pyptlib Documentation

Release 0.0.5.UNRELEASED

Brandon Wiley

March 21, 2014

pyptlib is a little Python library that makes the job of writing pluggable transports easier.

pyptlib README

- What is pyptlib?

pyptlib is a little Python library which understands the pluggable transport managed-proxy protocol.

- Who is interested in pyptlib?

You might be interested in pyptlib if you have an application that obfuscates TCP traffic and you want to integrate it easily with Tor.

- What does pyptlib do?

pyptlib speaks with Tor and informs your application about which pluggable transports Tor needs, in which ports they should listen for connections, in which filesystem directory they should keep state, etc.

- What does pyptlib not do?

pyptlib doesn't help your application do networking or obfuscation.

- What does pyptlib expect from an application?

pyptlib assumes that your application is executed by Tor as a managed-proxy.

pyptlib assumes that your application acts as a proxy: it listens for traffic on a TCP port and pushes the traffic somewhere else.

pyptlib assumes that your application has a SOCKS server when it acts as a client. This is needed because Tor needs to dynamically select where its data will be pushed to.

- How do I use pyptlib?

Read the documentation, the examples and the source.

- What are these buzzwords?

`glossary.rst`

CHAPTER 2

API overview

Be sure to read `API.rst` and `glossary.rst` before reading this file.

2.1 General Overview

Applications begin by initializing pyptlib.

Then pyptlib informs the application about which transports it should spawn, in which ports they should listen for connections, etc.

Then the application launches the appropriate transports as instructed, and for each transport it reports to pyptlib whether it was launched successfully or not. Finally, the application announces to pyptlib that it finished launching transports.

From that point and on the application should forget about pyptlib and start accepting connections.

2.2 Detailed API Overview

2.2.1 0) Find if it's a client or a server

An application using pyptlib should start by calling `pyptlib.config.checkClientMode()` to learn whether Tor wants it to run as a client or as a server.

You should then create a `pyptlib.client.ClientTransportPlugin` or `pyptlib.server.ServerTransportPlugin` as appropriate. This object is your main entry point to the pyptlib API, so you should keep it somewhere for later access.

2.2.2 1) Get transport information from Tor

The next step is to run `init` to parse the rest of the configuration and communicate the results to Tor. You should pass a list of names of the transports your application supports.

The application should be prepared for `pyptlib.config.EnvError`, which signifies that the environment was not prepared by Tor.

Consider an example of the fictional application *rot0r* which implements the pluggable transports *rot13* and *rot26*. If *rot0r*, in step 1, learned that Tor expects it to act as a client, it should now do:

```python
from pyptlib.client import ClientTransportPlugin
from pyptlib.config import EnvError

client = ClientTransportPlugin()
try:
    client.init(supported_transports=["rot13", "rot26"])
except EnvError, err:
    print "pyptlib could not bootstrap ('%s')." % str(err)
```

Afterwards, the API's `config` attribute provides methods to find out how Tor wants your application to be configured. For example, if you store state, it should go in `client.config.getStateLocation()`. For a complete list, see the documentation for that module.

2.2.3 2) Launch transports

Client case (skip if you are a server)

Your application should then use `client.getTransports()` to learn which transports it should launch.

Proceeding with the previous example:

```python
if 'rot13' in client.getTransports():
    launch_rot13_client()
if 'rot26' in client.getTransports():
    launch_rot26_client()
```

For a full list of the methods available, see the module docs for `client` and `client.config`.

Note: Since the application runs as a client, it should launch a SOCKS server in the upstream side of the proxy.

Server case (skip if you are a client):

Your application should then use `server.getBindAddresses()` to learn which transports it should launch.

Since the application runs as a server, it will push data to Tor's ORPort, which you can get using `server.config.getORPort()`.

Proceeding with the previous example:

```python
transports = server.getBindAddresses()
if 'rot13' in transports:
    launch_rot13_server(transports['rot13'], server.config.getORPort())
if 'rot26' in transports:
    launch_rot26_server(transports['rot26'], server.config.getORPort())
```

For a full list of the methods available, see the module docs for `server` and `server.config`.

2.2.4 3) Report results back to Tor.

For every transport that the application launches, it reports to pyptlib whether it was launched successfully or not. This way, Tor is informed on whether a transport is expected to work or not.

Client case (skip if you are a server):

Every time a transport is successfully launched, the application calls `client.reportMethodSuccess` with the name of the transport that was launched, the address where it is listening for connections, and the SOCKS version that the upstream SOCKS server supports.

For example, if *rot13* was launched successfully, waits for connections in ‘127.0.0.1:42042’ and supports SOCKSv4, the appropriate call would be:

```
client.reportMethodSuccess('rot13', 'socks5', ('127.0.0.1', 42042))
```

Every time a transport failed to launch, the application calls `client.reportMethodError` with the name of the transport and a message.

For example, if *rot26* failed to launch, the appropriate call would be:

```
client.reportMethodError('rot26', 'Could not bind to 127.0.0.1:666 (Operation not permitted)')
```

Server case (skip if you are a client):

Everytime a transport is successfully launched, the application calls `server.reportMethodSuccess` with the name of the transport that was launched, and the address where it is listening for connections.

For example, if *rot13* was launched successfully and waits for connections in ‘127.0.0.1:42042’, the appropriate call would be:

```
server.reportMethodSuccess('rot13', ('127.0.0.1', 42042))
```

Everytime a transport failed to launch, the application should call `server.reportMethodError` with the name of the transport and a message.

For example, if *rot26* failed to launch, the appropriate call would be:

```
server.reportMethodError('rot26', 'Could not bind to 127.0.0.1:666 (Operation not permitted)')
```

2.2.5 4) Stop using pyptlib and start accepting connections

When the application finishes launching connections, it should call `reportMethodsEnd()`, to announce to pyptlib that all transports were launched. This way, Tor knows that it can start pushing traffic to the application.

After this point, the API object (in this current version of pyptlib) has no other use.

CHAPTER 3

Pluggable transports glossary

3.1 pluggable transport (sometimes also called 'transport')

Pluggable transports obfuscate network traffic.

Specifically, pluggable transports transform the Tor traffic flow between the client and the bridge. This way, censors who monitor traffic between the client and the bridge see innocent-looking transformed traffic instead of the actual Tor traffic.

3.2 pluggable transport proxy

Pluggable transport proxies are programs that implement pluggable transports. They also implement the networking system that a pluggable transport needs (so that it can proxy data).

3.3 obfsproxy

obfsproxy is a pluggable transport proxy written in C. It implements the obfs2 pluggable transport.

3.4 upstream/downstream

The upstream side of a pluggable transport proxy is the side that communicates with Tor. Upstream data is non-obfuscated.

The downstream side of a pluggable transport proxy is the side that communicates with the other pluggable transport proxy. Downstream data is obfuscated.

3.5 client-mode / server-mode

A pluggable transport is a client if it has a Tor client in its upstream side.

A pluggable transport is a server if it has a Tor bridge in its upstream side.

3.6 external-mode proxy

A pluggable transport proxy is in external-mode if the user explicitly configures it using its command-line interface.

3.7 managed-mode proxy

A pluggable transport proxy is in managed-mode if it's launched and managed by Tor using the managed-proxy configuration protocol. The managed-proxy configuration protocol is defined in the pluggable transport specification.

3.8 pyptlib

pyptlib is a library that implements the managed-proxy configuration protocol and makes it easier for application to be used as managed proxies.

3.9 extended orport

Extended ORPort is an non-implemented feature of Tor that allows a pluggable transport proxy to communicate with Tor in real-time.

CHAPTER 4

pyptlib.client

Client-side pyptlib API.

CHAPTER 5

pyptlib.server

Server-side pyptlib API.

CHAPTER 6

pyptlib.util

pyptlib utilities.

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*