Cross Network Secure Service Networking

Georges Ankenmann - 100935237
School Of Information and Technology
Carleton University
Ottawa, Canada
georgesankenmann@cmail.carleton.ca

Fahmida Haque - 101137370
School Of Information and Technology
Carleton University
Ottawa, Canada
fahmidahaque@cmail.carleton.ca

Jeff Bailey - 101020551
Department of Systems
and Computer Engineering
Carleton University
Ottawa, Canada
jeffbailey@cmail.carleton.ca

Abstract—Cross network secure service networking is starting to be a more common requirement these days where certain services must be accessible but proper security controls must also be implemented to prevent unauthorized access. Traditionally this can be accomplished by means of a virtual private network (VPN), but this does not always work with services or applications that can't run the required VPN software. Cross network secure service networking allows for services to communicate with each other across networks by establishing an encrypted and authenticated overlay network that allows for secure communication at the service level. This project will compare existing frameworks, and propose a new framework that offers security as the preliminary design consideration.

I. Introduction

Cross network secure service networking is starting to be a more common requirement these days where certain services must be accessible but proper security controls must also be implemented to prevent unauthorized access.

VPN are present in a modern connected world. One instance is in corporate environments which allows secure access to protected services from untrusted environments, such as the public internet. Secure access to corporate services can be accomplished by means of a client to site or a site to site virtual private network (VPN). Generally with a site to site VPN, the tunnel are usually established by dedicated hardware. With a client to site VPN, the tunnel is usually established by software on a client device [1] [3].

Overlay networks are common in modern data centres where they allow abstraction of certain network functions that can run on top of physical networks [5]. A common network overlay is Virtual eXtensible Local Area Network (VXLAN) [6]. This framework allows data centres to offer tenants with their own isolated network domains without deploying additional infrastructure. Network encryption is available for VXLAN by means of IPSec.

Cross network secure service networking allows for services to communicate with each other across networks by establishing an encrypted and authenticated overlay network that allows for secure communication at the service level. Cross network secure service networking can also be leveraged by legacy applications that don't offer in transit data encryption by means of a bastion host [4] that can functions as a proxy which would take in unencrypted data then pipes it to a service or client over an encrypted tunnel. A common example of such a device are terminal servers [2] that connect to supervisory control and data acquisition (SCADA) and legacy equipment that only have serial connection or telent, and offer the user a secure session by means of a SSH session or VPN.

Multiple commercial offerings exist for software applications, notably Consul and Google Compute Engine (GCE) Identity-Aware Proxy (IAP). They offer software that enables cross network secure service networking. Consul can run on almost any operating system, while IAP is Google Cloud specific where it allows a identity aware proxy that allows secure access to non public services.

Consul is software that is majorly used for discovery and configuration for a range of applications and services, it provides an up to date outlook on services in an infrastructure. Consul was mainly built to manage services in a distributed system. One of the main challenges faced by many large scale industries, is the heightened use of distributed systems and service discovery has come as a blessing to the challenges faced by the use of distributed systems. Consul is made user-friendly with the use of a flexible and powerful interface. Consul requires a data plane and it runs with a built-in proxy. All the services discovered by consul in an infrastructure is stored in a single registry so that the services can find each other by storing the information of IP addresses or other location information, this makes consul a centralized service registry. Clients can find services that are registered with consul with the help of DNS or HTTP interfaces which leads to an easy procedure to discover resources [7].

Identity-Aware Proxy (IAP), is a service from Google which provides a "central authorization layer for applications accessed by HTTPS" [8]. IAP works by only granting users or so-called "members" access when they posses the correct role. This allows for access control policies to be specific to resources and applications [8]. IAP works by intercepting web requests, authenticating the user making the request, and only granting access when the user is in fact authorized. This technique of implementing authentication and authorization remotely removes the need for a VPN [9]. Google's goal with IAP is to create a method that users may connect to untrusted networks and access services without needing a VPN [10] –

this directly mitigates the issues surrounding VPN usage, such as reduced speed. As well, users of VPN's must either have full access or no access to the network, and service granularity is extremely limited – IAP solves this by providing authentication on an application by application basis.

As the previously mentioned Consul run in software while Google IAP runs only within Google Cloud. Consul is not designed to operate with physical equipment.

In this project we will be experimenting with available products along with a custom solution that we will design that will meet our requirements. We will be using Docker Engine with Docker Compose scripts to experiment with the software along with creating a consistent setup for a more accurate comparison.

II. Problem Statement

The issue you have with overlay networks is that it must be implemented at the network level, it is not implemented at the service level. This is where cross network secure service networking comes into play. This project will compare existing frameworks, and propose a new framework that offers security as the preliminary design consideration.

III. Objective

The primary objective is that we design a system that allows for encryption of data without changing the client or server implementation.

IV. Metrics

One of the first tasks we had to complete was to determine what metrics we were looking for to determine the "best" solution. We settled on the following metrics.

- Latency
- Throughput
- Resource utilization of host and containers
- Packets dropped

V. Software

After many hours of research we settled on the following software.

A. Backend Software (out of scope of this project)

For this project we used some software in the backend to make our lives much easier to analyze the data and provide meaningful results. The following list of software is not within the scope of this project as it does not actually relate to the problem statement, but it offers the possibility to accomplish research relating to the problem statement.

- Kibana
- Elasticsearch
- Mosquitto MQTT Broker

B. Lab Software (in scope of this project)

For this project we used some software in our lab that we determined that would be best for the project.

- Docker Engine
- Docker Compose
- Shadowsocks
- stunnel
- Wireguard
- Wireshark
- NGINX

VI. Lab Setup

This section of the report will go into detail on the different virtual laboratory environments that were designed to implement the idea of cross network secure service networking.

Each environment is set up with identical HTTP and MQTT client-server services and is built using docker compose. There are four different environments, first the base environment which is a non-secure implementation, secondly an Stunnel environment, thirdly a Wireguard environment, and lastly a Shadowsocks environment.

Figure 1 shows the generic lab environment for each secure implementation. A pair of proxy containers are used to set up a secure tunnel for the data being sent between the client and the server, without changing the client and the server themselves. The secure proxy container is what is changing between each environment. A Github repository has been provided with all our research and scripts [11].

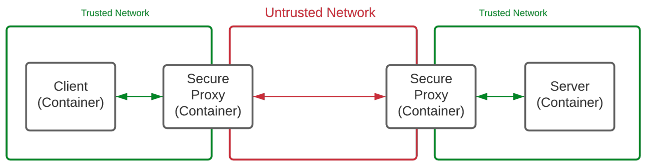

Fig. 1. Generic secure client-server environment

A. Base Lab Environment

The base environment is simply a system of containers to connect HTTP and MQTT client containers to their respective server containers. This framework excludes the secure proxy containers and is primarily used to create a baseline that each other environment can be compared to.

1) HTTP service containers: HTTP is a protocol for exchanging data between a client and server over the web [16]. For our purposes the client and server are both written in python using the requests library and flask respectively. Each program is running inside its own docker container. The client sends an HTTP POST request to the server with a timestamp and a "secret" which is a randomly generated string of characters. This

type of POST request is sent every two seconds to the server on port 5000 (at location "/process_data"). The HTTP server program listens on port 500 to receive the POST request from the client.

2) MQTT service containers: MQTT is a low-footprint, non-secure, publish-subscribe network protocol that connects a client and server via a broker, which receives messages with a defined "topic" from a client and forwards the message to all servers that are subscribed to that topic [2j]. In our case the broker accepts connections on port 1883, which is the default port for MQTT. The MQTT client publishes a message with a timestamp and a "secret" to a predefined topic to an MQTT broker. The server is subscribed to a generic wildcard topic that all clients would publish to the broker, thus when the client publishes its message, the broker passes the message on to the server. Similarly to the HTTP programs, the MQTT client and server programs are running in individual containers.

B. Stunnel Lab Environment

Stunnel is a type of proxy which is meant to add TLS encryption to existing clients and servers without changing the client-server code [13], which is the main goal of this project. For encryption, Stunnel uses OpenSSL and TLS-PSK for access control. In this environment the "secure proxy" containers in figure 1 would be implemented using stunnel to initiate a secure proxy tunnel. For HTTP the proxy containers, all requests from port 5000, the ports used for HTTP, are forwarded to port 9000. In MQTT the proxy containers listen on the MQTT port, port 1883, and forwards all requests to port 9001. Effectively this creates a secure tunnel between the stunnel proxy containers, which are used to send data over the untrusted network.

C. Wireguard Lab Environment

Wireguard is a lightweight VPN intended for the Linux kernel, but is now available for Windows and macOS as well. It is a performance-oriented VPN that functions using a concept called Cryptokey Routing [12]. This is a technique of assigning public keys to a list of allowable tunnel IP addresses, where each network interface is given a private key and a list of peers – keys are shared similarly to how SSH public keys are shared. Configuring Wireguard VPN's is a relatively simple task, once Wireguard is installed it simply needs to be configured on each device, where one device acts as the server and the other devices respectively operate as clients. Each device's configuration file contains the private key, public key of its peers, its listen port, and a list of allowed IP addresses. Wireguard can use IPv4 and IPv6 addresses [4j].

Like most VPN's Wireguard works on the network layer (layer 3) of the OSI model whereas the HTTP client-server and MQTT client-server programs of this environment operate on the application layer (layer 7) of the OSI model. As such, it must be interfaced properly to encrypt packets leaving the HTTP and MQTT services. The goal of this experimental environment is the same as in the other environments, encrypt the traffic coming from a service without changing the service itself, thus it is important not to alter the HTTP and MQTT containers. To connect the client and server through Wireguard an intermediate step is necessary connect from layer 7 down to layer 3. To do so, NGINX is used. NGINX is an open-source software that can be used for a multitude of web server-related use cases [17]. In the case of this environment, it is used to stream data from the layer 7 applications, HTTP and MQTT, to the layer 3 Wireguard VPN.

To create a secure tunnel between the services, the client and server which are docker containers, Wireguard is implemented as a proxy container to connect them over the untrusted network. Then a NGINX container is placed between the client/server and the Wireguard proxy that streams and redirects incoming data 5000 to 9000 for HTTP and port 1883 to 9001 for MQTT. The Wireguard proxy containers then connect over the untrusted network sending encrypted Wireguard messages. In figure 1 this combination of NGINX and Wireguard containers would go in place of the secure proxy container.

1) Shadowsocks Lab Environment: Shadowsocks is a secure SOCKs5 proxy. SOCKs5 is the most up to date version of the SOCKs protocol. SOCKs is a protocol used to establish a TCP or UDP session over a firewall, and since the protocol works on the session layer (layer 5 of the OSI model) it can readily handle HTTP requests [15].

This environment does not include an MQTT component because MQTT python library does not support SOCKs. As well, this implementation of a secure tunnel does not fulfill the goal of this project because the HTTP client and server programs needed to be modified to function with SOCKs5. In figure 1, the generic secure proxy containers would be the Shadowsocks containers.

VII. Results and Analysis

To gather metrics, searches must be performed in the Elasticsearch database in order to extract the data. Kibana is used to visualize the searches. Saved objects are available on the project's GitHub [11] to replicate our searches. The Kibana saved objects loads saved searches and visualizations that automatically colour results. The table I depicts the mapping of metrics to colour scheme used to automatically colour some of the data used in the graphs for this project. We determined these values after thinking what we believe to be "reasonable". The colour is just to make the graphs pretty and easy to quickly analyze, the do not impact the results in anyway.

For each lab setup, we would run the lab for a minimum of 25 minutes. Metrics are calculated on the latest 15 minutes of data based on the time of ingest into Elasticsearch. A table with our overall results is included in this report, see table II for details. If the reader wishes to replicate our project, Docker Compose scripts are available in the project's GitHub repository [11].

A. Base Results

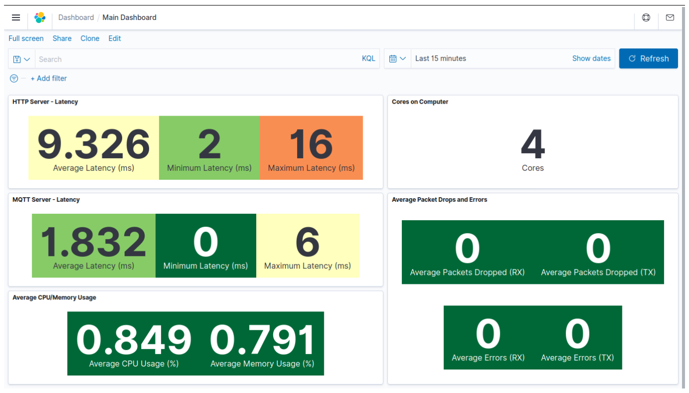
Dashboard / Main Dashboard
Full screen Share Clone Edit
Search KQL
Last 15 minutes Show dates Refresh
+ Add filter
HTTP Server - Latency
9.326 Average Latency (ms)
2 Minimum Latency (ms)
16 Maximum Latency (ms)
Cores on Computer
4 Cores
MQTT Server - Latency
1.832 Average Latency (ms)
0 Minimum Latency (ms)
6 Maximum Latency (ms)
Average Packet Drops and Errors
0 Average Packets Dropped (RX)
0 Average Packets Dropped (TX)
Average CPU/Memory Usage
0.849 Average CPU Usage (%)
0.791 Average Memory Usage (%)
0 Average Errors (RX)
0 Average Errors (TX)

Fig. 2. Kibana screenshot of the results from the base lab setup

Over the last 15 minutes of data from this setup, we found an average latency of 9.326ms and 1.832ms for HTTP and MQTT respectively. This setup averaged 0.849% in terms of CPU usage and 0.791% in terms of memory usage. It should be noted that this setup only uses 9 containers. HTTP and MQTT data in transit are is not encrypted in this lab.

B. Wireguard Results

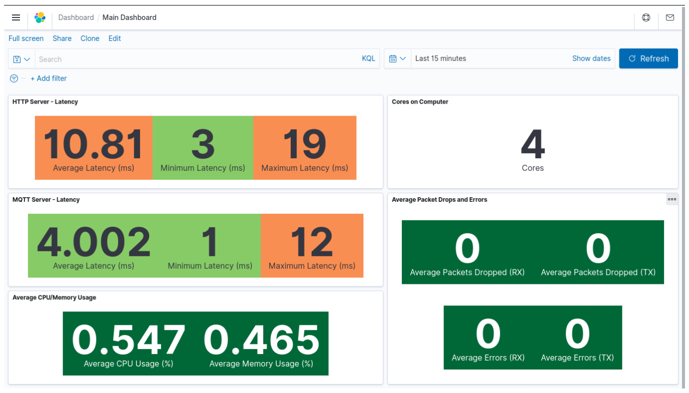
Dashboard / Main Dashboard
Full screen Share Clone Edit
Search KQL
Last 15 minutes Show dates Refresh
+ Add filter
HTTP Server - Latency
10.81 Average Latency (ms)
3 Minimum Latency (ms)
19 Maximum Latency (ms)
Cores on Computer
4 Cores
MQTT Server - Latency
4.002 Average Latency (ms)
1 Minimum Latency (ms)
12 Maximum Latency (ms)
Average Packet Drops and Errors
0 Average Packets Dropped (RX)
0 Average Packets Dropped (TX)
Average CPU/Memory Usage
0.547 Average CPU Usage (%)
0.465 Average Memory Usage (%)
0 Average Errors (RX)
0 Average Errors (TX)

Fig. 3. Kibana screenshot of the results from the Wireguard lab setup

Over the most recent 15 minutes of run time for this environment, we found an average latency of 10.810ms and 4.002ms for HTTP and MQTT respectively. For CPU and memory usage, the entire environment averaged 0.542% and 0.465% respectively. This is a 1.484ms and 2.170ms increase in latency for HTTP and MQTT respectively over the unencrypted base environment.

The interesting part of these results is the fact that in figure 3 we observed a slight decrease of 0.302% and 0.326% in CPU and memory usage respectively. While the designers that developed Wireguard claim that Wireguard "[...]to be faster, simpler, leaner, and more useful than IPsec[...]" [12]. While we don't refute these claims, our current working hypothesis is that this is due to the fact that in this environment we have 13 containers, and the CPU and memory is averaged over 13 containers. Further work will be required to determine how to gather the data required to determine the exact source of these interesting results. HTTP and MQTT data in transit are encrypted in this lab.

C. Stunnel Results

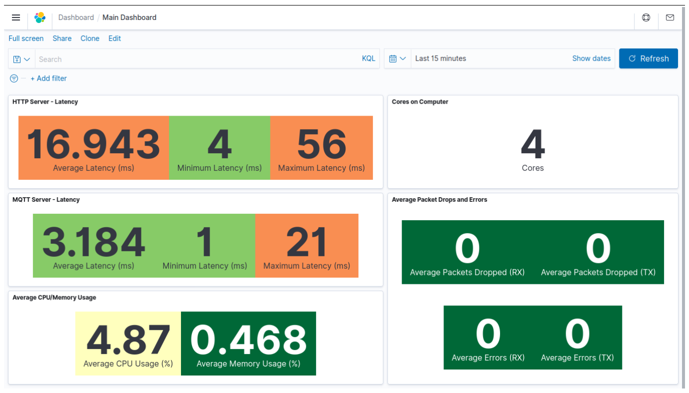
Dashboard / Main Dashboard
Full screen Share Clone Edit
Search KQL
Last 15 minutes Show dates Refresh
+ Add filter
HTTP Server - Latency
16.943 Average Latency (ms)
4 Minimum Latency (ms)
56 Maximum Latency (ms)
Cores on Computer
4 Cores
MQTT Server - Latency
3.184 Average Latency (ms)
1 Minimum Latency (ms)
21 Maximum Latency (ms)
Average Packet Drops and Errors
0 Average Packets Dropped (RX)
0 Average Packets Dropped (TX)
Average CPU/Memory Usage
4.87 Average CPU Usage (%)
0.468 Average Memory Usage (%)
0 Average Errors (RX)
0 Average Errors (TX)

Fig. 4. Kibana screenshot of the results from the Stunnel lab setup

Over the most recent 15 minutes of run time for this environment, we found an average latency of 16.943ms and 3.184ms for HTTP and MQTT respectively. For CPU and memory usage, the entire environment averaged 4.87% and 0.468% respectively. This is a 7.617ms and 1.352ms increase in latency for HTTP and MQTT respectively over the unencrypted base environment.

To us, the most interesting part of these results is the high CPU usage. This lab setup has consumed significantly more CPU usage when comparing to the other setups, over 4% above base. It is the highest recorded ever the entire project. Our current working hypothesis is that the stunnel process is really CPU intensive, more work will be required to chase down the cause of this. HTTP and MQTT data in transit are is encrypted in this lab.

D. Shadowsocks Results

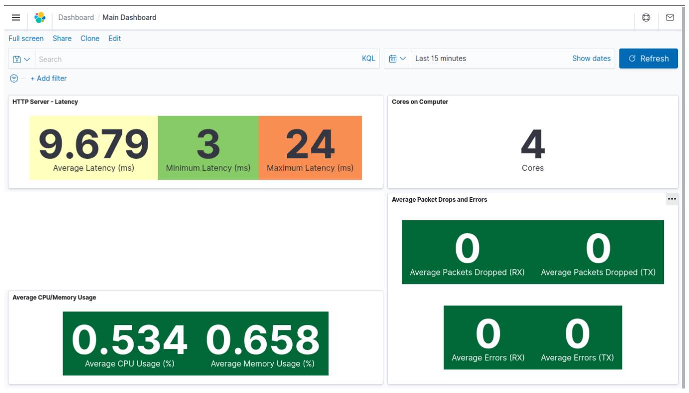
Dashboard / Main Dashboard
Full screen Share Clone Edit
Search KQL
Last 15 minutes Show dates Refresh
+ Add filter
HTTP Server - Latency
9.679 Average Latency (ms)
3 Minimum Latency (ms)
24 Maximum Latency (ms)
Cores on Computer
4 Cores
Average Packet Drops and Errors
0 Average Packets Dropped (RX)
0 Average Packets Dropped (TX)
Average CPU/Memory Usage
0.534 Average CPU Usage (%)
0.658 Average Memory Usage (%)
0 Average Errors (RX)
0 Average Errors (TX)

Fig. 5. Kibana screenshot of the results from the Shadowsocks lab setup

Shadowsocks [14], which is a SOCKS5 proxy [15], did not fulfill our project objective where we had do find or design a system that would allow for encryption of data without changing the client or server implementation. In the case of Shadowsocks, we had to modify the HTTP client implementation to use the SOCKS5 proxy.

We had an issue with the MQTT intance for this lab setup. The MQTT Python library does not support SOCKS5, therefore we were not able to gather metrics for MQTT via a Shadowsocks implementation.

Even if with these limitation, we opted to see what metrics would be generated by a modified HTTP setup. Over the most recent 15 minutes of run time for this environment, we found an average latency of 9.679ms for HTTP. For CPU and memory usage, the entire environment averaged 0.534% and 0.658% respectively. This is a 0.353ms latency increase for HTTP over the unencrypted base environment, which is certainly a respectable time difference considering that there are only 8 containers and the HTTP data is encrypted.

VIII. Conclusion and Future Work

This project compared three possible proxy architectures that allowed for a simulated legacy application to be upgraded to a secure connection without changing the legacy applications and without using a traditional site to site VPN system. We found that only two of the three compared actually worked for the chosen simulated applications.

For HTTP and MQTT, Wireguard and Stunnel fulfilled the objectives of this project with, what we consider to be, minimal and acceptable increased delay and resource utilization. Shadowsocks did not meet the requirements of the "no modification" requirement of this project as the HTTP instance had to be modified to support SOCKS5.

We believe that there are probably better and more efficient ways to accomplish this project and we would like to see further work dedicated to designing a solution for this.

IX. Literature Review

X. Appendix

Kibana Colour Boxes (CPU and Memory)		
Colour	Start	End
Green	0	1
|	>1	5
Red	>10	100

Kibana Colour Boxes (Latency, in ms)	Start	End
Green	0	
|	1	5
|	>5	10
|	>10	100
Red	>100	20000

Average Packet Drops and Errors (occurrences)	Start	End
Green	0	5
Red	>5	99999999

TABLE I
Kibana Colour Boxes

Scenario	HTTP Server Latency			MQTT Server Latency						
	Average (ms)	Minimum (ms)	Maximum (ms)	Average (ms)	Minimum (ms)	Maximum (ms)	Average CPU (%)	Average Memory (%)	# Containers	# Packet Drops
Base	9.326	2	16	1.832	0	6	0.849	0.791	9	0
Shadowsocks	9.679	3	24	N/A			0.534	0.658	11	0
Wireguard	10.81	3	19	4.002	1	12	0.547	0.465	13	0
Stunnel	16.943	4	56	3.184	1	21	4.87	0.486	13	0

TABLE II
Results table.

References

[1] Norton, What is a VPN?, Norton, 2020. Accessed on: September 20, 2020. [Online]. Available: https://us.norton.com/internetsecurity-privacy-what-is-a-vpn.html

[2] Raritan, Serial Console Server - Your Next Generation Solution, Raritan, 2020. Accessed on: September 20, 2020. [Online]. Available: https://www.raritan.com/products/kvm-serial/serial-console-servers/serial-over-ip-console-server

[3] J. Tyson, C. Pollette, S. Crawford, How a VPN (Virtual Private Network) Works, How Stuff Works, 2019. Accessed on: September 20, 2020. [Online]. Available: https://computer.howstuffworks.com/vpn.htm

[4] J. Kozlowicz, What's a Jumpbox or Bastion Host, Anyway?, Green House Data, 2019. Accessed on: September 20, 2020. [Online]. Available: https://www.greenhousedata.com/blog/whats-a-jumpbox-or-bastion-host-anyway

[5] SDxCentral, What are Network Overlays, SDxCentral, 2015. Accessed on: September 20, 2020. [Online]. Available: https://www.sdxcentral.com/networking/virtualization/definitions/get-on-top-of-network-overlays/

[6] IETF, Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, IETF, 2014. Accessed on: September 20, 2020. [Online]. Available: https://tools.ietf.org/html/rfc7348

[7] Consul, Consul by HashiCorp, HashiCorp, 2020. Accessed on: September 20, 2020. [Online] https://www.consul.io/

[8] Google Cloud, Identity-Aware Proxy (IAP), Google, 2020. Accessed on: September 20, 2020. [Online] Available: https://cloud.google.com/iap

[9] Google Cloud, Identity-Aware Proxy: Documentatio , Google, 2020. Accessed on: October 4, 2020. [Online] Available: https://codelabs.developers.google.com/codelabs/user-auth-with-iap/index.html?index=..%2F..index#0

[10] T. Treat, API Authentication with GCP Identity-Aware Proxy, Medium.com, January 25th, 2019. Accessed: October 4, 2020. [Online]. Available: https://blog.realkinetic.com/api-authentication-with-gcp-identity-aware-proxy-3a4147b30770

[11] G. Ankenmann, J. Bailey, F. Haque ITEC5102 Project Repository, Github.com. [Online]. Available: https://github.com/devagent42/ITEC5102F-Project

[12] J. A. Donenfeld, Wireguard, Wireguard, 2020. Accessed: October 4, 2020. [Online]. Available: https://www.wireguard.com/

[13] Stunnel, Stunnel, Stunnel, 2020. Accessed: October 4, 2020. [Online]. Available: https://www.stunnel.org/

[14] Shadowsocks, Shadowsocks, Shadowsocks, 2020. Accessed: October 4, 2020. [Online]. Available: https://shadowsocks.org/en/index.html

[15] A. Patil, SOCKS Proxy Primer: What Is SOCKs5 and Why Should You Use It?, Shadowsocks, September 27, 2019. Accessed: October 4, 2020. [Online]. Available: https://securityintelligence.com/posts/socks-proxy-primer-what-is-socks5-and-why-should-you-use-it/

[16] Mozilla, An Overview of HTTP, date unknown, Accessed on December 9, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[17] NGINX, What is NGINX?, date unknown, Accessed on: December 4, 2020. [Online]. Available: https://www.nginx.com/resources/glossary/nginx/

[18] MQTT.org, MQTT: The Standard for IoT Messaging, date unknown, Accessed on December 9, 2020. [Online]. Available: https://mqtt.org/