different proxy types, supporting proxying of both TCP and UDP traffic.

```
     +------------+                     +---------------------------+
     | Client App +---- Local Proxy ----+ PT Client (Dispatcher)    +--+
     +------------+                     +---+-------------------+---+  |
                                          | PT Client Library |        |
                                          +-------------------+        |
                                                                       |
                Public Internet (Transformed/Proxied traffic) =====>   |
                                                                       |
     +------------+                     +---------------------------+  |
     | Server App +---- Local Proxy ----+ PT Server (Dispatcher )   +--+
     +------------+                     +---+-------------------+---+
                                            |PT Server (Library)|
                                            +-------------------+
```

Figure 2. IPC Architecture Overview

A PT client-server setup may also mix and match interfaces, using Dispatcher IPC on one end of the connection and the Transport API on the other, as below (or vice-versa):

```
     +------------+                    +---------------------------+
     | Client App +---- Local Proxy ----+ PT Dispatcher Client      +-+
     +------------+                    +---+-------------------+---+ |
                                                                     |
                                                                     |
     +------------+                    +--------------------------+  |
     | Server App +-- Socket-like API --+ PT Server (Library )     +--+
     +------------+                    +--------------------------+
```

           Figure 3. Mixed IPC and Transport API example

Each invocation of a PT MUST be either a client OR a server.

PT dispatchers MAY support any of the following proxy modes: PT 1.0 with SOCKS4, PT 1.0 with SOCKS5, or any of the PT 2.1 modes: transparent TCP, transparent UDP, or STUN-aware UDP. Clients SHOULD prefer PT 2.1 over PT 1.0.

5. Specification

5.1. Pluggable Transport Naming

Pluggable Transport names serve as unique identifiers, and every PT MUST have a unique name. PT names MUST be valid C identifiers, which means that PT names MUST begin with a letter or underscore, and the remaining characters MUST be ASCII letters, numbers or underscores. No length limit is imposed. PT names MUST therefore satisfy the regular expression [a-zA-Z_][a-zA-Z0-9_]*.

5.2. Transports API Interface

5.2.1. Goals for interface design

The goal for the interface design is to achieve the following properties:

- Transport implementers have to do the minimum amount of work in addition to implementing the core transform logic.

- Transport users have to do the minimum amount of work to add PT support to code that uses standard networking primitives from the language or platform.

- Transports require an explicit destination address to be specified. However, this can be either an explicit PT server destination with the Server App is already known implicitly, or an explicit Server App destination with the PT server destination already known implicity.

- Transports may or may not generate, send, receive, store, and/or update persistent or ephemeral state.

- Transports that do not need persistence or negotiation can interact with the application through the simplest possible interface

- Transports that do need persistence or negotiation can rely on the application to provide it through the specified interface, so the transport does not need to implement persistence or negotiation internally.

- Applications should be able to use a PT Client implementation to establish several independent transport connections with different parameters, with a minimum of complexity and latency.

- The interface in each language should be idiomatic and performant, including reproducing blocking behavior and interaction with nonblocking IO subsystems when possible.

5.2.2. Abstract Interfaces

This section presents high-level pseudocode descriptions of the interfaces exposed by different types of transport components. Implementations for different languages should provide equivalent functionality, but should use the idioms for each language, mimicking the existing networking libraries.

5.2.2.1. Transport

- Transport takes a transport configuration and provides a Client Factory and a Server Factory.

- Transports may provide additional language-specific configuration methods.

- The only way to obtain Client Factories and Server Factories is from the Transport.

- The Server Factory of the Transport can fail if the Transport does not provide a server-side implementation. However, most transports provide both a client and server implementation.

- The transport configuration is specific to each Transport. Using a Transport requires knowing the correct parameters to initialize that Transport.

5.2.2.1.1. Client Factory

- Client Factory takes the connection settings and produces a Connection to that server.

- The connection settings are specific to each transport. Some transports will also require an argument indicating the destination endpoint. Producing a Connection may fail if the server is unreachable or if the transport configuration was incorrect.

5.2.2.1.2. Server Factory

- Server Factory takes the address on which the PT server should listen for incoming client connections and produces a Listener for that address

5.2.2.1.3. Listener

- Listener produces a stream of Connections

- New Connections are available whenever an incoming network connection from the PT client has been established. The language-specific API can adopt either a blocking or non-blocking API for accepting new connections, depending on what is idiomatic for the language. 3.2.2.2. Connection

- Connection provides an API similar to the environment's native socket type

- Connection is what is used to read and write data over the transport connection

- The transport-specific logic for obfuscating network traffic is implemented inside the Connection.

6. Adapters

This section covers the various different ways that the Pluggable Transport interfaces (both API and IPC) can be adapted to different use cases.

6.1. API to IPC Adapter

When an application and the transports it uses are written in the same language, either the Transports API or Dispatcher IPC can be used. When they are in different languages, they must communicate through the Dispatcher IPC interface. For maximum flexibility and to

minimize duplication of effort across languages, dispatcher can be implemented by wrapping transport implementations that implement the Transports API. For an example of this approach, see the Shapeshifter Dispatcher [PT2-DISPATCHER], which wraps transports implementing the Transports API in the Go language and provides a Dispatcher IPC interface to use them from other languages.

6.2. PT 1.0 Compatibility

The only interface defined in the PT 1.0 specification is an IPC interface. No standard API is defined. Therefore, PT 1.0 compatibility refers to compatibility between applications and transports where one side conforms to the PT 1.0 specification and the other conforms to the PT 2.1 specification. Fortunately, an adapter is not needed in this case as both the PT 1.0 and PT 2.1 specifications allow for version negotiation. The TOR_PT_MANAGED_TRANSPORT_VER environment variable or -ptversion command line flag is used by the application to specify a list of supported versions, for instance "1.0,2.1". The PT provider responds with the VERSION command on stdout in order to specify which version is supported by the PT provider, for instance "VERSION 2.1". Since the application can specify a list of supported versions, the PT provider can respond dynamically, supporting PT 1.0 when required and automatically upgrading to a PT 2.1 implementation when that is an available option. It is up to applications whether they want to support PT 2.1 exclusively or maintain backwards compatibility with PT 1.0 implementations.

6.3. Cross-language Linking

If two languages are compatible via cross-language linking, then a suitable adapter can be written that wraps the implementation of the Transports API in one language with an API for a compatible language. For example, on Android the Go implementation of the Transports API is wrapped in a Java API to create Java language bindings without the need for a native Java implementation or use of Dispatcher IPC.

6.3.1. Using the Dispatcher IPC Interface In-process

When using a transport that exposes the Dispatcher IPC interface, it may be more convenient to run the transport in a separate thread but in the same process as the application. Packets can still be routed through the transport's SOCKS5 or TURN port on localhost. However, it may be inconvenient or impossible to use STDIN and STDOUT for communication between these two threads. Therefore, in some languages it may be appropriate to produce an "inter-thread interface" that reproduces the Dispatcher IPC interface's semantics, but replaces STDIN and STDOUT with language-native function-call and

event primitives. This is the approach used by OnionBrowser [ONION_BROWSER], the Tor implementation on iOS. This approach is used because Tor uses the Dispatcher IPC mechanism to talk to the transports instead of the Transports API. However, iOS does not allow for applications to have multiple processes. Therefore, an in-process Dispatcher IPC approach must be used instead of traditional separate process Dispatcher IPC. An alternative would be to use the Transports API directly instead of Dispatcher IPC.

6.4. Anonymity Considerations

When designing and implementing a Pluggable Transport, care should be taken to preserve the privacy of clients and to avoid leaking personally identifying information. Examples of client related considerations are:

- Not logging client IP addresses to disk.

- Not leaking DNS addresses except when necessary.

- Ensuring that "TOR_PT_PROXY"'s "fail closed" behavior is implemented correctly.

Additionally, certain obfuscation mechanisms rely on information such as the server IP address and port being confidential, so clients also need to take care to preserve server side information confidential when applicable.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

7.2. Informative References

[ONION_BROWSER] "Onion Browser", 2019, <https://github.com/OnionBrowser/OnionBrowser>.

[PT2-DISPATCHER]
Wiley, B., "Shapeshifter Dispatcher", 2018, <https://github.com/OperatorFoundation/shapeshifter-dispatcher>.

[PT2.1] Wiley, B., "Pluggable Transport Base Specification", 2018, <https://github.com/Pluggable-Transports/Pluggable-Transports-spec/blob/master/releases/PTSpecV2.1Draft1/Pluggable%20Transport%20Specification%20v2.1%20-%20Base%20Specification%20v2.1%2C%20Draft%201.pdf>.

Acknowledgments

Many people contributed to the PT 2.1 specification. Major contributions were made by Dr. Brandon Wiley (Operator Foundation), Nick Mathewson (Tor), and Ben Schwartz (Jigsaw). Valuable feedback was provided by the attendees at the Pluggable Transport Implementers Meetings and the traffic-obf and tor-dev mailing lists. The PT 2.1 specification expands upon the "Pluggable Transport Specification (Version 1)" document authored by Yawning Angel (Tor). Inspiration for the PT 2.1 Go API was also inspired by the obfs4proxy implementation of the PT 1.0 specification in Go, also developed by Yawning Angel (Tor).

Authors' Addresses

Brandon Wiley
Operator Foundation

EMail: brandon@operatorfoundation.org
URI: https://operatorfoundation.org

David M. Oliver
Guardian Project

EMail: david@guardianproject.info
URI: https://guardianproject.info