Vamshi Arugonda
Luke Bakken
Zach Ryan
03 / 02 / 2020

Project 2: Emulating SSL and Password Verification

The *add_user.py* file generates a password file with usernames, salts, and password hashes. The user is prompted to input a username, followed by a password. These values are stored temporarily within python while the corresponding values are computed. A random 16 character hash is generated for each new user. This is done by randomly selecting 16 characters from a list of all lowercase, uppercase, and numerical characters. This function is not guaranteed to be unique, but for the expected number of users, it can be treated as such with a space of 62^{16}. The salt is appended to the end of the user input password and then hashed with sha256. The username, salt, and password hash are printed to the password file, then the program exits execution. This garbage collects all temporary python variables, including the plaintext password. The hashing algorithm, sha256, guarantees that the password plus salt will always output the same value and the password cannot be reverse-engineered from the salt and hash. Because of this, the password storage algorithm is secure and safe.

The public and private keys of the server and client were generated using the command: *$ ssh-keygen -f*. The keys were generated, without a password in this case, into both the client and server directories. The server and client each have unique RSA public and private key pairs. The library which uses the RSA keys for encryption is Crypto. This library was chosen for its ease of use. The key public file is read as bytes and then the library encrypts the message. The receiver then reads their private key file as bytes and decrypts the encoded message.

The handshake between server and client takes place using the RSA key pairs. The client generates an AES session key and encrypts it with the server's public key. Because of this, only the server can decrypt the AES session key broadcast by the client. The server makes note of the session key, and responds with 'okay'. The AES session key is now known only by the client and the server. This AES session key is used to encrypt the username and password sent by the client to the server. The server verifies the username and password against the username, salt, and password hash stored in the password file, then sends a ‘SUCCESS’ or ‘FAILURE’ message, encrypted with the AES session key, back to the client. Only the client and server have access to the session key, so they are the only two able to interpret the messages back and forth, making the session secure from third party listeners. AES was chosen because of its relative speed over similar encryption methods. Once the session is secured with RSA, AES makes continued interaction faster with the session key.

In the event of a replay attack, Eve could send the encrypted username password pair and would receive an encrypted success or failure response. The encrypted message received by Eve will be unreadable because she does not have the session key. Eve can, however, create her own session with the server and check username-password pairs.

The largest takeaway our group had from this project was the ease to implement both RSA and AES encryptions using python. With two files of ~150 lines of code, we can create a server and client capable of securely communicating across a network.