Last Name:	Champagne
First Name:	Steven
Course:	CPSC526
Assignment:	4: Encrypted File Transfer
Tutorial:	T03
Date:	2017-11-12
Files Submitted:	readme.pdf, myclient6.py, myserver6.py
python version:	3.6.2

===

HOW TO COMPILE SECTION

```
ARGS PROTOTYPE: myserver6.py <PORT> <SECRETKEY>
        myclient6.py <COMMAND> <FILENAME> <HOST:PORT> <CIPHER> <SECRETKEY>
```

```
HOW TO RUN (EXAMPLE):
  RUN SERVER FIRST: $ python3.6 myserver4.py 5555 mysecret

  RUN CLIENT SECOND:
    READ:
      $ python3.6 myclient4.py read a.txt localhost:5555 null mysecret
    WRITE:
      $ cat test.txt | python3.6 myclient3.py write a.txt localhost:5555 null mysecret
```

===

TESTING (AES256 upload/download/checksums/1MB file) SECTION

The client/server was tested with files of the following sizes: 0B, 1KB, 1MB, 1GB.

The files were generated using the dd command in linux. For example:

```
  $ dd if=/dev/urandom bs=1K iflag=fullblock count=1M > 1GB.bin
```

The SHA256 hashes were calculated using $ sha256sum *.bin

```
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855  0B.bin
c7a56487202a1b216a048c77cefdc374c2345c248caee945dfed0d1f693ca7dd  1GB.bin
f267497127be5cc7bf4866385fd23c895b68639134a4214be6ebdc4fce68eb3a  1KB.bin
14bd7ef141a787b931ce63bc4f259ada2390538b7250f93f024e85fa59c5ed7b  1MB.bin
```

The files were then read by the client and the sha256 hashes were recalculated and the hashes matched:

```
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855  0B.bin
c7a56487202a1b216a048c77cefdc374c2345c248caee945dfed0d1f693ca7dd  1GB.bin
f267497127be5cc7bf4866385fd23c895b68639134a4214be6ebdc4fce68eb3a  1KB.bin
14bd7ef141a787b931ce63bc4f259ada2390538b7250f93f024e85fa59c5ed7b  1MB.bin
```

The files were then resent to the server and again the hashes matched:

```
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855  0Bcpy.bin
f267497127be5cc7bf4866385fd23c895b68639134a4214be6ebdc4fce68eb3a  1KBcpy.bin
14bd7ef141a787b931ce63bc4f259ada2390538b7250f93f024e85fa59c5ed7b  1MBcpy.bin
c7a56487202a1b216a048c77cefdc374c2345c248caee945dfed0d1f693ca7dd  1GBcpy.bin
```

==

COMMUNICATION PROTOCOL DESCRIPTION

I decided to do the communication protocol as closely as described in the assignment description as possible.

1. client: generates the nonce and concatenates it with the cipher.
   client receives secret key from command line and generates IV and SK from it and nonce
2. client -> server: client sends cipher_nonce unencrypted.
3. server: receives cipher_nonce and generates IV and SK. server generates challenge.

The challenge is the client must sha256 hash a randomly generated bytestring with the secret key and return the hash to the server. (Though it would be possible to break this by simply listening to the transmission of the random string and the response from the client, then brute forcing until you find the same hash. But these communications are encrypted. But it is still possible to break this by simply knowing which encryption algorithm is used which was sent previously unencrypted.)

4. server -> client: sends challenge
5. client -> server: sends hash of challenge with secret key.
6. server: if hashes match then server proceeds. if not then server does not reply, and
   breaks the connection, logs the attempt.
7. client -> server: client sends the command_filename to the server.
8. server -> client: If the command is doable the server replies "GOOD". If not doable
   the server breaks the connection.

==

TIMING REPORT AND CONCLUSIONS

NOTE: These timings are calculated from the client side.
NOTE: These tests were run with client and server on the same machine.

	filename	average_time (s)	correct hash
	------------	----------------------	-----------------
cipher = null			
command = read			
	0B.bin	0.320	yes
	1KB.bin	0.323	yes
	1MB.bin	0.373	yes
	1GB.bin	40.782 (some failed)	not always
cipher = null			
command = write			
	0B.bin	0.327	yes
	1KB.bin	0.324	yes
	1MB.bin	0.345	yes
	1GB.bin	some failed	not always
cipher = aes128			
command = read			
	0B.bin	0.323	yes
	1KB.bin	0.331	yes
	1MB.bin	0.577	yes
	1GB.bin	3m25s	yes
cipher = aes128			
command = write			
	0B.bin	0.311	yes
	1KB.bin	0.338	yes
	1MB.bin	0.452	yes
	1GB.bin	some failed	not always
cipher = aes256			
command = read			
	0B.bin	0.315	yes
	1KB.bin	0.320	yes
	1MB.bin	0.525	yes
	1GB.bin	3m16s (and fails)	not always
cipher = aes256			
command = write			
	0B.bin	0.317	yes
	1KB.bin	0.327	yes
	1MB.bin	0.450	yes
	1GB.bin	failed	no

CONCLUSIONS

I have noted that the difference between the AES128 and AES256 encryption is insignificant.
Also, I have noted that my protocol is not reliable for large file sizes.

If I had more time and/or were to re-do this assignment I would:

1. Wait for data to fully populate the buffer before attempting to read from it. such as:
   if buffer should be = buffer.length: wait till message fully received, THEN proceeded.
2. Perhaps I would use a separate communication channel to do ACKS of packets.
3. Only finalize padding if the buffer is not full when sending.

I believe these would have solved my problems for the larger files.

===