Assignment #3

CPEN 442

October 13th 2019

Adriano Sela Aviles (31063150)
Christian Neufeld (36719152)
Israel Trujillo Quintero (16055155)
Felipe Ballesteros (10703156)

Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada

I. PROGRAM INSTALLATION

Download Go:

The only prerequisite to installing and running the program is the Go programming language for the host operating system. Go can be downloaded through most operating system's package manager (e.g. apt for Linux, brew for MacOS) or from the official Golang downloads page and following the instructions there (https://golang.org/dl/).

Get the source code:

Once you have a successful Go installation, you must acquire the source code for the program. If this was provided to you, fantastic -- otherwise you must get it from GitHub, either through cloning the repository using the 'git clone command':

```
$ git clone https://github.com/adrianosela/vpn
```

or the 'go get' command:

```
$ go get -u github.com/adrianosela/vpn
```

Build the program for the host OS:

Build the program by changing directory to /vpn and running the 'go build' command:

```
$ go build
```

Running the program:

Run the program by executing the built binary:

```
$ ./vpn
```

Additionally, a `-uiport` flag can be specified to override the default TCP port where the application's UI is served. If you are running two instances of the application on the same machine (i.e. to communicate through localhost), you will need to specify a non-default -uiport for at least one of the two instances:

```
$ ./vpn -uiport 8081
```

For more information you can use the '--help' flag on the binary:

```
$ ./vpn --help
```

II. DERIVING ENCRYPTION AND INTEGRITY PROTECTION KEY FROM SHARED SECRET

An insecure TCP connection is established between the application running in 'client' mode and another instance in 'server' mode. The users enter the shared secret value on both ends. The secret value is hashed as to ensure a length of 32 bytes. Note that hashing does not provide any security, we just happen to need a 32-byte long AES key and hashing is a nice easy way to ensure a fixed length of 32-bytes.

With this 32-byte long string, we then construct an AES-256 block cipher. The newly created AES-256 block cipher will be identical on both ends, and this is our encryption and integrity-protection key. The block cipher mode is Galois Counter Mode which utilizes a counter-type nonce.

III. ESTABLISHING A SHARED SECRET KEY WITH A KEY AGREEMENT PROTOCOL (ECDH)

Once both communicating parties have computed their AES block cipher from the shared secret value, the next step is to establish shared session keys for the rest of the communication. The server will generate a key pair based on an Elliptic Curve Diffie Hellman (ECDH) using Curve25519, a curve that is widely used for its performance benefits. The public key is sent to the client, and the server waits idle to receive the client's public key. The client receives the server's public key and generates its own ECDH key pair. The client then sends its public key to the server. At this point both parties have their own private key, and their peer's public key, and thus they can both construct their shared secret for a session key.

When performing the mentioned Diffie-Hellman, we encrypt the parameters as a form of authentication, even though they are public. This is to prevent a man-in-the-middle attack which would enable an intruder to establish shared keys with both communicating parties, serving as an all-seeing proxy for communication between the parties.

With this newly established shared secret (from the ECDH), both parties create a new AES-256 block cipher in Galois Counter Mode. This block cipher is used to encrypt all further communication between the server and the client.

IV. SENDING AND RECEIVING DATA SECURELY

Once a shared 32-byte key has been established through a Diffie-Hellman, the shared string is used to build an AES-256 block cipher. From this moment onward, all messages send over our DIY secure channel are encrypted with this symmetric block cipher. Additionally, messages are prepended with a nonce before encryption; this ensures that we do not accept replayed messages from an actor in the middle. The nonce we are using is in the form of a counter. That is, say we just got message 92733, we expect the next message's nonce to be 92734, and we reject the message otherwise. This is handled by having AES cipher block in GCM (Galois Counter Mode). This mode is designed to provide both data authenticity (integrity) and confidentiality.

Note that we do not expect messages to be 'lost' to the network since TCP handles retransmission and in-order delivery.

V. RECOMMENDATIONS FOR IMPLEMENTING A REAL SERVICE

If we were to implement a real-world VPN product for sale we would use the same design we used for this project. We would use the elliptic curve Diffie-Hellman algorithm which reduces the computational costs of the standard Diffie-Hellman algorithm. We used 32 bytes (256 bits) for both the encryption key and the integrity key of our project which we believe is secure enough to be used in most real world scenarios. Elliptic curve Diffie-Hellman does not use modulus operations, but if we were to use standard Diffie-Hellman we would have used 4096 bits for the modulus.

VI. DESCRIPTION OF THE PROGRAM

The program was written in Golang because the standard library contains reliable and production grade implementations of networking and encryption primitives. Furthermore, the team was particularly interested in learning about and experimenting with Go's concurrency model with 'goroutines' and 'channels'.

The program is entirely self-contained and can run on any major operating system. The size of the compiled binary for MacOS is 8.5 MB and is just over 900 lines of code, excluding the 160 lines of the UI's HTML.

The program consists of a few modular components:

App Controller: Handles HTTP connections from the application UI (a web app served locally). Common HTTP handler functions for the application operating in either client or server mode:

- serveApp(http.ResponseWriter, http.Request): serves data based on application state (multiplexes routes based on a state machine, rather than URL path)
- serveSharedSecret(http.ResponseWriter, http.Request): computes the shared secret value for the Diffie Hellman using a private key and the peer's public key. It then initiates threads to handle established TCP connection, decode and decrypt incoming messages through established TCP connection, and encrypt and encode messages through established TCP connection

- serveStateStep(http.ResponseWriter, http.Request): serves HTML for the current step of the authentication and key exchange. The served HTML is code-generated to show all current state messages for the application.
- serveChat(http.ResponseWriter, http.Request): serves HTML for the chat interface, which includes some Javascript code to open web socket connections to send the chat messages through
- serveWS(http.ResponseWriter, http.Request): serves web socket connections for sending and receiving messages between client and server and initiates thread to handle messages sent and received through the connection

Server-mode-only HTTP handlers include:

- serverConfigSetHandler(http.ResponseWriter, http.Request): serves HTML for the server configuration and deals with entered inputs (port, passphrase)
- serverListenTCP(http.ResponseWriter, http.Request): listens for incoming TCP connections
- serverGenerateDHHandler(http.ResponseWriter, http.Request): generates an Elliptic-Curve Diffie-Hellman key pair
- serverSendKeyHandler(http.ResponseWriter, http.Request): encrypts symmetric key and sends it to the client

Client-mode-only HTTP handlers include:

- clientConfigSetHandler(http.ResponseWriter, http.Request): serves HTML for the client configuration and deals with entered inputs (host name, port, passphrase)
- clientDialTCP(http.ResponseWriter, http.Request): establishes a TCP connection with the server and waits for Server's public key
- clientGenerateDHHandler(http.ResponseWriter, http.Request): generates an Elliptic-Curve Diffie-Hellman
- clientSendKeyHandler(http.ResponseWriter, http.Request): encrypts symmetric key and sends it to the server

TCP Conn Controller: Handles reading/writing from/to a TCP connection

- tcpConnHandler(net.Conn, chan []byte, chan []byte): reads incoming data from the TCP connection onto an RX channel, writes outgoing data from a TX channel to the TCP connection

Websocket Conn Controllers: Handles reading/writing from/to the Websocket connection for the secure chat

- wsConnHandler(websocket.Conn, chan []byte, chan []byte): reads incoming data from the websocket connection onto an RX channel, writes outgoing data from a TX channel to the websocket connection

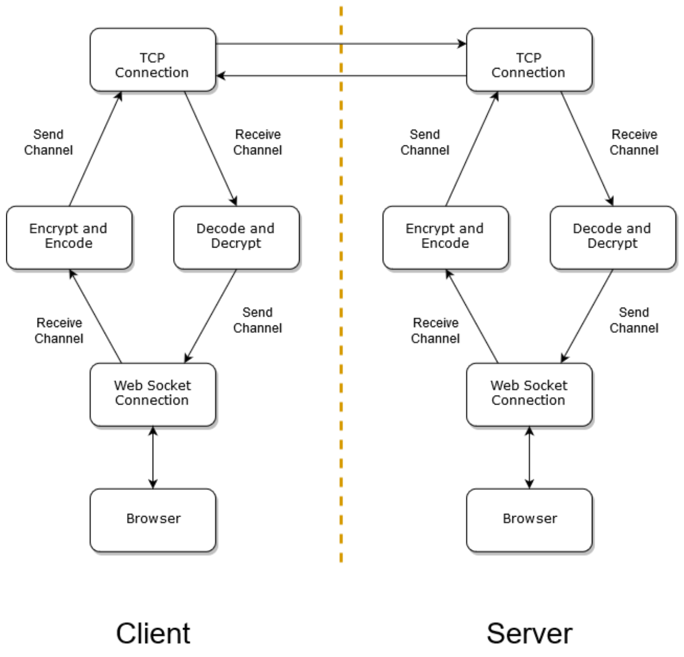
AES Encryption/Decryption Module: Contains utilities to perform AES block cipher encryption/decryption

- aesEncrypt([]byte, string): []byte, error - encrypts given ciphertext with an AES block cipher based on a string of arbitrary length (passphrase/secret shared value). Returns the generated ciphertext
- aesDecrypt([]byte, string): []byte, error - decrypts given ciphertext with an AES block cipher based on a string of arbitrary length (passphrase/secret shared value). Returns the decrypted plaintext

Protocol Controller: message processing functions

- encryptAndEncode(chan []byte, chan []byte, string): encrypts and encodes data from the input channel and writes it to the output channel
- decodeAndDecrypt(chan []byte, chan []byte, string): decodes and decrypts data from the input channel and writes it to the output channel

FIGURE 1: MAIN FUNCTIONAL TASKS AS A SYSTEMS DIAGRAM


TCP Connection
TCP Connection
Send Channel
Receive Channel
Send Channel
Receive Channel
Encrypt and Encode
Decode and Decrypt
Encrypt and Encode
Decode and Decrypt
Receive Channel
Send Channel
Receive Channel
Send Channel
Web Socket Connection
Web Socket Connection
Browser
Browser
Client
Server