Assignment 3 - VPN

October 15, 2014

Jorden Hetherington, Gorden Larson, Jeremy Lord, Rohit Singla

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

{jorden.hetherington, gordlarson, jeremy.lord200, rsingla92}@gmail.com

I. How to Install

This VPN can be found at the following Git repository:
https://github.com/rsingla92/a3-vpn

This VPN will work on any platform, but is advised to run on Windows or Linux as a noticeable lag has been viewed on Mac platforms. To note, a private network, where ports are open, must be used when running the application or else firewalls may be an issue (ex: UBC Secure does not work for communicating with the VPN, but UBC Private does). In order to "install" this VPN, the following must be done:

- Install Python3 interpreter
  - Linux: "$sudo apt-get install python3"
- Install Python3-tk
  - Linux: "$sudo apt-get install python3-tk"
- Run VPNApp.py
  - Linux: "$python3 VPNApp.py"

Both the client and the server computer must run the application on their respective machines.

To set up the VPN, the computer that is going to be the server must click the "Mode button". This switches the computer into server mode – this change is reflected in the updated button and the log. The server types a secret shared value that can be anything, the port they want to use, and clicks the "Connect" button. Then the server waits for the client to connect and authenticates it. Once fully authenticated, with keys exchanged, data can be typed into the text-box labelled "Data to be Sent", and the "Send button" will promptly send the data to the client.

Similarly, to set up a client, the client must enter the IP address of the server computer in the "IP Addr" text-box, the port they want to use, and the same-shared value as the server. The shared secret value must match the server in order to set up VPN. Then click the "Connect button". Once connected, data is sent in the same way as with the server, and is received in the "Data as Received" text-box.

The program's live log is displayed in the Log window, with both log messages about the state and data being sent.

II. Transfer of Data

Data is sent and received over a configurable TCP socket (the default is 50002). AES-CBC is used to encrypt messages after key establishment. CBC-MAC is used to ensure message integrity. All components of this data transmission were implemented within this project. Full AES-CBC was implemented to work with a 128-bit key. Further, it transmits data of any length. To do this, it pads the messages with 0x80, to fill any incomplete blocks, before encryption, and then strips any trailing 0x80 values after decryption. CBC-MAC was implemented to guarantee message integrity when performing regular data transmission. The respective encryption and integrity-protection key derivation is described further in section IV.

III. Mutual Authentication and Key Establishment

Authenticated Ephemeral Diffie-Hellman is used for key establishment. Using the shared secret value, converted to a 16-byte key, as described in section IV, an AES-encrypted Diffie-Hellman authentication protocol was implemented. This is done in five steps.

1. *Client initates connection*: client sends unencrypted client-ID and client-nonce to the server.
2. *Server responds*: after generating a response of unencrypted server-nonce, and an encrypted representation of the server-ID, the client-nonce, and the server-computed Diffie-Hellman modular value. Encryption is done using the shared secret key.
3. *Client responds*: client sends back an encrypted message consisting of the client-ID, server-nonce, and client-computed Diffie-Hellman modular value. modular value. This encryption is done with the shared secret key.
4. *Client authenticates*: the data sent by the server from step 2 is authenticated by unencrypting and ensuring server-ID and client-nonce are as expected.
5. *Server authenticates:* the data sent by the server from step 3 is authenticated by unencrypting and ensuring client-ID and server-nonce are as expected.

Performing these five steps, both the client and server can guarantee the origin of the data sent between them, as well as the validity (from the nonce) and can therefore perform authentication. Recall that the Diffie-Hellman key exchange works with modular arithmetic, so that both the client and server generate a value of $g^a \bmod p$ and $g^b \bmod p$, respectively, where g is a generator value, p is some large prime, and a/b are secret exponents that the client/server generate temporarily and discard after the exchange is complete. Then, passing these values as per the five steps described above, both parties can compute $g^{ab} \bmod p$, which is the shared session-key. By using this exchange, while encrypted with the shared secret value, no Man-in-Middle attacks are feasible, and Perfect Forward Secrecy (PFS) is achieved (since session keys are generated independently of previous keys/data). To note, this module implemented values that might be indicatively as large as in real systems. For example, the prime number used as the modulo-value is a 1536-bit value, with a generator value of two. Then, a random 128-bit exponent is used to generate the client/server transport data, which is passed between each other to generate the session-key.

IV. Encryption Derivation

To derive the symmetric encryption key, the following is done. The shared secret value is hashed using MD5 to convert the value into a 16-byte key, denoted the long-term-key, that can be used to encrypt the Diffie-Hellman key exchange protocol. This provides perfect forward secrecy. Once the users recover a shared secret session-key from the Diffie-Hellman exchange, this new shared secret is used as the session-key for AES encryption for regular data transmission.

To derive the integrity-protection key, the long-term-key is hashed once again using MD5 to produce a second unique 16-byte key. This key is used for MAC authentication techniques. Although less secure than having an independent integrity-protection shared-secret value (which would generate the integrity-protection key), this VPN determined it to be more user friendly to have the user only input one shared secret value, which generates both the encryption key and from that, the integrity-protection key. It would be more secure to generate both without a reliance on the other, but was intentionally not implemented for UX.

V. Real World Parameters

In designing a real-world VPN for sale, we would use similar algorithms implemented in this project. AES-CBC with a 256-bit key would be used for encryption. CBC mode would be chosen because it is more challenging to decrypt given its dependency on priori blocks of data. Given the size of keys in comparison to modern day processor power, there is no reason not to choose a higher bit key. This expands the search space significantly without a major power drain. The added benefit is that psychological acceptability increases with a bigger key. For authentication and integrity, the key size can also be 256-bit, making use of the same encryption algorithm.

For key exchange, it would depend on if the client and server have had prior communication. If there has been prior communication, then RSA works because a public key can be sent first. However, we assume there has not been. In this case, the Diffie-Hellman key exchange works well as there is no need for unsecure communication. The Diffie-Hellman should use a prime number for modulus, one that has multiple hundreds of digits. For example, the implemented VPN has 1536-bit prime number.

VI. Implementation Overview

The VPN was implemented in Python3. It uses Python's built in socket module for communicating over the network, and Tkinter for displaying the GUI. There are **833** lines of source code, with modules for AES, connection, MAC, and Diffie-Hellman. The code depends on Python3's standard libraries and takes up approximately **2.2MB**.

The GUI is the framework for the VPN as it ties several modules together. It has buttons and text fields for switching between client and server mode, start connection, sending messages, and stopping connection. There is also a log window for displaying time-stamped log messages as the application executes. Each of these is tied into the other modules through a series of function callbacks. The GUI primarily takes user input (in the form of button clicks and entering text) and provides output as log messages and decrypted messages received.

The connector module handles the client-server connection. It contains a Connector class that acts as the public interface to the module. This class gives the user the ability to connect as either the server or a client, and provides method to send and receive messages. The server binds a socket to an address specified by the server's IP and a port, and then listens for a single connection (the client). The client attempts to connect to the server's IP and port. Once each side has formed a connection, they each spin up two threads: one for receiving messages and one for sending messages. The receiver thread receives new messages and places them in a queue for the user to consume as needed. The sender thread looks at the sending queue, which is populated by the user, and sends any messages in the queue.

Notably, the AES implementation was done from scratch. It uses CBC mode to encrypt multiple blocks. The key used is 128-bits and the data blocks are 128-bits, and data is padded with the byte 0x80. This implementation of AES is done using lookup tables for many of the operations, including bytesub, mixcolumns and rcon (used in the key schedule). This section will not delve into the low level details of AES because it is an encryption standard. However, at a high level, both the encryption and decryption generate an extended key based on the key schedule algorithm for AES, and this extended key is used to encrypt and decrypt each block of plaintext or ciphertext. CBC mode is used to prevent patterns in the data and cut and paste attacks. As a result of using CBC mode, an initialization vector (IV) is used to encrypt multiple blocks. This IV is either generated randomly or provided by the user, and it is sent as the first block in the encrypted message. The code for AES is located in the aes module; the module contains two simple methods that can be used outside the module: aes_encrypt and aes_decrypt. Each of these accept a list of bytes or a string (plaintext or ciphertext, depending on which method) and the symmetric key. The method aes_encrypt also optionally accepts an initialization vector specified by the user.

The MAC module is a simple wrapper module to the AES module. In such, its fundamental requirements are to take in a key, and generate a 16-byte MAC (note that 16 bytes is the length of one AES block), as well as to verify if a plaintext's integrity is valid. To do the first task, the MAC module simply calls the AES module's encrypt function, and retains the MAC computed (i.e. the last block of the computed ciphertext) as well as the initialization vector (IV), so that the MAC can be properly compared against by another party. To do the second task of verifying a plaintext's integrity, the MAC module simply calls the AES module's encrypt function, specifying the IV to use, and then compares the computed MAC to the expected MAC. If they match up, then the data's integrity must have been upheld, since the probability of accidentally computing a matching MAC value is negligible.

The key exchange module implements an authenticated ephemeral Diffie-Hellman exchange. The goal is to have two parties, sharing some secret value, able to securely generate session keys. This means that confidentiality, and integrity are of utmost importance. Furthermore, authentication is fundamental as well. The key establishment procedure itself is described in section III. However, this section will describe its code implementation. Essentially, the GUI's connect button triggers the necessary connection protocol to be executed between the client and the host. Utilizing many helper functions to perform tasks such as generating nonces, generating random exponents, packaging/de-packaging tuples of transmitted data, and more, the key exchange module is able to reliably and securely generate session-keys to be used in the rest of the data transmission encryption.