Assignment #3: VPN

CPEN 442

21 October 2016

Derek Chan (33184128), Connie Ma (56047129), Jake Larson (32333122), Pascal Turmel (19449131)
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada

I. Description, Installation and Execution

Background:

We wrote our VPN to be run as a self-contained client / local server TCP session environment that is interfaced with the command-line through a main VPN configuration program. When executed on two separate processes, the VPN client and server programs may send and receive encrypted messages to each other by operating under a uniform security protocol that employs a selection of standard cryptographic functions.

The source code of our VPN was developed in Python (runtime v2.7) and much of the computational code is powered by library function calculations from the PyCrypto package (v2.6.1)[1]. More on that will be covered in section VI.

bash Installation (Linux or Mac OS X):

To install the program, simply clone the git repository located at https://github.com/dchanman/vpn and install the necessary program dependencies using Anaconda or pip. For simplicity it is not necessary to install the full Anaconda package as mentioned in the README, and using its lite version *miniconda* will suffice for running the program. To install[2] miniconda, download the bash package from this link (http://conda.pydata.org/miniconda.html), and execute:

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```

Next, install the required Python packages in the `vpn` repository, and switch to the new environment:

```
/vpn$ conda env create -f anaconda_env.yml
/vpn$ source activate cpen442
```

The VPN may now be run on the command-line.

Execution

1) On one terminal process, execute:

`/vpn$ ./main.py` (or `/vpn$ python main.py`)

2) To run the program in *server mode* (Fig.1)*:*

a) Select option “0” and input the “server” option when prompted by the interface program.

b) Set any other VPN configurations as desired (port, shared key, etc).

c) Select option “5” after verifying the settings. An instance of the VPN's server program will execute.

3) To run the program in *client mode:*

a) On a separate terminal process, execute the main program again as before.

b) Verify that it is in client mode with the same desired VPN configuration that was used in the previous step.

c) Select “5” to run the client program.

After establishing the connection, it is now possible to send and receive messages over TCP connection between the client and server, using our VPN program's security protocol (Fig. 2).

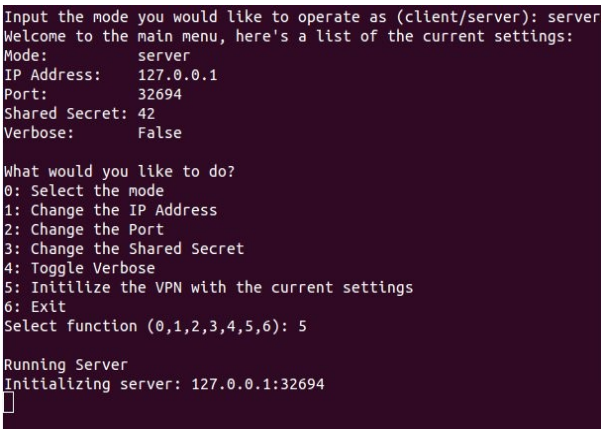

Fig. 1. Executing first process in server mode

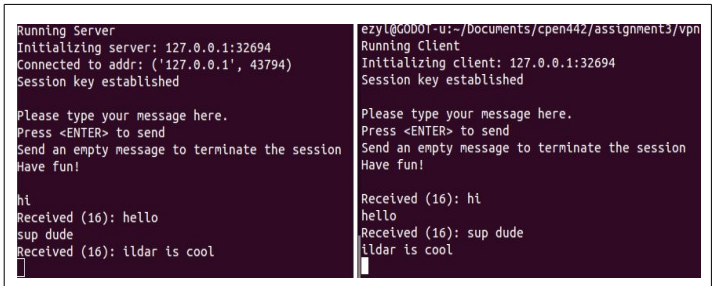

Fig. 2. Communication sending messages over the VPN connection

Additionally note that for debugging purposes, both the client and server programs may be independently run by executing `$ ./Client.py` and `$ ./Server.py`, respectively.

Enabling verbose mode whilst running the program (e.g., executing `$ ./Client.py –v`) is also very useful for analyzing the computations being made on either side. The shared key and port number may be changed in this way, as well. For more details, refer to the README in the repository.

To exit the program, simply enter an **empty message** into both terminal processes as confirmation to the program that the session transaction has been completed.

II. How the data is sent, received, and protected

Data is sent and received between a client and server through a local TCP port connection. Initialization of the server and client programs themselves follow typical socket programming protocol, as shown below, using an abstracted `Host` class for all connection functions. The following code from `Host.py` illustrates our implementation of basic socket programming using Python's built-in socket library.

```
def initServer(self):
    self.socket.bind((self.ipAddr, self.portNum))
    self.socket.listen(1)
    self.connection, addr = self.socket.accept()
    self.socket.close()

def initClient(self):
    self.socket.connect((self.ipAddr, self.portNum))
    self.connection = self.socket
```

Once the connection has been established, the VPN program will generate a unique session key for encryption of all data during the remainder of the transaction, following a handshake protocol. During this interaction, messages are not secured and are passed over the connection as plain strings. Both client and server use the `send()` and `recv()` function in `Host.py` (which are just abstractions of the `send` and `recv` functions from the Python socket library), but the key itself is individually computed by both the client and server program and will never be disclosed over the insecure connection. Detailed breakdown of the handshake and protocol computation will be covered in section III.

If the handshake fails, the program will quit. If the handshake is successful, the secure session has been established and both programs may now send and receive secured messages to one another through the command-line interface.

At this point, both client and server programs will call `startChat()`, which spins off a separate thread (`RecvEncryptedThread`) for sending secure messages. From this point, `send` and `recv` are layered in the thread by **`sendEncrypted()`** and **`recvEncrypted()`**. These two functions require the session key in order to encrypt/decrypt the messages that are sent/received over the connection, and also employ some other cryptographic functions to check for data integrity. Specific mechanics on those functions such as how the hashed MAC (HMAC) and initialization vector (IV) is computed for messages using the shared secret value will also be discussed in the following section.

```
def startChat(self):
    recvThread = self.RecvEncryptedThread(self)
    recvThread.start()
    ...
    while not self.connectionClosed:
          msg = raw_input()
          if self.connectionClosed:
              break
          self.sendEncrypted(self.sessionKey, msg)
          if not msg:
              self.connectionClosed = True
              break
    recvThread.join()
    self.close()
    print("Receiver thread closed. Session ended.")
```

Lastly, as demonstrated above in the code snippet for `startChat()`, sending an empty message will kick the program into a state where it waits to terminate. Once one program's corresponding `RecvEncryptedThread` has also detected that it has received an empty message (its `recvEncrypted()` will return a null value if it does not receive the expected data length), the TCP connection will be closed and the VPN program session will terminate.

III. Protocols used, why they were chosen, and the computation performed by the protocols used

The handshake protocol described in section II is a derivative of a standard Diffie-Hellman (DH) private key exchange algorithm, chosen because it is the cryptographic standard that many modern programs use for key exchange as it ensures *perfect forward secrecy* (i.e., an attacker cannot use a new DH session key to decrypt any old messages), and also because it is computationally expensive for attackers to solve the discrete logarithm problem of large primes, which is what the DH algorithm is based on. The primes chosen for this computation are all 2048-bits, which will ensure that it is extremely difficult to compute the primes that either program uses privately. Additionally, the protocol also uses the **shared secret** between the client and the server.

In terms of algorithms used— For all-purpose hashing, we chose the SHA256 hash function (`Cryptography.hash` in the code) because it is very difficult to break in a short period of time. For symmetric key encryption of messages using an IV and session key, we chose AES in CBC mode (`Cryptography.symmetricKeyEncrypt`) because it is relatively simple in implementation yet more than sufficient for the scope of this assignment (i.e., we do not have extremely time, noise or error-sensitive plaintexts), while still being complex enough that will not be cracked in Ildar's time limit of 1 hour. Below is a description of steps in the handshake (Fig. 3).

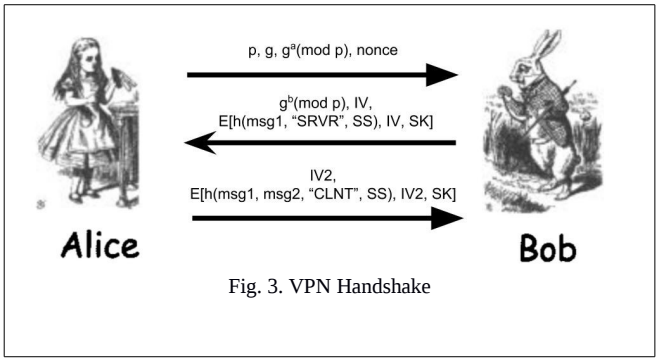

Fig. 3. VPN Handshake

STEP ONE

a) The client (denoted "A") computes the first step of the Diffie-Hellman exchange: **g^a (mod p)**, where **g** and **p** are **two** different, large random prime numbers, and **a** is a random number (unique to A for this session).

b) A then initiates the protocol by sending **p**, **g**, **g^a (mod p)**, and a generated **nonce** value to the server (denoted "B").

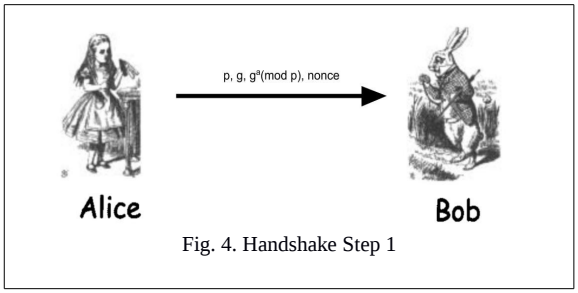

Fig. 4. Handshake Step 1

STEP TWO

a) B receives **msg1 = p**, **g**, **g^a (mod p), nonce**, and computes **g^b (mod p)** from the same **p** and **g** that the client sent, with **b** as a random number (unique to B for this session). B can also compute the DH session key, the value **g^{ab} (mod p)** (mathematically equivalent to [**g^a (mod p)]^b**).

b) B then:

i. Generates a random initialization vector (**IV**),

ii. Concatenates the previous message string that it received from A with the string "SRVR" with the **shared secret** value together, and hashes this value using SHA256,

iii. And encrypts the **IV** and hashed value in (ii) with the computed DH session key from a) using AES in CBC mode.

c) Finally, B sends a message back to A with **g^b (mod p)**, the encrypted value in b)(iii), and the IV.

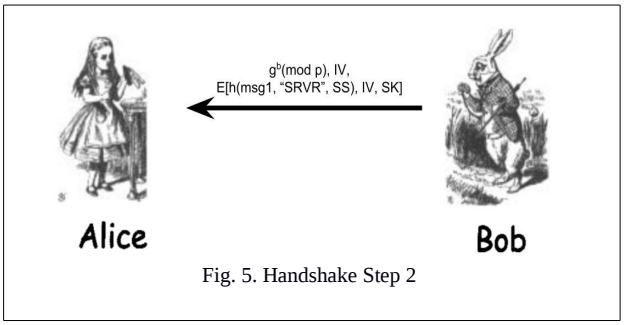

Fig. 5. Handshake Step 2

STEP THREE

a) After receiving **msg2 = g^b (mod p), E[h(msg1, SRVR, sharedSecret), IV, g^{ab} (mod p)], IV** from B, A now has the necessary information to privately compute the unique DH session key, **g^{ab} (mod p)**, by performing a similar calculation to what B did in step 2a): [**g^b (mod p)]^a**. Note that the shared key and both **a, b** are never transmitted over the A, and an attacker C listening over the connection will have a very difficult time solving this problem without knowing **a** or **b**.

b) A then:

i. Separately computes the encrypted hash **E[h(msg1, SRVR, sharedSecret), IV, g^{ab} (mod p)]** by using the IV and computed DH session key to re-encrypt the hash. This results in the same computation made by B in step 2b)(iii).

ii. A now compares the computed value from the previous part to the same encrypted hash that was encrypted by B with the DH session key. This serves to verify the authenticity of the over-the-air reply from B, as only B could have encrypted the hash. Even though the **IV** was sent in plain text right after this hash, it is useless to anybody except A, as only A can use the IV correctly to encrypt the hash and check for integrity.

c) We are almost done! A has now successfully confirmed the session key. To affirm this with server B, A then:

i. Generates another random initialization vector (**IV2**),

ii. Concatenates the previous message string that it received from B with the string "CLNT" with the **shared secret** value together, and then hashes this value using SHA256,

iii. And encrypts the **IV** and hashed value in (ii) with the computed DH session key from a) using AES in CBC mode.

d) Lastly, A sends a message back to B with the encrypted value in c)(iii), and IV2.

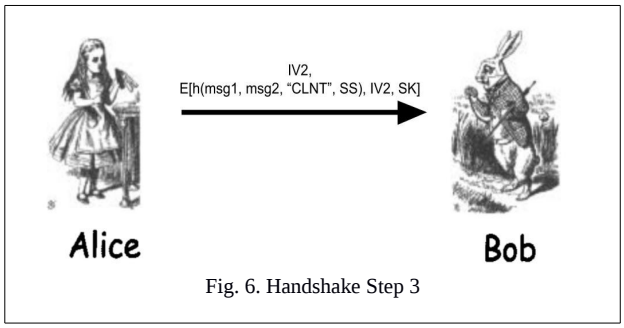

Fig. 6. Handshake Step 3

STEP FOUR

a) This is the last step of the key exchange! Having received **msg3 = E[h(msg1, msg2, CLNT, sharedSecret), IV2, g^{ab} (mod p)], IV2** from A, B now computes the SHA256 hashed **h(msg1, msg2, CLNT, sharedSecret)** and then encrypts it with the IV and DH session key to compare the two encrypted hashes.

b) If the the comparison is correct, then B can safely confirm that the session key has been established.

IV. How encryption/integrity-protection keys are derived from a shared secret value

Message authentication is achieved using a shared secret value and the process of "HMAC chaining". After the DH session key has been established, both A and B will keep a record of a "cumulative HMAC", which begins with the SHA256-hash of (msg1 + msg2 + msg3 + sharedSecret), where the messages are the same ones from the key establishment process demonstrated in the previous section. Every time a new message is sent, it will be encrypted with the DH session key and a new IV, and then hashed with the IV to create the message's HMAC. This new HMAC is then *added* to the old `cumulativeHMAC` and hashed to create a *new* `cumulativeHMAC` to be used in the next message, such that both sides will know when a replay attack has occurred.

Moving forward, the full message has the format of:
`IV | encrypted message | cumulative HMAC`

The code for `sendEncrypted()` in `Host.py` demonstrates how this is being done:

```
def sendEncrypted(self, key, msg):
   iv = Cryptography.generateRandomIV()
   encrypted = Cryptography.symmetricKeyEncrypt(key, iv,
msg)
   hmac = Cryptography.hmac(key, iv, msg)
   # update the cumulativeHmac
   self.cumulativeHmac =
Cryptography.hash(self.cumulativeHmac + hmac)
   fullMessage = iv + encrypted + self.cumulativeHmac

    return self.send(fullMessage)
```

Similarly, when the program receives a message with the `cumulativeHMAC`, it will know how to compute the message and check for integrity based on the old `cumulativeHMAC` that was collected prior to receiving the new message.

```
def recvEncrypted(self, key):
   # contents received
    msg = self.recv(1024)
```

```
    if len(msg) < Cryptography.SYMMETRIC_KEY_IV_SIZE +
Cryptography.HMAC_LENGTH:
        return None
    iv = msg[:Cryptography.SYMMETRIC_KEY_IV_SIZE]
    encrypted = msg[Cryptography.SYMMETRIC_KEY_IV_SIZE :
len(msg) - Cryptography.HMAC_LENGTH]
    receivedCumulativeHmac = msg[-Cryptography.HMAC_LENGTH:]

     # contents calculated from received
     decrypted = Cryptography.symmetricKeyDecrypt(key, iv,
encrypted)
     calculatedHMAC = Cryptography.hmac(key, iv, decrypted)
     self.cumulativeHmac =
Cryptography.hash(self.cumulativeHmac + calculatedHMAC)

     # check if we got a bad message
     if (self.cumulativeHmac != receivedCumulativeHmac):
        return None

    return decrypted
```

Note that due to the one-time pad property of the AES cipher we chose, the program is able to knowingly determine which portion of the received message is the IV, message, or the HMAC based on string length; which is another added benefit of using this cipher.

V. Real-World Implementation Considerations

As far as design of the program, all algorithms implemented in our VPN program are fairly standard, as all are used in many existing real-world implementations. AES is the current cryptographic standard for block cipher encryption, and hashed MACs are used frequently for verifying the integrity of received data. In fact, the only layer that is definitively missing from our VPN implementation is an added level of security for the key exchange process, which can be handled by a slower but more secure cryptosystem such as **RSA**. Using RSA for private key exchange is one of its most prominent use cases in real-world implementations.

Beyond algorithm choices, the more drastic change that we would have to make to the program for it to be realistic is to alter some of the bit sizes that are being used. Although the DH modulus that our program uses is currently a 2048-bit prime (generally sufficient in real-world applications, as the most well-known attack on a public key can crack a 1024-bit prime) and the HMAC length must be fixed at 64-bit for AES, the encryption key and IV sizes are quite small at 32 and 16 bits and may be lengthened or padded to 1024 and 512 bits, respectively.

VI. Software and program / architecture

Reiterating some of what was already discussed in section I: our program is written in Python and is roughly 850 lines long, divided up into some behavior-specific modules. As it is Python, these scripts are independently executable without the need of compiling an auxiliary executable file.

Our command-line interface is driven by the **`main.py`** module (its functionalities outlined previously in section I). Cryptographic helper functions (including imported PyCrypto library functions) are packaged into **`Cryptography.py`**—these include all hash functions used, symmetric key encryption/decryption, and generation of large primes/random numbers/nonces et al.

The **`Host.py`** module (*required inputs*: IP address, port number, shared secret value, and verbose flag) contains all shared server and client functionalities such as initializing the socket connection, `recv()`-ing and `send()`-ing data over the connection, encrypting and decrypting messages with the HMAC applied after the handshake protocol has been completed, and housing the `RecvEncryptedThread` logic (as described previously in section II). It is further abstracted into the **`Server.py`** and **`Client.py`** modules, which execute more program-specific code pertaining to the Diffie-Hellman handshake protocol (as described in section III).

The design of the program is quite simple as it is meant to be a bare-bones implementation. The VPN program does only exactly what it is meant to accomplish per the Assignment 3 guidelines, following the Economy of Mechanism design principle of secure systems.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*

[2] Continuum Analytics, “Linux Miniconda install”, 2016. http://conda.pydata.org/docs/install/quick.html#linux-miniconda-install.