Client/Server Interaction Results

Zack McKevitt, Connor Radeloff, Sahib Bajwa

Hashing

In our project, our hashing scheme involved the use of a 16 byte unique salt. To achieve this hashing, we used the python libraries os and hashlib. We generated our salt using the os.urandom() method, which allows us to generate a pseudo random salt. Further, we used the hashlib.pbkdf2_hmac() method to hash the password. This function takes in parameters which allow us to input the hashing algorithm we want to use, the password we are hashing, and the salt we previously generated. For our project, we decided to use sha256. We decided to use this hash member function because, as stated in the documentation, it is used for key derivation, which is useful in the context of our problem (more specifically, PKCS5 password based key derivation function 2).

Keys

To generate our keys, we used the command ssh-keygen. This allowed us to generate a public/private key pair for the server. The server's public key is used to encrypt the AES session key to send to the server. The server's private key is then used to decrypt messages sent over the handshake. We stored both keys server side in the server folder. In terms of the libraries used, we used the Crypto.Cipher and Crypto.PublicKey modules to import the AES, PKCS1_0AEP, and RSA libraries. We used the AES library to encrypt and decrypt the messages sent. Since our session key was generated with AES, that is the encryption scheme that we used after the connection had been established. We used RSA to generate the public and private key pairs, which allowed us to verify a connection between the client and server and to establish a unique session key.

Symmetric Encryption

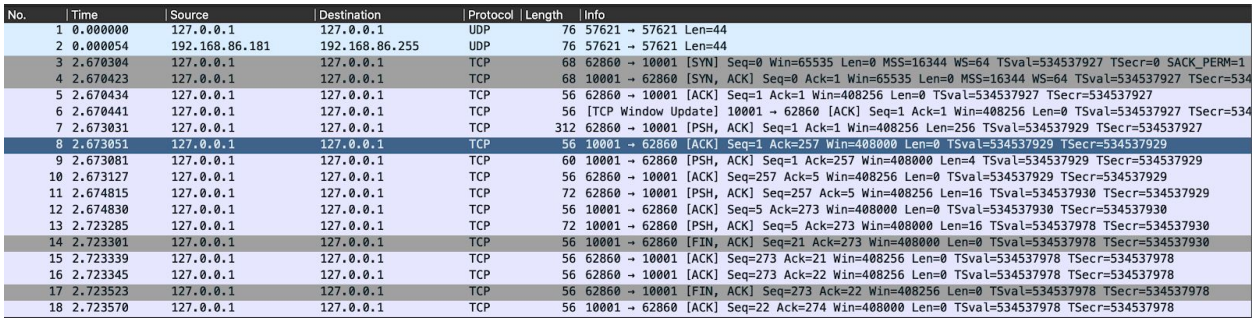
For encryption, we used AES for symmetric encryption and RSA for asymmetric encryption. As the server is listening for incoming connections, the client will begin the encryption method. The client generates a random AES key and encrypts it using RSA and the server's public key. The client will then send the encrypted AES key to the server, which will respond with an "okay" if successfully received. The server will then go on to decrypt the key with its own private key and wait for the client to send its message. The client, once receiving the "okay" from the server will go on to encrypt the message it wants to pass along to the server. The client will then Encrypt the message using AES and the randomly generated AES key. We decided to use AES.MODE_ECB to encrypt the message. ECB is a simple encryption mode which uses the same key to encrypt each block of plain text. As ECB encryption is simple in that it will encrypt identical text blocks to identical ciphertext blocks, it is not the safest form of encryption. For the simplicity of ECB

encryption we may lose confidentiality in the data if it gets decrypted by a third party. An alternative to this would have been to use CTR encryption as it is considered secure but is not as simple as ECB encryption. After encrypting the message using AES mode ECB, the client passes along the message to the server. The server then decrypts the message using the shared key and checks the message as to whether it is contained as a username + password combination. If it is, the server reports a success back to the client and closes the socket. If it is not, the server reports a failure to the client and closes the socket.

Eavesdroppers

Since our program makes use of private keys and a unique session key, an eavesdropper would be unable to sniff any messages sent over the connection.

Here is a wireshark capture of **one** login session:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	76	57621 → 57621 Len=44
2	0.000054	192.168.86.181	192.168.86.255	UDP	76	57621 → 57621 Len=44
3	2.670304	127.0.0.1	127.0.0.1	TCP	68	62860 → 10001 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=534537927 TSecr=0 SACK_PERM=1
4	2.670423	127.0.0.1	127.0.0.1	TCP	68	10001 → 62860 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=534537927 TSecr=534
5	2.670434	127.0.0.1	127.0.0.1	TCP	56	62860 → 10001 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=534537927 TSecr=534537927
6	2.670441	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 10001 → 62860 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=534537927 TSecr=534
7	2.673031	127.0.0.1	127.0.0.1	TCP	312	62860 → 10001 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=256 TSval=534537929 TSecr=534537927
8	2.673051	127.0.0.1	127.0.0.1	TCP	56	10001 → 62860 [ACK] Seq=1 Ack=257 Win=408000 Len=0 TSval=534537929 TSecr=534537929
9	2.673081	127.0.0.1	127.0.0.1	TCP	60	10001 → 62860 [PSH, ACK] Seq=1 Ack=257 Win=408000 Len=4 TSval=534537929 TSecr=534537929
10	2.673127	127.0.0.1	127.0.0.1	TCP	56	62860 → 10001 [ACK] Seq=257 Ack=5 Win=408256 Len=0 TSval=534537929 TSecr=534537929
11	2.674815	127.0.0.1	127.0.0.1	TCP	72	62860 → 10001 [PSH, ACK] Seq=257 Ack=5 Win=408256 Len=16 TSval=534537930 TSecr=534537929
12	2.674830	127.0.0.1	127.0.0.1	TCP	56	10001 → 62860 [ACK] Seq=5 Ack=273 Win=408000 Len=0 TSval=534537930 TSecr=534537930
13	2.723285	127.0.0.1	127.0.0.1	TCP	72	10001 → 62860 [PSH, ACK] Seq=5 Ack=273 Win=408000 Len=16 TSval=534537978 TSecr=534537930
14	2.723301	127.0.0.1	127.0.0.1	TCP	56	10001 → 62860 [FIN, ACK] Seq=21 Ack=273 Win=408000 Len=0 TSval=534537978 TSecr=534537930
15	2.723339	127.0.0.1	127.0.0.1	TCP	56	62860 → 10001 [ACK] Seq=273 Ack=21 Win=408256 Len=0 TSval=534537978 TSecr=534537978
16	2.723345	127.0.0.1	127.0.0.1	TCP	56	62860 → 10001 [ACK] Seq=273 Ack=22 Win=408256 Len=0 TSval=534537978 TSecr=534537978
17	2.723523	127.0.0.1	127.0.0.1	TCP	56	62860 → 10001 [FIN, ACK] Seq=273 Ack=22 Win=408256 Len=0 TSval=534537978 TSecr=534537978
18	2.723570	127.0.0.1	127.0.0.1	TCP	56	10001 → 62860 [ACK] Seq=22 Ack=274 Win=408000 Len=0 TSval=534537978 TSecr=534537978

Not very useful.

Replay Attack Vulnerabilities

This implementation is vulnerable to replay attacks. Because we are sending messages that do not identify ourselves, someone could conduct a replay attack. If we implemented a Needham-Schroeder Protocol or something similar, we could prevent replay attacks.

Takeaways

One primary thing we took away from this project is that implementations of secure messaging protocols and secure data transfer are much more complex than we had initially thought. It took us quite a while to wrap our heads around what we were doing and how we were implementing it. Lots of googling was done. I think it's fair to say that we believe the implementation of these functions and protocols should be left to the experts. This was a great exercise, but security vulnerabilities in the code itself is something not to be taken lightly, and one should use the most up to date libraries and protocols to ensure the most security in a product or service.

Sources

https://techtutorialsx.com/2018/04/09/python-pycrypto-using-aes-128-in-ecb-mode/

https://tools.ietf.org/html/rfc3686

http://www.crypto-it.net/eng/theory/modes-of-block-ciphers.html

https://github.com/Zackagawea/cybersecproject2 - (GitHub Repository)