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

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 inside the HTTP body and headers in multiple ways so 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 his support and guidance throughout the project. Additionally, I would like to thank my housemates, friends, family, kettle and coffee machine for their support throughout both this project and my degree as a whole.

Contents

1 Introduction

Censorship and monitoring of web traffic is a common occurrence around the world, as is deep packet inspection which is used to ensure that traffic on port 80 is HTTP (Hypertext Transfer Protocol) 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. Therefore it would be useful to be able to tunnel data over HTTP and make it virtually impossible to tell that additional data is being tunnelled through the protocol.

1.1 Aims

The aim of this project is to be able to tunnel data, and by extension operate a VPN over 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

In order to perform the above, a server and client pair would need to be created, a server that acts as an HTTP server, and can receive connections from a client that will act as the HTTP client.
In formal terms, the aims of the project are as follows:

1. To hide data in HTTP requests and responses
2. For the server to be able to function as a valid HTTP Server
3. To be able to communicate between the server and client using only HTTP
4. To make traffic appear as valid HTTP

As an example, it could be valuable to be able to SSH to a remote server, and make it very difficult for a third party to tell that a non-HTTP connection has been made.

The benefit of the server acting as a valid HTTP server is that if it is being investigated, the server could be browsed to, and it would not be obvious that it is not a real HTTP server, or at least that HTTP is not the only function of the server.

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
    - Explanation 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
    - Conclude whether or not the project was successful

2 Background

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

2.1 HTTP

HTTP is the most important protocol in this project and as such needs to be covered in detail. `HTTP/1.1` is defined in `RFC2616` [1] and all of the information about HTTP originates 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 the above request, the request line comes first (`GET / HTTP/1.1`), followed by the headers. The headers are a set of key-value pairs which form a dictionary that tells the server extra information about the client.

One of the headers is:
`User-Agent: curl/7.58.0`
Which is the client telling the server that the `User-Agent` of the program used to get the page is `curl` version `7.58.0`.

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 response, the response line (the first 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.

2.1.1 HTTP Encoding types

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

`Transfer-Encoding` additionally allows for `Chunked` encoding, as defined in RFC7230 [2], which splits the page or portions of data up into multiple sections, and sends the length for each section and the sections themselves separately. This is done that so web servers can send data as it is generated, rather than having to buffer the entire page. Chunked Encoding is built into HTTP/1.1, so that the client cannot request that the server does not use it.

2.2 TCP

TCP (Transmission Control Protocol) is defined in `RFC793` [3] and all of the information about TCP originates 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`. The passive, or listening socket is opened by a server and waits for a connection from an active, or client socket.

2.3 Steganography

Steganography is defined as [4]:

> '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 an HTTP stream.

2.4 Obfuscation

To obfuscate is defined as [5]:

> 'to be 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 makes data difficult to read and understand.

2.5 Encryption

> 'Encryption is the process of converting data to an unrecognisable or 'encrypted' form. It is commonly used to protect sensitive information so that only authorized parties can view it [6].'

This definition means that if some data has been encrypted with a sensible encryption scheme, it will be very difficult for a third party to decrypt. There are many types of encryption, AES and RSA are two major examples.

It is widely referred to as bad practice to '`roll your own crypto`' [7], which means that it is a bad idea to write a bespoke cryptographic algorithm, or to try and come up with your own cryptographic scheme.

2.6 Privacy

Privacy on the internet is often overlooked, however, it is vitally important, as explained by Bruce Schneier on his blog, 'Schneier on Security' [8]. 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 a person and build a model to predict their behavior, which can then be used for highly targeted advertising — not always desirable for the end user. An extreme example of highly targeted advertisement was a case in 2012 where a retailer inadvertently told a father that his daughter was pregnant by sending her coupons for baby clothes and cribs. Since this case was first reported by The Independent [9], statistical models have become significantly more complex and intrusive. Google, who are the worlds biggest advertiser, recently allowed users to mark Adverts as 'knowing too much' [10], which demonstrates that the problem exists, and is perhaps endemic in the advertising industry. As this case shows, lack of privacy is scary, and tools that increase privacy can only be a good thing.

2.7 Blocking of Services

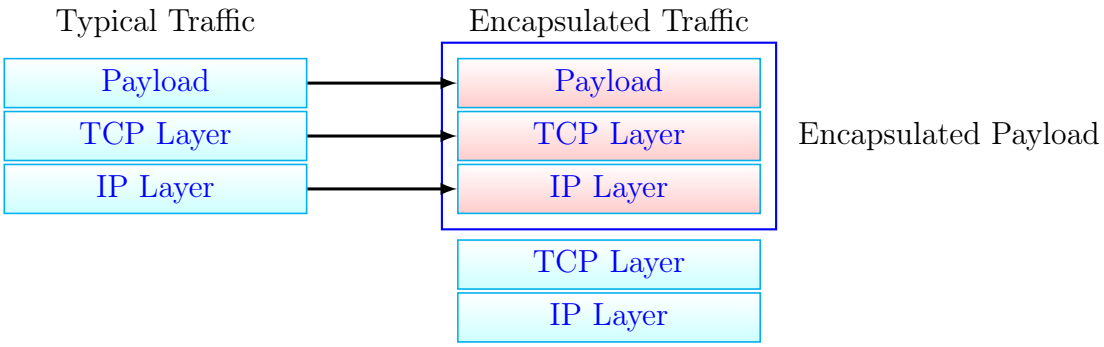
When you connect to Public Wi-Fi, the provider will often limit what content you have access to, typically by blocking ports or preventing access to certain websites, often known as a blacklist.
HTTP is run over port 80, which is often one of the only ports available.

2.8 Data Tunnelling

In the field of networking in Computer Science, networking is split into layers. Tunnelling is when some of the layers are encapsulated and sent as a regular payload. A real-world representation of data encapsulation could be putting a letter inside an envelope, addressing it, stamping it, and then putting that envelope inside a second larger envelope.

The larger envelope can then be addressed, stamped and posted independently. On receipt of the larger envelope, the encapsulated envelope can then be taken out and posted. This has the effect of concealing that two parties are communicating, and is a good analogy of how encapsulation on a computer network functions.

Below is a diagram explaining how tunnelling works:



2.9 VPN

A VPN or Virtual Private Network (defined in `RFC2764` [11]) is a piece of software, split into a server and a client, where the server and client (or clients) form a network which is both virtual and private.
A typical example of a VPN is OpenVPN, `https://openvpn.net/`.
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 has to only 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, whereas `Ethernet` frames can be read from and written to a `tap` device.

Tun and Tap devices are virtual devices that exist on a computer, packets routed down them can be read from the device as though it is a file. Whichever device type is used, the packets or frames are transmitted through an existing connection to a remote client or server.

3 Existing Work

In this section, related works are explored, and some literature surrounding steganography, HTTP, DNS tunnelling and detecting tunnelled traffic is discussed.

There has been much research done regarding steganography to hide data inside networking protocols, and a large proportion of this work has been done on DNS, as it is often below the radar for firewalls and checking if data is being exfiltrated.

3.1 Image Steganography

Steganography is most commonly used for hiding data in unused or unimportant areas of data [12], the most common place to hide data is inside images. This is the case because not all of the data in an image is required for a human to understand it, and the human eye is very good at filtering out noise. Basic image steganography is carried out by changing the least significant bit or bits in the data, whereas more advanced approaches to steganography can involve identifying redundant data in images [13] which is better, because it is more difficult to detect using steganalysis.

Steganography that is hidden from computers and is hidden from people are quite different things, and can require quite different approaches, it is therefore a much more significant challenge to hide data from both.

3.2 DNS Steganography

It is possible to hide data in DNS requests, otherwise known as DNS Tunnelling, which is often used to get around firewalls and hide which websites are being accessed, as Greg Farnham discusses in his paper 'Detecting DNS Tunnelling' [14]. The main benefit of this is that it often works when there is no direct access to the internet because DNS requests are often propagated through firewalls. The paper mentioned previously highlights the point raised in the previous section, that it is very easy for a human to look at the data and see it is abnormal, 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 tunnelled. DNS tunnelling as described in the aforementioned paper has a few advantages and disadvantages. The key advantage is that it 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 large amounts of overhead. DNS tunnelling is slow because there is a very limited amount of data that can be encapsulated inside DNS requests and responses, DNS requests are never hugely fast and the results can become cached.

3.3 HTTP Protocol

The HTTP 1.1 Protocol [1] is a protocol that describes how data is sent to and from a client, and it contains many areas where data could be included, such as: Images, HTML, CSS, Javascript, Binary files, and more, all of which can be used to hide data.

The HTTP headers could also be an area where data could be concealed, as they are whitespace insensitive, and they represent a dictionary which means that the order in which they are sent does not matter.

3.4 Detecting Tunnelled Traffic

There are a variety of ways to detect traffic being tunnelled, one of the most common is to perform entropy analysis [14]. Entropy analysis is looking at the '`randomness`' of the data to determine whether there is compressed or encrypted data within a stream. This works because compressed and encrypted data both strive to appear random (for different reasons), which means that they will have higher entropy than English text.

Another way of performing the lookup is to do character frequency analysis [15]. Frequency analysis looks at the difference in frequency of letters in expected tokens or English words and in random data. Frequency analysis is similar to but not quite the same as entropy analysis but can be more effective. This method of analysis is often more effective because English words or tokens have slightly different frequency characteristics to large passages of text, so determining whether something is likely to appear in a domain name is slightly different to determining whether it will appear in a block of text.

4 Specification

4.1 Functional Requirements

1. The system must consist of a server and a client
2. The system must allow duplex data transfer between the server and the client
3. The communication between the server and the client must all occur encapsulated within the HTTP protocol
4. The server must act as a valid HTTP server, if connected to by a standard HTTP client
5. The system should be able to serve multiple clients at a time without interference
6. The throughput of the system should be high enough to enable browsing of simple websites

4.2 Non-functional Requirements

1. The extra data encapsulated within the HTTP protocol must be carried out in such a way that is looks like typical HTTP traffic and cannot be spotted by conventional tools
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 and Architecture

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 which means that it is easier to maintain the codebase and also makes it easier to add functionality and repurpose the existing code in the future.

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. While this approach would get around many web filters, it would only require trivial inspection to determine that it is in fact not real HTTP traffic. In order to look more realistic, data is inserted into real websites and the client will exhibit realistic browsing patterns. Additionally, the server will act as a real website when accessed with a typical HTTP client.

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, the client uploading, or sending data to the server is called the 'Request' side, and when the client downloads, or receives data from the server it is the 'Response' side.

Putting data in the response side of the connection is trivial, as when a user makes 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 more difficult because when a user is browsing a website, the request does not significantly change, and it would be 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 does not matter and that they are also insensitive to whitespace.

5.2 Server Overview

The server is split into multiple layers:

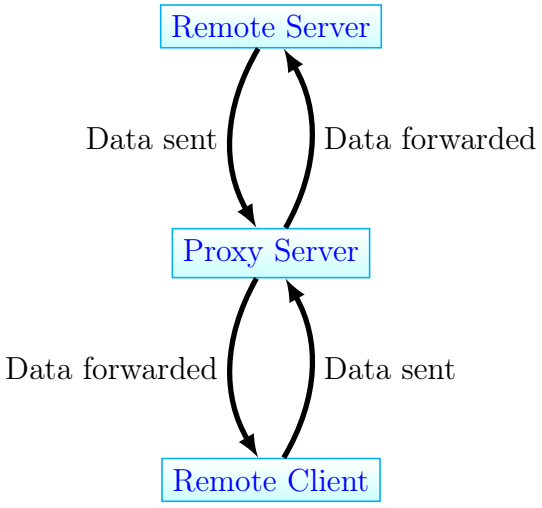
- Proxy Layer
- HTTP Layer
- Obfuscation Layer

Some of the reasons for choosing this design are:

- It allows the proxy layer to deal with keeping the connection alive, and the upper levels can assume that if they are running all of the resources they require are available
- It allows ease of customisation, because each layer can easily be switched out or modified
- It allows the proxy layer to provide a 'data storage' to the upper layers, which is provided thread safe and reliably so they can access current data about the session

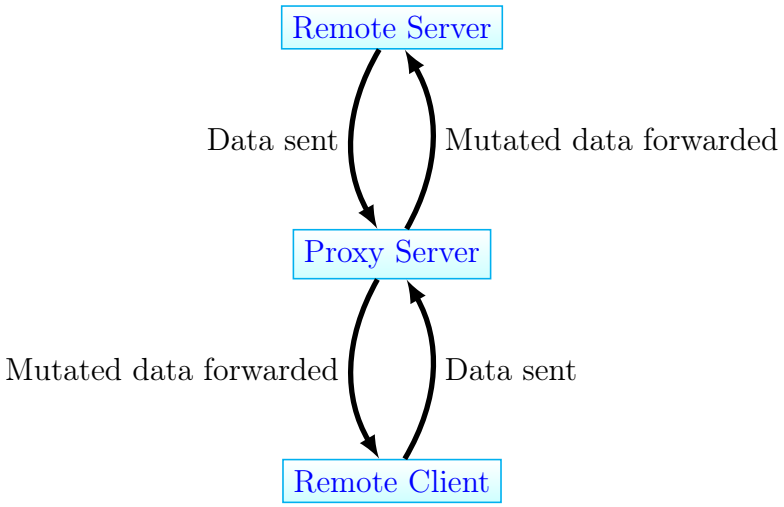
5.3 Proxy Layer

The proxy layer forwards data between a remote server, and a remote client and acts as a manager for the two duplex connections that are active. This creates a TCP listening socket which when it receives a connection, 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 completely independently, one from the client to the server, and one in the return direction.

This design was chosen because it allows traffic to be '`mutated`' on the fly by the proxy server:



5.3.1 Configurable Options

In the proxy layer, there are lots of options that are customisable.
Here is an example configuration file, and the options are explained below:

```
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, and it is defined as follows:

- '`name`' field is a unique identifier.
- '`remote`' and '`remote-port`' specify the remote host and port to connect to.

- '`local-port`' specifies the local port to listen on.
- '`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 implementation of new features was made significantly simpler. It has also left significant scope for future expansion of the project.

5.4 Obfuscation Layer

The obfuscation layer consists of three parts:

- Generic structured obfuscation
- List-order obfuscation
- Whitespace obfuscation

Generic Structured Obfuscation

Generic structured obfuscation is where a structure is defined, and then data can be placed into the defined structure. The structure is defined in a configuration file, called a '`page`' file, 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 below, and it essentially defines a grammar of the language, however the important elements are explained later.

```
PAGE#0.1
DELIM <-

bin <-0
bin <-1

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

This is a simple example that will print out a byte of the input at a time, but in binary. The obfuscation was designed like this because it makes it easy to modify, extend and debug. The obfuscation is completely reversible, and more details about how it works will be explained in the implementation section.

The Obfuscation layer also includes List-Order obfuscation, which stores data in the order of a list, and Whitespace Obfuscation which is when binary data is converted to unary and appended to the end of elements as whitespace.
Both of these are explained in detail in the implementation section.

5.5 HTTP Layer

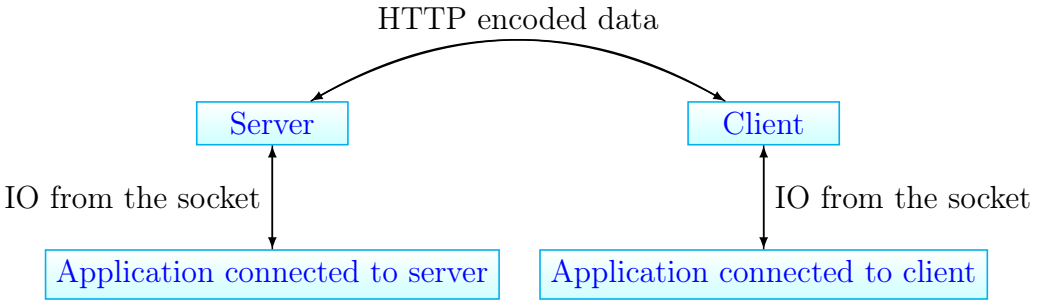
The HTTP layer consists of a '`mutator`' (as mentioned in the previous section), and a corresponding client. The HTTP layer understands the HTTP Protocol, and how data can be inserted into it.

The mutator is used to modify data on the HTTP layer.

The layer also calls the '`obfuscator`' which is used to insert data into the HTTP protocol.

The HTTP layer also contains the data endpoints, which are a pair of listening HTTP sockets, one on the server and one on the client.

This is where the data to be encoded is read from, and after decoding on the remote end, written to.

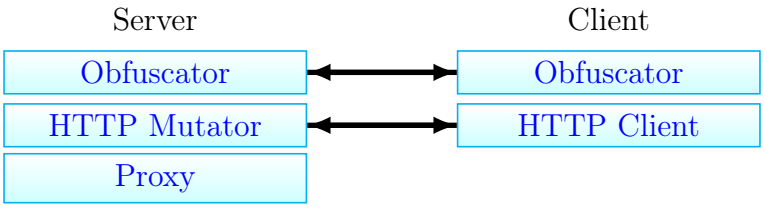


Using listening ports instead of directly interfacing with an application means that there is significantly more flexibility with regards to the applications that it can be used with, because multiple applications can be connected, with minimal configuration.

5.6 Modular Sections Overview

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

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 an HTTP section and an Obfuscator section. Either of these sections (on both the server and the client) could be switched out with an equivalent pair, for example if the HTTP layer was replaced with a DNS layer to tunnel data over DNS, the program would continue to function correctly without changing other parts of the program.

5.7 Data Buffer

As the entire project is about transmitting data, a lot of the data transfer is built on 'Data Buffers'. These are a bit-stream that encapsulates length, which means it is easy to tell when the bit-stream is complete because the correct number of bits will be received, and the next length header will not have been received.

5.8 Server-Server Adapter

On the client and server side of the project there is a TCP listening socket, which provides flexibility, however most applications form a server/client pair which means one is listening and the other connecting. Therefore, an adapter is needed to connect two listening sockets together — one on the project and the other on the application.

6 Implementation Details

In this section all of the areas of the project that have been talked about in the design section are discussed in detail.

6.1 Programming Language Choice

For the project, Python was chosen for a number of reasons:

- Quick to develop and prototype
- Generally Safe
- Solid language design and features
- 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. The sockets and remote server are defined in a configuration file, which is verified in the following ways:

- Checking the remote server has a port open and that it replies as expected
- Check that the specified mutator functions 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 sockets and rebuilds them. If the socket cannot come back up for any reason, 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 was created as part of the project can be used to take any input and generate an output of any highly structured data which can be defined in the format used by the program.

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.

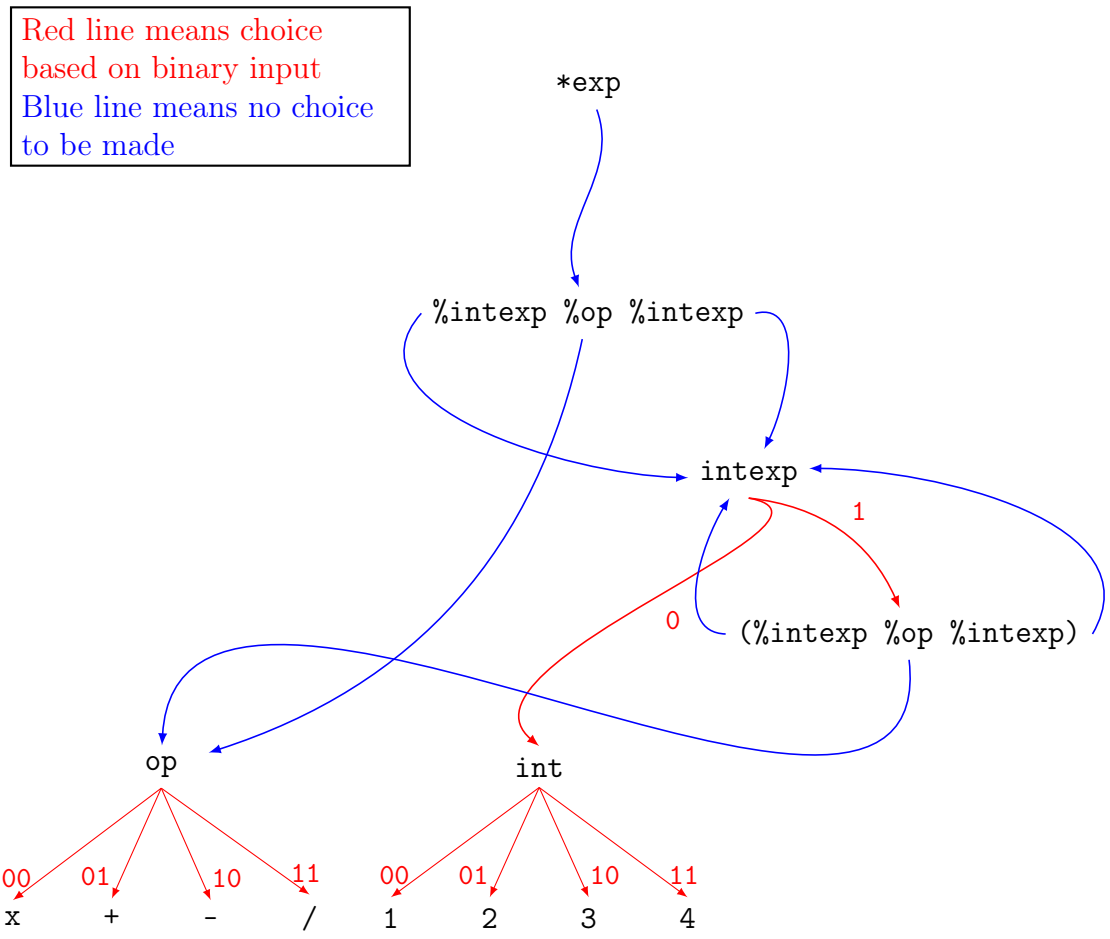
As an example, generating simple mathematical formulae from data is demonstrated below, because it is significantly simpler and less verbose than HTML, however the process is the same.

The format defined for simple mathematical equations is as follows:

```
1  op <- x
2  op <- +
3  op <- -
4  op <- /
5  int <-1
6  int <-2
7  int <-3
8  int <-4
9  intexp <-%int
10 intexp <-(%intexp %op %intexp)
11 *exp <- %intexp %op %intexp
```

As can be seen here, the format is defined recursively. A tree representing this format and how it is interpreted can be seen on the next page. The format is the built up by selecting the top level element which is 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, `%int`, or an expression, `(%intexp %op %intexp)`.
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, as demonstrated in the tree on the next page. When there are more than two choices, multiple bits are used to make the decision, however, when multiple bits are required, the order of the bits is reversed.
For example, the ASCII characters 'hi' are represented as: '`01101000 01101001`'.
When bits are read from the bitstream, they are consumed least significant bit first, which means the order they are used in is: '`00010110 10010110`'
To start with, we have:
`%intexp %op %intexp`
The `intexp` requires a single bit, which in this case is `0` as that is the first bit of the input. A `0` means that the first of the two choices are chosen, resulting in an `int`. An `int` requires 2 bits to choose what will represent it and the next two bits of the input are `00`, again selecting the first option. 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 that a `+` is chosen. This continues recursively, and results in the following: $1 + ((2 - 4) \times 1)$

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:



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. 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 List Order Shuffling

HTTP headers are stored as a dictionary, which means that we can reorder them without the remote server, or anyone listening to the traffic being likely to notice. In a list of, for example, 14 elements, there are 14! or 87178291200 different ways to order the list. This means that there are $\log_2 14!$, roughly 36 bits, or around 4 bytes of data that can be stored.

The algorithm devised resembles Quicksort, however instead of comparing two elements to each other, the next bit of a bitstream is used to decide the order.
Below is some side-by-side python to demonstrate this, with the differences highlighted:

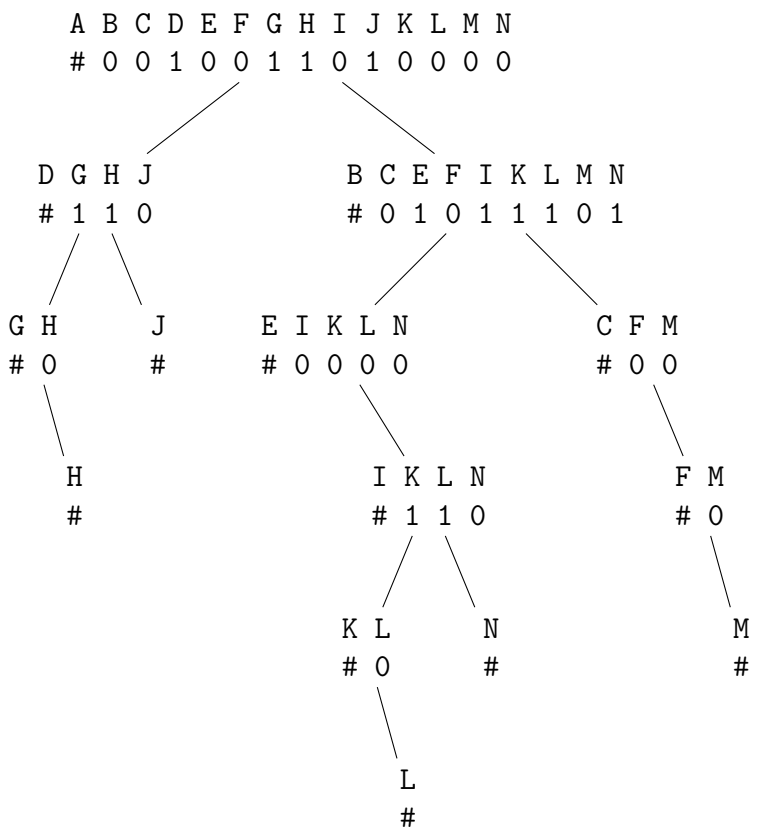
```python
def quicksort(lst):
    if len(lst) <= 1:
        return lst

    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)
```

An example list could be:
'A B C D E F G H I J K L M N'
If the datasource defined above is for the word 'data' (which in binary is: 00100110 10000110 00101110 10000110*), was input into the function, along with the list defined above, 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 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.

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'

To retrieve 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 can be encoded*, which means that different amounts of data would be sent with each request.

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 0b01010101).
The data above was encoded into lists of length 100.

It is notable that English text gets fewer bits encoded than random data, which is very different to typical compression schemes where English text is more compressible. This benefits this application, as the typical data that will be encoded will be more equivalent to random data because it will be compressed or encrypted.

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, the HTTP protocol had to be implemented to a level where every part of it can be manipulated. Using libraries was initially attempted, but the library support was not available 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. It was decided 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 could change
- The length of the initial HTML could change
- The position of the HTML-encoded data could change

The solution used to solve the above issues is twofold.

Firstly, the data is encoded with the 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 where 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:

```
def get_position(length):
    current = 1
    nextnum = 1

    while(nextnum < length):
        current = nextnum
        nextnum += current*1.2+2
        nextnum += nextnum*1.4 + 2
        nextnum = int(nextnum)
    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 slowly. There was a lot of trial and error to get a sensible function. Typically a HTML page does not even have `55,573` 'insertion' points, however in testing, pages with up to `300,000` insertion points were found.

HTTP has a feature called chunked encoding which differs from standard HTTP by not sending the 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 headers are added to the requests created by the requests library, it does not preserve the order they are added in thus 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.

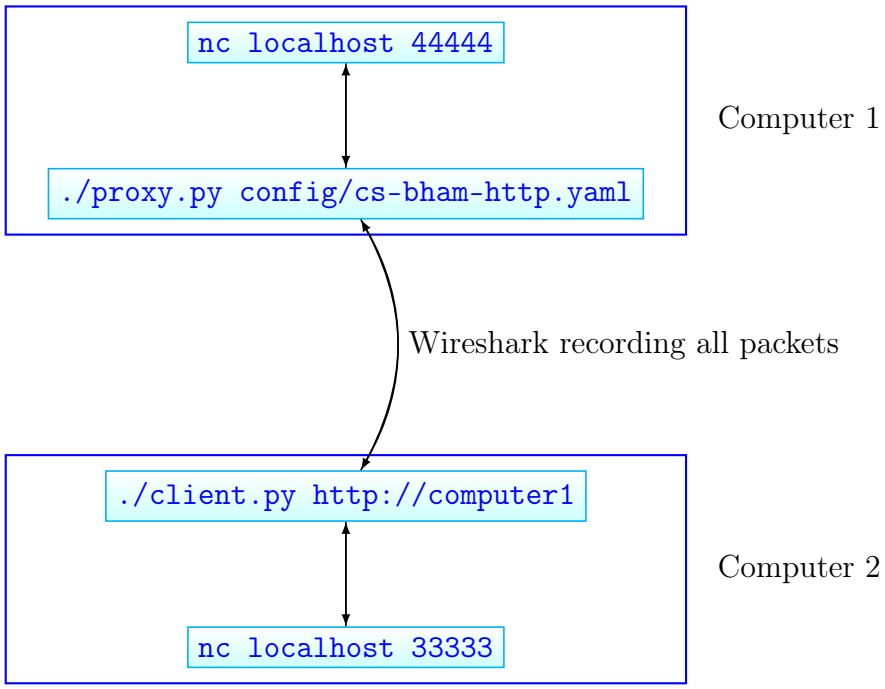
6.6 Complete Data Flow

In this section, an end to end data flow will 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 appendix 12.2. The focus is on decoding the data, but the process is the same for encoding.

6.6.1 Configuration

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

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



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 fully computed 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 alphabetically, 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 sub-lists are empty, the following data is the result:
`10001100100111000101111000010101000`.
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 Result

The rest of the data is in another request, and is:
`10110011101100111010000110111001101010010`
`10100001000101000000000000000000000000000`
`00000000000000000000000000000000000000000`
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 be not as expected, but this is because of the bit-order):

```
1  PAGE#0.1
2  DELIM <-
3
4  num <- 6 {00}
5  num <- 7 {10}
6  num <- 8 {01}
7  num <- 9 {11}
8
9  url <- https://en.wikipedia.org/wiki/Standards-compliant {000}
10 url <- https://en.wikipedia.org/wiki/The_Remaining_Documents_of_Talaat_Pasha {100}
11 url <- https://en.wikipedia.org/wiki/Millettia_pinnata {010}
12 url <- https://en.wikipedia.org/wiki/Lance_Tait {110}
13 url <- https://en.wikipedia.org/wiki/Doll,_Highland {001}
14 url <- https://en.wikipedia.org/wiki/Wilf_Spooner {101}
15 url <- https://en.wikipedia.org/wiki/May_Moustafa {011}
16 url <- https://en.wikipedia.org/wiki/London_District_Signals {111}
17
18 text <- Statement #4 {00}
19 text <- Rubber ducks are planning world domination {10}
20 text <- This is text? {01}
21 text <- Save water, drink beer {10}
22
23
24 3dnum <- %num%num%num
25
26 imgbase <- http://lorempixel.com {0}
27 imgurl <- %imgbase/%3dnum/%3dnum {1}
28
29 tags <- <h1>%text</h1> {000}
30 tags <- <h2>%text<h2> {100}
31 tags <- <p>%text</p> {010}
32 tags <- <div>%tags</div> {110}
33 tags <- <span>%tags</span> {001}
34 tags <- <img src=%imgurl /> {101}
35 tags <- <a href=%url>%text</a> {011}
36 tags <- <a href=%url>%tags</a> {111}
37
38 content <- <br /> {0}
39 content <- %content<br>%tags {1}
40
41 *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:
`0000110000001100010011001`
This is only a portion of the data in the request, but the entirety of it is:

```
0000110000001100010011001010110010000010011101100010011000000100001011100001011010010
0110110011100011010000000100100101101100111000000100100001100000010001001110101001100
1100111000001110111101100111011010011101010011001110100010100000
```

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.

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, the binary byte order is 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, the various ways the project was tested during development are detailed along with the more in depth testing upon completion. Unless otherwise stated, all of the testing was carried out successfully on the submitted project.

7.1 Development Testing

As described in the design section, the project was built in layers, and each of these have been tested individually.

Proxy

The proxy was tested in development by using it as a TCP passthrough by transmitting files through it, and verifying the files were identical before and after transmission. On top of this, time-sensitive tests were run to ensure that buffering was not massively increasing latency.

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

HTTP Layer

Testing the HTTP layer was done by parsing a variety of HTTP requests from a browser to multiple websites and the sites responses, ensuring that the layer correctly interpreted 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.

Obfuscation Layer

Testing the obfuscation layer consisted of lots of tests involving large files of 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, there are no complexities or edge cases that could cause issues with certain inputs, which means that large scale testing as with the other parts was not necessary. Testing the shuffling was carried out more methodically and extensively than other sections during development because as the shuffling algorithm is new work, there are no guarantees it would work correctly or even at all.

It was tested with large lists and large quantities 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.

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

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

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 a small program was created which connects to both servers and forwards all data between them.

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.

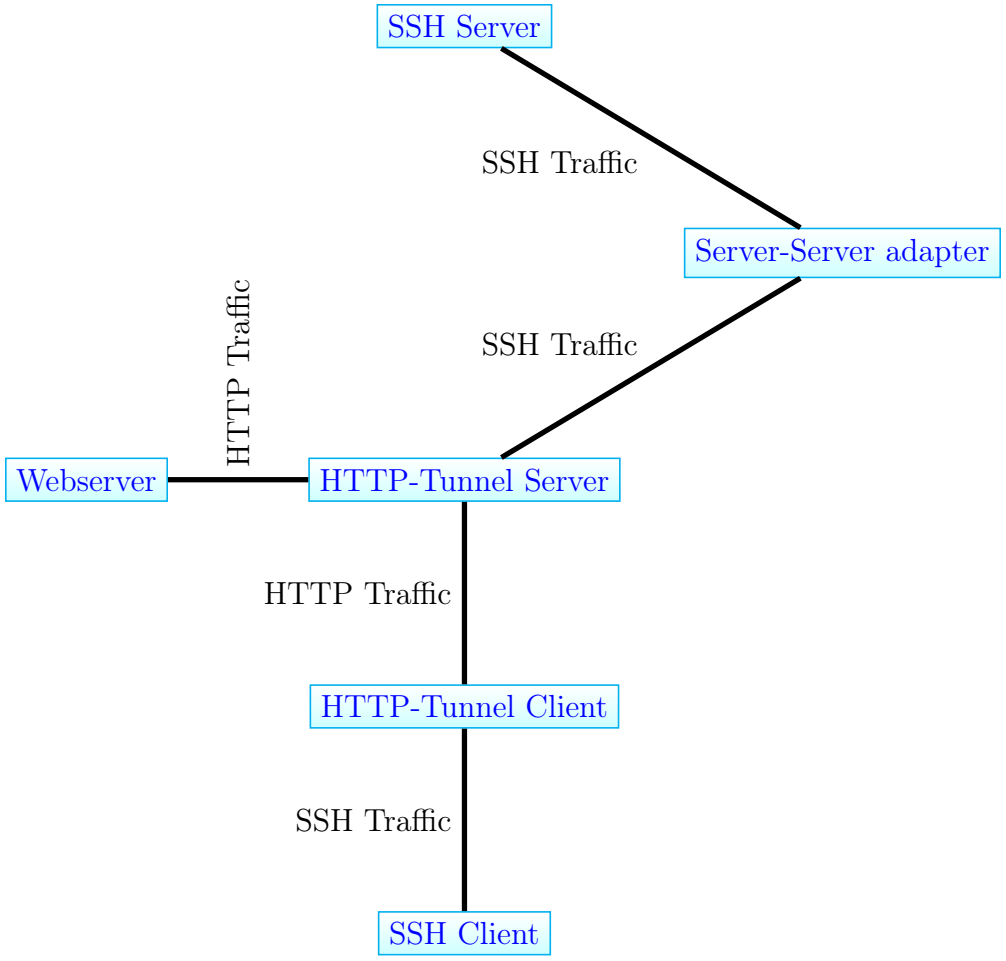
Testing With SSH

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.

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. All of the blue boxes represent separate processes and can occur on separate, or the same computers.



Testing With OpenVPN

OpenVPN is the most popular 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 tunnelled 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 the precise cause is unknown, however it seems to be caused by a combination of a few factors:

- 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 produced 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 packet from [AF_INET
    ]127.0.0.1:48830
Wed Apr 4 03:47:31 2018 127.0.0.1:48830 Fatal TLS error (check_tls_errors_co), restarting
...
Wed Apr 4 03:49:32 2018 127.0.0.1:49474 WARNING: Bad encapsulated packet length from peer (39657), which
    must be > 0 and <= 1627 -- please ensure that --tun-mtu or --link-mtu is equal on both peers -- this
    condition could also indicate a possible active attack on the TCP link -- [Attempting restart...]
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, client-instance
    restarting
```

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 it would serve little purpose and the documentation regarding the issue is conspicuously absent.

Testing With iPerf

iPerf is a TCP-based performance tester. A listening server was setup on the remote end, and a client on the local server. Over a variety of tests, the average performance was 32.7kb/s download, and 17.8kb/s upload.
While these speeds are not blisteringly fast, they are reasonable and roughly what can be expected as the overhead is so great.

7.3 Functional testing

The three initial aims for the project were:

1. To hide data in the HTTP requests and responses

2. For the server to be able to function as a valid HTTP server
3. To be able to communicate between the server and client using only HTTP
4. To make the traffic appear as valid HTTP

In this section, the aims are tested.

To hide data in the HTTP requests and responses

This aim was fulfilled, and is tested by transmitting data between the server and client.

For the server to be able to function as a valid HTTP Server

This aim was fulfilled, as the HTTP server could be communicated with by using a typical web browser and it would respond correctly.

To be able to communicate between the server and client using only HTTP

With only port 80 (HTTP) available to communicate on, the server and client were able to communicate and send data.

To make the traffic appear as valid HTTP

To test this, a network was created with a blacklist of websites, and only port 80 (HTTP) and DNS was allowed to leave the network.

Using the project, SSH was able to access remote servers, and a SOCKS proxy was able to be used to access the blocked websites.

A PFSense router was used as router/firewall for the network previously described, and PFSense comes with a suite of tools for deep packet inspection, and all of the traffic was shown as being HTTP traffic.

This aim was fulfilled, as even the sophisticated analysis tools packaged with PFSense were unable to detect that the traffic is not HTTP.

8 Project Management

8.1 Development Principles

When developing the project, standard software development methodologies were not followed, because these are not always the most suitable for one person.

The approach that was taken is incremental development and testing, so constructing a portion of the project and then testing it along with how it integrates with the previously developed sections.

The project was developed in a way that maintains good software engineering principles such as high cohesion and low coupling.

Regular meetings were held with the project supervisor to ensure the project was on track.

8.2 Organisation

The project was organised by taking regular logs and handwritten notes to keep track of which areas were completed, and where work had to be done.

Wakatime (`https://wakatime.com/`) was also used to track time spent on the project and track productivity.

8.3 Version Control

Git was used to track progress of the project, along with sensible commit messages which were used to track work done.

9 Discussion

In this section, the successes and failures of the project will be discussed, along with how it could be improved.

9.1 Successes

The project was successful in that it met all of its aims, and even surpassed some of them, with it going undetected by a number of open-source deep-packet-inspection tools in my own testing.

9.2 Failures and Limitations

Most of the project went reasonably well, however there are areas that I feel the project performs badly in, and design choices that were made that while they made sense at the time were not the most sensible choice upon reflection. Chief amongst these is to act as a TCP tunnel rather than a UDP tunnel. A TCP tunnel was chosen originally because it is easier to test and do the initial setup for the program. UDP would have been better, because it is split into packets, and UDP does not require any guarantees that the data arrives, instead relying on the applications at each end to keep track of everything that has and has not arrived. This would mean that the project internals could be streamlined, and everything could have been done on a packet-by-packet basis rather than breaking a TCP stream up into arbitrary chunks. Using UDP would also mean that OpenVPN would have worked considerably better than it does over TCP.

The other issue that is noteworthy is the transmission speed of the data. At sub dial-up speeds, accessing the internet is not the most seamless experience. One of the major reasons behind this stems from my choice of using Python, rather than using a lower level language, as Python is relatively slow.

9.3 Mitigations

While existing solutions do not seem to detect the presence of additional data being transferred over HTTP, if the project became widely used, or even noticed, it would be trivial to stop all of the methods of data insertion.

- Inserting HTML could be prevented by inserting any HTML towards the start of the page, or otherwise manipulating the HTML present in the page.
- Appending whitespace to the end of the headers could be prevented by sanitising the header fields
- Encoding data into the order of the headers could be prevented by sorting the headers

All of the methods highlighted above would be reasonably simple to implement, and would not adversely impact using HTTP for the typical user.

9.4 Further Work

If the project were to be continued, a sensible next step would be to work on improving the throughput. This could be done in a number of ways, however the immediate speed improvements would be seen by improving the speed of the generic obfuscation layer, as it relies heavily on regular expressions, which are slow.

More configuration would also be a benefit, as while there is a lot of configuration available there are options which are configurable in the way the code is written, but never exposed to the user.

Another improvement would be an option to use UDP as well as TCP, which would enable more applications to be used with the project.

9.5 Final Thoughts and Reflection

I spent a lot of time making this project, and I feel like it was a very valuable use of time from which I learnt a lot. Before starting the project, I had never used Python, which was one of the reasons I decided to develop it in Python, even though as previously stated it probably was not the best fit for the project.

I also experimented with lots of technologies for testing that did not fit into the report, one of which is using Docker, which allowed me to easily create varied testing environments to run the project in for testing.

If I were to do the project again, I would not use Python. I did not anticipate how the relatively low performance of Python would impact the project. The lack of strong-types caused me endless headaches, along with Python's interesting approach to including local files.

10 Conclusion

The aims defined in the introduction were:

1. To hide data in the HTTP requests and responses
2. For the Server to be able to function as a valid HTTP Server
3. To be able to communicate between the server and client using only HTTP
4. To make the traffic appear as valid HTTP traffic

As discussed in the testing section, all of these aims have been fulfilled, and it is possible to access services and send data over HTTP.
Creating a full end-to-end VPN connection was not completely successful, however it is possible to perform all of the functions of a VPN by using the project.

11 Bibliography

References

[1] 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.

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

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

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

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

[6] P Christensson. Encryption, November 2014.

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

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

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

[10] AATIF SULLEYMAN. Google lets you report ads that know 'too much' about you, September 2017.

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

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

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

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

12 Appendices

12.1 Structure of ZIP file

```
├──doc - contains the documentation for the project
│     ├──demonstration - contains a short write-up for the demonstration
│     ├──litreview - contains the literature review
│     └──report - contains the report
│        └──appendices
├──src - contains the source code for the project
└──www - a minimal clone of the Computer Science web page
```

12.2 capture.pcapng

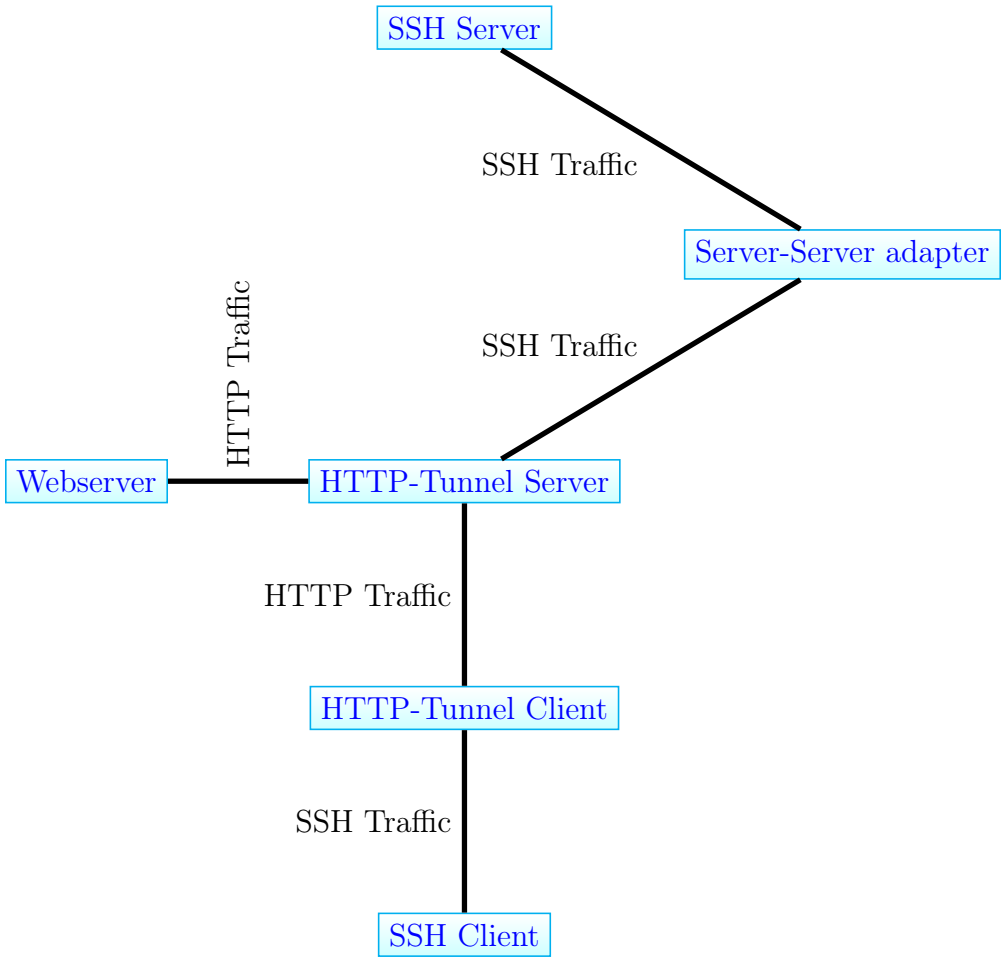
This is the capture file which was used when walking through the process step by step.

12.3 Running the code

The code can all be run separately, and should be platform independent, however the instructions are only provided for Linux using Docker, this is because it is multiple programs all which have to be run for the project to function. There is a `Dockerfile` and `docker-compose.yaml` file in the root of the included zip file, running `docker-compose up` will bring up 5 containers:

- A webserver serving the index page of the computer science webpage
- The proxy server
- The client
- An SSH Server
- An SSH Client acting as a SOCKS proxy.

The layout is as such:



This will start the computer you are running the containers on listening on port 8080, which will act as a SOCKS proxy. To see it in action, you can listen to the traffic created by the project on the local bridge called `wan`:

```
tcpdump -nni wan -vv
```

To do a `Curl` request for example, the following command requests google.com over the proxy:

```
curl google.com --socks5 localhost:8080
```