Pluggable Transports Go API

Overview

The Pluggable Transport Go API provides a way to use Pluggable Transports directly from Go code. It is an alternative to using the SOCKS proxy interface. The Go API may be an appropriate choice when the application and the required transport are both written in Go. When either the application or the transport are written in a different language, the SOCKS proxy interface provides a language-neutral method for configuring and using transports running in another process using the SOCKS protocol communicating over a TCP socket.

This current draft of the Pluggable Transport Go API is only descriptive. The source code of the existing implementation of Pluggable Transports written in Go found at https://git.torproject.org/pluggable-transports/obfs4.git/obfs4proxy has been examine and the present API has been documented. This provides a basis from which a formal API can be developed. The scope of this effort to capture the existing API is limited to those types and methods necessary to implement and use Pluggable Transports available in the Go implementation. The obfs4proxy codebase has other functionality that may be of interest when developing future drafts of the API. In particular, https://git.torproject.org/pluggable-transports/obfs4.git/common contains utility functions that are not strictly necessary to implement a Pluggable Transport, but which many PTs may find useful.

This draft specification is divided into three parts. The “Modules” section provides documentation of the types and methods provided by the Pluggable Transports Go API. The “Implementing a Transport” and “Using a Transport” sections then provide context on how the API is used.

Modules

The Pluggable Transports Go API provides three modules: pt, base, and transports. The pt module contains the Args type, which is a data structure used for specifying transport-specific parameters. The base module provides the interfaces types necessary for implementing a Pluggable Transport. The transports module provides the methods necessary to register a Pluggable Transport with obfs4proxy.

Module pt

This module provides the Args data structure type and associated methods for adding and getting values from the data structure. The Args type is used to provide transport-specific parameters to client and server instances of a transport.

type Args map[string][]string

The Args type provides key-value mappings for the representation of client and server options. Args maps a string key to a list of values. It is similar to url.Values.

func (args Args) Get(key string) (value string, ok bool)

The Get method returns the first value associated with the given key. If there are any values associated with the key, the value return has the value and ok is set to true. If there are no values for the given key, value is "" and ok is false.

If you need access to multiple values, use the map directly.

func (args Args) Add(key, value string)

The Add method appends a value to the list of values for the given key.

Module base

This module provides the interface types for implementing a Pluggable Transport.

Transport is an interface that defines a pluggable transport protocol. It provides a name and factories for generating client and server instances of this transport.

Transport Interface

```
type Transport interface {
  Name() string
  ClientFactory(stateDir string) (ClientFactory, error)
  ServerFactory(stateDir string, args *pt.Args) (ServerFactory, error)
}
```

Name() string

The Name method returns the name of the transport protocol. It must be a valid C identifier. Therefore, the first character must be an underscore or uppercase letter or lowercase letter. Subsequent characters can be any of these and also include numbers.

ClientFactory(stateDir string) (ClientFactory, error)

The ClientFactory method takes a string representing the path to the state directory and returns an instance of the ClientFactory interface for this transport protocol.

ServerFactory(stateDir string, args *pt.Args) (ServerFactory, error)

The ServerFactory method takes a string representing the path to the state directory and transport-specific parameters encapsulated in the pt.Args type. It returns an instance of the ServerFactory interface for this transport protocol.

ClientFactory Interface

The ClientFactory interface defines the factory for creating pluggable transport protocol client instances.

```
type ClientFactory interface {
  Transport() Transport
  ParseArgs(args *pt.Args) (interface{}, error)
  WrapConn(conn net.Conn, args interface{}) (net.Conn, error)
}
```

Transport() Transport

The Transport method returns the Transport instance to which this ClientFactory belongs.

ParseArgs(args *pt.Args) (interface{}, error)

The ParseArgs method parses the supplied arguments into an internal representation for use with WrapConn. The return type “interface{}” is the Go idiom for an unknown or wildcard type. The actual type should be whatever the WrapConn method is expecting as an argument.

WrapConn(conn net.Conn, args interface{}) (net.Conn, error)

The WrapConn method takes a net.Conn and the output of ParseArgs are arguments. It returns a net.Conn instance that wraps the given net.Coon with the transport protocol implementation. It also does whatever handshaking steps are necessary so that the returned net.Conn can be used to send and receive data.

ServerFactory Interface

The ServerFactory interface defines the factory for creating plugable transport protocol server instances. As the arguments are the property of the factory, validation is done at factory creation time.

```
type ServerFactory interface {
  Transport() Transport
  Args() *pt.Args
  WrapConn(conn net.Conn) (net.Conn, error)
}
```

Transport() Transport

The Transport method returns the Transport instance to which this ClientFactory belongs.

Args() *pt.Args

The Args method returns the Args required on the client side to handshake with server connections created by this factory.

WrapConn(conn net.Conn) (net.Conn, error)

The WrapConn method takes a net.Conn and returns a net.Conn instance that wraps the given net.Coon with the transport protocol implementation. It also does whatever handshaking steps are necessary so that the returned net.Conn can be used to send and receive data.

Module transports

func Register(transport base.Transport) error

The Register method takes a Transport instance and adds it to a private data structure, such that it will be retrievable by the Transports and Get methods.

func Transports() []string

The Transports method returns a list of strings representing the names of Transport instances previously added with the Register method.

func Get(name string) base.Transport

The Get method takes a string representing the name of a Transport instance previously registered with the Register method. It returns that Transport instance.

Implementing a Transport

In order to implement a transport, instances must be provided for the Transport, ClientFactory, and ServerFactory interfaces types.

The Transport instance has two main pieces of functionality. The first is dealing with transport-specific parameters provided in a pt.Args data structure. The second is wrapping a net.Conn representing a plain TCP socket with a net.Conn providing an obfuscated connection.

Dealing with Parameters

On the client side, the transport needs to implement ParseArgs. This should validate the arguments, including ensuring that all required arguments are present. If validation fails, then ParseArgs should return an error. If validation succeeds, then ParseArgs should return a data structure that can be of any type, but must be compatible with the WrapConn method.

On the server side, the transport parameter handling is done by the Transport interface's ServerFactory method. Since this method returns a ServerFactory instance, it can also delegate parameter handling to the ServerFactory instance's constructor method. The ServerFactory must also provides an Args method that returns matching parameters for the client side ClientFactory. These are may be different that those provided to the ServerFactory method.

Wrapping the Connection

On the client side, the WrapConn method accepts the validated parameters data structure from ParseArgs and a plain TCP connection. The transport must use this connection to do any

handshaking required and then return an instance of the net.Conn interface that the application can use to communicate using the transport. This net.Conn instance must perform any obfuscating transforms and then handle reading and writing to the plain TCP connection.

On the client side, the WrapConn method accepts a plain TCP connection. The transport must use this connection to do any handshaking required and then return an instance of the net.Conn interface that the application can use to communicate using the transport. This net.Conn instance must perform any obfuscating transforms and then handle reading and writing to the plain TCP connection.

Using a Transport

Applications using transport have two main responsibilities. The first is gathering transport-specific parameters to pass to the transport. The second is managing TCP connections.

Dealing with Parameters

The application can gather the transport parameters in many different ways, it is up to the application. The specific parameters required vary depending on the transport being used. Once the parameters have been obtained, on the client side they are passed to the ClientFactory.ParseArgs method and on the server side they are passed to the Transport.ServerFactory method.

Managing TCP Connections

Most transports do not create their own TCP connections, although some do. The transport has full sovereignty to perform any computation, including opening sockets, once WrapConn is called. However, many transports will expect to use the net.Conn passed into WrapConn for all actual network communication. Therefore, it is up to the application to make this TCP connection so that it can be made available to the transport. On the client side this will require opening a TCP connection to the server, and on the server it will require accepting a TCP connection from the client. How the client gets the destination address for the server is a matter for the application to handle. Additionally, the application will need to provide data to the transport to send. In the case of a proxy, this would require opening other TCP connections on which to receive the data to proxy. For other uses cases, where the application is use a transport directly and not acting as a proxy, the data could be generated directly by the application.