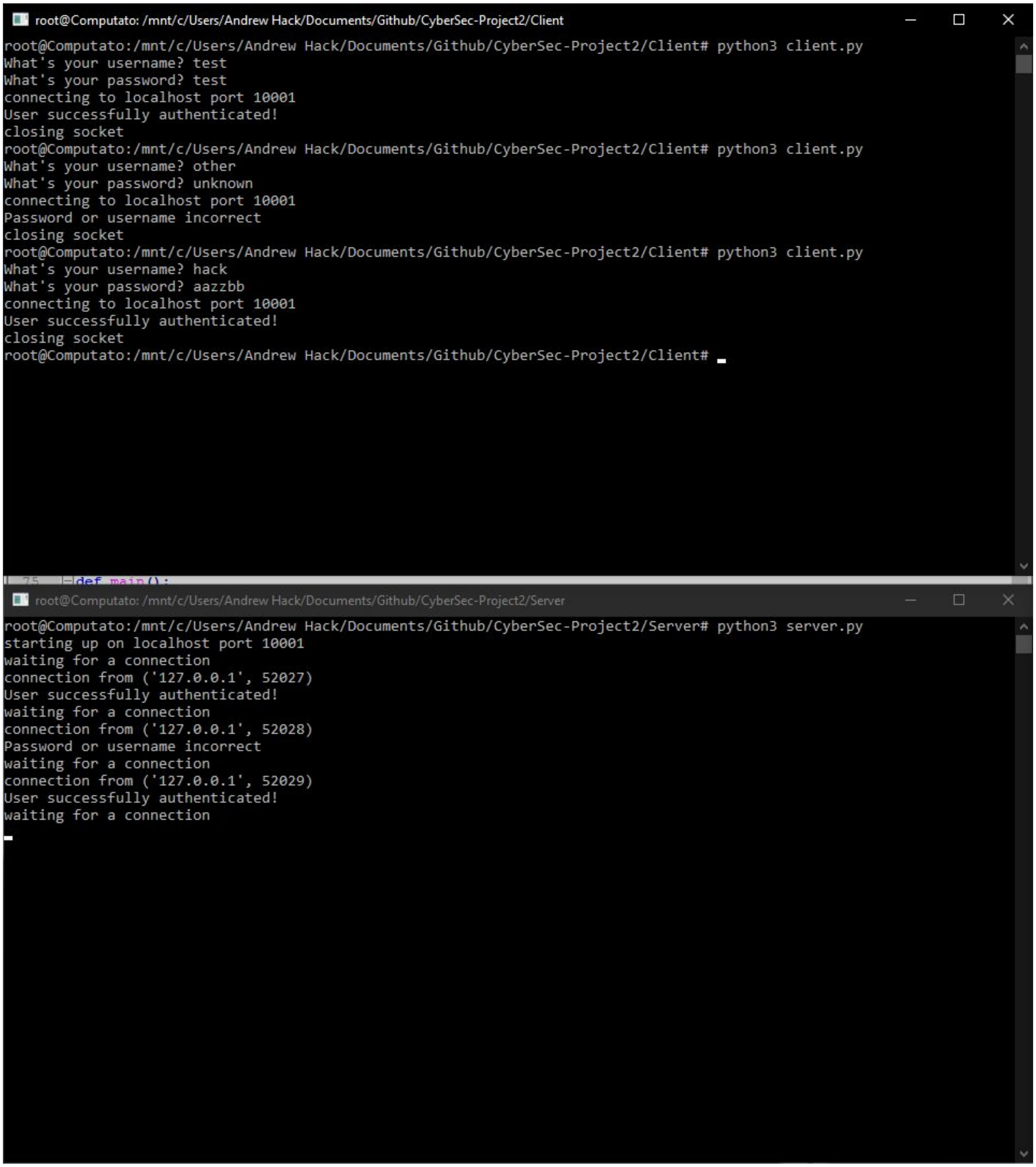
RESULTS

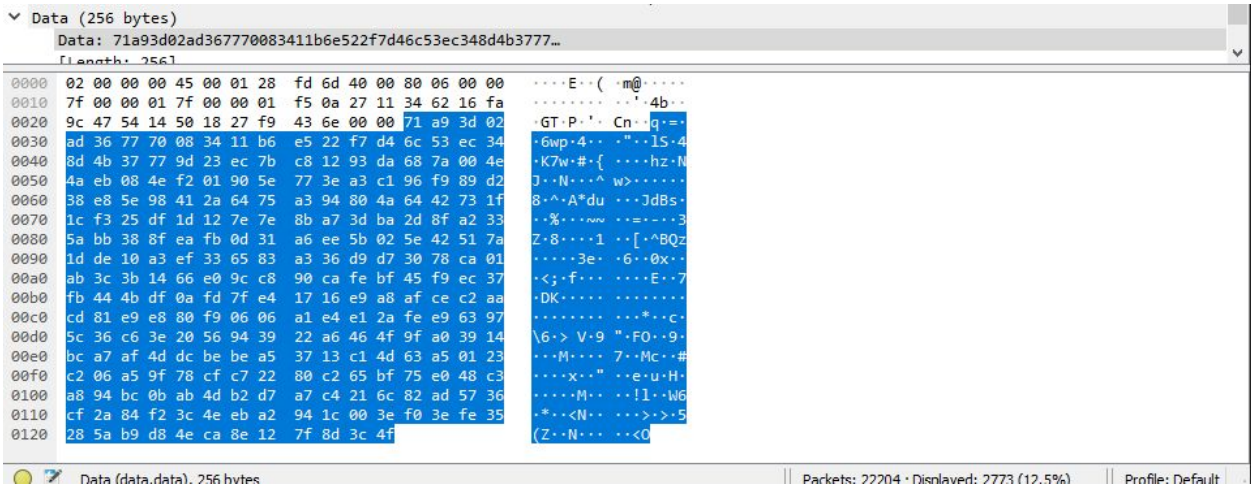


root@Computato: /mnt/c/Users/Andrew Hack/Documents/Github/CyberSec-Project2/Client

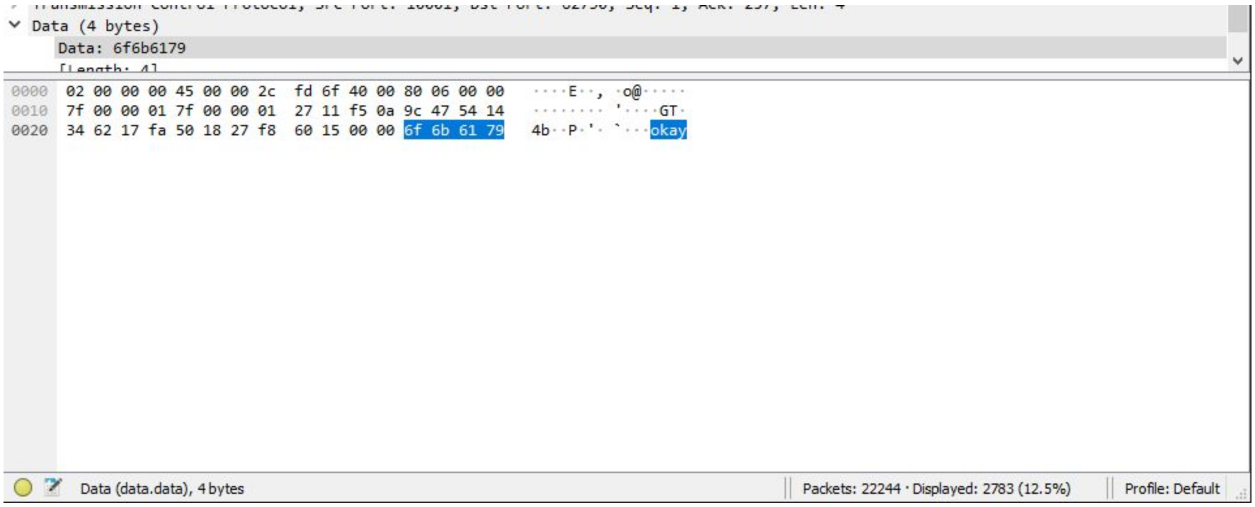
```
root@Computato:/mnt/c/Users/Andrew Hack/Documents/Github/CyberSec-Project2/Client# python3 client.py
What's your username? test
What's your password? test
connecting to localhost port 10001
User successfully authenticated!
closing socket
root@Computato:/mnt/c/Users/Andrew Hack/Documents/Github/CyberSec-Project2/Client# python3 client.py
What's your username? other
What's your password? unknown
connecting to localhost port 10001
Password or username incorrect
closing socket
root@Computato:/mnt/c/Users/Andrew Hack/Documents/Github/CyberSec-Project2/Client# python3 client.py
What's your username? hack
What's your password? aazzbb
connecting to localhost port 10001
User successfully authenticated!
closing socket
root@Computato:/mnt/c/Users/Andrew Hack/Documents/Github/CyberSec-Project2/Client#
```

root@Computato: /mnt/c/Users/Andrew Hack/Documents/Github/CyberSec-Project2/Server

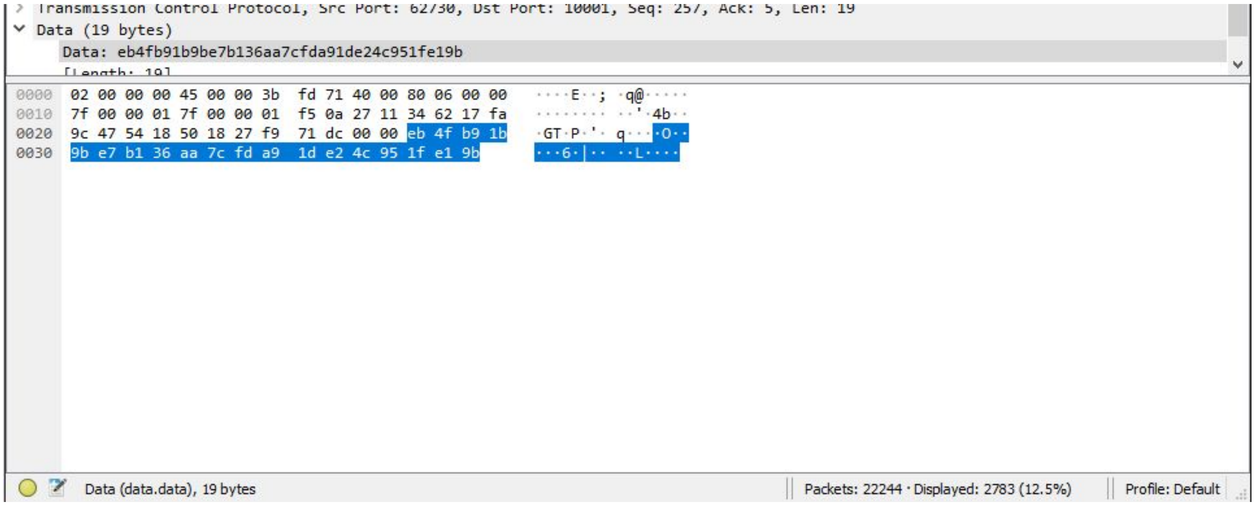
```
root@Computato:/mnt/c/Users/Andrew Hack/Documents/Github/CyberSec-Project2/Server# python3 server.py
starting up on localhost port 10001
waiting for a connection
connection from ('127.0.0.1', 52027)
User successfully authenticated!
waiting for a connection
connection from ('127.0.0.1', 52028)
Password or username incorrect
waiting for a connection
connection from ('127.0.0.1', 52029)
User successfully authenticated!
waiting for a connection
```



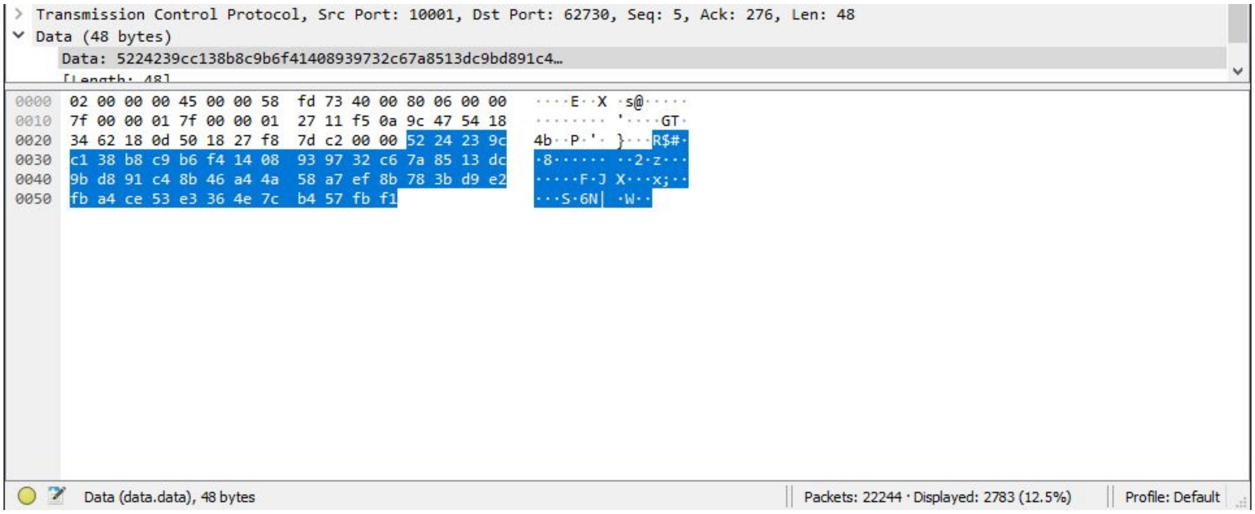
```
Data (256 bytes)
    Data: 71a93d02ad367770083411b6e522f7d46c53ec348d4b3777…
    [Length: 256]

0000  02 00 00 00 45 00 01 28  fd 6d 40 00 80 06 00 00   ····E··( ·m@·····
0010  7f 00 00 01 7f 00 00 01  f5 0a 27 11 34 62 16 fa   ········ ··'·4b··
0020  9c 47 54 14 50 18 27 f9  43 6e 00 00 71 a9 3d 02   ·GT·P·'· Cn··q·=·
0030  ad 36 77 70 08 34 11 b6  e5 22 f7 d4 6c 53 ec 34   ·6wp·4·· ·"··lS·4
0040  8d 4b 37 77 9d 23 ec 7b  c8 12 93 da 68 7a 00 4e   ·K7w·#·{ ····hz·N
0050  4a eb 08 4e f2 01 90 5e  77 3e a3 c1 96 f9 89 d2   J··N···^ w>······
0060  38 e8 5e 98 41 2a 64 75  a3 94 80 4a 64 42 73 1f   8·^·A*du ···JdBs·
0070  1c f3 25 df 1d 12 7e 7e  8b a7 3d ba 2d 8f a2 33   ··%···~~ ··=·-··3
0080  5a bb 38 8f ea fb 0d 31  a6 ee 5b 02 5e 42 51 7a   Z·8····1 ··[·^BQz
0090  1d de 10 a3 ef 33 65 83  a3 36 d9 d7 30 78 ca 01   ·····3e· ·6··0x··
00a0  ab 3c 3b 14 66 e0 9c c8  90 ca fe bf 45 f9 ec 37   ·<;·f··· ····E··7
00b0  fb 44 4b df 0a fd 7f e4  17 16 e9 a8 af ce c2 aa   ·DK····· ········
00c0  cd 81 e9 e8 80 f9 06 06  a1 e4 e1 2a fe e9 63 97   ········ ···*··c·
00d0  5c 36 c6 3e 20 56 94 39  22 a6 46 4f 9f a0 39 14   \6·> V·9 "·FO··9·
00e0  bc a7 af 4d dc be be a5  37 13 c1 4d 63 a5 01 23   ···M···· 7··Mc··#
00f0  c2 06 a5 9f 78 cf c7 22  80 c2 65 bf 75 e0 48 c3   ····x··" ··e·u·H·
0100  a8 94 bc 0b ab 4d b2 d7  a7 c4 21 6c 82 ad 57 36   ·····M·· ··!l··W6
0110  cf 2a 84 f2 3c 4e eb a2  94 1c 00 3e f0 3e fe 35   ·*··<N·· ···>·>·5
0120  28 5a b9 d8 4e ca 8e 12  7f 8d 3c 4f               (Z··N··· ··<O

Data (data.data), 256 bytes    Packets: 22204 · Displayed: 2773 (12.5%)    Profile: Default
```



```
Transmission Control Protocol, Src Port: 10001, Dst Port: 62730, Seq: 1, Ack: 257, Len: 4
Data (4 bytes)
    Data: 6f6b6179
    [Length: 4]

0000  02 00 00 00 45 00 00 2c  fd 6f 40 00 80 06 00 00   ····E··, ·o@·····
0010  7f 00 00 01 7f 00 00 01  27 11 f5 0a 9c 47 54 14   ········ '····GT·
0020  34 62 17 fa 50 18 27 f8  60 15 00 00 6f 6b 61 79   4b··P·'· `···okay

Data (data.data), 4 bytes    Packets: 22244 · Displayed: 2783 (12.5%)    Profile: Default
```



```
Transmission Control Protocol, Src Port: 62730, Dst Port: 10001, Seq: 257, Ack: 5, Len: 19
Data (19 bytes)
    Data: eb4fb91b9be7b136aa7cfda91de24c951fe19b
    [Length: 19]

0000  02 00 00 00 45 00 00 3b  fd 71 40 00 80 06 00 00   ····E··; ·q@·····
0010  7f 00 00 01 7f 00 00 01  f5 0a 27 11 34 62 17 fa   ········ ··'·4b··
0020  9c 47 54 18 50 18 27 f9  71 dc 00 00 eb 4f b9 1b   ·GT·P·'· q····O··
0030  9b e7 b1 36 aa 7c fd a9  1d e2 4c 95 1f e1 9b      ···6·|·· ··L····

Data (data.data), 19 bytes    Packets: 22244 · Displayed: 2783 (12.5%)    Profile: Default
```



```
> Transmission Control Protocol, Src Port: 10001, Dst Port: 62730, Seq: 5, Ack: 276, Len: 48
v Data (48 bytes)
    Data: 5224239cc138b8c9b6f41408939732c67a8513dc9bd891c4…
    [Length: 48]

0000  02 00 00 00 45 00 00 58  fd 73 40 00 80 06 00 00   ····E··X ·s@·····
0010  7f 00 00 01 7f 00 00 01  27 11 f5 0a 9c 47 54 18   ········ '····GT·
0020  34 62 18 0d 50 18 27 f8  7d c2 00 00 52 24 23 9c   4b··P·'· }···R$#·
0030  c1 38 b8 c9 b6 f4 14 08  93 97 32 c6 7a 85 13 dc   ·8······ ··2·z···
0040  9b d8 91 c4 8b 46 a4 4a  58 a7 ef 8b 78 3b d9 e2   ·····F·J X···x;··
0050  fb a4 ce 53 e3 36 4e 7c  b4 57 fb f1               ···S·6N| ·W··
```

Data (data.data), 48 bytes | Packets: 22244 · Displayed: 2783 (12.5%) | Profile: Default

All of the messages except for the “okay” message were encrypted. The server correctly identifies user/password combos and successfully returns the correct response to the client.

REPORT

We used PBKDF2 in order to ensure that it takes longer to guess a hash if the passfile was to be compromised. In this algorithm, the salt will be used repeatedly in a series of hashes, slowing attempts to guess the passwords that match the hashes. The encryption algorithm utilized is the *SHA-512* algorithm. This algorithm was selected for having high resistance against collision attacks and length extension attacks, as can be seen here: https://en.wikipedia.org/wiki/Secure_Hash_Algorithms. *SHA-3* was not used due to being incompatible with the pbkdf2_hmac() function.

To generate our key pair, we used the Python Cryptography Toolkit (pycrypto) that has RSA encryption methods included. We generated a one time pair of keys and stored these in folders within the project directory. We specifically used the RSA library in pycrypto to create the keys. The keys were generated one time so that there would be access in both the server.py and the client.py files. The private key is stored within the server.py file and the public key is stored within the client.py file.

We are managing symmetric encryption by using AES (Advanced Encryption Standard), a symmetric encryption algorithm. Within that, we used the mode of Cipher Feedback (CFB), which we decided to use because the block cipher is only used in the encrypting direction, meaning that the message does not need to be padded to the multiple of a block size. Downfalls are that CFB cannot encrypt data in parallel, slowing down the encryption process. But this isn't something that bothered us, given the small size of the message we are encrypting. Another downfall of CFB is that it is susceptible to replay attacks. However, it is safe from a chosen plain-text attack, because the attacker has no way of knowing if they got the correct data.

This program would be secure from eavesdroppers on a publicly visible network because the data being sent over the network at all times is encrypted data. To start, when the session key is being sent over the network, it is encrypted with the public key of the server. This means that the server is the only thing that will be able to decrypt the session key. Therefore the session key is safe when it is sent over. Once the client knows that the server has received the session key, it then creates an AES encryption for the username and password, so this information will be encrypted before it is sent over a public network. The server then sends back an encrypted message telling the client whether the hashed username and password match. Therefore, at all times the information being sent over the public network would be safe from sniffing. The only way that the information could be compromised is if the public and private keys were stolen from the local files or if the session key was stolen from either of the local files.

Our program is susceptible to replay attacks. If someone grabbed the encrypted message during its passing from the client and the server, they could pass the same encrypted message through our system again, and our system would have no way of knowing the encrypted message has been sent before, and everything would be successful for the attacker. If we wanted to protect against this, we could include a unique tag with every message, and have the decryption function check to make sure that the program has not already seen that unique tag before, as it would compare it to a stored database of seen unique tags.

I learned that trying to create a secure connection between two different ports is much more difficult than it may sound. I certainly had to read lots of the documentation in regards to the implementation of both RSA from the pycrypto file and the AES encryption implementation to get the encryption methods to work when sending the information from the client to the server. I was surprised that while the concepts may seem relatively simple, the implementation of them can be quite difficult.