Jordan Smart: josm7269@colorado.edu

Jonathan Bluhm: jobl6075@colorado.edu

Laura Kaiser: laka4851@colorado.edu

Project 2 Report

In this project we had a server and a client. The goal was for the client to be able to log in using a username and password, and after they logged in they would be able generate a session key to send messages to the server. For this project there was a total of 6 TODO's: 3 in the client section and 3 in the server.

One of the first obstacles was to create a hashed password. To make the hashing possible we added the library SHA256 from crypto.hash into our code. We also salted the password before encryption by taking the password and adding the salt (password+salt). To encrypt the password we used the command SHA256.new(str.encode(password_and_salt)).hexdigest() in order to hash the password, which was taken from the crypto.hash library mentioned earlier. We chose these ways to hash, not only because it was our goal, but with salting we made it so if an attacker somehow got their hands on the passwords, the hashes would be different causing the attacker to only be able to guess one client's password at a time. This added a little more security to our program.

We also used the random library from Crypto. We generated the 32 byte salt using the get_random_bytes method from Random class, and then converted it to a string so that we could concatenate it with the password string. We chose to make the salt 32 bytes so that our password hashes are even longer than normal. We then re-encoded it so it could be hashed. We hashed the password so that the password is not stored anywhere in plaintext. If an attacker saw our

password file, they would have no idea what the passwords are because they're just a bunch of gibberish.

After the client was able to log in, they would be able to send messages using the server for authentication. To start we needed to create keys. We created a private and a public key using the command ssh-keygen in a terminal. The public key was then stored in the client folder, and the private key was stored in the server folder. Inside the programs, we imported the library AES from crypto.cypher for encryption purposes later in the program.

To encrypt the message, we generated a random 16 byte key for the session, using this as the symmetric key that would be used to encrypt and decrypt all future messages. For the encryption process, we used the public key to encrypt the symmetric key, then we sent the encrypted symmetric key. The server would then use the private key to decrypt the symmetric key. Then the server sends confirmation to the client. Unfortunately for us, our encryption is also padded, which makes the code a bit more predictable, which to an attacker can be a big advantage.

We used AES to encrypt the message using the symmetric key. It is a padded message, again making it a bit more predictable. We were under the impression that AES was required, and we understand that this is an unideal case. We would have preferred to use a symmetric cipher instead of a block cipher, as the block cypher makes our encryption more predictable and easier to decrypt.

Our program is secure from eavesdroppers because nothing sent over the network is unencrypted. An attacker could not decrypt the symmetric key since only the server has the private key which can decrypt the symmetric key. The symmetric key is then used to create

cipher objects which encrypt all future messages. An attacker could never get the symmetric key which generates the cipher object; therefore, messages encrypted with the cipher are secure from attackers. Wireshark verified this. We sent an unencrypted message called “WIRESHARK” from the client to the server (see unecnrpyted_msg image) .This was intercepted and readable by wireshark. The server also sent an unencrypted message and this was viewable on wireshark. However, the rest of the communications were all gibberish. First message (see first_msg file) is a lot longer than the rest of the messages, and this is because we are sending over the session key encrypted with the public key, so it is a lot longer. Despite our protection against eavesdroppers, our program still isn’t perfect. We still have some vulnerabilities when it comes to replay attacks. Replay attacks are a low tier Man in the Middle attack in which the attacker delays or resends a valid message to one of the recipients. This type of attack can happen regardless of whether the information is encrypted or not. We have nothing that stops a man in the middle from delaying or resending the message.

We learned to use hex digest and not digest when storing hashes. Comparing the digest of the hashes was not possible, but with hex digest it worked like a charm. We discovered that wireshark is not good at monitoring local host traffic; however, opening files and reading them in wireshark is not too hard. We also learned about different types of filters in wireshark. Finally, we bolstered our understanding of public key infrastructure, and managed to create a functioning and secure communication line.