Rumen Kasabov
ID: 10128754
Tutorial Section: T02

Group partner: Michael Radke

~~~

Running the code

To run the code you need to start the server first on terminal as follows:

$python3.6 server.py port key

Where:

- port is the port number the server will be listening on
- key is the secret key of the server

Then you need to start the client as follows:

$python3.6 client.py command filename hostname:port cipher key

Where:

- command is the file operation the client will be performing (either 'read' or 'write')
- filename is the file that the server will read from to the client or write to on the server side
- hostname:port is the server address of the server the client is connecting to and port is the port number of that server. A colon separates them.
- cipher is the cipher being used (either aes128, aes256 or the null cipher)
- key is the secret key of the client
~~~

Test for correctness:

We upload (write) a 1MB bin file we create to the server from the client. Then download (read) the uploaded file to a different file. This yields the same checksums for all three files.

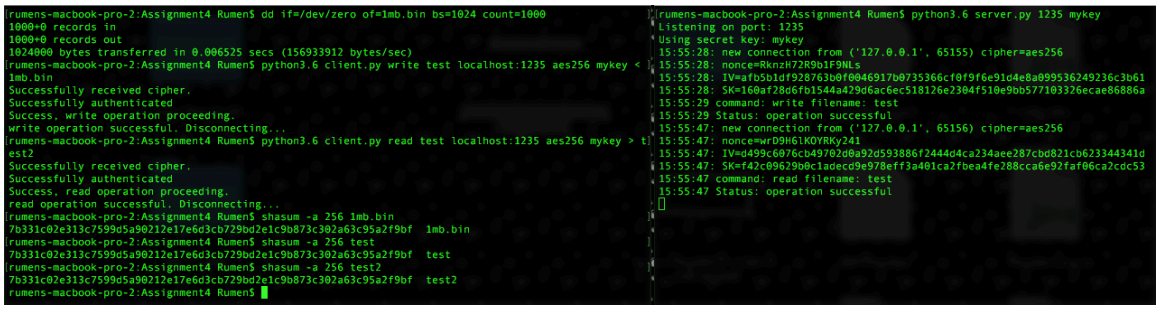
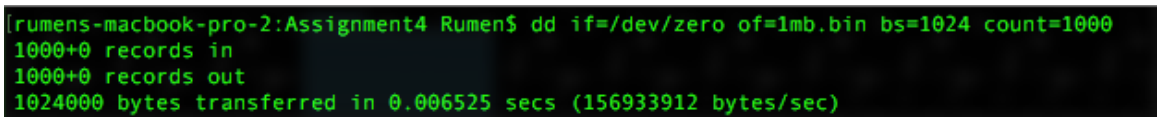


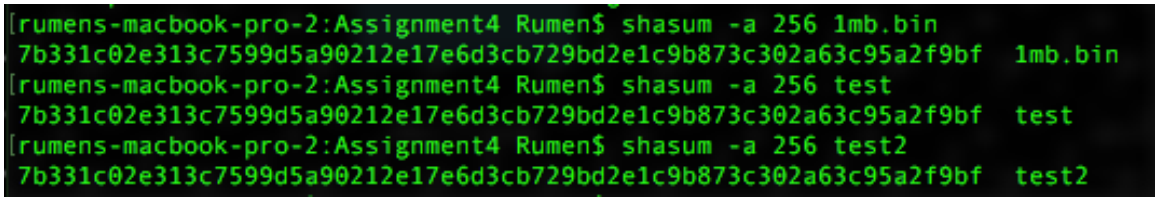
Figure 1. *Note: Left terminal is the client.py program and right terminal is the server.py program.*

As seen in this image taken from the Figure 1, we create a 1MB bin file:



We then connect to the server and read 1mb.bin from standard input and write it to a file called test. Afterwards we connect to the server again, reading the file test into another file called test2 through standard output.

Then we compute the checksums using sha256, which as seen in the left terminal of Figure 1 yields the same three checksums for 1mb.bin, test, and test2:



Communication Protocol

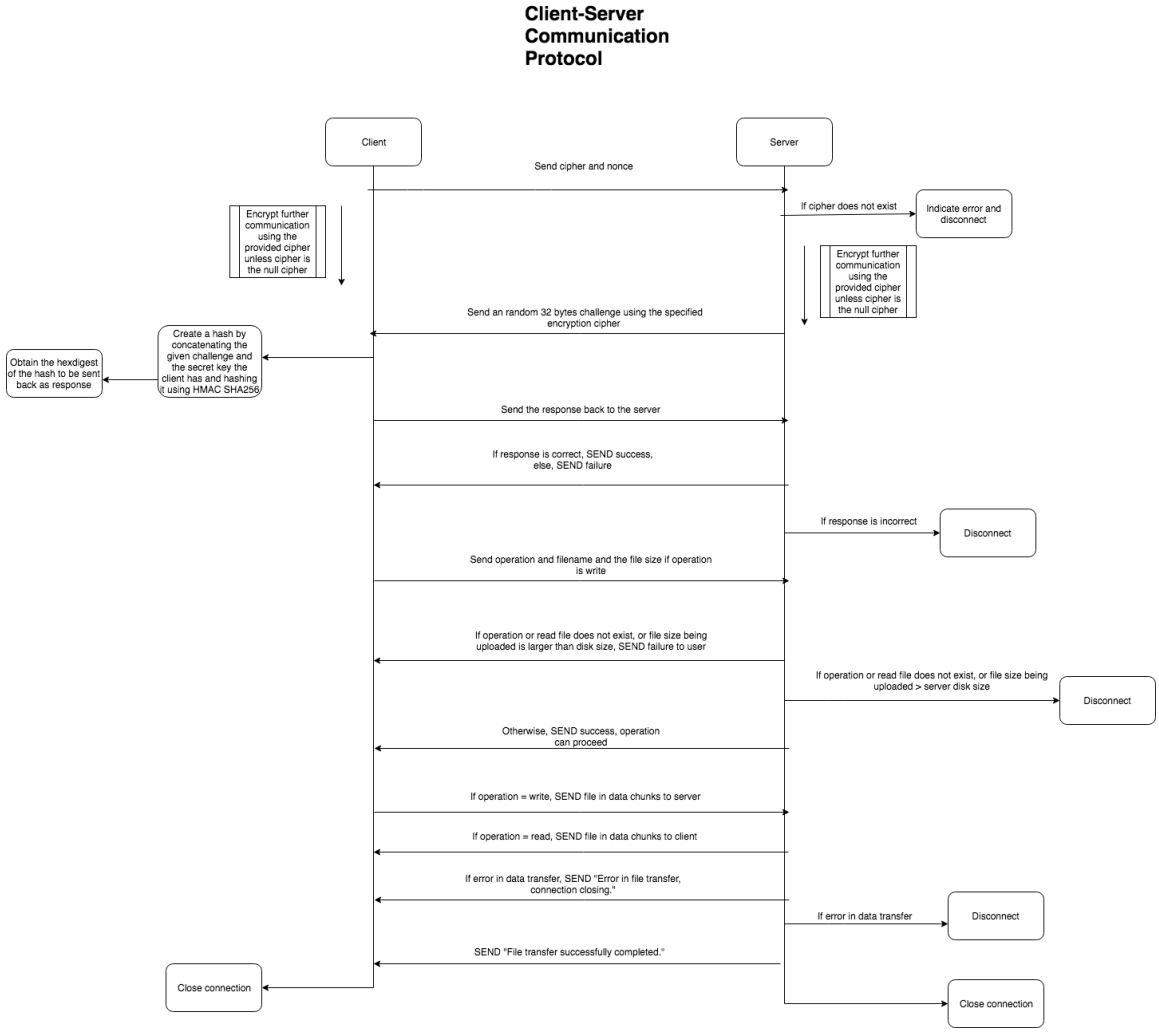


Figure 2. *Note: Connection diagram explaining our protocol procedure.*

As seen in Figure 2, our protocol starts of when the client is connected to the server and sends the cipher they intend to use including a random string generated nonce. The server then checks if such a cipher is supported and either responds with a success or indicates an error and disconnects the user.

If cipher is valid, all communication from that point on both sides is encrypted. The server then sends a random 32-byte string challenge to the client. The client then hashes the challenge with their secret key, creates a hex digest and sends it back as a response to the server. The server in return will create its own hash hex-digest using the same challenge and its own secret key. If the digests match, the keys are matching and the communication continues indicated by a success, otherwise the server indicates an error and disconnects from the client.

Given a successful authentication, the client sends an operation, a filename, and a file size if the operation they are requesting is a write. If the operation does not exist or if the operation is a read on a filename of a file that is not contained in the server, the server will send an error and disconnect from the client. If the operation is a write, the server then checks if the size of the file is less than the disk size it has. If the file is too large it indicates an error to the client and it disconnects. Otherwise the server sends a success acknowledgement to the client and the provided operation proceeds.

If the operation is a write, the client sends the file from standard input in data chunks of at max 1024 bytes at a time and the server takes each byte, writing it to a file with the specified filename by the client. If the operation is a read, the server opens the file with the given file name, reads lines of at most 1024 bytes, and sends the lines one chunk at a time to the client where the client writes it to standard output.

If there is any error in the data transfer, the client sends an error response to the client and disconnects from the client. Otherwise, if transfer is successful, it sends success to the client and both the client and server close the connection between each other. The server keeps listening for other connections.

Timing Results:

READ 1GB AES256: 2m 42.234s

WRITE 1GB AES256: 2m 36.720s

READ 1GB AES128: 2m 09.027

WRITE 1GB AES128: 2m 02.541

READ 1GB NULL: 2m 01.645

WRITE 1GB NULL: 1m 58.988

READ 1MB AES256: 0m 0.984

WRITE 1MB AES256: 0m 0.765

READ 1MB AES128: 0m 0.870

WRITE 1MB AES128: 0m 0.932

READ 1MB NULL: 0m 0.673

WRITE 1MB NULL: 0m 0.611

READ 1KB AES256 0m 0.441

WRITE 1KB AES256 0m 0.474

READ 1KB AES128 0m 0.412

WRITE 1KB AES128 0m 0.489

READ 1KB NULL 0m 0.351

WRITE 1KB NULL 0m 0.393

Explanation:

Median of results = 0.803 seconds

As you can see from the data given, it is usually the case that AES256 takes longer than AES128, which takes longer than non-encryption.

The times are similar for AES256 and AES128 due to the same 16 byte block size.

The larger the file, the longer the encryption takes