

By Saif Suleman

Table of Contents

[By Saif Suleman](#)

[Table of Contents](#)

[Analysis](#)

[Problem Identification](#)

[Stakeholders](#)

[Why it is suited to a computational solution](#)

[Problem Recognition](#)

[Problem Decomposition](#)

[Divide and Conquer](#)

[Interview](#)

[Interview Questions](#)

[General End Users](#)

[Enterprise Users](#)

[Analysis](#)

[Research](#)

[Existing Similar Solutions](#)

[RdpGuard](#)

[Features of the proposed solution:](#)

[Initial concept of my solution considering this research:](#)

[Limitations of my solution:](#)

[Further meeting with stakeholders:](#)

[This is an email I sent to my stakeholders \(Attique and ASTCO Solutions\)](#)

[These are the responses that I received:](#)

[Requirements](#)

[Software and Hardware Requirements](#)

[Hardware](#)

[Software](#)

[Stakeholder Requirements](#)

[Design](#)

[Functionality](#)

[Hardware and Software](#)

[Success Criteria](#)

[Design](#)

[User Interface Design](#)

[Connection Log](#)

[Active Connections](#)

[Authorised IPs](#)

[Installation Process](#)

[Stakeholder Input](#)

[I sent them the following email:](#)

[These are the replies I received:](#)

[Algorithms](#)

[Decomposing the problem](#)

[Decomposition](#)

[Abstraction](#)

[Subroutines](#)

[TCP Proxy Server Algorithm](#)

[Connection Pipe Structure Pseudocode](#)

[Connection Piping Algorithm Pseudocode](#)

[TCP Listener Algorithm](#)

[Proxy Server Struct](#)

[Proxy Server Listen Pseudocode Algorithm](#)

[Proxy Server Handle Connection Pseudocode Algorithm](#)

[Networking HTTP Server Algorithm](#)

[SMTP Email Sending Algorithm](#)

[Logger Algorithm](#)

[Configuration System](#)

[IP Whitelist Algorithm](#)

[MFA Code Generation & Verification Algorithms](#)

[Explanation and justification of this process](#)

[Validation & Testing Method](#)

[Iterative Development](#)

[Inputs](#)

[Testing Checklist](#)

[Development and Testing](#)

[Connection Piping Logic](#)

[Testing](#)

[Configuration System Module](#)

[Testing](#)

[Testing Framework](#)

[Testing Modification \(Connection Piping Logic\)](#)

[Use 'defer' to close connections and listeners](#)

[Avoid hardcoding port numbers](#)

[Use in-built copy function for reading and writing TCP stream data](#)

[Testing Modification \(Configuration System Module\)](#)

[Authentication Data Handler](#)

Computer Science Coursework - MFA-enabled TCP Proxy

Multi Factor Authentication Handler

Function to determine whether a code exists (returning a boolean):

Function to get the IP address an authentication code corresponds to:

Function to generate a new authentication code:

HTTP Server Handler

Logger System

Main Proxy TCP Listener

Evaluation

Criteria Met

Success Criteria

Evidence

Ability to get email alerts

Connection log endpoint

Ability to authenticate connections based on IP addresses

TCP connection proxy between two points for seamless communication

The option to close the program

Configuration file to configure the program in a simple format

Usability Features

Limitations

How to avoid these limitations

Maintenance

Post Development Testing

Analysis

Problem Identification

Currently there is no way to implement multi-factor authentication with many TCP based services such as remote desktop, web servers, game servers and more. These services typically have some form of authentication such as password-based authentication, but nothing is as secure as MFA. Multi factor authentication can be handled in many ways, such as email based authentication or using an Authenticator app on your phone.

The requirements for enabling this type of authentication on these services would be control over the network. If you can host a separate proxy service inside the network, that proxy will be able to access your private services and will pipe TCP communications through, provided that this proxy service allows you to connect only if it has been authenticated against that single IP address.

Stakeholders

The clients and demographic for this software would be users of various services hosted on the network. Because of the range of applicability for something like this, the demographic is very wide, ranging from users of RDP to network administrators wishing to be able to access administration panels from anywhere in the world with security in mind.

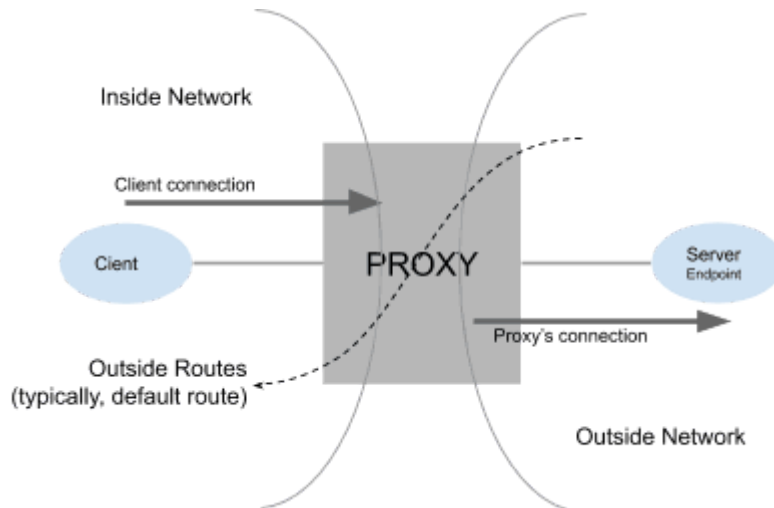
The stakeholders for this project mostly represent end users of these services, but can also vary and apply to larger enterprises.

A large number of services are naturally accessible to the outside world providing that the port is opened up by the firewall. These services alone may have poor security because of lack of MFA implementation. The stakeholders for the casual demographic is Attique Suleman, who is a user of Remote Desktop Protocol and wishes to further secure his computer by implementing wider security.

Another type of stakeholder demographic is larger enterprises. These enterprises may utilise several services such as RDP and private hosted web servers. If the websites of the web servers are hosted on-premise and may have poor security, these enterprises can use this project to secure their services. The stakeholder for this demographic is ASTCO Solutions, which is a tech company.

Why it is suited to a computational solution

The problem itself means that a computational solution is required for a variety of reasons. Firstly, there needs to be a computational effort to relay all TCP communications between the client and server once a connection has been made and the proxy has authorised it. Secondly, there also needs to be a method of verifying that the incoming connection is authenticated, by evaluating against a database.



Problem Recognition

There are numerous problems that need to be solved for this project. There needs to be a way to pipe network communications between the client and the server, with my application acting as the proxy.

There also needs to be a way for users to be able to verify if an IP address is authenticated or not through emails. This means that my project should be able to send emails, and also have a HTTP server with an REST API so that users can click on links from the emails to authenticate an IP address.

When this is finished, all that needs to be done is having a method of saving certain IP addresses to a database, and logging connections to the proxy.

Problem Decomposition

The problem can be decomposed into a series of smaller steps. My initial idea for these steps are:

- Receive incoming TCP connections on the proxy
- Evaluate if the IP address connected is in the authenticated IPs list
- If not, send an authentication request to the configured email address the proxy remembers

- If they are in the IPs list, pipe the TCP communications through between the client and the proxy's configured endpoint.

Divide and Conquer

Smaller problems on their own are much easier to implement, and combining them into a modular program makes use of the divide and conquer method of problem solving.

Interview

Interview Questions

I will outline some of the key questions I will be asking all of my stakeholders to find out the optimal solution for the software. I may also ask them to elaborate on their answers and ask follow up questions

General End Users

My questions for Attique Suleman (who represent General End Users) are:

1. Are you satisfied with the current security of services such as Remote Desktop Protocol?
2. Have you ever used multi-factor authentication?
3. If so, how have they been advantageous/disadvantageous to you?
4. What do you think of reinforcing your network with multi-factor authentication?
5. How would you like the software to be operated?
6. Do you have anything else to add?

The first question establishes confidence in current existing services that rely on password protection solely. This is important because it helps understand the current need for this type of software.

The second question helps to establish whether or not the client is familiar with this type of authentication. This is important because if we know about the user's familiarity with MFA, we can ask follow up questions. For example, the third question helps us understand advantages and disadvantages with existing software like this, which will help us in designing a better solution for our program.

The fourth question is to grasp the interest of the general end user when it comes to this type of security application. This is important because it helps us ask a follow up question which is the fifth question which is about how the user expects the application to behave which is important as it will help us in our design section.

The last question helps us understand what else the user would like us to know when designing and implementing the application, which is important as it helps us to gather as much information as possible when it comes to the expectations of the end user.

Are you satisfied with the current security of services such as Remote Desktop Protocol?

No. As RDP is only password protected, anybody can access my computer if they can guess my password. My computer has been hacked before and all my files have been stolen.

Have you ever used multi-factor authentication?

Yes, I have used multi-factor authentication before for many other web services.

If so, how have they been advantageous/disadvantageous to you?

It has been very advantageous to me as I have peace of mind and can control all logins to my computer. However, sometimes it is annoying as I have to repeatedly authenticate to login despite being on the same computer and network.

What do you think of reinforcing your network with multi-factor authentication?

It sounds great!

Enterprise Users

My questions for ASTCO Solutions (who represent Enterprise Users) are:

1. Are you satisfied with the current security of services such as Remote Desktop Protocol?
2. What services do you currently use on your intranet?
3. What do you think of reinforcing your network with multi-factor authentication?
4. How would you like the software to be operated?
5. Do you have anything else to add?

The first question helps us to understand the enterprise user's current need for a security reinforcement software such as this.

The second question is asked as we need to know the type of services they use so that we will be able to design our software appropriately to tailor it to the typical software that an enterprise user will use as well.

The third question helps us get a firm understanding of the user's need and opinion of a software like ours and identify the stakeholders' opinion on this project.

The fourth question is to get an understanding on how the enterprise will like the software to behave which will be taken into account during our design of the software.

The last question helps us understand what else the enterprise would like us to know when designing and implementing the application, which is important as it helps us to gather as much information as possible when it comes to the expectations of the enterprise user.

Are you satisfied with the current security of services such as Remote Desktop Protocol?

No. A lot of these services such as RDP do not have a good level of security, such as no support for multi-factor authentication and it is not even possible to change the ports that some of these services operate on - which leaves us very vulnerable to hackers that can take our important information.

What services do you currently use on your intranet?

Our local network has multiple computers, with remote access via RDP. We also have a central server machine that has an administration panel that controls all the logistics of our company, and is accessed via a HTTPS website.

What do you think of reinforcing your network with multi-factor authentication?

It sounds great and this is something that we have been looking for, for a very long time.

How would you like the software to be operated?

I would like to receive notifications by different means when people try to connect to my network, and also an administration panel that only I can access and view to control who is allowed to connect to the proxy, and read the logs of who has ever attempted to connect before.

Analysis

For security management software such as this which is targeted towards both enterprise users and end users, it is important to have an easy to use user interface with a positive user experience. It should be very easy to set up with minimal effort. There should also be support for an administration panel on a webpage that can be accessed from anywhere, and also support for multiple different means of authentication such as email based authentication.

Research

Existing Similar Solutions

RdpGuard

RdpGuard Dashboard

RdpGuard Service: ● Running (Protection is enabled)

Windows Firewall: ● Enabled (Protection is enabled)

Monitoring

RDP: ● Enabled

FTP: ● Enabled

IMAP: ● Enabled

POP3: ● Disabled

SMTP: ● Enabled

MySQL: ● Disabled

MS-SQL: ● Disabled

VoIP/SIP: ● Disabled

Web Forms: ● Disabled

RD Web Access: ● Disabled

Blocked IP Addresses

Local (115) Cloud (5428) GeoIP (5244) All (10787) < 01 02 >

IP Address	<u>Block Date</u> ▼	Unblock Date	Protocol
242.103.12.230	10.03.2020 16:55:10	10.03.2021 16:55:10	RDP
114.61.110.189	10.03.2020 16:55:10	10.03.2021 16:55:10	RDP
165.181.125.4	10.03.2020 16:55:10	10.03.2021 16:55:10	IMAP

Attacker's IP addresses are blocked automatically. Click image to view other screenshots.

This program allows you to see all attempts to connect to a service on your computer and log them - it also allows you to see the protocol they attempt to connect to.

RdpGuard is a host-based intrusion prevention system designed to protect your server from brute-force attacks. This service works on various protocols and it monitors the logs on your server and detects failed logon attempts. If the number of

failed log-on attempts from a single IP address reaches a set limit, the attacker's IP address will be blocked for a specific period of time. The user interface is very simple, it allows you to see the logs and services the program is monitoring.

Parts that I can apply to my solution:

There are some components in the program that are adaptable to my solution. Specifically, the program employs a method of identifying attempted connections from particular IP addresses and filtering them accordingly. This is a beneficial feature that I can incorporate into my own program as it provides a streamlined way of distinguishing between different individuals trying to access the service. Through comparing IP addresses, I can then determine which connections should be whitelisted and which ones should be blacklisted. The program also features a user interface that displays a log of attempted connections, including relevant information such as the time and protocol utilised. However, I do have some concerns regarding the accessibility of the user interface, which is currently only available on the Windows machine itself. To ensure maximum convenience for remote desktop protocol, my solution will include a web interface that can be accessed from anywhere, in addition to incorporating email authentication.

Features of the proposed solution:

Initial concept of my solution considering this research:

My solution will be an application that when the user runs on the host machine, will start a network proxy that will pipe all communications to the Remote Desktop Port on the loopback network interface. It will be a terminal window opened up that will display the port the proxy is listening on, and the port the web app is listening on. The web app is used for API requests to authenticate certain IP addresses, and will also be used to capture connection attempt logs. The logs section of the web app will authenticate based on the same IP address filter that filters incoming TCP connection traffic.

Limitations of my solution:

My solution has the limitation of not providing its own firewall, so upon setup, the user will have to configure their router's port forwarding so that it does not allow direct incoming connections on the default service's port (i.e. RDP: 3389) and rather only allows incoming traffic to connect to the proxy directly, so that the proxy will choose what connections are allowed through.

Another limitation of my solution is that when an unauthorised IP address attempts to connect, it simply terminates the connection early without any error message or explaining why the connection was terminated. This means that it could be confusing for first time users to figure out why the proxy isn't working properly, when in reality they need to authorise the connection.

Further meeting with stakeholders:

This is an email I sent to my stakeholders (Attique and ASTCO Solutions)

"Hi,

I have been investigating different designs for my software and how it will be operated and I have a few ideas about how it could work. When running the software for the first time, it will load a default configuration file which will contain the default ports for the API and the proxy, and it will display to a terminal the ports. Then, you can close the software, modify the configuration files and open the software again. The requirements are also that you set up port forwarding in a way that only the proxy port is allowed through.

Thanks,
Saif."

These are the responses that I received:

ASTCO Solutions:

"That sounds good! It seems very intuitive to use and we have our own networking infrastructure so setting up the port forwarding will be very easy to us. I like the design."

Attique:

"I like the design, the configuration sounds complicated. Maybe add a way to configure the program from the terminal GUI?"

Requirements

Software and Hardware Requirements

Hardware

CPU supporting either x86 or ARM instruction set - We are writing the program in Go, which supports multiple compilation targets across different operating systems, such as Windows & Mac (x86) or Mac: Apple Silicon / Linux (ARM) and the CPU on the computer will need to run one of these two architectures to be able to run

Software

Windows/Linux/Mac Operating System - The software will be written in Go, a compiled language which supports operating systems, and the OS used will have to be one of the mentioned operating systems.

Stakeholder Requirements

Design

Requirement	Explanation & justification
User interface to configure the program	This is so the user can change configuration options such as the ports for the API and proxy endpoints, authenticated IP addresses and the correct email to send alerts to
Lightweight Performance	As this is a proxy that is intended to be used for things such as RDP amongst others, it needs to have very fast network performance to prevent stuttering and provide a smooth and seamless experience.
Clear Instructions	This should be included in our user interface so that the program is easy to get started with for our stakeholders as we have both end-users and enterprise users for our stakeholders.

Functionality

Requirement	Explanation
Ability to get email alerts	Users should be able to get alerts and authorise connections from anywhere, so we will use email alerts as they can click on a link that has a unique secure code embedded to authorise a connection.
Simple and easy installation	Flow for the setup in the terminal so users can type in the options they want for the program such as email to send to, API endpoints, port to proxy to, etc.
Connection log endpoint	So the user can view attempted connections and successful connections to the proxy via a web page. For security the authentication of this web page will check against the authorised IP addresses list
Ability to authenticate connections based on IP addresses	This allows the user to add more IP addresses to the authorisation list usually via email alerts, they will do this by clicking a link they receive and this link has an embedded security code which is randomly generated by the program, and the program only allows the authorisation request if the embedded security code matches
TCP connection proxy between two points for seamless communication	This is the main function of the program and it allows users to use the applications they typically would through the proxy. This just allows the proxy to act as a passthrough for all network communications once an IP address has been authenticated and a connection has been successfully established.

Hardware and Software

Requirement	Explanation
Computer with x86/x64 Architecture	The user needs one of these CPU

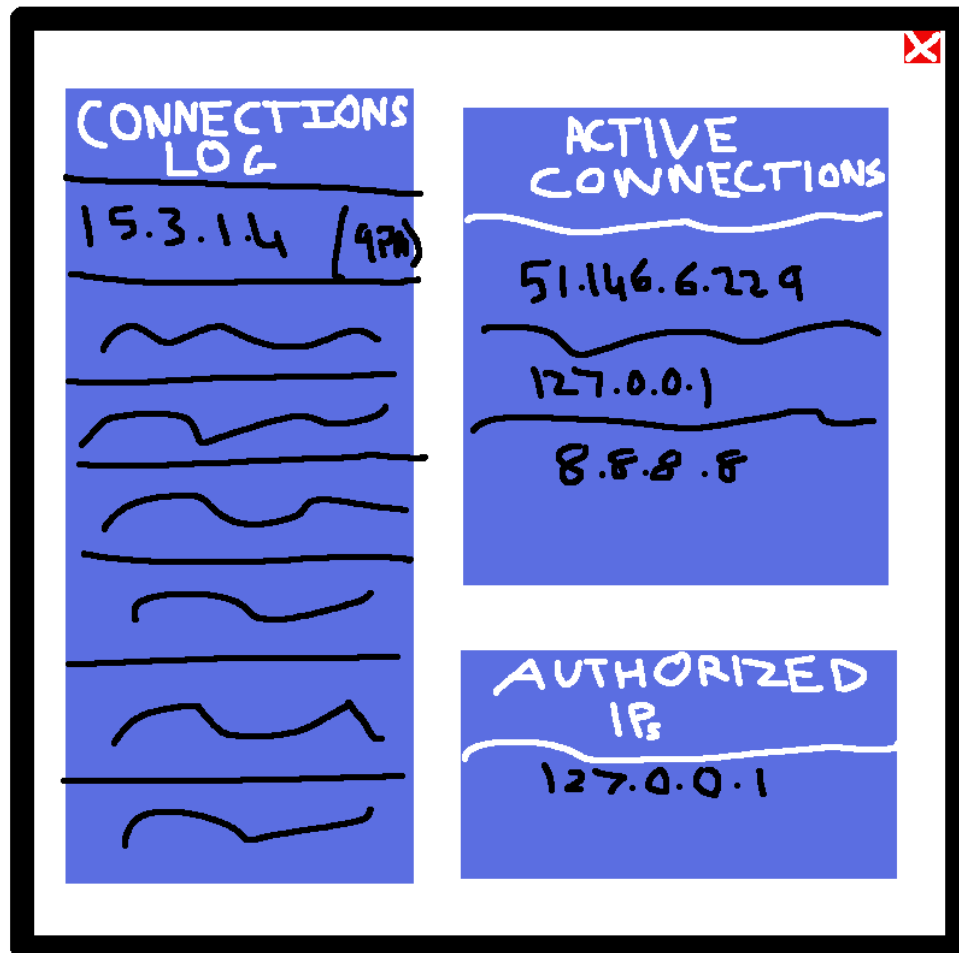
	architectures as these are the architectures supported by our programming language Go
Windows/Mac/Linux	These are the operating systems supported by our programming language Go
Access over network administration such as port forwarding and firewall control	The user will need to do this to allow the ports on the network proxy to be accessible from anywhere - this is crucial to access the REST API and the proxy itself from any location. They also need this so that they can disable access to the services they are trying to protect from the outside world, so that the only means of connection to these services are through the proxy.

Success Criteria

Requirement	How to show evidence
User interface to configure the program	Screenshot of the user interface for the configuration wizard
Clear Instructions	Screenshot of the instructions to setup the program
Ability to get email alerts	Screenshot of the email alert you get from an attempted connection and the code and testing to show that it works
Connection log endpoint	Screenshot of the connection log and code and testing to show that it works
Ability to authenticate connections based on IP addresses	Code for the authentication system and testing to ensure that you cannot connect if you are not authenticated, and then authenticating and then seeing that you can connect once you are authenticated.
TCP connection proxy between two points for seamless communication	Code for the connection stream proxy and testing to show that it works
The option to close the program	Screenshot of button to close the program and testing to prove that the program is fully stopped once it is closed
Configuration file to configure the program in a simple format	Screenshot of JSON file used to configure the program after the installation setup has been run, and code for configuration loading and testing to show that it works

Design

User Interface Design



This is the main design of the program. This allows the user to see all the relevant information regarding the application in a web panel they can access from anywhere in the world.

Links to the success criteria:

- User interface to configure the program (this is because they would be able to add additional IP addresses to the authorised IPs list)
- Connection log endpoint

Connection Log

The connection log box will contain a log of all past attempted and successful connections so that the user can monitor connections and view all the information.

Links to the success criteria:

- Connection Log endpoint

Active Connections

The active connections box will contain information of all existing connections that the proxy is currently allowing, and entries from this box will disappear once the specific connection has been terminated.

Authorised IPs

The “Authorised IP Addresses” box displays the current list of IPs that are authenticated and allowed to connect to the proxy. There will also be a text field after the last entry to add new IP addresses to be permitted to connect to the proxy.

Installation Process

The installation process of this application will be very simple and easy to use. Upon running the program for the first time, a CLI tool will be used to set up the program. In this tool, the user will type the email address they'd like alerts to be sent to, the SMTP server configuration for the email address of the application's sender component itself, default authenticated IP addresses and more. For an easier setup process, there will be no additional database such as MongoDB/SQL required - instead the program will cache in authorised IPs and to add more to persistent storage, JSON files will be used. This allows for a seamless and easy setup process where everything required is set up using this tool.

Stakeholder Input

At this point I have a good idea of what the final program's functionality and usability will be like. I have arranged a meeting with my stakeholders to obtain their input on the program.

I sent them the following email:

"Hi,

Please see the attached documents on the design for the program, which displays the User Interface and Installation/Setup progress.

Please let me know if this is in line with your ideals for the program and if this is anything you would like me to change.

*Regards,
Saif."*

These are the replies I received:

Attique Suleman:

"The design is very simple and seems very easy to use. I like the setup process of using the command line tool and the fact that we do not need to set up any additional database services or anything."

ASTCO Solutions:

"The designs are great and I like the large buttons - it seems very useful as the installation process seems easy."

Overall, the stakeholders were happy with the design so it is time to proceed. There were no strong objections about anything, so I will continue using this design.

Algorithms

Alongside user interfaces, the functionality of the program is equally as important. Some of the most important pieces of logic for the program are:

- **Piping TCP data between two connections**, using the server as the intermittent middle man, it will be designed to handle large volumes of data and used to reliably ensure that data is delivered between the client and server. This will utilise a concurrency tool known as coroutines in Go to allow handling of as many connections as needed.
- **Authentication Algorithm**: This algorithm will be responsible for authenticating the administrator who receives the email when a connection attempt is made to the proxy server. The algorithm will generate an access token in the link that is sent to the administrator, so that when the administrator accesses the API endpoint using that link, the token will be used to verify the authenticity of the administrator.
- **Whitelisting Algorithm**: This algorithm is responsible for maintaining a persistent list of IP addresses which have been authenticated by the administrator. Whenever an unauthenticated IP address attempts to connect to the proxy server, the program will reject the connection and send an email alert to the administrator, in which the administrator can choose to authenticate the IP.
- Processing files to allow for a persistent store of data for the IP whitelist, and a persistent configuration file to modify the options of the program.
- **Logging Algorithm**: This algorithm will be responsible for logging all events that occur on the proxy server. It will store logs in persistent file information on connection attempts, including the IP address, the date and time. Example events that will show up in the logging algorithm would be:
 - Unsuccessful connection attempts
 - Successful connection attempts
 - IP authenticate event
 - IP blacklist event (blocks all alerts and connections in the future until manually removed from the blacklist)
- **Configuration File Processing Algorithm**: This algorithm is responsible for processing the configuration file that defines the options of the program. It will be designed to read and modify the configuration file as needed and ensure that the program is always running with the correct settings. This should also have a system to update the configuration file in real time without needing to restart the program.
- **Email Sending Algorithm**: This algorithm will be responsible for sending emails to the registered administrator whenever a connection attempt is made to the proxy server. It will be designed to integrate with an email server and use standard email protocols like SMTP to send emails to the administrator.

Pipelining TCP Data between two connections

In this project I will be using Go, which supports easy concurrency using “goroutines”. This means that on the proxy server, I can have a running loop that will accept incoming connections, and if the connection is authorised, it will start a new goroutine and pipe data between the incoming connection and the proxy’s target (usually the RDP server).

This will be very simple and all it will do is read bytes from one endpoint and re-emit the bytes to the other endpoint. As two-way communication is always required, I will have to create two new goroutines for each connection piping instance. The first piping goroutine will pipe connections from the incoming connection to the target service, and the second pipe will pipe communications from the target service to the incoming connection.

This will allow for seamless communication between the incoming user connection and the target service. For each incoming connection, I will instantiate a new TCP connection to the target service, allowing for support for multiple isolated connections to the proxy.

Authentication Algorithm

This algorithm will be responsible for authenticating the administrator who receives the email when a connection attempt is made to the proxy server. The way that this will work is that once a connection attempt has been made to the proxy, the proxy server instance will call my authentication service module to check whether or not the connection should be allowed to connect.

If a connection has been made that is not allowed to connect, the authentication module will first generate a 16 digit code and store this code in a HashMap. After this, the authentication module will send an email to the configured administrator of the program including a link to the proxy server’s API endpoint with the 16 digit security code embedded. This link will be clicked to authenticate the connection’s IP address and allow future connections to be allowed.

Once somebody has attempted to click on a link used for authenticating IP addresses, the server will take the 16 digit code included in the link’s contents and compare that against the values in the HashMap. If it finds a pair, it will grab the IP address stored in the security code cache and add that IP address to the list of authenticated IP addresses.

The authentication module will use the **Email Sending Algorithm** to send emails, which is explained in full depth later here.

Whitelisting Algorithm

The whitelisting algorithm is responsible for maintaining the persistent file for the IP address whitelist. When the program loads, it will load a connection to the registry file and each time a connection attempt is made, it will fetch the list and identify whether or not the IP address that attempted to connect is included in the whitelist.

Logging Algorithm

The logging algorithm is responsible for logging all events that happen in the program, such as successful connection, unsuccessful connection, IP authenticated, aborted connection and more. This algorithm will store logs in a persistent JSON file and provide detailed information regarding the event such as event type, timestamp, and any additional data required. I am using a JSON file for this as this is a file format that allows the data to be easily processed and analysed later.

The module will be initialised on the program's start, and have functions to log certain events. It will store a cache of the list of events in a list, and on each event additional, it will append to the cache and then serialise the cache into JSON and write to the file.

Decomposing the problem

In order to make the problem easier to tackle and solve, I will be using various programming techniques such as **decomposition** and **abstraction**.

Decomposition

Decomposition is a programming technique that involves breaking down a complex problem or system into smaller, more manageable parts, modules or subroutines. Each subroutine performs a specific task and can be developed and tested independently of the others, making the overall development process more efficient and easier to manage.

The process of decomposition involves identifying the major components of a system and defining the interfaces and relationships between them. This can be done using a variety of techniques, including flowcharts, diagrams, and pseudocode. Once the system has been decomposed into its components, each component can be developed separately, either by a single programmer or by a team.

One of the primary benefits of decomposition is that it allows for the reuse of code. If a particular subroutine is used in multiple parts of a system, it can be developed once and reused as needed. Additionally, decomposition can make it easier to identify and fix bugs or issues in a system, since each module can be tested and debugged independently.

In my program, I will be using decomposition in many places. For example, the authentication handler will have a “save” function that will save the contents of the caches to a file, such as saving the list of whitelisted IP addresses. Decomposition is useful here because by moving the ‘save’ function to its own subroutine, I can use the shared code to save the list of IP addresses when you add an IP address and when you remove an IP address without having to repeat code.

Abstraction

Abstraction is when a programmer removes unnecessary information and focuses on the essential information. It simplifies the complexity and removes the cognitive burden of the system. It allows you to use fundamental concepts to break down a large problem into simple actionable pieces. I will be using abstraction in many places - one of them for example will be used in functions, which will encapsulate the fundamental details of a process and hide the unnecessary details behind parameters, so that the function can be called anywhere by passing the important information through the parameters.

For example, I can use abstraction to make a function that pipes TCP streams from one endpoint to another. It will simply have a constant while loop that reads from one endpoint to another. This is a good example of abstraction because I am removing the unnecessary details such as which TCP streams it should pipe and moving that to parameters. This means that on every successful connection, I can just call this function to pipe streams from the client to the target service, and pipe streams from the target service to the client using goroutines.

Subroutines

Now that I know how I want the program to work, I can start by implementing different subroutines that will be instrumental to the flow of the program.

TCP Proxy Server Algorithm

This subroutine will pipe a TCP stream between two endpoints. This is the core functionality of the program as it is a TCP proxy. When a successful connection is made to the proxy, it will call the function twice. The first time, it will use a concurrent goroutine (coroutine) to create another thread that will pipe TCP data from the user to the target service. Then it will run another blocking function that will pipe the TCP data from the target service to the user. This works as it allows for two-way communication between the user and the target service.

When a pipe has been created, it will also create a struct that will contain all the information required to maintain the connection between the two endpoints. This will consist of the two TCP sockets involved in the communication, and a boolean value "Alive". This boolean value will be used to keep track of whether or not the connection should be terminated or not. This means that from any thread, I can set **connectionPipe.Alive = false** in order to terminate the pipe from anywhere.

Connection Pipe Structure Pseudocode

```
type ConnectionPipe struct {
    Alive: boolean // boolean to determine if the connection
                  // should stay alive or not
    Left: TCPConnection // LHS (Left hand side) of the connection
    Right: TCPConnection // RHS of the connection
}
```

This is the required structure data for the **Connection Pipe** structure. The **Left** and **Right** fields are network connections and would typically represent the connecting user and the target service (i.e. RDP)

Connection Piping Algorithm Pseudocode

```
while (connectionPipe.Alive) {
    incomingData = connectionPipe.Left.Read()
    if (incomingData.length == 0 || incomingData == EOF) {
        connectionPipe.Alive = false
        break
    }
    connectionPipe.Right.Write(incomingData)
}
```

In a while loop that is dependent on the connectionPipe.Alive field being true, it will read data from the left hand side, and then if the data is null or is a terminated connection packet, it will terminate the proxy pipe. If the proxy pipe is not terminated, it will take the data that it has read from the left hand side and then write that same data on the right side, effectively proxying all communication between the two endpoints.

TCP Listener Algorithm

This is the TCP listener that will bind to a port and handle the incoming connections. This will work by accepting all incoming connections, and on every connection attempt it will compare the incoming connection's IP address with the list of whitelisted IP addresses. If that matches, it will use the piping algorithm described above to pipe the streams, and append to the log.

Proxy Server Struct

```
type ProxyServer struct {
    Address:      string
    Redirect:     string
    Connections:  Map(TCPConnection -> ConnectionPipe)
    Auth:         MultiFactorAuth
    APIAddress:   string
}
```

This struct has all the necessary and important fields for the proxy server. It has the address the server is bound to (listening on) as the '**Address**' field. It has the address of the target service so it knows where to pipe incoming connections to be marked as the '**Redirect**' string. The **Connections** field is a map/dictionary that maps network connections to Connection Pipe instances. The **Auth** field is an instance of the MultiFactorAuth struct, which has all the necessary functions relating to sending new alerts and verifying existing IP addresses. Lastly the **APIAddress** field is the address that the HTTP listener is listening on. This HTTP server is described later here.

Proxy Server Listen Pseudocode Algorithm

```
func (this *ProxyServer) Listen() {
    go this.Auth.Start(this.APIAddress)
    listener = net.Listen("tcp", this.Address)
    while (true) {
        incoming = listener.Accept()
        go this.handleConnection(incoming)
    }
}
```

In this algorithm, it will first create a goroutine that starts the HTTP API server in a different coroutine (denoted by the **go** keyword). This is because I want to start that HTTP listener without blocking the main thread, so that the code after is able to run in parallel with this code. After the HTTP API server is instantiated and listening, I create a TCP listener that listens on the address defined in the struct, and then I use a while loop to constantly accept incoming connections and once a connection has been determined, it handles the connection in a different coroutine. The reason I handle the connection in a different coroutine is so that I can continue to keep listening and accepting incoming connections while another connection is being handled. This allows me to handle thousands of requests at once with no issues - allowing the program to be scaled infinitely.

Proxy Server Handle Connection Pseudocode Algorithm

```

func (this *ProxyServer) handleConnection(conn net.Conn) {
    ip = GetIP(conn)
    whitelisted = this.Auth.IsAuthenticated(ip)
    if !whitelisted {
        conn.Close()
        return
    }
    redirect = net.Dial("tcp", this.Redirect)
    connectionPipe = NewConnectionPipe(conn, redirect)
    this.Connections[conn] = connectionPipe
    defer delete(this.Connections, conn)

    connectionPipe.Pipe()
}

```

This is the algorithm that will be used to handle incoming connections to the TCP server. It has already been referenced in the code above as **this.handleConnection** and it is a method on the **ProxyServer** struct. First it gets the IP address of the incoming connection serialised as a string, and then it references the **Auth** field on the struct to check whether or not the incoming connection is authenticated on the IP address whitelist. If the connection is not whitelisted, it closes the connection and exits from the function. If it is whitelisted, however, it will proceed to pipe the connection. It does this by first instantiating a TCP socket with the target service (i.e. RDP), and then instantiating a new connection pipe which pipes both the target service's connection socket and the incoming connection. Then it uses the **Connections** map field to map the connection to the pipe instance.

After this, it will use Go's **defer** keyword to defer deleting the connection from the map. This means that the connection pipe will not be deleted from the map until after the connection is terminated. Go's **defer** keyword achieves this as it delays the execution of that function/statement until the function it was called from ends. In this case, as our **Pipe()** function is thread blocking, the parenting function will only end once the pipe has been terminated, therefore the connection will only be removed from the map once the connection is terminated.

Lastly, it calls the **Pipe()** subroutine that we defined above, that will continuously pipe the data between connections in both directions, allowing for a seamless proxy experience.

Networking HTTP Server Algorithm

This is what will handle the REST api endpoints for the program. These endpoints will be used when an administrator clicks on a link to validate an IP address, or use a HTTP GET method to get the program's log.

There will need to be a way to listen on different HTTP endpoints and extract the data from the request. For this, I will be using a HTTP router. A HTTP router can perform routing based on the data that the URL and response processing are mapped to. This route mapping is commonly known as a route map. The router would map different request URLs to Handlers, which are subroutines that are called and defined in my code. A Request URL would consist of the path, which would be a string, and the HTTP method such as GET or POST. The HTTP method would depend on the functionality of the specific API request and varies with each function.

This is an example of a route map for further understanding

Request URL	Handler
GET /	<i>IndexHandler</i> This handler is called when no data has been added onto the request.
GET /foo	<i>FooHandler</i> This route is called when it is a GET request and the path is /foo
POST /foo	<i>FooPostHandler</i> This route is called when it is a POST request and the path is /foo
GET /foo/bar/:id	<i>FooParamHandler</i> The route is /foo/bar/ and the method is GET . This also accepts additional data, a param called "id" that you can put any data into and it will be extracted in the code.

The route map I will be using for my program is:

Request URL	Handler
POST /api/authenticate/:code	<i>AuthenticationHandler</i>

	Adds an IP address to the IP whitelist. Requires a cryptographically secure authorization code generated and signed by the server to authorise the request.
GET /api/log	<i>LogHandler</i> Returns a string of the program's entire log. Requires the user to be part of the whitelisted IP address in order to view this log.

I will need to write a handler in order to listen on a port and run different functions based on the HTTP route accessed. For this I can use a router library and then map different routes on my route-map to different handler subroutines.

This is the code to instantiate a new HTTP router using the gorilla mux library and declare all the necessary routes:

```
func (mfa *MultiFactorAuth) Start(address string) {
    mfa.Router.HandleFunc("/api/authenticate",
mfa.HandleAuthenticate)
    mfa.Router.HandleFunc("/api/log", mfa.ViewLog)

    fmt.Printf("Listening on: %s\n", address)
    if err := http.ListenAndServe(address, mfa.Router); err !=
nil {
        panic(err)
    }
}
```

This is a function on the **MultiFactorAuth** struct, it starts the listener on a certain HTTP address. The reason I did this on the MultiFactorAuth struct is because it needs to have handlers that reference other important functions and data on the struct. This struct declares the routes that reference the functions below, and then begins the listener. It calls the ListenAndServe function on the http module, and if an error is thrown then it will log the error.

This is the code for my **/api/authenticate:code** handler:

```
func (mfa *MultiFactorAuth) HandleAuthenticate(w
http.ResponseWriter, r *http.Request) {
```

```

code := r.FormValue("code")
if code == "" {
    _, _ = fmt.Fprint(w, "you must enter a code")
    return
}
ip, valid := mfa.GetCodeIP(code)
if !valid {
    _, _ = fmt.Fprint(w, "invalid code")
    return
}
delete(mfa.AuthCodes, code)
err := mfa.ProxyAuthHandler.AddWhitelistIP(ip)
var response string
if err == nil {
    response = "success"
} else {
    response = fmt.Sprintf("error: %s", err)
}
_, _ = fmt.Fprint(w, response)
}

```

When a request is made to this route, the provided function will be executed with **w** as the HTTP response writer and **r** as the incoming HTTP request. It processes a user's authentication code, checks its validity, and adds the associated IP address to a whitelist if the code is valid. If the operation is successful, it returns a "success" message; otherwise, it returns an error message.

This is the code for my **/api/log** route:

```

func (mfa *MultiFactorAuth) ViewLog(w http.ResponseWriter, _
*http.Request) {
    _, _ = fmt.Fprintf(w, string(mfa.Logger.CachedLog))
}

```

This code is very simple, when this route is accessed, it will simply return with the cache of the existing log and send that to the browser.

SMTP Email Sending Algorithm

This is the core algorithm used for sending email alerts to the administrators. These email alerts will be used for alerts when people attempt to try to connect to the server, and for when people want to authorise new IP addresses. As these two functions will be part of the same email, I can use one function to send the alert to the administrators.

The code will first construct all the data required for the email, including the body, subject, and fetch the authentication code to be used to authenticate the IP address. Then, after all the data has been constructed, it will loop through the list of administrator email addresses and send the email to each one of them. There will be a list of emails, and in this loop, it will append to that list. After the loop has finished, it will send all the emails in one request to the SMTP server. The reason I do this is to limit the number of network calls to the SMTP server, meaning that this is the most efficient solution.

This is the code:

```
func (mfa *MultiFactorAuth) SendEmailAlerts(ip string) {
    code, err := mfa.GenerateCode(ip)
    if err != nil {
        panic(err)
    }
    link := fmt.Sprintf("%s/authenticate?code=%s",
mfa.DefaultApiUrl, code)

    hostname, err := os.Hostname()
    if err != nil {
        panic(err)
    }

    body := fmt.Sprintf("RDP Login Attempt from %s.\nClick below
to verify this IP.\n\n%s", ip, link)

    dialer := gomail.NewDialer("smtp.gmail.com", 587,
"rdpgatekeeper@gmail.com", "Password123")
    dialer.TLSConfig = &tls.Config{InsecureSkipVerify: true}

    var messages []*gomail.Message

    for _, email := range mfa.Emails {
        m := gomail.NewMessage()
```

```

        m.SetHeader("From", "RDP Gatekeeper
<rdpgatekeeper@gmail.com>")
        m.SetHeader("To", email)
        m.SetHeader("Subject", fmt.Sprintf("RDP Access Attempt
on machine: %s", hostname))
        m.SetBody("text/plain", body)
        messages = append(messages, m)
    }

    if err := dialer.DialAndSend(messages...); err != nil {
        panic(err)
    }
}

```

As you can see here, first I fetch the authentication code that will be used to authenticate the IP address, and if an error is returned, I throw that error. Then, I construct that code into a string that represents the whole link. After that, I generate the body of the email address and its contents, including information of the IP address that attempted to connect and include the link in there. Then we instantiate a new SMTP dialer using the **gmail** package. This dialer is what will be used for sending emails.

We then instantiate a new variable, **messages** as a list of all our emails we will send using the dialer. After this, we loop through all the administrator email addresses and create a new email that we append to the array. After this, we send that list of emails using the dialer.

Logger Algorithm

The logging algorithm will be used for maintaining a log of all connection attempts to the proxy. It will write this log both to a memory cache for serving up to the REST API, and to a file to maintain persistence and retain data in the case of a program restart.

There will be a **Logger** struct to store the data and functions of the logger:

```

type Logger struct {
    File *os.File
    CachedLog []byte
}

```


This has a `CachedLog` field which is an array of bytes to represent the data, and a pointer to our `File` object so that we can make write calls to the file and append to the data.

We will have a function to initialise the logger, acting as our constructor:

```
func InitializeLogger(filepath string) Logger {
    if _, err := os.Stat(filepath); os.IsNotExist(err) {
        file, err := os.Create(filepath)
        if err != nil {
            panic(err)
        }
        file.Close()
    }

    file, err := os.OpenFile(filepath, os.O_RDWR, 0644)
    if err != nil {
        panic(err)
    }
    cached, err := ioutil.ReadFile(filepath)
    if err != nil {
        panic(err)
    }
    logger := Logger{
        File: file,
        CachedLog: cached,
    }
    log.SetOutput(&logger)
    return logger
}
```

Firstly, this checks if our output log file exists, and if not it attempts to create the output file - throwing an exception if any errors are returned. After the file is confirmed to exist, it will make a connection to that file with the **READ** and **WRITE** flags - also declaring this in our file permissions. Then, it will instantiate the logger with the attributes of our **file** variable, and the **cached** variable which is initially the contents of the log itself. Then it will use the **log** module in Go to set the output to a reference to our logger.

In order for our logger to be valid in our **log.setOutput** call, the logger must implement the Logging interface. This means that it must have two functions on the struct: **Write([]byte)** and **Close()**

For our **Write()** function, we simply need to print the string to the console window, append to the cached log and append to the file object we have. This will look something like:

```
func (l *Logger) Write(data []byte) (n int, err error) {
    fmt.Print(string(data))
    l.CachedLog = append(l.CachedLog, data...)
    return l.File.Write(data)
}
```

Lastly, for our **Close()** function, we simply need to close the connection to the file to avoid memory leaks and ensure the file is no longer locked, so that other processes can access it:

```
func (l *Logger) Close() error {
    return l.File.Close()
}
```

Configuration System

In order to ensure the program is extremely configurable and easy to use, I will implement my own custom JSON configuration system so that both personal and enterprise users can easily tweak the parameters of the program. Firstly, we will have a struct as this is the easiest way to encapsulate all the fields and methods of this utility into one place, and it maintains the same code style I have been consistent with. This “*ApplicationConfig*” struct will have all the fields that will be defined in the default configuration, with json accessors so that the JSON library will know what field maps to what.

```
type ApplicationConfig struct {
    ProxyAddress    string `json:"proxyAddress"`
    RedirectAddress string `json:"redirectAddress"`
    ApiAddress      string `json:"apiAddress"`
    LoggerPath      string `json:"loggerPath"`
    DefaultApiUrl   string `json:"defaultApiUrl"`
    ApiWhitelist    []string `json:"apiWhitelist"`
    Emails          []string `json:"emails"`
}
```

As you can see, I have all the fields necessary in the configuration. Below is a description of each field that will be included in the config and its purpose

Field Name	Description
ProxyAddress	The network address that the proxy should be listening on for incoming connections to the TCP proxy.
RedirectAddress	The network address of the service that the proxy should redirect communications to once a connection has been authorised.
ApiAddress	The URL that the REST API should be listening on - this will typically be the FQDN of the host machine so that it can be accessed from anywhere in the world.
LoggerPath	The path to the output logging file the logger should use
DefaultApiUrl	The URL that should be embedded in the email alerts for easy access to the API - this will usually be a domain name that points to this program.
ApiWhitelist	The list of IP addresses that are permitted to access sensitive information from the API such as the administrator log.
Emails	The list of administrator email addresses that the service should send email alerts to.

Now that the format for our config has been determined, I will have a default configuration file included in the code so that upon the program's first launch, from the code I can copy the contents of this default configuration to the output configuration file if the configuration file does not already exist.

Here is what the default configuration will look like:

```
{
  "proxyAddress": ":7777",
  "redirectAddress": "127.0.0.1:3389",
  "apiAddress": ":8182",
  "loggerPath": "gatekeeper.log",
  "defaultApiUrl": "https://rdp.plasmoid.io:8182/api",
```

```

"apiWhitelist": [
    ":::1",
    "127.0.0.1"
],
"emails": [
    "example@gatekeeper.io",
    "network@randomemail.com"
]
}

```

After this, I will need a way to embed this default configuration file into our code. In order to do this, I will use the **go embed** library. As Go is a compiled language, I can embed other files as resources into my executable to be used by the program. In order to embed this file, I just have to include this default JSON configuration file in the source code, and then reference it like so:

```

import (
    _ "embed"
)

//go:embed config.json
var defaultConfig string

```

Importing the library alone has a side effect in Go that allows me to annotate variables to reference embedded files. The contents of the default JSON configuration file is automatically injected into this variable, so accessing it allows me to directly read the contents of the file itself.

After this, in the case of configuration loading from the existing configuration file, I will need a way to validate a JSON configuration in order to make sure it's in the valid format. This is to ensure that there are no errors upon attempting to load the configuration, and if the configuration file is in the incorrect format, then I can override the existing configuration with our default configuration.

The validation function looks like this:

```

func IsTextValidConfig(text string) error {
    var config ApplicationConfig
    if err := json.Unmarshal([]byte(text), &config); err != nil {
        return err
    }
    return nil
}

```

This first declares a new variable with the type of our **ApplicationConfig** struct. Then it attempts to unmarshal (decode) the JSON configuration with the contents of the text given as a function argument, and a reference to our newly defined struct. Then, if an error is returned from the unmarshal attempt, it will return that error, if not, it will return nil. By returning an error, this means that it will return an error if the text is not valid, and return nil if it is valid - meaning that this function is a good way of determining whether or not a text parameter is a valid JSON.

Lastly, there will be a constructor for our configuration struct, this constructor will first read the existing config file if it exists and validate it using our **IsTextValid** function, if it is valid, then it will proceed to load that configuration file and return its contents in the form of our defined struct. If it is not valid, then it will override the existing file with our default configuration determined in our embedded file.

The code for something like that will look like this:

```
func NewApplicationConfig(filepath string) ApplicationConfig {
    if err := IsTextValidConfig(defaultConfig); err != nil {
        panic(err)
    }

    var config ApplicationConfig
    if _, err := os.Stat(filepath); os.IsNotExist(err) {
        err := ioutil.WriteFile(filepath,
[]byte(defaultConfig), 0644)
        if err != nil {
            panic(err)
        }
    }

    file, err := os.Open(filepath)
    if err != nil {
        panic(err)
    }
    err = json.NewDecoder(file).Decode(&config)
    if err != nil {
        panic(err)
    }

    return config
}
```

This first validates our default config to ensure that there are no errors made by the programmer in the config from compilation, then it checks if the existing output file exists and if not, it creates it, throwing an error if any is returned. Then it opens a connection to the existing file and uses a JSON decoder to decode the contents of that file into our **config** variable reference. If any errors are returned, it throws them, else it returns the config to be accessed from anywhere in the program.

IP Whitelist Algorithm

This is the algorithm responsible for adding to the list of whitelisted IP addresses, removing from the list of whitelisted IP addresses and reading from the list of whitelisted IP addresses. This will have a struct called **ProxyAuthHandler** that will contain the path to the whitelist file, and the whitelist itself as a list of strings representing the IP address of each entry as a string.

This is the pseudocode for the ProxyAuthHandler struct:

```
type ProxyAuthHandler struct {  
    WhitelistFilepath string  
    Whitelist          []string  
}
```

For reading and writing against this list of whitelisted IP addresses, I will need a function that will return the index of an entry in the whitelist, which will return either an integer representing the index, or -1 to represent that no index was found. This is crucial because I can use this to determine whether or not an IP address is whitelisted by determining whether or not the index is not equal to -1, and I can also use this to locate the index of an entry so that I can splice the list at that index in order to remove an entry from the whitelist.

This function solves two problems at once so this is an example of decomposition, which greatly simplifies the program.

Below is an example of the code for locating the index. I will be using a linear search for this as the whitelist will usually be very small, so a linear search will be just as effective and simple enough rather than any other form of search.

```
func (p *ProxyAuthHandler) GetWhitelistIPIndex(ip string) int {  
    for i, v := range p.Whitelist {  
        if v == ip {  
            return i  
        }  
    }  
}
```

```

    }
    return -1
}

```

This function accepts the **ip** it will be searching against as a parameter, and will return an int. This function exists on the **ProxyAuthHandler** struct so that it will be able to read from the struct's whitelist field. It will loop through all index and values in the whitelist, and if the value it is checking at the time is equal to the **ip** parameter, then it will return the index at that point. If the loop successfully ends with no indexes being returned, then it will return -1 to represent that there is no index in this list.

Now I will need a function to determine whether or not an IP address is in the whitelist. This is easy now that I have my function to get the index of an IP address in the whitelist. I can simply fetch the index of the entry, and if the index is greater than -1, then the IP address does exist in the whitelist. If the index is less than or equal to -1, then the IP address does not exist in the whitelist and we can return false. This is what the code will look like:

```

func (p *ProxyAuthHandler) IsWhitelisted(ip string) bool {
    return p.GetWhitelistIPIndex(ip) > -1
}

```

As always, this function exists on our ProxyAuthHandler struct for tighter coupling and encapsulation, and to keep with the same code style used in the rest of the project.

Now that we have functions ready for reading from the whitelist, we need to write functions to write to the whitelist. This will consist of both writing to the whitelist cache in memory which is a field on our ProxyAuthHandler struct, and saving the contents of that field to the file for persistence in case of the program restarting.

For the first part, there will be two functions: a function to add an IP address to the whitelist, and a function to remove an IP address from the whitelist. In order to add an IP address to a whitelist, first we should check if the IP address already exists in the whitelist, and if it does, we return an error. If no error is thrown, then we simply append to our list and save the whitelist to the file. In the case of removing an IP address from the whitelist, it follows the same concept. First it checks if the IP address is whitelisted, and if the IP is not whitelisted, then it returns an error. Then, it will fetch the index of the IP address in the whitelist and splice the array at that index in order to successfully remove it from the list.

Here is some pseudocode for adding and removing IP addresses to and from the whitelist:

```
func (p *ProxyAuthHandler) AddWhitelistIP(ip string) error {
    if p.IsWhitelisted(ip) {
        return fmt.Errorf("ip already whitelisted")
    }
    p.Whitelist = append(p.Whitelist, ip)
    return p.Save()
}

func (p *ProxyAuthHandler) RemoveWhitelistIP(ip string) error {
    index := p.GetWhitelistIPIndex(ip)
    if index < 0 {
        return fmt.Errorf("ip not whitelisted")
    }
    last := len(p.Whitelist) - 1
    p.Whitelist[index], p.Whitelist[last] = p.Whitelist[last],
p.Whitelist[index]
    p.Whitelist = p.Whitelist[:last]
    return p.Save()
}
```

Adding the whitelist consists of running our precondition checks, appending to the array and then saving the file. When removing from the whitelist, it consists of first grabbing the index it is at, and then calculating the index of the last element in the array. Then, we swap the elements of the index the element is at and the last element in the array, so that we can run a splice and set the whitelist to remove the last element by assigning the array to a smaller slice of itself. This is because in Go, there is no in-built way to remove elements from an array (also known as a slice in Go), so you have to assign it to a modified copy of the array, hence this solution.

In the code snippets above, I have referenced a **Save()** function that I use. The purpose of this **Save()** function is to copy the contents of our whitelist cache into the file that the whitelist is supposed to output to. This subroutine will first check whether or not the file exists, and if it does not exist, it will create a new file. After this, it will open the file with read and write permissions and dump the contents of the Whitelist variable into the file and then safely close the file by using the **defer** keyword that I explained earlier. It will dump the contents of the whitelist into the file by using a JSON encoder to encode the list as serialised JSON format, so that it can be successfully read again by our JSON decoder. The code that carries this process is below:

```
func (p *ProxyAuthHandler) Save() error {
```



```

var file *os.File
if _, err := os.Stat(p.WhitelistFilepath); os.IsNotExist(err)
{
    file, err = os.Create(p.WhitelistFilepath)
    if err != nil {
        return err
    }
} else {
    file, err = os.OpenFile(p.WhitelistFilepath,
os.O_WRONLY, 0644)
    if err != nil {
        return err
    }
}
defer file.Close()

return json.NewEncoder(file).Encode(&p.Whitelist)
}

```

Lastly, I will need to write the code for the constructor of my struct that will take care of ensuring that the file exists and reading the existing whitelist file and dumping that file into memory using our Whitelist field that we have in our struct.

```

func NewProxyAuthHandler(filepath string) (ProxyAuthHandler,
error) {
    var handler ProxyAuthHandler

    if _, err := os.Stat(filepath); os.IsNotExist(err) {
        file, err := os.Create(filepath)
        if err != nil {
            return handler, fmt.Errorf("line 21: %s", err)
        }
        if _, err := file.Write([]byte("{}")); err != nil {
            return handler, fmt.Errorf("line 24: %s", err)
        }
        file.Close()
    }
    file, err := os.Open(filepath)
    if err != nil {

```

```

        return handler, fmt.Errorf("line 29: %s", err)
    }

    defer file.Close()
    var whitelist []string
    if err := json.NewDecoder(file).Decode(&whitelist); err !=
nil {
        return handler, fmt.Errorf("line 36: %s", err)
    }

    handler = ProxyAuthHandler{
        WhitelistFilepath: filepath,
        Whitelist:          whitelist,
    }

    return handler, nil
}

```

First, we check whether or not the file exists, and if it does not exist, then we create that file and close it. Then, after we have verified that the file exists, we open a connection to it, and if an error is thrown we exit the function early. Then, we use Go's **defer** keyword to halt the execution of closing the file until the function ends. After this, we will return a JSON decoder on the existing contents of the file and pass in a pointer to an empty **whitelist** variable so that the output of that JSON decoder is assigned to that whitelist variable. If an error is thrown here, we will return the empty handler object we have and an error. Lastly, we assign the **handler** struct we have to a new struct that has the **WhitelistFilepath** field to be the **filepath** we were passed in the method's parameters, and the **Whitelist** field of the struct to be the newly calculated **whitelist** variable that we have processed. Then, we return this new handler with the error being **nil** to show that no errors were thrown and the program completed successfully.

MFA Code Generation & Verification Algorithms

I will need an algorithm to generate cryptographically secure authentication codes that will be sent over email and used to verify an administrator's identity in the event of authenticating an IP address. These codes will be one-time tokens that are included in the link of an email to all administrators and are different each instance they are made. When somebody attempts to connect to the proxy, a code will be generated and stored in a map that maps the code to the particular IP address that requires it to be authenticated, then this code is sent as a link to all the

administrators. Once the **/api/authenticate** endpoint has been accessed, it will scan the code from the link and compare that to the map. If that code corresponds to an IP address in the map, it will append that IP address in the whitelist so that future connection attempts will be automatically authorised and the user can connect to the intended services via the proxy.

The code generation algorithm will first set the key equal to an empty string, and then in a constant while loop that is dependent on: while the key is an empty string or the key is equal to an existing code, it will generate new 256-bit cryptographically secure authentication codes. It will generate these authentication codes by first allocating an empty array of 32 bytes ($32 * 8 \text{ bits} = 256 \text{ bits}$) - then it will use Go's **rand** library to fill this array with completely random bytes. If an error occurs it will throw the error, and if no error is thrown, then it will encode this array of bytes into a base64 string so that it is URL-friendly and can be sent easily over email as a string.

This is a pseudocode algorithm of the cryptographically secure code generation:

```
// 256 bits
func (mfa *MultiFactorAuth) GenerateCode(ip string) (string,
error) {
    key := ""
    while key == "" || mfa.DoesCodeExist(key) {
        buf := make([]byte, 32)

        _, err := rand.Read(buf)
        if err != nil {
            return key, err
        }

        key = base64.RawURLEncoding.EncodeToString(buf)
    }
    mfa.AuthCodes[key] = ip
    return key, nil
}
```

As you can see, first it utilises a loop to generate a cryptographically secure base64 code that does not already exist, and after this code is generated, it utilises the map, using the key as the cryptographically secure code, and the value as the IP address this code corresponds to.

This subroutine above utilises a function to check whether or not a code exists in the map - this is extremely trivial as fetching a key-value from a map in Go also returns whether or not that key-value pair exists, so we can utilise this behaviour by creating

our own function to check whether or not a code exists. That will look something like this:

```
func (mfa *MultiFactorAuth) DoesCodeExist(code string) bool {  
    _, has := mfa.AuthCodes[code]  
    return has  
}
```

Lastly, we will need to make a function to get the IP address a code corresponds to, if one does at all. The idiomatic way to do this in Go is to return both the value and a boolean to represent whether or not that value is present. This is also very simple, it will look something like this:

```
func (mfa *MultiFactorAuth) GetCodeIP(code string) (string, bool)  
{  
    ip, has := mfa.AuthCodes[code]  
    if !has {  
        return "", false  
    }  
    return ip, true  
}
```

As you can see, first we fetch the value and whether or not the value is present from the map. If the value is not present, we return an empty string and false. If the value is present, we return the IP address and true. We write this wrapper for the map for simplicity purposes as it is easier to have a function that is correctly named for reading from the map rather than reading from the map directly. This is also useful as if we want to change the function of how reading from the map works, we can do it once here rather than repeating that change in functionality from every place we read from the map.

Explanation and justification of this process

The initial task seems very complex, however I have broken down the large task into small actionables that can be solved one at a time, independently of each other. Breaking down the solution into small modules is extremely important in managing large projects such as this one. The code itself will be extremely modular and have a very organised file structure and code stylings in order to maintain the cleanest possible solutions. I will be doing things in Go idiomatically so that my solution is the most optimal solution for this task. Separating the authentication handler, TCP handler, different servers and different modules such as logging and code generation

allows me to think more directly on each individual task and manage it one at a time rather than all at once, which has huge benefits.

Validation & Testing Method

To ensure that the program is working properly and is robust, I will be testing it by sending a variety of different requests to it from multiple different sources and seeing if the program responds appropriately. Firstly, I will test by using the program as intended with all inputs correct and seeing that it proxies the connection correctly with lack of issues, and that it outputs to the logger correctly. Next I will attempt to access the logger using the API from an authenticated, trusted connection, in which it should give me the log correctly. Next I will attempt to access the API from an incorrect location and see that it blocks me from connecting.

During development, all functions should be tested to ensure that they work correctly. I will do this so that as I write a module, I will run my own tests to ensure that the module is working correctly to my expectations of the program and I will document any errors that will arise. If any errors are printed to the console, I will show the stack-traces and error messages as they arise. For debugging purposes, I will log the contents of certain variables and ensure that they are what I intended them to be, and if they are not, I will isolate the issue, fix it and document it appropriately.

After this process, and that I am confident development is complete, I will carry out one final testing phase that will include destructive testing too. Destructive testing is when I input to the program invalid and incorrect inputs to see how it will behave, and then measure whether or not it will respond appropriately.

I will record all testing data I input and show screenshots of the testing method, and once I am confident that all testing has been completed and all bugs have been resolved, I will schedule a meeting with my shareholders to discuss if any revisions need to be made, and if it is up to their expectations.

Iterative Development

Iterative development is a style of development that consists of breaking the problem into small actionable modules (as demonstrated above in the design section), and then working on developing each of the modules and then as each module is developed, I will run testing on that module until I am sure it works perfectly. This

process will repeat for any new features I add or any new modules until I have a fully formed program, in which then I can run my later stages of testing.

Inputs

For the main part of the program that will consist of proxying TCP communication, there is no easy way to test it as it requires running the proxy while mapping to a service. This service will typically be used for Remote Desktop Protocol, however any TCP enabled service will be adequate for testing. For example, we can use a simple HTTP server to act as our target service.

You simply need to run the target service on your local machine as well as the proxy, and then configure the proxy correctly and attempt to connect to it and you should find that after a successful connection to the proxy, it acts as the target service you are trying to connect to and all TCP data is seamlessly piped.

Testing Checklist

This is the checklist to make sure I have tested all the functionality of the program:

Action to test	Working?
<ul style="list-style-type: none">• Check that the TCP Proxy Piping procedure is working correctly by testing a proxy connection and validating that no data is corrupted/destroyed upon transit during the proxy connection	
<ul style="list-style-type: none">• Verify the HTTP server by accessing the different routes and verifying the different functions do the different things based on which route has been accessed	
<ul style="list-style-type: none">• Verify the logger is working in correctly outputting to the console, the API cached log and the persistent file log	

<ul style="list-style-type: none"> • Check the the log API endpoint is working correctly and that you are able to view the log - but only if your IP address has been authorised, and that the logger returns an error if you attempt to access it from an unauthorised network connection 	
<ul style="list-style-type: none"> • Check that the email sending alerts is working correctly by attempting to make a connection to the RDP proxy from an unauthorised IP address and observing the alert 	
<ul style="list-style-type: none"> • Check that the code generation and validation is working correctly by first modifying the link sent by the email alert to be an invalid code, checking that it doesn't work, and then using the link with the correct code and checking that it works 	
<ul style="list-style-type: none"> • Check that the config system is working correctly by changing an admin email or the port listening on for the server, and verifying it changes correctly upon restart 	
<ul style="list-style-type: none"> • Check that the whitelist file handler is working correctly by adding to the IP address list via email alert and verifying that the file changes 	

Once this comprehensive list is complete, I will be sure that the functionality of the program is working exactly as it is intended and that I will be able to present this to my stakeholders and collect further revisions and input from them.

Development and Testing

Connection Piping Logic

This is the core functionality of the program - the ability to pipe TCP data is what the foundation of this program is built on, therefore I started on this as the first thing I worked on.

The first step is to write code that pipes TCP streams from one network connection to another network connection. This is a monodirectional TCP stream piping, and utilising this function is the building blocks for creating bidirectional TCP stream piping (by utilising Go's goroutines)

This is my initial code for the monodirectional TCP stream piping

```
func (cp *ConnectionPipe) pipeConnection(read net.Conn, write
net.Conn) {
    buf := make([]byte, 1024) // creates the buffer to be used
    for the data

    for cp.Alive { // while connection pipe is alive
        length, err := read.Read(buf) // reads from the 'read'
network socket
        if err != nil { // if error is thrown break from the
loop by marking cp.Alive = false
            cp.Alive = false
        }
        length, err = write.Writer(buf[:length]) // writes the
data from the buffer to the 'write' socket
        if err != nil { // if an error is thrown, terminate the
connection
            cp.Alive = false
        }
    }
}
```

From this code, I first instantiate a buffer for the data of 1 kilobyte, and then while the connection pipe is alive, I read from the 'read' network socket and then write to the 'write' network socket. This code generally works, however there are some logic errors I have noticed that I need to adapt for.

Firstly, when terminating the loop by **cp.Alive = false**, I should also call **break** to halt the execution of the code block early as the execution does not finish until it reaches

the end of the loop block. Secondly, when checking if **err != nil**, I should also check if **length == 0** as that would typically represent an EOL/EOF byte where the connection has been gracefully terminated by one of the partners in the connection.

This version of my code with the tweaked parameters and additional comments is below:

```
// function existing as a method on the ConnectionPipe to pipe
connections,
// it reads from one connection and directly pipes it to the other
// this alone is NOT bidirectional
func (cp *ConnectionPipe) pipeConnection(read net.Conn, write
net.Conn) {
    // creates our buffer of 2048 bytes (2KB)
    buf := make([]byte, 2048)

    // while this connection pipe is alive
    for cp.Alive {
        // reads from the 'read' connection and dumps that data
into buf
        length, err := read.Read(buf)

        // if an error was thrown or the length of the data
read is 0
        if err != nil || length == 0 {
            // kill the connection pipe and break from the
loop
            cp.Alive = false
            break
        }

        // writes to the 'write' connection the contents of the
buf
        // up to the length returned from the read function
        length, err = write.Write(buf[:length])

        // if an error was returned or the length successfully
written is 0:
        // terminate the connection pipe and break out of the
loop
        if err != nil || length == 0 {
            cp.Alive = false
            break
        }
    }
}
```

```

    }
}

```

This is the improved version of my code, it checks if the length received from reading the data and writing the data is zero, and if it is, OR if an error is thrown, it terminates the loop correctly whilst also breaking from the loop. I have also increased the buffer size for 2048 bytes (2KB) for increased network throughput.

This function is defined on a struct called **ConnectionPipe** which we have not defined yet - so we should define this now.

```

// struct for an active or inactive connection pipe
type ConnectionPipe struct {
    Alive bool // represents whether or not the connection pipe
               is actively piping data
}

```

This now allows our code to successfully compile and work correctly now. After we have determined how to do monodirectional TCP piping, we will now build the ground for the bidirectional TCP piping.

This will utilise Go's goroutines. The way it will work is that we will have our **left** and **right** variables that are network connections that we need to pipe. In one goroutine, we will pipe **left->right**, and then in the main thread of that function we will pipe **right->left**. This is so that both functions can run at the same time parallel with each other, however if we run both functions as a goroutine, they will both run in the background and nothing will run in the foreground, resulting in the function ending early and nothing happening (there needs to be a main context active to every function).

In order for this, we should actually store both network connections as part of our ConnectionPipe struct named as the **left** and **right** fields. The adapted version of our new struct will look like this:

```

// struct for an active or inactive connection pipe
type ConnectionPipe struct {
    Alive bool // represents whether or not the connection pipe
               is actively piping data
    Left net.Conn // left-hand-side of this connection pipe
}

```

```

    Right net.Conn // right-hand-side of this connection pipe
}

```

This allows us to directly reference these two connections without needing to provide them as parameters in our `Pipe()` function of the connection pipe. This means that instead we pass the two network connections in the constructor for our network pipe and then when we call the `connectionPipe.Pipe()` function, it will automatically reference the **left** and **right** network connection fields in our struct.

Using the concepts we described above, we can begin to write our **connectionPipe.Pipe()** function utilising the initial monodirectional TCP piping function. This is the code I ended up with:

```

// function to begin piping on the connection pipe bidirectionally
func (cp *ConnectionPipe) Pipe() {
    // first, set the Alive field equal to true to represent
    // the connection pipe as now being active
    cp.Alive = true

    // in a goroutine, pipe connection from one way to the other
    go cp.pipeConnection(cp.Left, cp.Right)

    // blocking this thread context, pipe the connection in a
    // reverse direction to the initial way
    cp.pipeConnection(cp.Right, cp.Left)
}

```

When calling this **Pipe()** function that is defined on the `ConnectionPipe` struct, we first mark the **ConnectionPipe** as **Alive** using the boolean flag, then in one goroutine we pipe the connection from left to right, and then in the main context of the function we pipe it from right to left. Initially, I was using a goroutine for both of these function calls, but that instead resulted in nothing happening for the reasons I described above: *“if we run both functions as a goroutine, they will both run in the background and nothing will run in the foreground, resulting in the function ending early and nothing happening (there needs to be a main context active to every function)”*

Now that we have finalised the format for our `ConnectionPipe` struct, we need to create a new constructor function for this struct. This will simply accept the **left** and **right** connections as parameters, and default **Alive** = false, as it will be set to true once the connection pipe starts piping and the **Pipe()** is called directly. Here is our revised constructor:

```
// main constructor function for a connection pipe, accepting
// left and right as parameters and default 'Alive' = false
func NewConnectionPipe(left net.Conn, right net.Conn)
ConnectionPipe {
    return ConnectionPipe{
        Alive: false,
        Left: left,
        Right: right,
    }
}
```

This connection pipe module is now complete and we are free to continue to work on other modules now and integrate that with this module.

Testing

In order to test this, I wrote a testing program that used this module to simply pipe over an RDP connection, using the server as a middle man.

Here's what the code looks like:

```
func testConnectionPiping() {
    listener, err := net.Listen("tcp", ":8080")
    for {
        incoming, err := listener.Accept()
        if err != nil {
            panic(err)
        }
        redirect, err := net.Dial("tcp", "127.0.0.1:3389")
        if err != nil {
            panic(err)
        }
        connectionPipe := pipe.NewConnectionPipe(incoming, redirect)
        connectionPipe.pipe()
    }
}
```

This code simply starts a testing TCP listener, and when a connection is made to it, it dials port 3389 on the loopback machine and pipes connections to it. Now, I just need to run this code at the start of my program and attempt to make an RDP connection to **127.0.0.1:8080**.

I have made a connection to port 8080 and as I am able to connect to the RDP machine through this proxy and use the remote machine with no issues, I am confident that this logic works.

Configuration System Module

The next logical step is to work on our configuration system as it is one of the crucial dependencies of most of the other modules. For this configuration system, I will be using Go's built in library of structuring and destructuring JSON. This is because this library is extremely effective as we can directly convert a JSON file into a pre-defined struct and have all the data ready for us and easily accessible from there.

This JSON library will utilise struct field tags to identify the mappings between the JSON field names and the fields defined in the struct.

The fields that we will be using for the configuration have been established in our design section:

```
type ApplicationConfig struct {  
    ProxyAddress    string    `json:"proxyAddress"` // the address  
the tcp proxy server is listening on  
    RedirectAddress string    `json:"redirectAddress"` // the  
address the tcp proxy server will use as its target service to  
piping  
    ApiAddress      string    `json:"apiAddress"` // the address  
the REST API will be listening on  
    LoggerPath      string    `json:"loggerPath"` // the path to  
the output file of the program's log  
    DefaultApiUrl   string    `json:"defaultApiUrl"` // the  
publicly accessible link to the REST API to be used in embedded in  
the email links  
    ApiWhitelist    []string `json:"apiWhitelist"` // the IP  
address whitelist to access sensitive information from the REST  
API such as the log  
    Emails          []string `json:"emails"` // the list of  
administrator email addresses that the program should email alerts  
to  
}
```

As you can see in this struct above, we are using struct tags (for example: ``json:"proxyAddress"``) to denote the struct's field's mapping to their corresponding JSON field names.

Now that we have the struct for the configuration itself defined, we need to write a function to determine whether or not a text string is a valid JSON configuration. To do this, we can simply attempt to unmarshal (decode) the JSON into the struct we defined based on the certain input text and return the error if one is made. That way, if **IsTextValidConfig(text) != nil** then that means that the text is **NOT** a valid configuration.

```
// determines whether or not a text string is a
// valid configuration JSON
func IsTextValidConfig(text string) error {
    // variable for our config
    var config ApplicationConfig

    // attempts to decode the text into our config variable,
    // and if an error is present, then return the error
    if err := json.Unmarshal([]byte(text), &config); err != nil {
        return err
    }

    // return no error if no error has been thrown
    return nil
}
```

This was my initial implementation of this function. Whilst reading through it however, I noticed that it could be simplified into simply returning the value of the **json.Unmarshal** call directly. This is because we fetch the return value (the error), and if it exists: we return it, but if it's nil, we return nil. This can be simplified to simply returning the error itself as in the function above, the value we return will always be equal to the return value of the **json.Unmarshal** call.

This is what the revised function will look like:

```
// determines whether or not a text string is a
// valid configuration JSON
func IsTextValidConfig(text string) error {
    // variable for our config
    var config ApplicationConfig
    // attempts to decode the text into our config variable,
    // then return the error
    return json.Unmarshal([]byte(text), &config)
}
```

We can now write our constructor function for the `ApplicationConfig` struct. This function will first check whether or not the configuration file exists, if it does not exist, then we will use the default configuration file, otherwise we will load the configuration file, check if it's valid and then JSON decode it into our struct and return the output value.

```
// constructor for our ApplicationConfig
func NewApplicationConfig(filepath string) ApplicationConfig {
    // first checks if our defaultConfig is valid to prevent
    // errors that occurred from compilation and make them
    obvious
    if err := IsTextValidConfig(defaultConfig); err != nil {
        panic(err)
    }

    // variable for our new ApplicationConfig
    var config ApplicationConfig

    // checks if the file exists, if it doesn't, then paste in
    // the default configuration
    if _, err := os.Stat(filepath); os.IsNotExist(err) {
        err := ioutil.WriteFile(filepath,
[]byte(defaultConfig), 0644)
        if err != nil {
            panic(err)
        }
    }

    // opens a connection to the file path, if an error is
    // present, then throw the error
    file, err := os.Open(filepath)
    if err != nil {
        panic(err)
    }

    // decodes the file contents into the config using
    // a pointer, and if an error is returned, then
    // throw the error
    err = json.NewDecoder(file).Decode(&config)
    if err != nil {
        panic(err)
    }
}
```

```

    }

    // returns our newly defined config
    return config
}

```

This code works great, however this references a variable known as **defaultConfig** which we will now have to define. As **Go** is a compiled language, we can embed resource files into the executable and load that in our code. This is particularly useful for this case because we can define the default configuration in our source code as a normal JSON file, and then reference that from code by using the **go:embed** package to embed the resource into the code.

This is what our default configuration file will look like:

```

{
  "proxyAddress": ":7777",
  "redirectAddress": "127.0.0.1:3389",
  "apiAddress": ":8182",
  "loggerPath": "gatekeeper.log",
  "defaultApiUrl": "https://rdp.plasmoid.io:8182/api",
  "apiWhitelist": [
    "::1",
    "127.0.0.1"
  ],
  "emails": [
    "example@gatekeeper.io"
  ]
}

```

As you can see, all these JSON fields have keys that directly correspond to our struct, using our JSON struct tag annotations to map the field names between the two sources.

Now that we have defined what our default configuration will look like, we simply need to save this file in the same source directory as the file we wish to call it from. Then, in Go, we simply need to declare the variable **defaultConfiguration** to be an empty string, and then we write a comment above it to be **//go:embed config.json** - this combined with importing the go embed library will automatically inject the

contents of **config.json** into that string. This means we are now ready to use the **defaultConfiguration** variable in our code as we reference it above.

```
import (
    _ "embed"
)

//go:embed config.json
var defaultConfig string
```

We import the “**embed**” library with the tag of `_` because this library isn’t directly accessed or used, instead we take advantage of the side effect of importing the library. We do it like this because it is the most idiomatic way to do embeds in Go.

Testing

For testing the configuration system, there are two elements that we need to test:

- Loading the default configuration from the embedded resource when a configuration file does not exist
- Loading the configuration from the configuration file when the file exists

In order to test these two components, I wrote a very simple program to test this module:

```
func TestConfigurationSystem() {
    config := NewApplicationConfig("testing_configuration.json")
    fmt.Printf(
        `
        LOADED TEST CONFIGURATION:
        ProxyAddress: %s,
        RedirectAddress: %s,
        ApiAddress: %s,
        LoggerPath: %s,
        DefaultApiUrl: %s,
        ApiWhitelist: %v,
        Emails: %v
        `, config.ProxyAddress, config.RedirectAddress, config.ApiAddress,
        config.LoggerPath, config.DefaultApiUrl, config.ApiWhitelist, config.Emails)
}
```

This code simply loads a new ApplicationConfig with the path **test_configuration.json** as we would in our main program. Then after we load the configuration, we simply print out all the fields from the struct in order to ensure that it has correctly loaded the data.

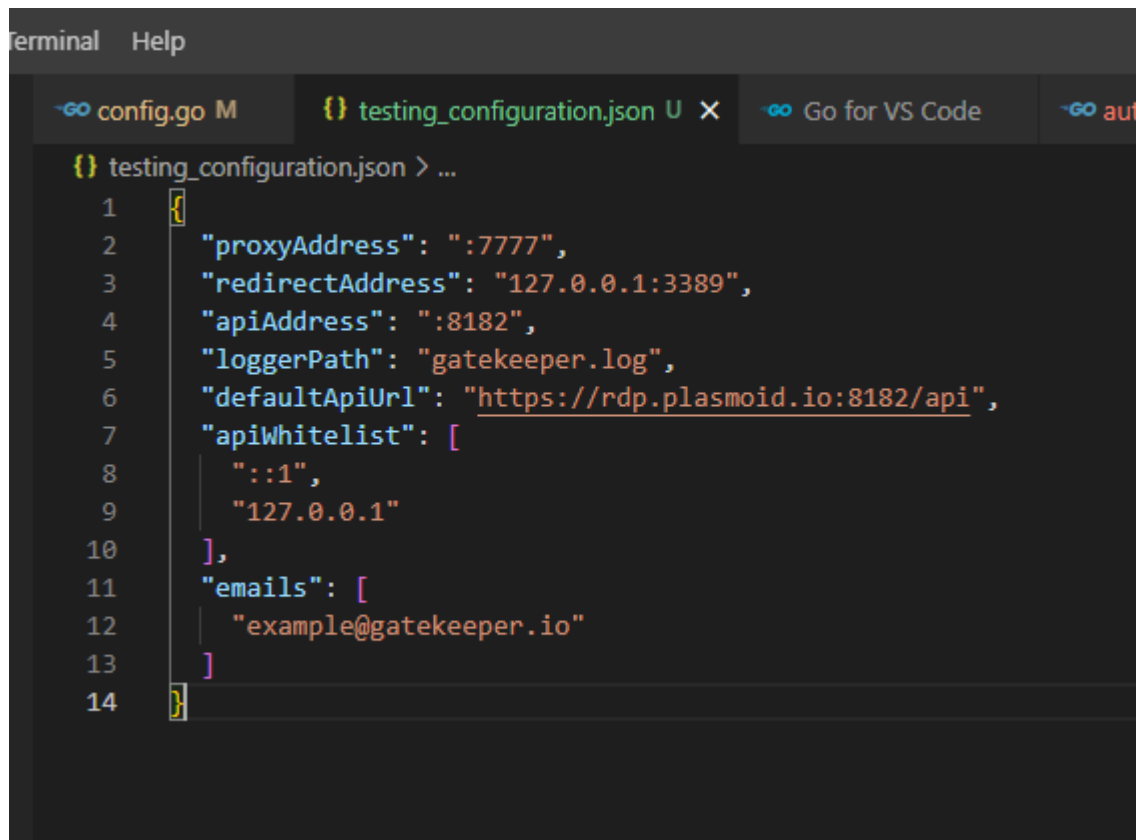
Running the test for the first time, we get this message:

```
PS C:\Users\Saif\dev\gatekeeper> go run .

LOADED TEST CONFIGURATION:
ProxyAddress: :7777,
RedirectAddress: 127.0.0.1:3389,
ApiAddress: :8182,
LoggerPath: gatekeeper.log,
DefaultApiUrl: https://rdp.plasmoid.io:8182/api,
ApiWhitelist: [::1 127.0.0.1],
Emails: [example@gatekeeper.io]
```

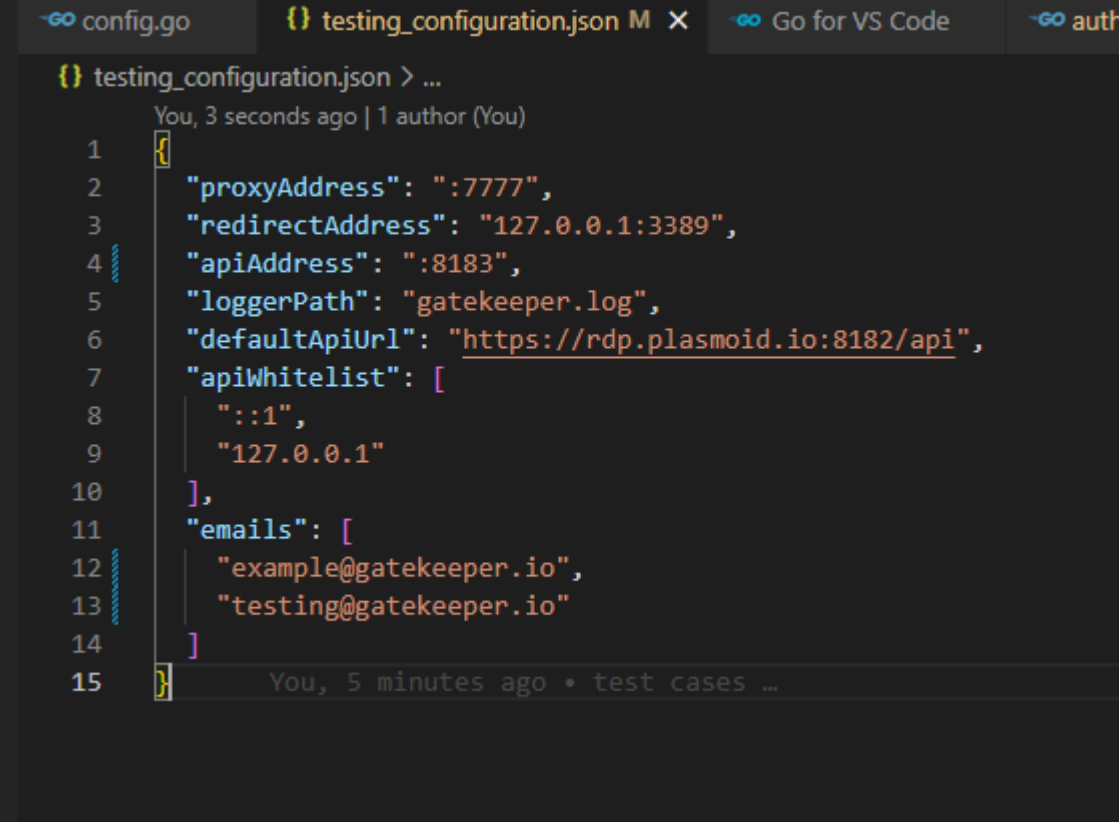
This first test was run when the configuration file we reference does not exist, we can see that it has loaded the configuration from our default configuration file and all the fields correspond with the fields defined in the default configuration.

We can also see that it has created the configuration file and saved it to the file system.

A screenshot of a Visual Studio Code editor window. The top bar shows 'Terminal' and 'Help' menus. Below the menu bar, there are tabs for 'config.go M', 'testing_configuration.json U X', 'Go for VS Code', and 'aut'. The active tab is 'testing_configuration.json U X', which displays the JSON content of the file. The JSON is as follows:

```
{
  "proxyAddress": ":7777",
  "redirectAddress": "127.0.0.1:3389",
  "apiAddress": ":8182",
  "loggerPath": "gatekeeper.log",
  "defaultApiUrl": "https://rdp.plasmoid.io:8182/api",
  "apiWhitelist": [
    "::1",
    "127.0.0.1"
  ],
  "emails": [
    "example@gatekeeper.io"
  ]
}
```

Now, let's change a field in the test configuration and re-run our testing function. We will add an extra email to the "emails" list, and change the apiAddress port from 8182 to 8183.

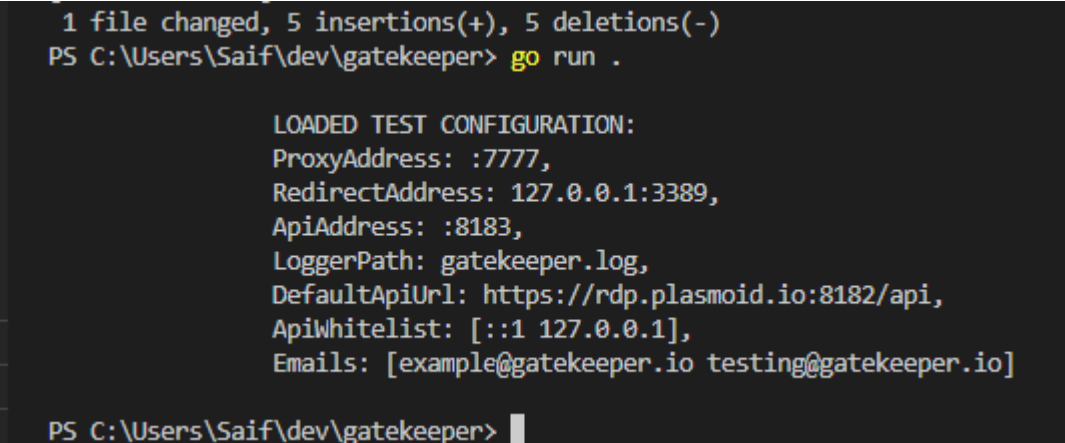


The screenshot shows a VS Code editor with the file `testing_configuration.json` open. The file contains the following JSON configuration:

```
{
  "proxyAddress": ":7777",
  "redirectAddress": "127.0.0.1:3389",
  "apiAddress": ":8183",
  "loggerPath": "gatekeeper.log",
  "defaultApiUrl": "https://rdp.plasmoid.io:8182/api",
  "apiWhitelist": [
    "::1",
    "127.0.0.1"
  ],
  "emails": [
    "example@gatekeeper.io",
    "testing@gatekeeper.io"
  ]
}
```

This is what our new configuration file looks like.

This is the log of the output of our testing function



The screenshot shows a PowerShell terminal window with the following output:

```
1 file changed, 5 insertions(+), 5 deletions(-)
PS C:\Users\Saif\dev\gatekeeper> go run .

LOADED TEST CONFIGURATION:
ProxyAddress: :7777,
RedirectAddress: 127.0.0.1:3389,
ApiAddress: :8183,
LoggerPath: gatekeeper.log,
DefaultApiUrl: https://rdp.plasmoid.io:8182/api,
ApiWhitelist: [::1 127.0.0.1],
Emails: [example@gatekeeper.io testing@gatekeeper.io]

PS C:\Users\Saif\dev\gatekeeper>
```

As we can see, the function works well and the configuration module successfully reads from the configuration file.

Testing Framework

Up until this point, I was doing testing by writing some simple functions to test and then temporarily commenting out code in my main function to run my test functions. This isn't very nice and there is a better way of doing testing in Go, I can use Go's official framework. The reason that I chose to move to this is because I can test ALL my modules at once, without having to modify any code.

The way to use Go's testing framework is to first import the "testing" module, and then write a function with the method signature: **func TestXxx(*testing.T)** where **Xxx** does not start with a lower case letter. Then I can use the command-line tool **go test**. The file containing the test must also end in the suffix "_test.go".

Now, with this great testing utility, I can write tests for my modules.

Testing Modification (Connection Piping Logic)

For the connection piping logic, I used a lazy testing function. I can move it from that one function into my unified testing framework method.

This is what the function looked like previously:

```
func testConnectionPiping() {
    listener, err := net.Listen("tcp", ":8080")
    for {
        incoming, err := listener.Accept()
        if err != nil {
            panic(err)
        }
        redirect, err := net.Dial("tcp", "127.0.0.1:3389")
        if err != nil {
            panic(err)
        }
        connectionPipe := pipe.NewConnectionPipe(incoming, redirect)
        connectionPipe.pipe()
    }
}
```

Now, I can move this to my testing framework. To do this, I will follow the instructions I listed above. First, I made a new file in the same directory called **pipe_test.go**. Then, I declared the package name to be “pipe”, the same as the connection piping package name itself. After that, I declared a new function called **TestConnectionPiping(*testing.T)** and copy and pasted the same logic as above. After this, instead of panicking with each error, I instead return the error back to the testing framework.

This is what it looks like:

```
func TestConnectionPiping(t *testing.T) {
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        t.Error(err)
    }
    for {
        incoming, err := listener.Accept()
        if err != nil {
            t.Error(t)
        }
        redirect, err := net.Dial("tcp", "127.0.0.1:3389")
        if err != nil {
            t.Error(t)
        }
        connectionPipe := NewConnectionPipe(incoming, redirect)
        connectionPipe.Pipe()
    }
}
```

This is nice, but there are a few things I don't like about this method of testing this specific behaviour:

- It requires the tester to manually start some form of TCP connection with the test program and observe there are no issues
- This method of human-assisted pairing does not ensure the byte-by-byte integrity of the entire connection piping stream.
- This also locks up the testing framework as it is an infinite loop of accepting incoming connections, acting more as a server rather than a test.

Instead of this, I have decided to write a new method of testing. This method of testing will be completely automated, and will work by starting a server and a proxy server, and then connecting to the proxy server and sending some data and receiving data from the proxy, and then verifying the integrity of the data both sent and received to ensure that all communication is being proxied on a byte-perfect level.

I wanted to ensure that my connection piping module worked as expected, allowing the data to be proxied on a byte-perfect level between the client and the server. To accomplish this, I decided to create a fully automated test that simulates the entire process, including starting a server, a proxy server, and a client.

I began by generating two payloads, **serverToClientPayload** and **clientToServerPayload**, filled with random bytes. I used these payloads to simulate data transfer between the server and the client. The random data helped me ensure that the test covers various edge cases and different byte sequences.

Next, I set up two goroutines to simulate the proxy and backend servers. The first goroutine starts a proxy server listening on port **8080**. When it receives a connection, it dials the backend server on port **8081**, establishes a connection, and creates a connection pipe between the incoming client connection and the backend server connection. The piping of data between the two connections is then initiated. The use of goroutines allows me to run the proxy and backend servers concurrently, simulating a real-world scenario.

The second goroutine sets up a backend server listening on port 8081. When it receives a connection, it writes the **serverToClientPayload** to the client and reads the data from the client until it has received x number of bytes. It then compares the received data with the **clientToServerPayload**. If they don't match, an error is raised to indicate that the data transfer was not byte-perfect.

In the main test thread, I simulated the client's behaviour by connecting to the proxy server on port 8080 and reading data from the proxy until I received x bytes. I then compared the received data with the **serverToClientPayload**. If they didn't match, I raised an error to indicate that the data transfer was not byte-perfect. Finally, I wrote the **clientToServerPayload** to the proxy, simulating the client sending data to the server.

Throughout the test, I used the **byteSliceEq** function to compare byte slices, ensuring that the data sent and received by the client and server was indeed equal.

By designing the unit test this way, I created a realistic environment that allowed me to thoroughly test the connection piping module and verify its functionality. This gave me confidence that my TCP proxy would work as intended when deployed in a production environment.

This is what the code looks like:

```

import (
    "crypto/rand"
    "net"
    "testing"
)

const (
    PAYLOAD_LENGTH = 8192
)

func TestConnectionPiping(t *testing.T) {
    serverToClientPayload := make([]byte, PAYLOAD_LENGTH)
    clientToServerPayload := make([]byte, PAYLOAD_LENGTH)

    _, err := rand.Read(serverToClientPayload)
    if err != nil {
        t.Error(err)
    }

    _, err = rand.Read(clientToServerPayload)
    if err != nil {
        t.Error(err)
    }

    go (func() {
        listener, err := net.Listen("tcp", ":8080")
        if err != nil {
            t.Error(err)
        }
        incoming, err := listener.Accept()
        if err != nil {
            t.Error(err)
        }
        redirect, err := net.Dial("tcp", ":8081")
        if err != nil {
            t.Error(err)
        }
        pipe := NewConnectionPipe(incoming, redirect)
        pipe.Pipe()
    })()

    go (func() {

```

```

    listener, err := net.Listen("tcp", ":8081")
    if err != nil {
        t.Error(err)
    }

    incoming, err := listener.Accept()
    if err != nil {
        t.Error(err)
    }

    incoming.Write(serverToClientPayload)

    data := []byte{}
    dataSize := 0
    for dataSize < PAYLOAD_LENGTH {
        buf := make([]byte, 2048)
        length, err := incoming.Read(buf)
        if err != nil {
            t.Error(err)
        }
        dataSize += length
        data = append(data, buf[:length]...)
    }

    if !byteSliceEq(data, clientToServerPayload) {
        t.Error("invalid payload sent from client ->
server")
    }
}

conn, err := net.Dial("tcp", ":8080")
if err != nil {
    t.Error(err)
}

data := []byte{}
dataSize := 0
for dataSize < PAYLOAD_LENGTH {
    buf := make([]byte, 2048)
    length, err := conn.Read(buf)
    if err != nil {
        t.Error(err)
    }

```



```

    }
    data = append(data, buf[:length]...)
    dataSize += length
}
if !byteSliceEq(data, serverToClientPayload) {
    t.Errorf("invalid payload sent from server -> client\n
SENT: %v\n EXPECTING: %v", data, serverToClientPayload)
}

conn.Write(clientToServerPayload)
}

func byteSliceEq(a []byte, b []byte) bool {
    if len(a) != len(b) {
        return false
    }

    for i := range a {
        if a[i] != b[i] {
            return false
        }
    }

    return true
}

```

There are some further optimisations and fixes I can make to this code.

Use 'defer' to close connections and listeners

When I create network connections and listeners, it's essential to close them properly to avoid resource leaks. Every time I open a listener or connection, I handle errors, but now I have added **defer conn.Close()** or **defer listener.Close()** to ensure that there are no resource leaks.

Avoid hardcoding port numbers

Instead of hardcoding the port numbers (8080 and 8081), I use an empty port now (":0") to pick a free port, and I retrieve the port number itself using the **Addr()** method. I do this instead to avoid potential port conflicts while running tests.

Use in-built copy function for reading and writing TCP stream data

Instead of manually reading and writing data in a loop, I use the **io.CopyN** function to copy a specified number of bytes between two **io.Reader** and **io.Writer** interfaces.

This is what the code looks like after the optimizations and tweaks:

```
func TestConnectionPiping(t *testing.T) {
    proxyPortChan := make(chan int)
    backendPortChan := make(chan int)

    serverToClientPayload := make([]byte, PAYLOAD_LENGTH)
    clientToServerPayload := make([]byte, PAYLOAD_LENGTH)

    _, err := rand.Read(serverToClientPayload)
    if err != nil {
        t.Error(err)
    }

    _, err = rand.Read(clientToServerPayload)
    if err != nil {
        t.Error(err)
    }

    go (func() {
        listener, err := net.Listen("tcp", ":0")
        if err != nil {
            t.Error(err)
        }
        defer listener.Close()

        backendPortChan <- listener.Addr().(*net.TCPAddr).Port

        incoming, err := listener.Accept()
        if err != nil {
            t.Error(err)
        }
        defer incoming.Close()

        incoming.Write(serverToClientPayload)

        data := &bytes.Buffer{}
```

```

    _, err = io.CopyN(io.MultiWriter(data, io.Discard), incoming,
PAYLOAD_LENGTH)
    if err != nil && err != io.EOF {
        t.Error(err)
    }

    if !bytes.Equal(data.Bytes(), clientToServerPayload) {
        t.Error("invalid payload sent from client -> server")
    }
}

backendPort := <-backendPortChan

go (func() {

    listener, err := net.Listen("tcp", ":0")
    if err != nil {
        t.Error(err)
    }
    defer listener.Close()

    proxyPortChan <- listener.Addr().(*net.TCPAddr).Port

    incoming, err := listener.Accept()
    if err != nil {
        t.Error(err)
    }
    defer incoming.Close()
    redirect, err := net.Dial("tcp", fmt.Sprintf(":%d", backendPort))
    if err != nil {
        t.Error(err)
    }
    pipe := NewConnectionPipe(incoming, redirect)
    pipe.Pipe()
}())

proxyPort := <-proxyPortChan

t.Logf("Initializing connection piping test with proxy running on port %d
and backend running on port %d", proxyPort, backendPort)

conn, err := net.Dial("tcp", fmt.Sprintf(":%d", proxyPort))

```

```

if err != nil {
    t.Error(err)
}

data := &bytes.Buffer{}
_, err = io.CopyN(io.MultiWriter(data, io.Discard), conn, PAYLOAD_LENGTH)
if err != nil && err != io.EOF {
    t.Error(err)
}
if !bytes.Equal(data.Bytes(), serverToClientPayload) {
    t.Errorf("invalid payload sent from server -> client\n SENT: %v\n
EXPECTING: %v", data, serverToClientPayload)
}

conn.Write(clientToServerPayload)
}

```

I have made sure the test is working as expected, by testing my test. I have tested it specifically when I know it's going to fail and ensure that it fails, and specifically with data I know it will pass with and ensure that it passes.

```

ok      github.com/saifsuleman/gatekeeper/config    0.288s
=== RUN TestConnectionPiping
pipe_test.go:88: Initializing connection piping test with proxy running on port 56937 and backend running on port 56936
--- PASS: TestConnectionPiping (0.00s)
PASS

```

Testing Modification (Configuration System Module)

Rewriting the initial configuration system test with the same instructions we define for unit testing, I created a file in the **config** directory called **config_test.go** and rewrote the logic as a unit test:

```

package config

import (
    "fmt"
    "testing"
)

func TestConfigurationSystem(t *testing.T) {
    config, err := NewApplicationConfig("testing_configuration.json")
}

```

```

if err != nil {
    t.Error(err)
}

fmt.Printf(
    `
    LOADED TEST CONFIGURATION:
    ProxyAddress: %s,
    RedirectAddress: %s,
    ApiAddress: %s,
    LoggerPath: %s,
    DefaultApiUrl: %s,
    ApiWhitelist: %v,
    Emails: %v
    `, config.ProxyAddress, config.RedirectAddress, config.ApiAddress,
    config.LoggerPath, config.DefaultApiUrl, config.ApiWhitelist, config.Emails)
}

```

Authentication Data Handler

Now that we have our configuration system finished and our TCP piping done, we can begin to work on our authentication handler. This has been talked about in the design section, but this handler will be responsible for reading and writing to the list of the authenticated IP addresses. Firstly, there will be the struct we have to define for this handler, this struct will hold the information of the whitelist cache itself and the file path to the whitelist file.

```

// IP whitelist handler for the proxy
type ProxyAuthHandler struct {
    WhitelistFilepath string // string field of the path to the
    whitelist file
    Whitelist          []string // list of IP addresses to
    represent the whitelist
}

```

As you can see, this struct itself is very simple in its structure, however we will now begin to implement the most important functions of the whitelist handler.

Firstly, we will need a function to save the contents of the whitelist array to the file itself - this will work by first fetching the file path of the whitelist, opening (or creating) the file and saving a reference of it to a variable, and then using a JSON encoder to encode whitelist as a JSON array and then saving the contents of what that returns to the file itself. Once the function has ended, we will also need to close the file.

```
// function to save the contents of the Whitelist array to the
// file
func (p *ProxyAuthHandler) Save() error {
    // pointer to the whitelist file
    var file *os.File

    // if whitelist file does not exist
    if _, err := os.Stat(p.WhitelistFilepath); os.IsNotExist(err)
{
        // create the file and assign it to our 'file' variable
        file, err = os.Create(p.WhitelistFilepath)
        // if an error is returned, return the error
        if err != nil {
            return err
        }
    } else {
        // if the file does exist, open it and assign the
        // opened file to our 'file' var
        file, err = os.OpenFile(p.WhitelistFilepath,
os.O_WRONLY, 0644)
        // if an error is returned, return the error
        if err != nil {
            return err
        }
    }

    // defers closing the file until after the function ends
    defer file.Close()

    // creates a new json encoder, encodes the Whitelist as a
    // reference and returns the encoded string
    return json.NewEncoder(file).Encode(&p.Whitelist)
}
```

This code follows the routine described above. As mentioned, we need a way to close the file once the function has ended, and for this we use Go's **defer** function mentioned above.

Now that we have a way to save our whitelist once it's been changed, we need to add some functions in order to modify that whitelist.

This was my initial idea for functions to add and remove from the list of whitelisted IP addresses:

```
/**
**  Functions to add and remove IP addresses
**  from the whitelist.
**/

func (p *ProxyAuthHandler) AddWhitelistIP(ip string) error {
    p.Whitelist = append(p.Whitelist, ip)
    return p.Save()
}

func (p *ProxyAuthHandler) RemoveWhitelistIP(ip string) error {
    for index, v := range p.Whitelist {
        if v == ip {
            last := len(p.Whitelist) - 1
            p.Whitelist[index], p.Whitelist[last] =
p.Whitelist[last], p.Whitelist[index]
            p.Whitelist = p.Whitelist[:last]
            return p.Save()
        }
    }

    return fmt.Errorf("IP address is not whitelisted!")
}
```

This code is fine, for adding a whitelist we just append to the array and return the response of saving the whitelist. For removing from the list of IP addresses, we simply loop through the list to find the index of the existing whitelist, and then splice the list at that index by swapping the value at the index and the value at the end, and then setting the list to be its own value without the last value.

While it works, this code is slightly nested, and we do not have any validation of ensuring that when adding an IP, that IP we are adding does not already exist. We can take the logic from the `RemoveWhitelistIP` function and take its method of locating the index of an element and put that into its own function called **GetWhitelistIPIndex**, which will return the index of an element in the list, or `-1` if the element is not present in the list.

```
// Function to find the index of an existing
// IP address if one exists, or returns -1
// to represent no indexes found
func (p *ProxyAuthHandler) GetWhitelistIPIndex(ip string) int {
    // Loop through every element in the list
    for i, v := range p.Whitelist {
        // if the element we are at is equal to the IP, return
index
        if v == ip {
            return i
        }
    }

    // return -1 to represent no indexes found
    return -1
}
```

Now that we have this function, we can modify our last two functions for adding and removing the list to include this function for better code readability and safety.

This is what our new **add** and **remove** functions look like:

```
/**
** Functions to add and remove IP addresses
** from the whitelist.
**/
func (p *ProxyAuthHandler) AddWhitelistIP(ip string) error {
    // precondition check to ensure that the IP does not
    // already exist in this list.
    if GetWhitelistIPIndex(ip) > -1 {
        return fmt.Errorf("IP address already exists in the
whitelist!")
    }
}
```



```

    // appends to the whitelist
    p.Whitelist = append(p.Whitelist, ip)

    // saves the contents of whitelist to file
    // and returns the error if any
    return p.Save()
}

func (p *ProxyAuthHandler) RemoveWhitelistIP(ip string) error {
    // gets the index of this IP and returns error if not exists
    index := GetWhitelistIPIndex(ip)
    if index == -1 {
        return fmt.Errorf("IP address is not whitelisted!")
    }
    // swaps the elements of this IP and last and then changes
    // length of list
    last := len(p.Whitelist) - 1
    p.Whitelist[index], p.Whitelist[last] = p.Whitelist[last],
    p.Whitelist[index]
    p.Whitelist = p.Whitelist[:last]

    // saves the content of the modified whitelist to the file
    return p.Save()
}

```

Now that we have all the fields and functions defined for our struct, we can create the constructor. This constructor should take its efforts to ensure that the whitelist file exists, open the file, and decode the existing whitelist info from the file. This essentially is just loading the whitelist that was previously stored so that the program retains its persistence in case of restarts.

This is the code for our ProxyAuthHandler constructor - it simply does what was described above. You can see the comments for a line-by-line explanation

```

// constructor for proxy auth handler - loads the file
// and decodes the existing whitelist as JSON and loads into array
// field
func NewProxyAuthHandler(filepath string) (ProxyAuthHandler,
error) {
    // ProxyAuthHandler variable
    var handler ProxyAuthHandler

```

```

// checks if file is present, if not:
if _, err := os.Stat(filepath); os.IsNotExist(err) {
    // creates the file
    file, err := os.Create(filepath)
    // error handling
    if err != nil {
        return handler, fmt.Errorf("line 21: %s", err)
    }
    // attempts to write an empty JSON list and handles any
errors
    if _, err := file.Write([]byte("[]")); err != nil {
        return handler, fmt.Errorf("line 24: %s", err)
    }
    // closes the file as we are now done with it
    file.Close()
}
// opens the file and handles errors
file, err := os.Open(filepath)
if err != nil {
    return handler, fmt.Errorf("line 29: %s", err)
}

// defers closing the file until after function ends
defer file.Close()

// local variable for whitelist to be loaded
var whitelist []string

// decode the JSON of the file into the whitelist ptr and
handles errors
if err := json.NewDecoder(file).Decode(&whitelist); err !=
nil {
    return handler, fmt.Errorf("line 36: %s", err)
}

// instantiates the proxy auth handler struct
handler = ProxyAuthHandler{
    WhitelistFilepath: filepath,
    Whitelist:         whitelist,
}

```

```
// returns handler and no error to represent successful load
return handler, nil
}
```

Multi Factor Authentication Handler

Now that we have the core systems out of the way, it is time to work on some of the most important functionality of the program. The Multi Factor Authentication Handler is this program's unique selling point, the problem that it solves. We need a way for users to add new IP addresses to the IP whitelist in an easy-to-use manner. The way that we have solved this in our analysis and design section is by initially, when somebody attempts to connect to the proxy it blocks the connection and sends email alerts to our list of administrators, in which they click on a link in order to authenticate the IP address. This is what we will be implementing now.

The flow of this algorithm will be that when somebody attempts to connect for the first time, it generates a cryptographically secure authentication code, packages an alert into an email and then sends that email to all our administrators. In this email, there will be a link that accesses our **/api/authenticate** endpoint and includes our cryptographically secure code as a form value in that link. That means that when an administrator clicks on the link, the server will notice that, fetch the authentication code from the link and then validate the code.

In order to handle our cache of authentication codes, what we should do is that when a code is generated, we update a dictionary that maps authentication codes to different IP addresses. The reason that we do this is so that when an authentication request is made to our API, it gets the code and then determines exactly which IP address is attempting to be verified by getting the value from the map. This code will interact with our proxy authentication data handler to add an IP address to the whitelist once a valid code has been sent.

We will implement this part now. First, there will be our **MultiFactorAuth** struct that will hold all the data of this module.

```
// multi-factor authentication
type MultiFactorAuth struct {}
```

Now, we can add fields to our struct. First, there will be our **ProxyAuthHandler** which is our data handling of our whitelist file itself, and there will be a dictionary called **AuthCodes** which will map authentication codes to IP address string literals.

```
// multi-factor authentication
type MultiFactorAuth struct {
    ProxyAuthHandler ProxyAuthHandler // instance of
    ProxyAuthHandler (Whitelist file storage handler)
    AuthCodes         map[string]string // a map of authentication
    codes to the IP addresses they should whitelist
}
```

Now, we can add some functions that will add and read data from our **AuthCodes** map. These functions will be a function to determine whether or not a code exists, a function to get the IP address a code corresponds to, if one exists, and a function to **generate** a new code and update our **AuthCodes** dictionary to include our newly generated code-IP pair.

Function to determine whether a code exists (returning a boolean):

```
// checks whether or not the cryptographically secure code for an
IP exists
func (mfa *MultiFactorAuth) DoesCodeExist(code string) bool {
    _, has := mfa.AuthCodes[code]
    return has
}
```

This function simply fetches a value and whether it exists (bool) from the map by using the **code** parameter to lookup the map, and returns whether it exists as a boolean.

Function to get the IP address an authentication code corresponds to:

```
// function to get a code's IP and if it exists
func (mfa *MultiFactorAuth) GetCodeIP(code string) (string, bool)
{
    // searches the map
    ip, has := mfa.AuthCodes[code]
    if !has {
        return "", false
    }
    return ip, true
}
```

This code simply looks up the map, and returns the values you get from if you index a map. Reflecting back, this code looks very inefficient, so this is something that we will change later in our code refactoring, however for development purposes it is fine to be clear on what we are doing.

Function to generate a new authentication code:

We will now write our function to generate a new random authentication code and update our **AuthCodes** map. This code will be a 256-bit secure code, so we should use a buffer of a fixed-length and then dump random bytes into that buffer using Go's 'rand' library. After we have done this, we need a way to encode these random bytes back into strings, and they have to be safe to embed into URLs, so we will use Base64 URL encoding for this.

This is what the code looks like:

```
// uses the 'rand' library to generate a 256-bit secure base64
unique code
func (mfa *MultiFactorAuth) GenerateCode(ip string) (string,
error) {
    // variable for our code to return later
    key := ""

    // while the key is empty or the key is already present in
the map
    for key == "" || mfa.DoesCodeExist(key) {
        // creates a buffer of 32 bytes (32 * 8 = 256 bits)
        buf := make([]byte, 32)
```

```

        // uses rand to dump random bytes into the buffer and
        handles errors
        _, err := rand.Read(buf)
        if err != nil {
            return key, err
        }

        // encodes our random buffer into a base64 string and
        assign that to our 'key'
        key = base64.RawURLEncoding.EncodeToString(buf)
    }

    // updates the AuthCodes map to include the key as the key
    and ip as the value
    mfa.AuthCodes[key] = ip

    // returns the key (secure code) with a nil error (represents
    success)
    return key, nil
}

```

In this code, first we declare our variable **key** which is what we will return as an empty string. After this, we assign an empty 32 byte buffer ($32 * 8 = 256$ bit) and use `rand` to dump random bytes into this buffer and then encode it into a string. After this, we encode the key into the buffer and update the **AuthCodes** map. Finally, we return the generated key and a nil error. We handle any errors by simply returning them to the calling function, which is the idiomatic way to do this in Go.

HTTP Server Handler

Now that we have handled our authentication code system, we can begin to implement another important part of this module. We have briefly discussed how this module is very dependent on the route map and HTTP server handling for our REST API. This includes functionality such as authenticating IP addresses and viewing the log. We will implement our HTTP server API here in this module.

Firstly, we will need a **HTTP Router**. We will be using an external library to handle our HTTP routing for us so that we can easily declare routes. This is for a few reasons. The standard **net/http** package in Go does not provide a lot of routing features. For example, you can define routing for each individual HTTP method, you cannot declare routes using regular expressions, and much more.

I have decided to use the **gorilla/mux** router in Go for its simplicity, ease of use and network performance. Using this library is extremely useful and simple, as I will show you in the code below.

Routes in the route-map for this router will be defined with two components: first there will be the string of the route address that we are creating the definition for, and then there will be the handler function of this route. A HTTP handler function in Go accepts two parameters: a pointer to the HTTP request itself, which contains all the information of the request such as the form values. For our **/api/authenticate** route we will be using the **code form** value to get the authentication code they included in the link, so this is something extremely useful for us. Another parameter that is required in a HTTP handler function is the HTTP ResponseWriter. This is a writer that you write to in order to send a response back to the browser or service that requested this API.

Here is an example of a HTTP handler function that responds to the browser with *"hello world"*:

```
func HandleExampleRequest(w http.ResponseWriter, r *http.Request){  
    _, _ = fmt.Fprint(w, "hello world!")  
}
```

Now that we have determined how our route handling will work, we can begin to implement it.

First, we will include our HTTP Router in our struct that we defined earlier. This is now what the struct will look like:

```
// multi-factor authentication
type MultiFactorAuth struct {
    ProxyAuthHandler ProxyAuthHandler // instance of
    ProxyAuthHandler (Whitelist file storage handler)
    AuthCodes         map[string]string // a map of authentication
    codes to the IP addresses they should whitelist
    Router             *mux.Router // a reference to our HTTP
    router handler
}
```

As you can see, this is the same exact struct that we had defined above, except now there is an extra field: **Router *mux.Router**. This is a new field named **Router** with the type of a reference to a **mux.Router**. The reason that we use a reference/pointer is because we can then update the router's data and change its route map without needing to re-assign the field.

Now, to instantiate this router, we will do this in our constructor that we define later, however it will look something like this:

```
Router = mux.NewRouter()
```

We can now begin to define our **Start()** function. This function will define our HTTP routes on our route map and begin listening on our HTTP server, which is the primary component of our REST API.

This function will be run in the context that the **Router** field has already been instantiated, so in this function we simply need to use our **Router** to define our routes, and then use the **net/http** module to listen on a specific port using our **Router**.

This is what the function will look like:

```
// starts our HTTP server
func (mfa *MultiFactorAuth) Start(address string) {
    // declares HTTP route-map
    mfa.Router.HandleFunc("/api/authenticate",
mfa.HandleAuthenticate)
    mfa.Router.HandleFunc("/api/log", mfa.wrapApiFunc("/api/log",
mfa.ViewLog))

    // prints to the console window the address the API server is
    listening on
    fmt.Printf("Listening on: %s\n", address)

    // uses 'http' module to listen on the address with our
    router and handles error
    if err := http.ListenAndServe(address, mfa.Router); err !=
nil {
        panic(err)
    }
}
```

This defines our routes by linking the route's literal as a string to a reference of a function. For example, we have **HandleAuthenticate** and **ViewLog** functions, so we pass in references to these subroutines/procedures. After we define our route-map, we print to the console the address we will be listening on and then use the **net/http** module to listen on the address passed in as a parameter and we handle any errors by throwing them.

As discussed in the design, we have a **wrapApiFunc** which is a **high-order function** as this returns a handler function for the route. The reason we do this is to generate a handler function that only runs if the user who accessed this API route is classified as an administrator. It checks whether or not the accessor is an administrator by checking if the IP address of the person who connected is part of the API IP Whitelist.

We will first define a function to check if a network connection has API access, this function will be used by our **wrapApiFunc** function.

This is what the **HasApiAccess(*net.Conn)** function will look like:

```
// returns whether or not a certain IP address has access to the
// API
func (mfa *MultiFactorAuth) HasApiAccess(r *http.Request) bool {
    // if the ApiWhitelist is empty, return true as all IPs are
    // allowed
    if len(mfa.ApiWhitelist) == 0 {
        return true
    }

    // splits the address into its host and port and gets the
    // address variable
    // the port is unused so we call it _, we also handle errors
    // by silently
    // returning false
    address, _, err := net.SplitHostPort(r.RemoteAddr)
    if err != nil {
        return false
    }
    // for every value in the ApiWhitelist
    for _, v := range mfa.ApiWhitelist {
        // if the value is equal to the address, return true
        if v == address {
            return true
        }
    }
    // no values were found so we return false
    return false
}
```

This function will work by first checking if the ApiWhitelist is empty, if this is the case, then the whitelisting feature has been disabled. If it is not empty, first it gets the address of the connection that made the HTTP request as a string. It splits the address and the port and only compares the address portion (i.e. the IP address of Fully Qualified Domain Name).

After we get the address string, we loop through the API Whitelist and if the value we are checking is equal to the address literal, we return true, else we return false.

Now that we have this function to check if a connection has been API Whitelisted, we can write our **wrapApiFunc** function.

This is my first draft of the **wrapApiFunc** function.

```
// this function is a generator function so that only requests
// that are from
// an API-allowed IP address is able to be called - this utilises
// callbacks
func (mfa *MultiFactorAuth) wrapApiFunc(path string, f func(w
http.ResponseWriter, r *http.Request)) func(w http.ResponseWriter,
r *http.Request) {
    return func(w http.ResponseWriter, r *http.Request) {
        if !mfa.HasApiAccess(r) {
            if address, _, err :=
net.SplitHostPort(r.RemoteAddr); err == nil {
                log.Printf("Unauthorized API attempt [%s]
from: %s\n", path, address)
            }
            _, _ = fmt.Fprint(w, "unauthorized")
            return
        }
        f(w, r)
    }
}
```

This function accepts the route path as a string, and a HTTP handler function, and it returns a new HTTP handler function that first checks if the connection request is whitelisted, and if it is not, we write an error to our logger and send an error response back to the request. If the request connection is whitelisted, we simply pass through to the callback HTTP handler function we were passed in as a parameter.

This function will be used for our HTTP routes that require administrator privileges to be accessed - one example for this is our **/api/log** route which allows the user to view the log of the proxy.

This will work where instead of passing a reference to a HTTP handler function like **mfa.HandleAuthenticate**, we will pass in **wrapApiFunc("/api/authenticate", mfa.HandleAuthenticate)**.

Now that we have the routes fully defined, we simply start our HTTP server listener by using the **net/http** module, and we handle any errors by throwing the error, which prints it to the console and halts the program.

Now that our routes are defined, we need to implement the functionality for the handler functions we reference. These functions are **mfa.HandleAuthenticate** and **mfa.ViewLog**.

Let's first implement the **ViewLog** handler function as it is the simplest to implement. This will simply get the **Logger** from the **MultiFactorAuth** struct and write a string of the cached log to the HTTP response writer. As this is a HTTP handler function, it needs to match the criteria of having two parameters: a **http.ResponseWriter** and a pointer to the HTTP request.

This is what the **ViewLog** code will look like:

```
// function to write the logger's cachedLog to a http response writer
// all return values of the logger is ignored, so we name them '_'
func (mfa *MultiFactorAuth) ViewLog(w http.ResponseWriter, _
    *http.Request) {
    _, _ = fmt.Fprint(w, string(mfa.Logger.CachedLog))
}
```

As you can see, when this function is called, it simply uses the **fmt** module to write the cached log as a string to the HTTP response writer. We also cast the **CachedLog** to a string as originally it is an array of bytes. We name the return values (and the pointer to the HTTP request) as **_** which signifies that these variables are unused, and this is the idiomatic way to do this in Go.

Now that we have implemented the handler function for our **/api/log** route, we can implement the code for our **/api/authenticate** route. This will follow the same function signature as before. This handler function will first get the **code** form value, check if it is empty, and if it is, it will return an error back to the browser. After this, it will get the IP address of this security code from our map and check if it is a valid code. If it is an invalid code, we will return an error to the browser and stop the function. If it is valid, however, we will delete the authentication code from the map as it is now being processed, and we don't want the code to be processed again. We then use our **ProxyAuthHandler** module to add the IP address to the whitelist, and if there is an error returned we will write the error to the browser, or we will return with "success".

This is what the code looks like:

```
// handler function for our /api/authenticate route
func (mfa *MultiFactorAuth) HandleAuthenticate(w
http.ResponseWriter, r *http.Request) {
    // gets the "code" form value
    code := r.FormValue("code")
    // if the code is empty, we return an error
    if code == "" {
        _, _ = fmt.Fprint(w, "you must enter a code")
        return
    }
    // gets the IP address of a certain secure code
    // from the map and checks if its a valid code
    ip, valid := mfa.GetCodeIP(code)
    // if invalid, write an error back to the browser
    if !valid {
        _, _ = fmt.Fprint(w, "invalid code")
        return
    }
    // delete the auth code from the map as its now being
    processed
    // and we don't want to authenticate it twice
    delete(mfa.AuthCodes, code)

    // adds this IP address to the IP whitelist
    err := mfa.ProxyAuthHandler.AddWhitelistIP(ip)

    // calculates a response to send back to the browser
    // if no error from adding IP to whitelist, return "success",
    // else, return the error
    var response string
    if err == nil {
        response = "success"
    } else {
        response = fmt.Sprintf("error: %s", err)
    }

    // writes our calculated response back to browser
    _, _ = fmt.Fprint(w, response)
}
```

Lastly, we need to implement our **SendEmailAlerts(string)** function which will send an email alert of a specific IP address connection to our list of administrators. First, this will generate our secure 2FA code, and then it will generate the link as a string. Then, it will generate the email's dialer using a specific username and password. Then it will generate a list of emails and iteratively go over all administrators and append another email to the list, each individual email in the list is to target a different administrator. Then, we use our dialer to send this list of emails.

```
// function to send email alerts to all the administrators
// for a connection attempt of a certain IP address
func (mfa *MultiFactorAuth) SendEmailAlerts(ip string) {
    // generates the secure unique code to identify
    // the IP in the email
    code, err := mfa.GenerateCode(ip)

    // if an error is returned, throw the error
    if err != nil {
        panic(err)
    }

    // uses the code to generate a link based off the
    // DefaultApiUrl struct field
    // and the secure code
    link := fmt.Sprintf("%s/authenticate?code=%s",
mfa.DefaultApiUrl, code)

    // gets the OS hostname to use in the email
    // and handles errors by throwing
    hostname, err := os.Hostname()
    if err != nil {
        panic(err)
    }

    // generates the text body of the email alert
    body := fmt.Sprintf("RDP Login Attempt from %s.\nClick below
to verify this IP.\n\n%s", ip, link)

    // uses the gomail library to generate a new SMTP dialer for
    // the email
    // and configures TLS to work appropriately
    dialer := gomail.NewDialer("smtp.gmail.com", 587,
"rdpgatekeeper@gmail.com", "Password123")
```

```

    dialer.TLSConfig = &tls.Config{InsecureSkipVerify: true}

    // array of emails to send all at once as a batch request to
    limit
    // network calls
    var messages []*gmail.Message

    // loops through all administrator email addresses
    for _, email := range mfa.Emails {
        // creates a new email message object and appends to
        the 'messages' array
        m := gmail.NewMessage()
        m.SetHeader("From", "RDP Gatekeeper
<rdpgatekeeper@gmail.com>")
        m.SetHeader("To", email)
        m.SetHeader("Subject", fmt.Sprintf("RDP Access Attempt
on machine: %s", hostname))
        m.SetBody("text/plain", body)
        messages = append(messages, m)
    }

    // uses the dialer to send the array of emails in one
    // batch network call and handles errors
    if err := dialer.DialAndSend(messages...); err != nil {
        panic(err)
    }
}

```

Now that we have all our subroutines finished, this is the final definition of our **MultiFactorAuth** struct.

```
// multi-factor authentication
type MultiFactorAuth struct {
    ProxyAuthHandler ProxyAuthHandler // instance of
    ProxyAuthHandler (Whitelist file storage handler)
    Emails            []string        // list of administrator
    email addresses
    AuthCodes         map[string]string // a map of authentication
    codes to the IP addresses they should whitelist
    DefaultApiUrl     string          // the API URL to encode
    in the links sent to the email
    ApiWhitelist      []string        // an IP address whitelist
    of the authenticated IP allowed to use administrator functions of
    the API
    Router            *mux.Router    // a reference to our HTTP
    router handler
    Logger            logger.Logger // an instance of our
    custom logger
}
```


Logger System

For the logger system, I implemented the same code as described in our design section.

This is the final code:

```
package logger

import (
    "fmt"
    "io/ioutil"
    "log"
    "os"
)

type Logger struct {
    File *os.File
    CachedLog []byte
}

func (l *Logger) Write(data []byte) (n int, err error) {
    fmt.Print(string(data))
    l.CachedLog = append(l.CachedLog, data...)
    return l.File.Write(data)
}

func (l *Logger) Close() error {
    return l.File.Close()
}

func InitializeLogger(filepath string) Logger {
    if _, err := os.Stat(filepath); os.IsNotExist(err) {
        file, err := os.Create(filepath)
        if err != nil {
            panic(err)
        }
        file.Close()
    }

    file, err := os.OpenFile(filepath, os.O_RDWR, 0644)
    if err != nil {
        panic(err)
    }
}
```

```

    }
    cached, err := ioutil.ReadFile(filepath)
    if err != nil {
        panic(err)
    }
    logger := Logger{
        File: file,
        CachedLog: cached,
    }
    log.SetOutput(&logger)
    return logger
}

```

As you can see in the code above, first we define our **Logger** struct, which contains a pointer to the file the logger is using to output its logs, and an array of bytes which is the cache of the log. This is effective because when logging something, we simply need to append to the array of bytes with the new log, and then we call a **Save()** function that simply dumps the array of bytes into the file output's pointer.

The logger struct we have implements two functions: **Write([]byte)** and **Close()** - with these two functions, the logger now implements the correct interface to be considered a logger for Go, so we can simply do **log.SetOutput(&logger)** in order to have all logs made from the program output to our custom log.

Lastly, our **InitializeLogger** function creates an instance of the Logger struct by successfully loading the file, loading the existing file cache into our CachedLog byte array, and successfully handling any errors and using the **log** module to set the output of Go's main logger to our custom logger, and then it returns the **logger** we have created.

Main Proxy TCP Listener

Now we have to write the code for our main server. This server is what will listen on our proxy listener port, accept all incoming connections, and pipe TCP communication if the IP is authenticated. It will call all the modules we have created and connect them together.

First, we will have our ProxyServer struct, which has all the fields necessary for the proxy server - the info for this struct was defined in our design.

```
// The struct for the main ProxyServer
// contains all the relevant data required to work
type ProxyServer struct {
    Address      string // the address this proxy server should
listen on and accept incoming connections
    Redirect     string // the address this proxy server should
pipe incoming connections to
    Connections  map[net.Conn]pipe.ConnectionPipe // a map of all
connections to connection pipe instances
    Auth         authentication.MultiFactorAuth // the instance of
the MultiFactorAuth object
    APIAddress   string // the address the API listener is
listening on
}
```

This fields this struct has is:

- The address the proxy server is listening on to accept incoming connections
- The address the proxy server is piping TCP data to
- A map of all connections which map to instances of connection pipes
- An instance of the MultiFactorAuth module
- The address the REST API listener should listen on

After this, I wrote the constructor for this struct. This constructor will take in an instance of the ApplicationConfig, and an instance of the logger. Then, it will instantiate a new ProxyServer based on this ApplicationConfig and return the ProxyServer it created.

```
// the main constructor for the ProxyServer struct
func NewProxyServer(config config.ApplicationConfig, logger
logger.Logger) ProxyServer {
```

```

    // instantiates a new ProxyAuthHandler which is responsible
    for maintaining the list
    // of whitelisted IP addresses
    proxyAuthHandler, err :=
authentication.NewProxyAuthHandler("whitelist.json")
    // if an error is returned, throw the error which halts the
    program
    if err != nil {
        panic(err)
    }
    // instantiates a new MFA instance which is required for
    email alerts & more
    auth := authentication.NewMFA(proxyAuthHandler, logger,
config.ApiWhitelist, config.DefaultApiUrl, config.Emails...)

    // constructs the struct and returns it
    return ProxyServer{
        Address:      config.ProxyAddress,
        Redirect:     config.RedirectAddress,
        Connections:  map[net.Conn]pipe.ConnectionPipe{},
        Auth:         auth,
        APIAddress:   config.ApiAddress,
    }
}

```

Now that this section is complete, I can begin to work on the listener. For our TCP listener, it will need to be able to handle connections separately in a goroutine, so that the handling of one connection does not block other incoming connections. In order to do this, I should write another function which will be my **handleConnection(conn net.Conn)** function. This is what that code looks like:

```

// connection handler function which is called upon every
// connection to
// the proxy server's TCP listener
func (p *ProxyServer) handleConnection(conn net.Conn) {
    // gets the IP address of the incoming connection
    ip := GetIP(conn)

    // Leverages the AuthHandler to determine whether or not this
    // IP address is whitelisted
    whitelisted := p.Auth.IsAuthenticated(ip)
}

```

```

    // if it's not whitelisted, log this event and
    // close the connection and return
    if !whitelisted {
        log.Printf("Connection dialled from %s - IP not
authenticated!\n", ip)
        _ = conn.Close()
        return
    }

    // log the successful connection
    log.Printf("Connection dialled from %s - IP
authenticated!\n", ip)

    // dial TCP to the target service of this proxy (used for
piping)
    redirect, err := net.Dial("tcp", p.Redirect)
    // if an error is returned, throw the error
    if err != nil {
        panic(err)
    }

    // instantiate a new connection pipe instance and pipe the
incoming connection
    // and the dialed TCP connection to the target service
    connectionPipe := pipe.NewConnectionPipe(conn, redirect)

    // updates the connection map with the network connection as
the key
    // and the connectionPipe as the value
    p.Connections[conn] = connectionPipe

    // as the connectionPipe.Pipe() is thread blocking, we can
defer
    // the execution of deleting this from the map because we
know that
    // this host function will only end once the connection pipe
has been terminated
    // (it's quite smart really)
    defer delete(p.Connections, conn)

    // using our connection pipe instance, we begin piping the

```

```
connection
    connectionPipe.Pipe()
}

// function to get an IP address of an existing network connection
// it takes the whole IP address and splits it at the colon and
// then returns the
// LHS (left-hand side) of that statement - for example:
// 51.146.6.229:5274 -> 51.146.6.229
func GetIP(conn net.Conn) string {
    return strings.Split(conn.RemoteAddr().String(), ":")[0]
}
```

This code works well and as intended, it handles the connection successfully and makes the correct calls to all the other modules we have implemented, I have done extensive testing and have encountered no issues.

Evaluation

Criteria Met

Success Criteria

Requirement	How to show evidence	Met?
User interface to configure the program	Screenshot of the user interface for the configuration wizard	No
Clear Instructions	Screenshot of the instructions to setup the program	Yes
Ability to get email alerts	Screenshot of the email alert you get from an attempted connection and the code and testing to show that it works	Yes
Connection log endpoint	Screenshot of the connection log and code and testing to show that it works	Yes
Ability to authenticate connections based on IP addresses	Code for the authentication system and testing to ensure that you cannot connect if you are not authenticated, and then authenticating and then seeing that you can connect once you are authenticated.	Yes
TCP connection proxy between two points for seamless communication	Code for the connection stream proxy and testing to show that it works	Yes
The option to close the program	Screenshot of button to close the program and testing to prove that the program is fully stopped once it is closed	Yes

Configuration file to configure the program in a simple format	Screenshot of JSON file used to configure the program after the installation setup has been run, and code for configuration loading and testing to show that it works	Yes
--	---	-----

Evidence

Ability to get email alerts



RDP Gatekeeper <rdpgatekeeper@gmail.com>

to me ▾

RDP Login Attempt from 90.250.8.59.

Click below to verify this IP.

<https://gatekeeper.plasmoid.io/authenticate?code=x1ZrneV1Dy1kLiQtnqv8t-EwIQ2I7E4xg4NpJiSULzM>

Above is the email that was sent to the list of administrators while testing the program. As you can see, the URL to click to verify the IP address is based on the format defined in the config, and it points to the REST endpoint to authenticate and has a parameter of the cryptographically secure unique code to verify the authentication request.

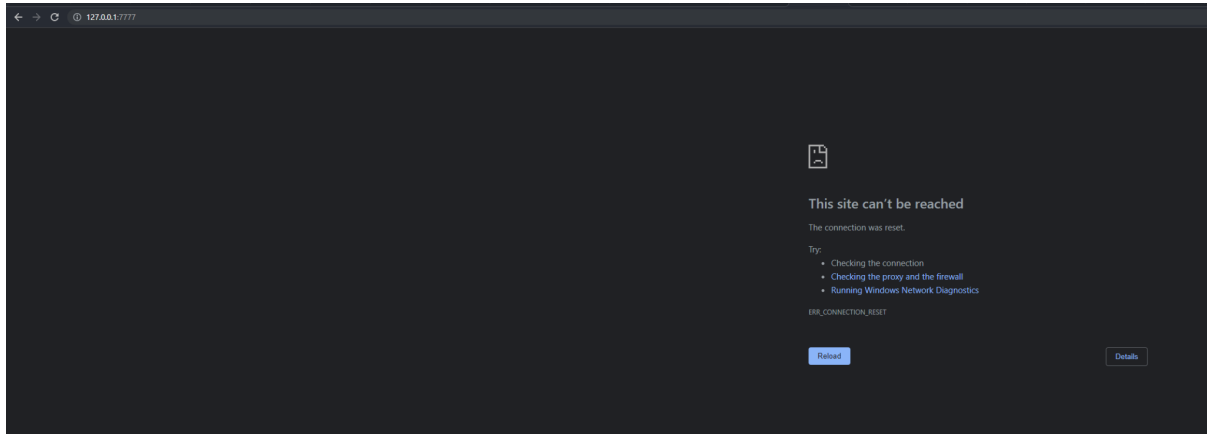
Connection log endpoint

```
localhost:8182/api/log
2021/06/28 14:29:08 Unauthorized API attempt [/api/log] from: 138.68.183.151
!
2021/06/28 14:24:03 Unauthorized API attempt [/api/authenticate] from: 90.250.8.59
2021/06/28 14:24:24 Unauthorized API attempt [/api/log] from: 192.168.0.1
```

This is what the connection log endpoint looks like, with the REST endpoint being on the HTTP server at **/api/log**

Ability to authenticate connections based on IP addresses

I launched the proxy, and set up a mock HTTP server - then I attempted to connect to the HTTP server through the proxy, however as we can see, the connection is terminated as our IP address is not authenticated.



Once this connection attempt was made, this email was sent to the administrators of the proxy.



RDP Gatekeeper <admin@hot.fish>

to me ▾

RDP Login Attempt from 127.0.0.1.

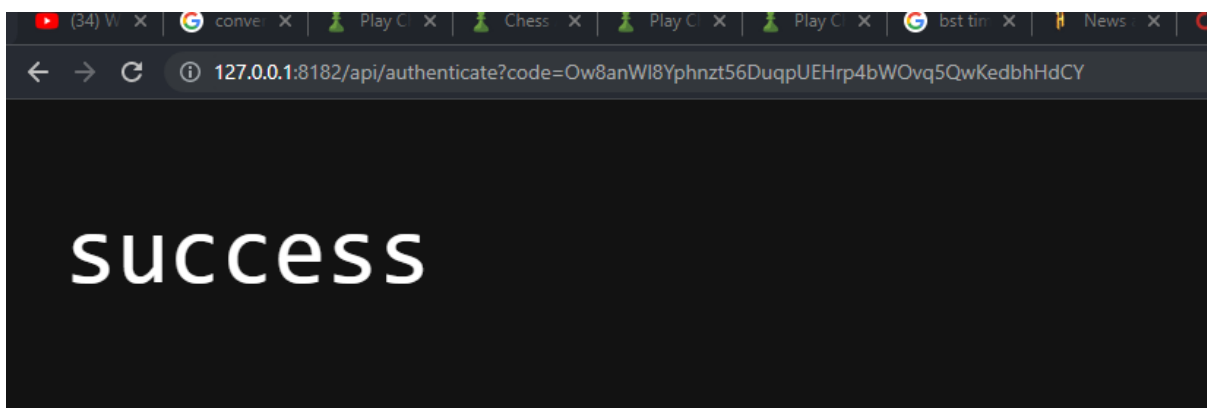
Click below to verify this IP.

<http://127.0.0.1:8182/authenticate?code=Ow8anWl8Yphnzt56DuqpUEHrp4bWOvq5QwKedbhHdCY>

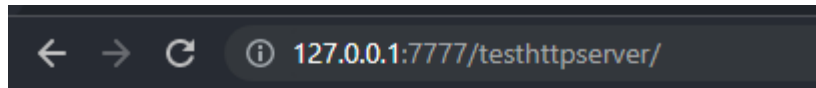
↩ Reply

➦ Forward

After clicking on the link to authenticate the IP address, we can see it successfully authenticated.



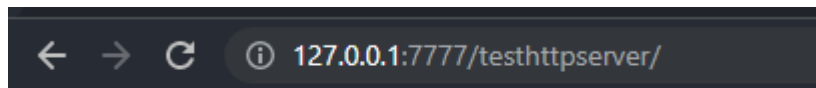
Now that we are authenticated, we can see we can access the HTTP server through the proxy, meaning this works.



hello there

TCP connection proxy between two points for seamless communication

We can see that this works from both ways. Firstly, we can see accessing a HTTP server through the proxy works and all the TCP data is piped through



hello there

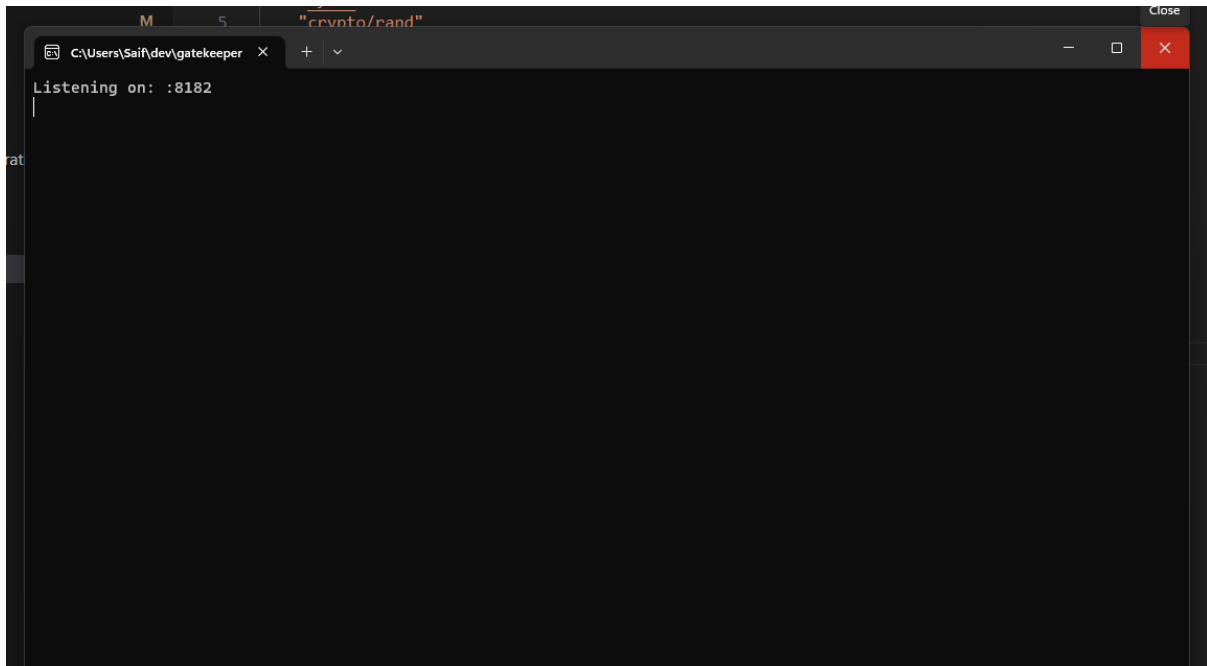
We can also verify that the proxy between two points works from our unit test that we wrote previously.

```
PASS
ok      github.com/saifsuleman/gatekeeper/config    0.298s
=== RUN   TestConnectionPiping
    pipe_test.go:88: Initializing connection piping test with proxy running on port 63123 and backend running on port 63122
--- PASS: TestConnectionPiping (0.00s)
PASS
ok      github.com/saifsuleman/gatekeeper/pipe    0.322s
PS C:\Users\Saif\dev\gatekeeper>
```

We can see the test passes with an 8KB payload, meaning that we know this works.

The option to close the program

Running the program opens this terminal window - pressing the red close button will stop the program cleanly.



Configuration file to configure the program in a simple format

We wrote a unit test in order to ensure that this module works and did all the testing and explanations in the previous section.

```
?      github.com/saifsuleman/gatekeeper/server      [no test files]
=== RUN   TestConfigurationSystem
    config_test.go:14:
        LOADED TEST CONFIGURATION:
        ProxyAddress: :7777,
        RedirectAddress: 127.0.0.1:3389,
        ApiAddress: :8182,
        LoggerPath: gatekeeper.log,
        DefaultApiUrl: https://rdp.plasmoid.io:8182/api,
        ApiWhitelist: [::1 127.0.0.1],
        Emails: [example@gatekeeper.io]

    --- PASS: TestConfigurationSystem (0.00s)
PASS
```

We can see the test passes.

This is what the configuration file format looks like:

```

onfig > {} testing_configuration.json > ...
You, yesterday | 1 author (You)
1  {
2    "proxyAddress": ":7777",
3    "redirectAddress": "127.0.0.1:3389",
4    "apiAddress": ":8182",
5    "loggerPath": "gatekeeper.log",
6    "defaultApiUrl": "https://rdp.plasmoid.io:8182/api",
7    "apiWhitelist": [
8      ":",
9      "127.0.0.1"
10   ],
11   "emails": [
12     "example@gatekeeper.io"
13   ]
14 }
You, yesterday • what a test

```

Usability Features

The program is very simple to use as you simply need to modify the configuration file which uses a JSON format, and then launch the program.

```

run > {} config.json > ...
You, 21 hours ago | 2 authors (Saif and others)
1  {
2    "proxyAddress": ":7777",
3    "redirectAddress": "127.0.0.1:5500",
4    "apiAddress": ":8182",
5    "loggerPath": "gatekeeper.log",
6    "defaultApiUrl": "http://127.0.0.1:8182/api",
7    "apiWhitelist": [":", "127.0.0.1", "138.68.183.151"],
8    "emails": ["saif@visionituk.com"]
9  }
10

```

As this is statically compiled, there is no need to install an additional interpreter or virtual machine backend such as Python or Java Virtual Machine. This means that this program will support a very large variety of devices with very simple setup.

Limitations

The biggest limitation the program has right now is lack of ability to configure the SMTP server credentials that the program uses for sending emails. This means that if an enterprise has a private self-hosted SMTP server, they won't be able to use their own SMTP server.

As of right now, the email authentication credentials will be hardcoded. I could add email authentication credentials and SMTP server setup to the `ApplicationConfiguration`, however this is an added level of complexity to the program that users may find confusing, and have potential to stop the program from working if configured incorrectly.

Another limitation is lack of support for other forms of two-factor authentication such as SMS based authentication. This will not be an issue for most stakeholders, however it may be convenient to authenticate a proxy connection from your SMS.

How to avoid these limitations

The lack of ability to configure SMTP server credentials could simply be solved from additional development time. I would first add the necessary fields to the `ApplicationConfig` struct, and then add a check on the application's pre-initialization to ensure that the SMTP details provided are correct.

Support for SMS based authentication would also be solved from additional development time, however I would also need to modify the MFA system slightly to be more abstract. It would be something like writing an interface that defines a function to send an alert, and then implementing this interface from email-based authentication and SMS-based authentication. Abstracting my system more would help it be more extendable.

Maintenance

The code is structured quite well - there are different packages for different modules which allows future development to be very easy. The code is commented and I have written unit tests for the most crucial parts of the programs allowing an easy way to see if a change breaks a feature.

Future maintenance could change the multi-factor authentication algorithm to be more abstract and instead depend on an interface to handle sending alerts and verifying authentication requests. Abstracting this module in this way using interfaces would allow for easy support for additional methods of authentication such as SMS two-factor authentication.

Post Development Testing

[This is a link to my video for post-development testing.](#)