Isaac Llewellyn
Professor Bulent

Cryptography and Network Security

Secure Socket Layer in Python

Implementation of this project started ambitious, as we wanted to implement not only secure socket connections, but also a secure chat server with anonymous voting. This project has seen battle and adapted to the meet time requirements, as such, we switched to implementing a negotiable secure socket connection, which once key negotiation and data encryption type have been established, will echo back the data sent.

In the code base, there exists capability for processing commands as sub-shells but does not establish a new login. Instead, it runs the data as a shell command with the same privilege level as the python process running the server. For security purposes and to not confuse this with SSH, we have decided to not implement this as a standard feature. If we want to use this feature in the future, it will be as simple as modifying one line of code and commenting out another.

For our project, we must note that there will be malicious programmers attacking our code. We can mitigate this via obfuscation, however, Professor Bulent was quite clear that we should not make the code hard to read, or use. Therefore, we have made it explicitly easy for the attackers to step through our program. Part of this is that our server currently acts as an echo server, which allows for a chosen cipher-text attack on our code. This should aid in testing and exploitation for the black hat team.

Our secure socket layer has two major processes. The client and the server. Everything is written in python which was a major design choice by our group. Of course, having everything written in the same language does not solve all issues, but it does allow us to focus more on the task at hand versus the alternative of stitching a cryptographic suite out of multiple dependencies.

One recurring issue we encountered was that some of our group built the code in Python 2.7 and some Python 3.6. Python interacts with sockets differently in both versions. The fix for this has been to strictly encode our messages instead of leaving it up to the interpreter. Our code has been written so that it may run on Python 2.7 and Python 3.6, although running it on Python 3.6 is recommended.

Messages are prone to interception due to the nature of this course. As there is no real way to protect from a man in the middle attack without prior authentication or shared keys, we must assume that any message sent or received can be an attack vector. Our major mitigation to this is using Hased Message Authentication Codes (HMAC). They are relatively easy to implement and provide a relatively secure way to know that there is low chance the message is not what is intended. We do this by hashing our data with SHA1, which we do know is, at this date, broken for most purposes. One reference that may prove interesting to the reader is https://shattered.io/, which help you preform a collision attack on any arbitrary source of data.

A major consideration in implementing a secure socket is the choice of encryption schemas used, and the public key negotiation methods used. Our secure socket supports key negotiation via Elliptic Curve Diffie-Hellman Cryptography (ECC), and the original prime exponent based

Diffie-Hellman (DFH) Cryptography, and data encryption via SimpleData Encryption Standard (SDES), or our version of Rivest Cipher 4.

Generally, this process is automated by both client and server, but for the purpose of our project, we have made the secure socket negotiation process much more verbose and client dependent. We do this by first requiring the client to choose what initial key sharing method they will use. Then, this is sent and verified by the server, which then allows both the client and server to set up public keys.

If at any point the server encounters an error with authentication, the server process simply returns -1, and will wait for a new connection to be established. One issue that is present in our code is that we do not handle packets in a segmented fashion; Instead, we send entire chunks with python's handy 'sendall' socket function. This can prove troublesome when considering packet loss or interference but proves quite convenient.

Both authentication types use the mathematical difficulty of solving discrete logarithms as a bottleneck to prevent our keys from being brute forced. However, our Elliptic Curve authentication functionality should allow for a greater security than standard Diffie-Hellman prime generator. Our data transmission functionality supports SDES for simple block encryption based cryptography, but one should note that their key size is easily brute forced. We have fixed the SDES Substitution Box vulnerability as discussed in class and our homework. Secondly, we support Rivest Cipher 4 (RC4), which allows for a stream of data to be encoded. RC4 has been discussing in lecture but not implemented. As such, we have added it to our cryptography suite. Streaming bit by bit via Rivest Cipher 4 is not implemented, as the

communication done between both parties is encapsulated into chunks before being sent over the wire.

Overall our program is not without flaws; SDES is vulnerable to a brute force attack, and we never share keys between client and server, which makes it impossible to know if we are talking to an adversary or not. One major cryptography suite we were debating on implementing was G/PGP, however, we have not gone over it in lecture and found it too over the top for the scope of this project.

The client is run after the server, but it can be modified to run in any way needed for enterprise solutions

Running the program:

- First, run server.py.

- Then, run client.py.

    - As a client, sent 0 or 1 to initiate Elliptic Curve or Diffie Hellman key exchange.

    - After a successful key exchange has been established the server prompts the client to pick between SimpleData Encryption Standard(SDES), or our version of Rivest Cipher 4.

    - From there the server echoes back encrypted data with HMAC / SHA1 authentication.