University of Birmingham

School of Computer Science
Final year project

VPN over HTTP

Project Report

Author: Daniel Jones (1427970)

BSc Computer Science

supervised by
Dr Ian Batten

Submitted in conformity with the requirements
for the degree of Bsc Computer Science
School of Computer Science
University of Birmingham

VPN over HTTP

Daniel Jones

April 4, 2018

Abstract

Problem: VPN traffic is easy to block, and commonly blocked on free public networks.

Solution: HTTP traffic is rarely blocked, so encoding data into HTTP traffic is one possible way to bypass filtering and blocking on public networks.

Conclusion: It is possible to encode data in such a way that it is difficult to detect that this has been done.

All code that was developed can be found at:
`https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2017/dgj470`

Keywords: VPN, HTTP, Tunnelling, Obfuscation, Steganography

Acknowledgements

I would like to thank my supervisor, Dr Ian Batten for support and guidance throughout the project.

Additionally, I'd like to thank my housemates, friends and family for support throughout both this project and my degree as a whole.

Contents

1 Introduction

The aim of this project is to be able to tunnel data, and by extension operate a VPN over the Hypertext Transfer Protocol (HTTP). There are multiple reasons why this would want to be done, for example:

- To access services that are blocked by the current network
- To hide the fact that blocked services are being accessed
- To maintain privacy regarding services that are being accessed

This project enables connections to be made and transmit data only over the HTTP protocol, and provide all of the benefits highlighted above.

1.1 Problem Defininition

Censorship and monitoring of web traffic is a common occurrence around the world, as is deep packet inspection to ensure that traffic on port 80 is in fact HTTP traffic. On top of this, organisations have been known to man-in-the-middle HTTPS traffic by adding their own certificates to devices owned by the organisation. The aim of this project is to be able to tunnel data over HTTP and make it virtually impossible to tell that data transfer is occurring.

1.2 Paper Overview

1. Introduction
    - Summary of the aims the project intends to fulfill
    - Short overview of the project
2. Existing Work
    - Review of literature surrounding the project
3. Background
    - High level overview of concepts and designs essential for the project:
        - HTTP
        - TCP
        - VPN
        - Privacy
4. Specification
    - Project specification

5. Design
    - High level program architecture and design
    - Configurable options
    - Design choices
6. Implementation
    - Detailed information about how each component functions
    - High level overview about data flows
7. Testing
    - Testing methodologies
    - Testing strategy
8. Project Management
    - Explaination of how the project was managed
9. Discussion
    - An overview of about what was successful
    - A review of what was learned
10. Conclusion
    - Compare the project to the aims
    - Concludes whether or not the project was successful

2 Existing Work

In this section, I explore some related works, and discuss some literature surrounding Steganography, HTTP, DNS tunneling and detecting tunneled traffic.

There has been a lot of work done on the study of steganography, and using different protocols to hide data. The most work has been done on DNS, as it is often below the radar for firewalls and checking if data is being exfiltrated.

2.1 Image Steganography

Steganography is most commonly used for hiding data in unused or unimportant areas of data[1], and the most common form of data for this is images. This is because all of the colour data in an image is not required for a human to see it, and the human eye is very good at filtering out noise[1].

More advanced approaches to steganography can involve identifying redundant data in images[2] which can be better than changing the least significant bit in an image which can be detected by steganalysis[2].

Steganography that is hidden from computers and is hidden from people are quite different things, and can require quite different approaches. The real challenge is to hide data from both.[2].

2.2 DNS steganography

It is possible to hide data very easily in DNS requests, and this is called DNS Tunneling, and it is often used to get around firewalls and hide which websites are being accessed.[3] This paper highlights the point raised in the previous section, that it is very easy for a human to look at the data and see it's not normal, but non-trivial for a computer. The paper describes how the data is detected, and in doing so describes in depth how the data is encoded and tunneled. DNS tunneling as described in the aformentioned paper has a few advantages and disadvantages. The key advantage is that is can be used in locked down networks, as DNS traffic is often let out, but the main disadvantage is that data transfer is very slow with a lot of overhead.

2.3 HTTP protocol

The HTTP 1.1 Protocol[4] is a protocol that describes how data is sent to and from a client, and it describes many areas where data could be included, HTTP traffic can include: Images, HTML, CSS, Javascript, Binary files, and more.
All of which can be used to hide data.

The headers in HTTP are also a potential vector to hide data, as HTTP Headers are whitespace insensitive, and the order in which the headers are sent do not matter[4].

2.4 Detecting tunneled DNS traffic

There are a variety of ways to detect tunneled traffic, from entropy analysis to performing DNS requests[3]. Another way of performing the lookup is to do character frequency analysis[5]. Frequency analysis looks at the difference in frequency of letters in domain names/english words and in random data. It is similar to but not quite the same as entropy analysis, and it is also more effective[5].

3 Background

This section provides an overview of all of the technologies mentioned in the remainder of the project.

3.1 HTTP

HTTP (Hyper Text Transfer Protocol) is the most important protocol in this project and as such needs to be covered in detail. `HTTP/1.1` is defined in `RFC2616`[4] and all of the information about HTTP comes from that document unless otherwise specified. HTTP is a Request/Response protocol, which means the client Requests data from the server, and the server returns it.
An example request is as follows:

```
GET / HTTP/1.1
Host: localhost
User-Agent: curl/7.58.0
Accept: */*
```

In this request, there is first the request line, and then a series of headers. The headers are a set of key-value pairs which form a dictionary that tells the server extra information about the client.

An example response is as follows:

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.6.4
Date: Tue, 20 Mar 2018 14:25:01 GMT
Content-type: text/html
Content-Length: 10088
Last-Modified: Tue, 13 Mar 2018 15:27:09 GMT

<!doctype html>
<html>
<head>
...
```

In this request the response line is sent, along with some headers which provide important meta-data about the response itself, which is then included when the headers are finished.

3.1.1 HTTP Encoding types

HTTP has a variety of different encoding types and methods. These are specified in the `Transfer-Encoding` and `Content-Encoding`, both of which are used to perform compression and other encoding of data.

`Transfer-Encoding` allows for `Chunked` encoding[6], which splits the page or portions of data up into multiple sections, and sends the length for each section separately. This is done as then servers can send data as it is generated, rather than having to buffer the entire page.

3.2 TCP

TCP (Transfer Control Protocol) is defined in `RFC793`[7] and all of the information about TCP comes from there unless otherwise specified. TCP is a connection based protocol that allows for reliable data transfer between a server and a client.
TCP connections are a connection between two sockets, one active and one passive.
A passive socket is listening, and is typically a `server` whereas an active socket is connecting and is typically a `client`.

3.3 Steganography

Steganography is defined as[8]:

> `the art or practice of concealing a message, image, or file within another message, image, or file`

An example of this could be encoding data into an image, or into a HTTP stream.

3.4 Obfuscation

To obfuscate is defined as[9]:

> `to be evasive, unclear, or confusing`

Obfuscation is simply the act of being `evasive, unclear, or confusing`.

When applied to computers, this is similar to steganography and the two terms are often used interchangeably, however steganography is the art of concealing and disguising data, whereas obfuscation just makes data difficult to read and understand.

3.5 Encryption

> `Encryption is the process of converting data to an unrecognizable or ‘encrypted’ form. It is commonly used to protect sensitive information so that only authorized parties can view it[10].`

There are many types of encryption, AES and RSA are two major examples.

It's generally referred to as bad practice to ‘`roll your own crypto`’[11], which simply means that it is a bad idea to write a bespoke cryptographic algorithm, or to try and come up with your own cryptographic scheme.

3.6 Privacy

Privacy on the internet is often overlooked, however, it is vitally important[12]. Almost everything that is done on the internet is tracked by multiple parties: Internet service providers (ISPs), the websites visited, DNS servers, third party servers and many more. These parties could all be gathering information in order to profile and build a model to predict behavoir. This can then be used for highly targeted advertising which is not always desirable for the end user. An extreme example of this was a case in 2012 where a retailer inadvertently told a father his daughter was pregnant by sending her coupons for baby clothes and cribs.[13] This case was first reported in early 2012, and since then statistical models have become significantly more complex and intrusive. This is demonstrated by Google recently allowing you to mark Adverts as '`knowing too much`'[14].

3.7 Blocking of services

When you connect to Public Wi-Fi, often the provider will limit what content you have access to, often by blocking 'ports' or preventing a blacklist of websites.
HTTP is run over port 80, and this is often one of the only ports available. ISP's have also been known to block certain websites.

3.8 Data Tunnelling

Data Tunnelling is a term for transmitting data in a different form to how it is usually transmitted. A VPN is an example of this, as is an SSH (Secure SHell) tunnel.

3.9 Binary

Binary is how a computer represents data, and it is a series of 1's and 0's, each one is called a bit. All characters have a binary representation, and typically characters are made up of 8 bits. Some file formats may not have a concept of characters, and instead may utilise a stream, which means data is processed one bit at a time.

3.10 VPN

A VPN or Virtual Private Network which is defined in `RFC2764`[15], is a piece of software, split into a server and a client, where the server and the client (or clients) form a network which is both virtual and private.
For a network to be private, it needs to be encrypted so no one else can see the data, and for it to be virtual, it only has to exist inside computers, and not physically connected by cables.

In practical terms this means that data is tunnelled from one computer to another, generally by use of `tun` or `tap` devices. `IP` packets can be read from and written to a `tun` device.

`Ethernet` frames can be read from and written to a `tap` device.
Which ever device type is used, the packets/frames are transmitted through an existing connection to a remote client or server, which will also read and write from a corresponding device.

A typical example of a VPN is OpenVPN, `https://openvpn.net/`.

4 Specification

4.1 Functional Requirements

1. The system must allow duplex communication with a remote server and local client
2. All communication between the user and the server must be encapsulated within the HTTP Protocol
3. The user must be able to browse websites
4. The user must be able to connect to a remote server, over SSH for example
5. The server and client must expose a TCP port which can be used for multiple applications
6. The server must act as a valid HTTP server serving a valid web page if navigated to
7. The server must act as a mirror to another HTTP server when serving HTTP to both the client and any other clients that may discover it
8. The system should not encrypt data traveling over it, data should be assumed to be already encrypted
9. The client should poll the server for data
10. When the client or server has data to send, the client should increase the frequency of requests
11. The system should be able to serve multiple clients at a time without interference

4.2 Non-functional Requirements

1. The extra data encapsulated within the HTTP protocol must be done in such a way that is looks like typical HTTP traffic and is not easy or obvious to spot.
2. The connection speed must be such that browsing the internet is possible without undue difficulty.
3. If the connection drops, the server and client must work to re-establish the connection

5 Design

This section sets out the High level program architecture and design, along with areas that are configurable and the different design choices that were made in the development of the project.

During the design of the project, efforts were made to keep the entire program modular, so all sections are reusable and replaceable.

5.1 HTTP Analysis

To embed data in HTTP on its own would be trivial, putting the data to send and receive as the request/response body as data, while this would get around many web filters, it would only require trivial inspection to determine it is in fact not real HTTP traffic. For this reason, I decided that I wanted to insert the data into real websites and realistic browsing patterns, and to do this I decided that the VPN-Server would act as a website mirror when browsed to in the browser.

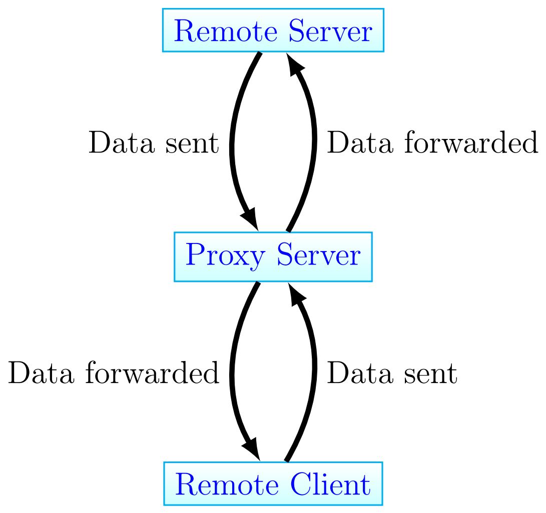
This means that to an observer, the VPN-Server will appear as a live mirror of a real website that updates as the original does.

When talking about how data is transferred in detail, I will refer to the client uploading, or sending data to the server as the 'Request' side, and when the client downloads, or receives data from the server as the 'Response' side.

Putting data in the Response side of the connection is trivial, as when you make a HTTP request, data is returned, and it is often different even if the same request is sent multiple times. The Request side, however is significantly more difficult. This is because when you are browsing a website, the request doesn't significantly change, and it would be incredibly obvious to an observer if for example the 'User Agent' field continuously changes. Therefore, the changes have to be subtle, and rely on the Header fields being a dictionary, so order doesn't matter, and insensitive to whitespace.

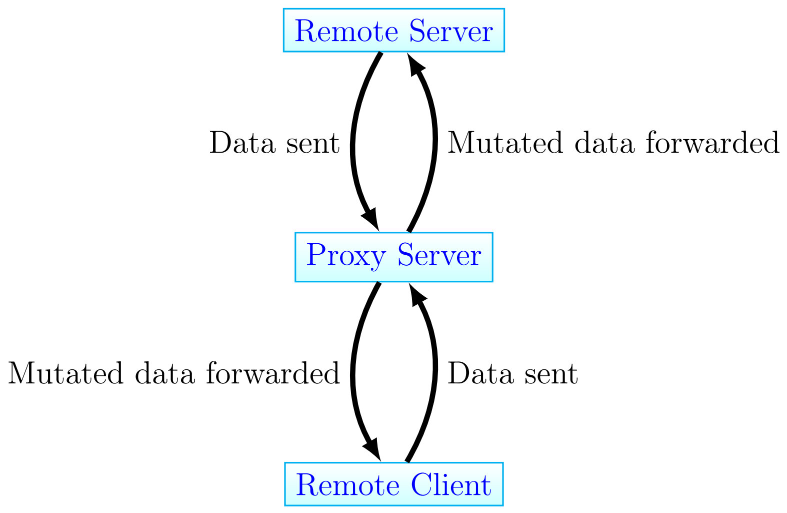
5.2 Server Overview

The server of the project is split into two parts.
The actual 'server' that runs, which acts as a manager for the two duplex connections that are active. The server (called the proxy) creates a TCP listening socket, and then when that socket is connected to it, it creates a connection to the remote server. In the most basic configuration, data is read from one socket and written directly to the other, acting as a TCP-Forwarder:



The server runs two threads, one from the client to the server, and one in the return direction.

One of the reasons I chose this design is because it allows traffic to be '`mutated`' on the fly:



The other reasons I chose this design are:

- It allows the lower level to deal with keeping the connection alive, and the upper levels can assume that if they are running, both of the TCP sockets are alive.
- It allows ease of customisation, because the mutators can be easily switched out.
- It allows the lover level to provide a 'data storage' to the upper threads, which is provided thread safe and reliably so they can access current data about the session.

5.2.1 Configurable Options

In the main 'proxy' part of the program, there are lots of options that are customisable. Here is an example configuration file:

```
config:
    name: localhost-http
    remote: localhost
    remote-port: 8000
    local-port: 80
    test-request: |+
        GET / HTTP/1.1
        Host: localhost
        User-Agent: Test-Agent


    test-response: HTTP/1.0 2\d\d.*
    test-response-length: 16
    proxy-mutator-location: mutators/http.py
    proxy-mutate-receive: receive
    proxy-mutate-send: send
```

The config file is in yaml format.

The '`name`' field is just a unique identifier.
The '`remote`' and '`remote-port`' specify the remote host and port to connect to.
The '`local-port`' specifies the local port to listen on.
The '`test-request`' provides an example request to send to the remote server to check it is active, and the first '`test-response-length`' bytes are read from the socket and compared to the regular expression defined in '`test-response`'.
'`proxy-mutator-location`' is the location of the file that stores the mutator code. '`proxy-mutate-receive`' and '`proxy-mutate-send`' are the names of the functions that are called when data is received from the remote server and needs mutating before sending to the client, and vice versa.

Because of all of this configuration testing connections, debugging, and attempting new things were made significantly simpler. It has also left significant scope for future expansion of the project.

5.3 Obfuscation Overview

Early on in the project, I created a '`generic`' obfuscation program.
The program reads a configuration file, called '`pages`', which sets out the structure the obfuscated data should fit into. It uses a custom configuration file format, for a number of reasons:

- Easy to extend

- Concise
- Easy to understand

An example of the file format is as follows:

```
PAGE#0.1
DELIM <-

bin <-0
bin <-1

byte <- %bin%bin%bin%bin%bin%bin%bin%bin
*out <- %byte
```

This is a very simple example that will simply print out a byte of the input at a time, but in binary. The reason I designed the obfuscation like this is because it allows for easy expansion, and testing. The obfuscation is completely reversible, and more details about how it works will be explained in a later section.

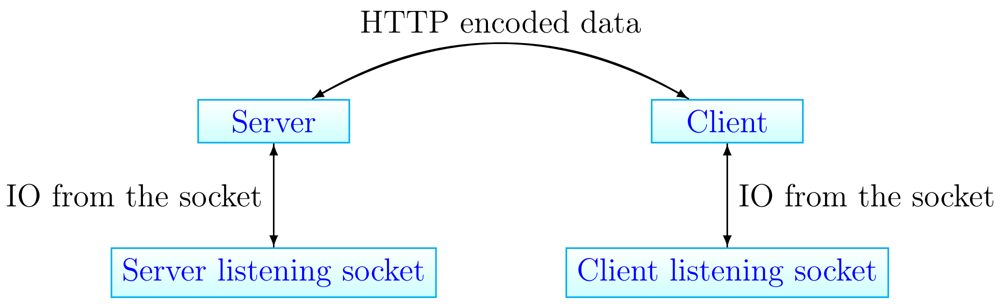
The obfuscation layer also contains functionality to store data in the order of lists, and by appending whitespace to whitespace independent places, all of which is replaceable.

5.4 HTTP Layer

The HTTP Layer consists of a '`mutator`' (as defined in the previous section), and a corresponding client. The HTTP layer understands the HTTP Protocol, and how data can be inserted into it. The layer also calls the '`obfuscater`' which is used to insert data into the HTTP protocol.

5.4.1 Listening Ports

As a part of the HTTP layer (on both server and client), a TCP socket is listening, which is where all data that is transferred from client to server (and vice versa) is read from and written to.

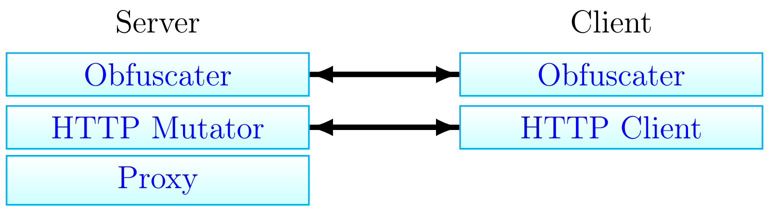


Using listening ports instead of directly interfacing with an application means that there is significantly more flexibility.

5.5 Modular Sections Overview

The overall design of the project is to be very modular, so things can be switched out and replaced.

Below is a diagram showing all of the parts that could be switched out and replaced between the server and client.



Both the server and the client have a HTTP section and an Obfuscater section. Either of these sections (on both the server and the client) could be switched out with an equivalent pair, and it would continue to function correctly without changing other parts of the program.

5.6 Data Buffer

As the entire project is about transmitting data, a lot of the data transfer is built on 'Data Buffers' which are essentially a bit-stream that encapsulates length, which means it's easy to tell when the bit-stream is complete.

6 Implementation

In this section I am going through all of the areas of the project that I have talked about in the design section, along with discussing programming language choice and the use of libraries.

6.1 Programming Language Choice

For the project, I opted to use Python for a number of reasons:

- Quick to develop and prototype
- Generally Safe
- Solid language design and features

On top of this, I did not know any python when I started the project, and I wanted to learn something new, which I definitely feel I did.
Python also has a wealth of useful libraries.

6.2 Proxy

As discussed previously, the proxy creates and maintains two sockets, and acts as a man in the middle between them, receiving data from one socket, mutating it with the current mutator, and then passing it on to the second socket.

It also verifies the config file passed to it in the following ways:

- Checking the remote server has a port open and that it replies as expected
- Check that the functions defined exist

This helps make sure the project is stable.

The Proxy maintains both sockets by storing a buffer each way of data that is ready to send, but not yet sent. When the data is sent, it checks the sockets to make sure it has been sent successfully, and if it has not been it tears down the socket and rebuilds it. If the socket cannot come back up, it terminates the program.

6.3 Obfuscator

The Obfuscator is split into three parts:

- Generic obfuscation for structured data
- Steganography by mutating the order of lists
- Steganography by appending whitespace to non-whitespace-sensitive fields

6.3.1 Generic Obfuscation

The generic obfuscator that I created can be used to take any input and generate an output of structured data. This is very useful for generating things like HTML, as it is highly structured and forms a tree structure.

As this is for generating structured data, and in the context of this project, HTML, it is only used on the response side of the program to inject valid HTML into the page that is returned from the webserver.

As an example, generating simple mathematical formulae from data is demonstrated below because it is significantly simpler and less verbose than HTML, however the process is the same and it is trivial to understand.

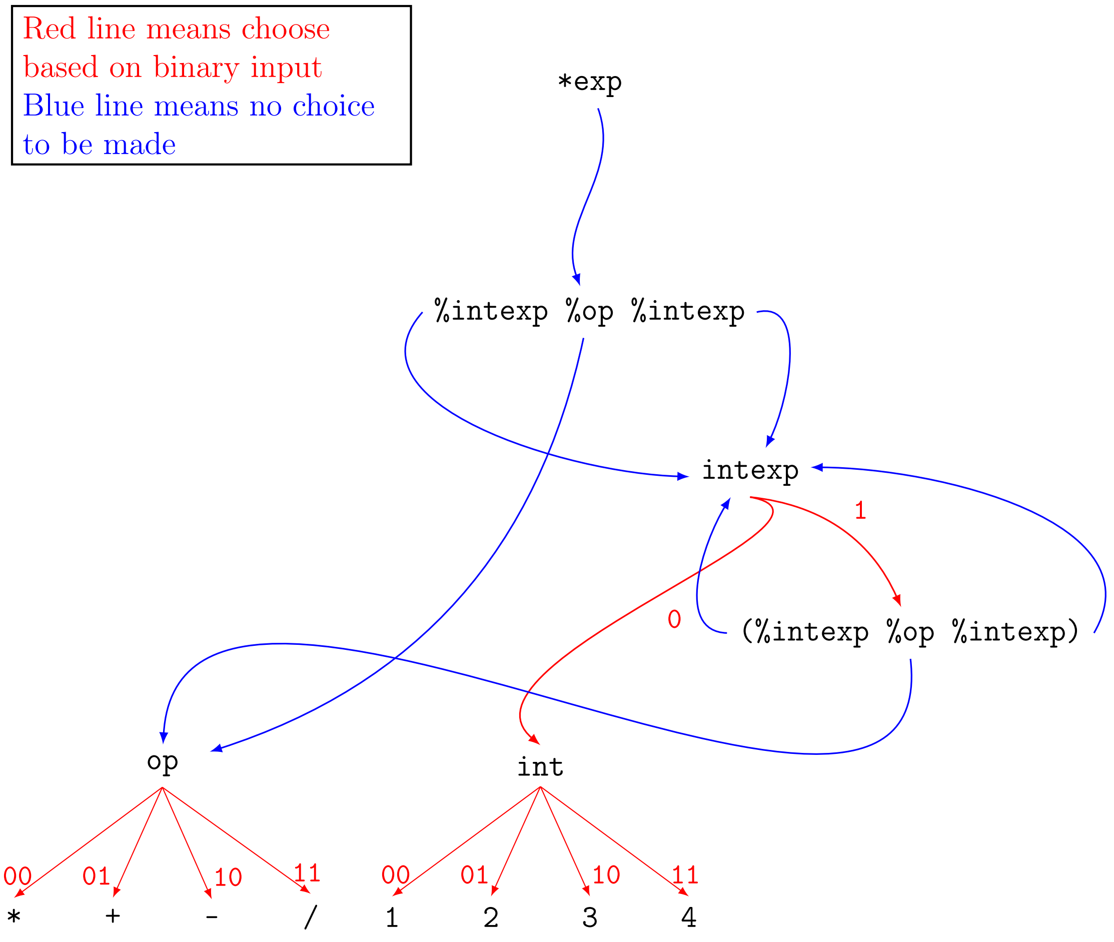
The format defined for simple mathematical equations is as follows:

```
op <- *
op <- +
op <- -
op <- /
int <-1
int <-2
int <-3
int <-4
intexp <-%int
intexp <-(%intexp %op %intexp)
*exp <- %intexp %op %intexp
```

As can be seen here, the format is defined recursively. The format is the built up by selecting the top level element which in this case '`exp`' (the asterisk is only to mark that it is the top level element). `exp` can only be represented by `%intexp %op %intexp`.
`intexp` can be represented by a single integer, or an expression. As there are two choices, the first bit in the bitstream will be selected and this will be used to choose between the two possible choices. When there are more than two choices, multiple bits are used to make the decision. This is made slightly more complicated by the fact that the bytes are read backwards, but when multiple bytes are required, they are interpreted the other way around for ease of decoding.

For example, the ASCII characters 'hi' are represented by the binary '01101000 01101001'. And as the bits are read backwards, they are used in the order: '00010110 10010110' To start with, we have: `%intexp %op %intexp` The `intexp` requires a single bit, and which in this case is `0`, which means it is an `int`. An `int` requires 2 bits to choose, which is `00`. Therefore the first int is 1. The next part is the `op` which also requires two bits, `10`.
The order here is again flipped, which means a `+` is chosen. This continues recursively, and results in the following:
`1 + ((2  4) * 1)`

To make this clearer, the recursive tree that is generated by the format above, along with the choices to navigate each branch are defined below:


Red line means choose based on binary input
Blue line means no choice to be made
*exp
%intexp %op %intexp
intexp
1
0
(%intexp %op %intexp)
op
int
00
01
10
11
*
+
-
/
1
2
3
4

The tree is navigated until either the input is exhausted, which means the stream is considered a stream of 0's, or the tree is completed, in which case a new tree is started.

6.3.2 Appending Whitespace

Appending whitespace to whitespace insensitive fields is a simple way to insert more data into the header of the HTTP request, because as in previous sections it is assumed that if the traffic was being inspected, it would be sanitised before inspection.
Whitespace can be added up to a customisable maximum, which represents data. The way this works is by utilising '`unary`', which is where a number is represented by a number of items. In the project, a maximum of 16 spaces (or 4 bits) is used, for example, 7 spaces out of a maximum of 16 would represent '0111'.

6.3.3 Shuffling the order of lists

HTTP headers are stored as a dictionary, which means that we can reorder them, and the remote server, nor anyone listening to the traffic are likely to notice. If we have a list of 14 elements:
'`A B C D E F G H I J K L M N`' Simple maths would dictate that data can be stored in the order of the list.
In a list of 14 elements, there are 14! or 87178291200 different ways to order the list. This means that there are $\log_2 14!$, or roughly 36 bits, or around 4 bytes of data that can be stored.

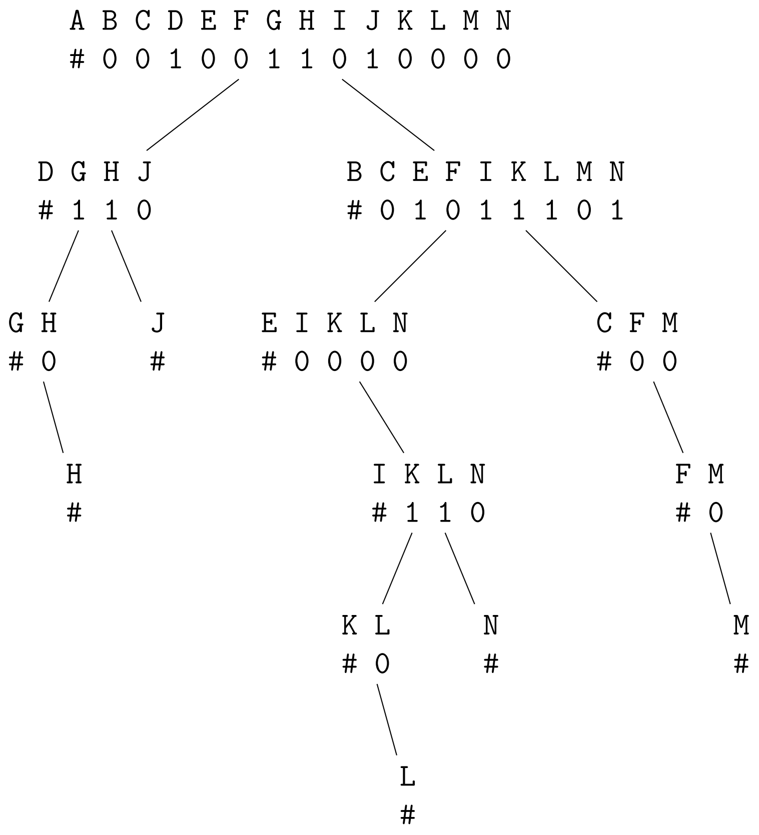
The algorithm that I came up with resembles quicksort, however instead of comparing two elements to each other, the next bit of a bitstream is used to decide the order. Here is some side-by-side python to demonstrate this, and differences are highlighted:

```python
def quicksort(lst):
    if len (lst) <= 1:
        return lst

    pivot = lst[0]
    lst = lst[1:]
    first = []
    second = []
    for item in lst:
        if item <= pivot :
            first.append (item)
        else:
            second.append (item)
        return quicksort(first)
            +[pivot]
            + quicksort(second)
```

```python
def shuffle (lst ,datasource ):
    if len (lst) <= 1:
        return lst

    pivot = lst[0]
    lst = lst[1:]
    first = []
    second = []
    for item in lst:
        if datasource.getbit () == 1 :
            first.append (item)
        else:
            second.append (item)
        return shuffle (first, datasource )
            +[pivot]
            + shuffle (second, datasource )
```

If the datasource defined above is for the word '`data`' (which in binary is: `00100110 10000110 00101110 10000110`*), and it was input into the function, it would be broken down into a tree as follows:

*Note that the binary is least significant bit first, rather than the more conventional most significant bit first.



At each node, the first element (marked with a '`#`') represents the 'pivot' in the quicksort. The binary below each element is for deciding whether the element should go left (`1`) or right (`0`).

The output from encoding 'data' into the order of:
'`A B C D E F G H I J K L M N`' is:
'`G H D J A E K L I N B C F M`'

The binary is put into the tree in a depth first fashion, and in order to get the order of the list back out of the tree, it is traversed in-order.

To get the binary data back out from the reordered list is trivial, the tree is just built up, and then the binary can be easily extracted.

One side effect of building a tree based on binary data is that depending what the data to be encoded is, a different number of bits will be encoded*:

Type of data	English Text	Random Data	Binary 0's	Repeated 'U's
Bits Encoded	482	493	4851	474

* The Random data test was run 100 times and this is the average.The english data test was run on different data from Lorem Ipsum 100 times and this is the average. The binary 0's is a repeated binary steam of 0's.The Repeated U's is binary switching between 1 and 0 (The binary representation of U in ASCII is 0×10101010).

It is notable that English text gets fewer bits encoded than random data, which is very different to different compression schemes.

This idea can be applied to HTTP headers, to get data from the client to the server.

6.4 HTTP Mutator

The HTTP Mutator is the layer that understands HTTP, and where data can be inserted. To make this work, I had to implement a the HTTP protocol to a level where every part of it can be manipulated. I started of by using libraries, but the library support was not there for chunked encoding, or modifying requests and forwarding them on.

Implementing HTTP was not too difficult, as it is a plaintext protocol that is reasonably well defined, however in order to make it work, all of the different compression types have to be handled. I chose to strip the compression out rather than spend time implementing lots of variations on the same compression scheme.

6.4.1 Inserting HTML-encoded data

In order to insert HTML-encoded data in a sensible place, there are a some problems to solve:

- The length of the HTML-encoded data will change
- The length of the initial HTML will change
- The position of the HTML-encoded data needs to be calculated

The solution I came up with to solve the above issues is twofold.

Firstly, the data is encoded with a length appended to the front. The length is in ASCII which while being more inefficient from a data-storage perspective, provides a token that allows the program to be certain that if for a given section of HTML the HTML can be decoded into data, and the data is the same length as expected that it is in fact the data to be decoded.

Secondly, there are only a limited number of places the data can be inserted. The number of valid 'insertion' points (where HTML can be correctly inserted) is counted, and then a function is run which generates the following sequence of numbers:

```
1
12
70
376
1992
10524
55573
293432
1549327
8180453
```

The last valid insertion point is used to insert the data. The code used to generate the numbers is:

```python
def get_position(length):
    current = 1
    next = 1

    while(next < length):
        current = next
        next += current*1.2+2
        next += next*1.4 + 2
        next = int(next)
    return current
```

This sequence was used because it provides a small number of points to test to see if data is present, and it expands exponentially, but starts quite slow. There was a lot of testing and trial and error to get a sensible function. Typically a HTML page doesn't have even `55573` 'insertion' points, however in testing, pages with up to `300,000` insertion points were found.

6.4.2 Chunked Encoding

Chunked encoding differs from standard HTTP by not sending a length of the entire response, but sending it in chunks, and sending the length of each chunk individually. More headers can also be sent with each chunk, and the webserver does not have to inform the client when all of the chunks have been sent. All of these complicate the matter of receiving data via chunked encoding.

To solve these issues, the program concatenates multiple chunks together before inserting the extra HTML-encoded data inside.

6.5 HTTP Client

The HTTP client makes HTTP requests to the server, and data is encoded by changing the order of the headers and appending whitespace, as previously explained.

To make the HTTP Requests, the Python Requests library is used. When requests are added to the request use by the requests library, it does not preserve the order they are added in, so to bypass this, the client modifies the fields directly inside the library.

HTTP is a response/request protocol, so the client has to poll the server at regular intervals to check to see if the server has any data to return. If the server returns any data encoded into the HTML response, the client waits for a short amount of time before requesting another page. This continues until the server returns no data when the client goes back to polling at regular intervals. The same behavior occurs if the client has any data to send to the server, it makes frequent requests until all of the data has been sent.

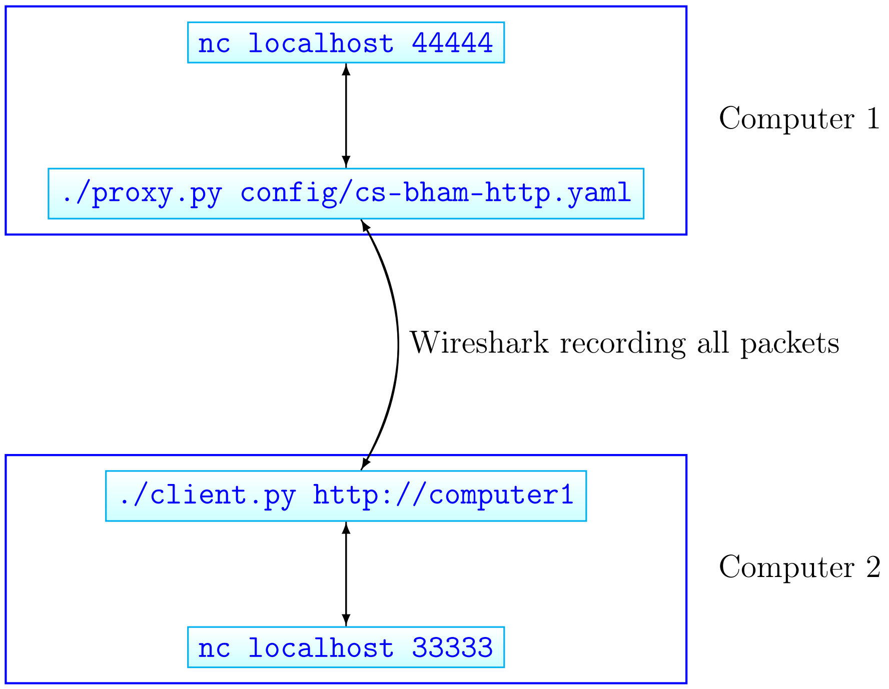
6.6 Complete data flow

6.6.1 Configuration

In this section, an end to end data flow is going to be explained by sending short messages in both directions with a direct connection to the TCP sockets exposed by the client and server of the program. All of the data here is present in a wireshark capture which is submitted as an appendix.

`netcat` is used to connect to listening socket on each computer.

The program has been set up as follows on two separate computers:



nc localhost 44444
./proxy.py config/cs-bham-http.yaml
Computer 1
Wireshark recording all packets
./client.py http://computer1
nc localhost 33333
Computer 2

6.6.2 Encoded Whitespace

The first request with data encoded in it is:

```
GET / HTTP/1.1
Content-Length: 0                     (26)
X-Request-ID: 8a5da39b-a61f-44eb-8952-c19ad81f3817
                                  (41)
Accept-Encoding: gzip, deflate        (13)
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
    (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36 (7)
Pragma: no-cache      (8)
Cookie: _ga=GA1.3.1924440076.1506628216
   (41)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp
   ,image/apng,*/*;q=0.8  (13)
Referer: google.com    (7)
Accept-Language: en-US,en;q=0.9,en-GB;q=0.8   (8)
Connection: keep-alive                           (33)
Host: www.cs.bham.ac.uk (1)
Cache-Control: no-cache              (18)
DNT: 1                      (27)
Upgrade-Insecure-Requests: 1                              (37)
```

The numbers at the end of each line are the number of spaces. For the purposes of demonstration, the number of bits transferred in whitespace have been increased, and there are a maximum of 64 spaces per line, which means 6 bits/header. The first number is 26, which is `11010` in binary. However, this is only 5 binary digits, and each line represents 6, so a 0 is appended to the start, resulting in: `011010`. As previously, the binary is transferred as least significant bit first, which means the order is reversed to `010110`. Therefore, the binary from appending whitespace is the following:
`010110100101101100111000000100100101101100`
`111000000100100001100000010010110110101001`

6.6.3 Reordering of headers

In the example above, the headers are out of order, and data is stored in the order of them. To get the data back out, the list is sorted, and the first item is taken to act as a 'pivot', which is the `Accept:` header. The sorted list is then compared to the original list, and for every element in the sorted list that appears before `Accept:` a 1 is output, and if it appears afterwards, a 0 is output. For the top level, this results in the following binary output: `1000110010011`, and this is recursed upon on each sub-list (before and

after the pivot). When this has been repeated for all of the sub-lists until the sublists are empty, the following data is output:
`1000110010011100010111000010101000`.
This is appended to the start of the previous binary stream, and is half of the data we need for the current data.

6.6.4 Client Side remainder

The rest of the data is in another request, and is:
`1011001110110011101000011011100110101001`
`1010000100010100000000000000000000000000`
`000000000000000000000000000000000000000`
Lots of the data is 0's, and this is because when the whitespace is appended to the headers, if no whitespace is appended, that represents only 0's. To get the initial data out, we append the current data to the end of this, and then reverse all of it.
The first 24 bits of this are:
`100011001001110001011100`
Which is `:91` in ascii. However, this is backwards, and when it is reversed it tells us that the length is 19. Therefore, to get the full data out, we have to read the next $19*8 = 152$ bits.
This means the resulting message is:

```
This is a message!
```

6.6.5 Data encoded in HTML

To encapsulate data in the HTTP Response (from the server to the client), data is encoded into the HTML response.

The config file used to generate the HTML to insert into the page is as follows, and the binary value for each choice is appended to end of each line and highlighted (the order may seem strange, but this is because of the bit-order):

```
PAGE#0.1
DELIM <-

num <- 6 {00}
num <- 7 {10}
num <- 8 {01}
num <- 9 {11}

url <- https://en.wikipedia.org/wiki/Standards-compliant {000}
url <- https://en.wikipedia.org/wiki/The_Remaining_Documents_of_Talaat_Pasha {100}
url <- https://en.wikipedia.org/wiki/Millettia_pinnata {010}
url <- https://en.wikipedia.org/wiki/Lance_Tait {110}
url <- https://en.wikipedia.org/wiki/Doll,_Highland {001}
url <- https://en.wikipedia.org/wiki/Wilf_Spooner {101}
url <- https://en.wikipedia.org/wiki/May_Moustafa {011}
url <- https://en.wikipedia.org/wiki/London_District_Signals {111}

text <- Statement #4 {00}
text <- Rubber ducks are planning world domination {10}
text <- This is text? {01}
text <- Save water, drink beer {10}


3dnum <- %num%num%num

imgbase <- http://lorempixel.com {0}
imgurl <- %imgbase/%3dnum/%3dnum {1}

tags <- <h1>%text</h1> {000}
tags <- <h2>%text<h2> {100}
tags <- <p>%text</p> {010}
tags <- <div>%tags</div> {110}
tags <- <span>%tags</span> {001}
tags <- <img src=%imgurl /> {101}
tags <- <a href=%url>%text</a> {011}
tags <- <a href=%url>%tags</a> {111}

content <- <br /> {0}
content <- %content<br>%tags {1}

*htmlbody <- %content
```

In the example, the data is inserted into position 70, and below is the first section of this:

```
<br />
<br />
<br />
<br />
<br /><br><h1>Statement #4</h1><br><div><span><span><h2>Rubber ducks are planning world domination<h2></
    span></span></div>
<br /><br><div><p>Statement #4</p></div>
<br />
```

The lines with just `<br />` are just the bit `0`, as defined in the file above. The next line is slightly more complex. To calculate the value of it, the '`leaf`' nodes are replaced with their binary value and the type of node like so:

```
%content:0%<br><h1>%text:00%</h1><br><div><span><span><h2>%text:10%<h2></span></span></div>
```

The next step is to parse valid patterns, so for example a `tag` can be a `<h1>%text</h1>`, and repeat.
Each step is shown below:

```
%content:1000000%<br><div><span>%tags:00110010%</span></div>
%content:1000000%<br><div>%tags:00100110010%</div>
%content:1100000011000100110010%
%htmlbody:1100000011000100110010%
```

As `htmlbody` is the top-level, this is the output:
`1100000011000100110010`
Adding in the previous 0's, we get:
`00001100000011000100110010`
This is only a portion of the data in the request, but the entirety of it is:

```
00001100000011000100110010101100100000100111011000100110000000100001011100000101101001
0110110011100011010000000010010010110110011100000010010000110000001000100111010100110
1100111000001110111101100111011011001110101001100111010001010000
```

Which when converted to ASCII is:
`0025And this, is a response.`
The 4 digit number at the start is the length in bytes of the message that has been sent, and the length is sent in ASCII for the reasons described in the previous section.

6.6.6 Note on binary reordering

In the previous sections, there is a lot of reversing the order of bits and bytes etc, this is because in the implementation, I used treated the binary byte order as least significant bit first, which is different to the typical use. There is no real reason for this, and no advantages/disadvantages, aside from that it makes it slightly more difficult to explain.

7 Testing

In this section, I will detail the various ways I tested the project during development and the more in depth testing upon completion.

7.1 Development Testing

As described in the design section, the project was built in layers, and each of these could be testing individually.

7.1.1 Proxy

The proxy was tested in development by using it as a TCP passthrough, and sending files through it and checking the hash of the file at both sides to ensure it was the same.

On top of this, time-sensitive tests were run to ensure that buffering was not massively increasing latency.

The next tests I ran was disconnecting the sockets at both ends at different times and reopening the sockets to see if the connection could be re-established.

7.1.2 HTTP layer

Testing the HTTP layer was done by parsing a variety of HTTP requests from a browser to multiple sites and the sites responses and making sure that the layer correctly understood all of the data that it was being given.

Data was then manipulated in the requests to ensure that both chunked and normal encoding types were supported when the data was manipulated.

7.1.3 Obfuscation layer

Testing the obfuscation layer consisted of lots of tests involving large files and random data to check that data that has been obfuscated can be recovered correctly.

Testing the whitespace was simple, as it is a linear data transfer, and if some data gets through, it can be safely assumed that more will follow. Testing the shuffling is slightly more difficult to test because as the shuffling algorithm is new work, there are no guarantees it would work correctly.

It was tested with large lists and lots of data. Testing the generic obfuscation was a multi stage process, firstly testing that simple obfuscation can work, with large random data files. Once this was fully tested, it was tested with more complex obfuscation and similar large files, finally resulting in being tested with the full HTTP obfuscation and large files.

7.1.4 End to end testing

The end to end testing was similar to testing the proxy, as the goal was for the behavior to be the same, namely reliable data transfer without mounting latency caused by overzealous buffering.

7.2 Application testing

As the project acts as a TCP tunnel, some sensible testing would be to test it with applications that make it useful. The applications that were tested are the following:

- cUrl
- SSH
- SSH Socks Proxy
- OpenVPN
- iperf

7.2.1 Testing setup

The client was set up on one computer, and the server was setup on a second, connected over a fast local network. The webserver that the server connects to was also hosted on the local network.

7.2.2 TCP Server-Server adapter

For all of the following tests as the server exposes a TCP listening socket, and as does the service with which to connect, a bridge is needed between the two, so I made a small program which connects to both servers and forwards all data between them.

7.2.3 Testing with cUrl

cUrl is a program that can be used to download web pages and make HTTP requests. The remote server was pointed at a webserver, and then curl requests could be made to the local listening port.
cUrl was able to get web pages of varying sizes correctly.

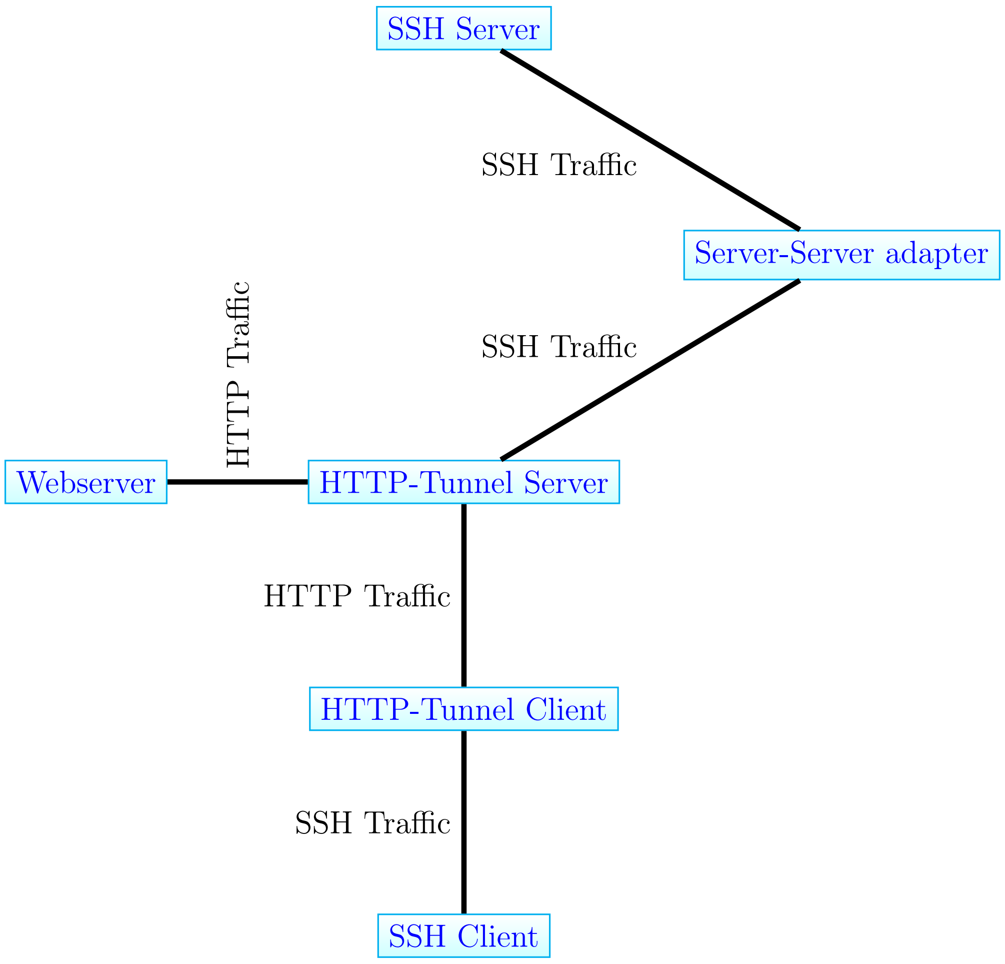
7.2.4 Testing with SSH

SSH is a program that can be used to remotely manage computers and perform tasks on them. The remote server was pointed at an SSH server, and ssh was used to connect to the local socket and make an SSH connection. It was possible to do server administration and even file editing with programs like `vim`, and the responsiveness that was provided was surprising, given the very limited bandwidth that was available.

7.2.5 Testing with SSH Socks Proxy

A feature that SSH provides is the ability to advertise a local socks proxy on the client side. A socks proxy receives HTTP requests for remote webservers and forwards the requests on to get the response. When testing with this, the setup was similar to the SSH test previously, except the client connected and acted as a SOCKS5 client.

The below diagram is applicable for both SSH tests, and provides a good overview of how all of the parts connect for all tests.



7.2.6 OpenVPN

OpenVPN is the main open source VPN server and client.
To test OpenVPN, it was setup and tested as a server and client pair separately to verify the configuration was correct.

OpenVPN was then used on top of the HTTP-Tunnel.

A connection was successfully established, and data is transferred, however there are a couple of issues:

- OpenVPN adds substantial overhead, and the connection is incredibly slow
- OpenVPN sometimes detects it is being tunneled and thinks it is an attack on TCP

The first issue is to be expected, as a VPN is quite a heavy piece of software, and it was possible to access services over the VPN link. The second issue was intermittent, and as far as I can work out is caused by a combination of a few things:

- OpenVPN reads and writes packets directly to the wire, so treats TCP as a datagram based protocol rather than as a stream based protocol
- When the link gets saturated packets get fragmented and concatenated by my program
- As the packets get broken up the HMAC for each datagram becomes invalid

The errors I got when running openvpn are the following:

```
    Wed Apr  4 03:47:31 2018 127.0.0.1:48830 TLS Error: cannot locate HMAC in incoming
Wed Apr  4 03:47:31 2018 127.0.0.1:48830 Fatal TLS error (check_tls_errors_co), restar
...
Wed Apr  4 03:49:32 2018 127.0.0.1:49474 WARNING: Bad encapsulated packet length from
Wed Apr  4 03:49:32 2018 127.0.0.1:49474 Connection reset, restarting [0]
Wed Apr  4 03:49:32 2018 127.0.0.1:49474 SIGUSR1[soft,connection-reset] received, clie
```

They seem to be linked and caused by the issues highlighted above.
It may have been possible to debug and solve all of the issues, however the throughput from OpenVPN was so low, it was almost unusable so there was not much point.

7.2.7 iperf

Iperf is a TCP-based performance tester. I setup a listening server on the remote end, and a client on the local server. Over a variety of tests, I was able to get 32.7kb/s download, and 17.8kb/s upload.

8 Project Management

9 Discussion

10 Conclusion

11 Bibliography

References

[1] Johnson Neil F and Jajodia Sushil. Exploring steganography: Seeing the unseen. *Computer*, 31(2):26–34, Feb 1998.

[2] Provos N and Honeyman P. Hide and seek: an introduction to steganography. *IEEE Security & Privacy*, 99:32–44, June 2003.

[3] Greg Farnham. Detecting dns tunneling. *SANS Institute Reading Room*, Feb 2013.

[4] Fielding R, Irvine UC, Gettys J, Compaq/W3C, Mogul J, Compaq, Frystyk H, W3C/MIT, Masinter L, Xerox, Leach P, Microsoft, and Berners-Lee T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999.

[5] Born Kenton and Gustafson Dr. David. Detecting dns tunnels using character frequency analysis. *CoRR*, abs,1004.4358, 2010.

[6] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, RFC Editor, June 2014.

[7] University of Southern California Information Sciences Institute. TRANSMISSION CONTROL PROTOCOL. RFC 7230, RFC Editor, September 1981.

[8] Merriam-Webster Online:. Steganography, February 2018.

[9] Merriam-Webster Online:. Obfuscate, February 2018.

[10] P Christensson. Encryption, November 2014.

[11] Bruce Schneier. Memo to the amateur cipher designer, October 1998.

[12] Bruce Schneier. The value of privacy, May 2006.

[13] CHARLES DUHIGG. How companies learn your secrets, February 2012.

[14] AATIF SULLEYMAN. Google lets you report ads that know âĂŸtoo muchâĂŹ about you, September 2017.

[15] B. Gleeson, A. Lin, Nortel Networks, J. Heinanen, Telia Finland, G. Armitage, A. Malis, and Lucent Technologies. A Framework for IP Based Virtual Private Networks. RFC 2764, RFC Editor, February 2000.

12 Appendices