CPSC 526 - Assignment 3

Due date: March 10, 2017 @ 23:59.
Weight: 30% of your final grade.
Group work allowed, max. group size of 2.

Implement a network data-transfer system that uses symmetric encryption for communication. The system will consist of a client application and a server application. The client application will connect to the server application, and either send data to be stored on the server, or retrieve data already stored on the server. To make the client flexible, it will read/write data to/from standard output. The user of the system will have the option of having the communication between the client and server encrypted using AES. The purpose of the server application is to fulfill clients' requests. It will either receive data and write it to files, or read data from files and send them to clients.

Client

The client application will be used to:

- upload data to the server; or to
- download data from the server.

The client application will take 5 command line arguments

```
$ client command filename hostname:port cipher [key]
```

The arguments are:

- The **`command`** argument will determine if the client will be uploading or downloading data to/from the server. Valid values are **`write`** and **`read`**.
- The **`filename`** argument specifies the name of the file to be used by the server application.
- The **`hostname:port`** argument specifies the address of the server, and the port on which the server is listening.
- The **`cipher`** argument specifies which cipher is to be used for communication with the server. Valid values are **aes256**, **aes128** or **none**.
- The **`key`** parameter specifies the key to be used for encryption. It is not necessary to specify this key if cipher=none.

The client will attempt to connect to the server running on **`hostname`** on the given **`port`**. If successful, it will request the cipher to be used with the server. After that it will either try to upload data to the server, or download data from the server, depending on the value of the **`command`** argument. If the supplied secret **`key`** is invalid, the system should somehow detect this, and the client should report an error message on standard error.

Uploading

When **`command=write`**, the client will read data from standard input and send it encrypted to the server. The server will decrypt the data, and write the results to a file called **`filename`**. The client will keep reading the data from standard input until EOF. Once all data has been transferred to the server, and the server confirms the data has been received, the client can disconnect.

Please note that it is possible the data supplied to the client on standard input will be much bigger than the available memory or even the available disk space. The client should not try to read all this data into memory/disk, and then send it all to the server. Instead, the client should read a small block of the data and then immediately encrypt & send the block to the server. It should then repeat this process until all data from standard input has been processed.

Downloading

When **`command=read`**, the client will ask the server to send the contents of a file called **`filename`**. The server will send this data to the client encrypted. The client will decrypt the received data, and then write the results to standard output. The client will terminate once all data from the server is processed.

Server

The purpose of the server application is to listen for and accept connections from clients, and then to fulfill a request from a client. Each client connection can only make one request, which is either: (a) save data to a file, or (b) retrieve data from an existing file. The communication with the client will need to be encrypted using the cipher selected by the client.

The server application should take two command line arguments:

```
$ server port [key]
```

The **`port`** argument will specify the listening port. The **`key`** argument, if given, will be the secret key to be used by the symmetric ciphers. If the key is not specified, it will be randomly generated as a 32 character long string, composed of alphanumeric characters.

Once the server starts, it will listen for clients on the given port. When a client connects, it will request a read or write operation. Once the request is handled and the client is disconnected, the server will resume listening for another client. The server should run forever and it only needs to handle one client at a time.

Error handling

If the client requests a download of data from a non-existent file, the server should indicate to the client that such a file does not exist. This will be then reported by the client to the user.

If the client requests an upload of data and asks the server to save the data to a file which does not exist, the server will simply create the file. If the destination file already exists, the file should be overwritten. If there is any error while writing the file to the disk, eg. the client specifies a filename that is not writable, the server should report an error that the client can display to the user.

Logging

During it's operation, the server application should log client activity to standard output. This will be very useful during debugging. In particular the following events should be logged:

- when a new client connects, log the IP address and the encryption method selected;
- log the client command;
- log any errors;
- log 'done' when the server is done serving the client.

Each log entry should be prefixed with the current server time. Here is a possible output of a server:

```
$ ./server 9999
Listening on port 9999
Using secret key: Z3DmjUGaK4ZQjhEe45q4WnwuW5KDCQcP
19:05:33: new client: 136.159.55.221 crypto: NONE
19:05:33: command: write f1.txt
19:05:34: done
19:07:16: new client: 136.159.55.221 crypto: AES256 iv: 59HBcSYHHTUhMg7m
19:07:16: error: could not decrypt
19:07:16: done
```

Ciphers

You need to implement 3 different ciphers:

`cipher = none`	With this cipher *ciphertext* = *plaintext*. Neither the secret key nor the initialization vectors are used. This cipher will be useful for you to debug your protocol.
`cipher = aes128`	128 bit AES using CBC. This requires 128 bit keys, and 128 bit blocks.
`cipher = aes256`	256 bit AES using CBC. This requires 256 bit keys, and 128 bit blocks.

You should not try to implement the AES cipher yourself. Instead, you should use the *libcrypto* library, which is part of the OpenSSL project. A good description of how to use *libcrypto* for the purposes relevant to this assignment can be found here:

https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption

For block cipher mode of operation you need to use the CBC (cipher block chaining). This will be handled for you automatically if you use the right ciphers from the *libcrypto* library. For 128bit AES encryption, you should be using the **`EVP_aes_128_cbc()`** cipher, and for 256bit AES you should be using **`EVP_aes_256_cbc()`**.

If you are implementing the assignment in Python, you have many choices for cryptographic libraries. I suggest the *cryptography* library: https://cryptography.io.

Both *libcrypto* (for C/C++) and *cryptography* (for Python) should be available on the Linux workstations in the MS labs.

Initialization vectors

CBC needs an initialization vector (IV) in order to work. This initialization vector should be a nonce that is synchronized between the client and the server for each communication. The IV must be randomly generated by the client and then sent to the server in the clear (see Protocol below for more information).

Padding

The *libcrypto* library implements padding automatically for you. Just beware that your ciphertext may end up being larger than the original plaintext.

The protocol

You will need to design and implement a communication protocol to support all of the functionality described so far. With the exception of the very first message, the **entire** communication must be encrypted by the selected cipher. The first message is something that is sent by the client to the server to establish the cipher. This message should only contain two parts: (a) the cipher to be used and (b) the initialization vector for CBC. All subsequent communication after this first message must be encrypted by the selected cipher. Note: of course if the selected cipher=none, then the communication will end up being sent in the clear.

When you design the protocol, design it so that an invalid secret key will be detectable by both the server and the client. When this happens, appropriate message should be shown to the user by the client, and appropriate message should be logged by the server.

Key “stretching”

The keys accepted by the client program might be too short for *libcrypto* to accept them. To increase the size of the key one would usually use a proper key-stretching algorithm such as Argon2. However, to keep things simple, for this assignment you can stretch the key simply by appending the key to itself enough times until you get a key of desired size.

Examples

The following examples describe the system in more detail.

Example: starting the server with a given key

To start the server, we need to tell the server the port and optionally the secret key. For example, here we tell the server to listen on port **9999** and to use the key **secret123** for encryption:

```
$ ./server 9999 secret123
Listening on port 9999
Using secret key: secret123
```

The server then starts and echoes back the port and the secret key.

Example: starting the server without a key

We can start the server without specifying the secret key. In this case the server generates a random key:

```
$ ./server 9999
Listening on port 9999
Using secret key: Z3DmjUGaK4ZQjhEe45q4WnwuW5KDCQcP
```

Example: uploading a file to the server – using no encryption

The client reads data from standard input and sends it to the server. We can use this to upload a file to the server:

```
$ cat file1.txt | ./client write f1.txt localhost:9999 none
OK
```

The example above connects to a server running on localhost on port 9999, and the server saves the data received to a file called **f1.txt**. No encryption is used. The filename **f1.txt** can be used later to retrieve the saved data.

Example: uploading a file to the server – using encryption

Below are two examples that would accomplish the same thing as above, but this time using 128-bit and 256-bit AES encryption for communication, respectively, with the *secret key=secret123*:

```
$ cat file1.txt | ./client write f1.txt localhost:9999 aes128 secret123
$ ./client write f1.txt localhost:9999 aes256 secret123 < file1.txt
OK
```

Example: failing to upload a file to the server because of bad key

In the next example we try to upload a file with encryption enabled, but we specify the wrong secret key. The system will indicate the operation failed by printing a suitable message to standard error:

```
$ cat file1.txt | ./client write f1.txt localhost:9999 aes128 badkey123
Error: Wrong key.
```

Example: downloading a file using no encryption

In this example we try to extract some saved data from the server. Since the client writes the received data to standard output, we redirect it to a file:

```
$ ./client read f1.txt localhost:9999 none > f1.txt
OK
```

Example: downloading a file using encryption

Same as in previous example, but using 256-bit AES encryption:

```
$ ./client read f1.txt localhost:9999 aes128 secret123 > f1.txt
OK
```

Example: failing to download a file because file does not exist on server

If the client tries to download data from a file that does not exist on the server, the client displays an error message on standard error:

```
$ ./client read xyz.txt localhost:9999 aes128 secret123 > f1.txt
Error: File xyz.txt does not exist.
```

Example: failing to download a file because of wrong secret key

Here we try to download some data, but supply the wrong secret key. The client reports an error:

```
$ ./client read f1.txt localhost:9999 aes128 badkey > f1.txt
Error: Wrong key.
```

Testing

You need to test your implementation for correctness and for speed.

To test your system, for each of the three ciphers you should try to upload and then download at least 3 different binary files of different sizes, and compare their checksums before and after. The file sizes you should test are 1KB, 1MB and 1GB. You can use **`dd`** and **`/dev/urandom`** to create these files, eg.:

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

To create checksums for your files, you can use sha256 like this:

```
$ sha256sum 1*.bin
5c258626a2a49167d3d645062dec214065065ace3599ca25acbfd67819d99e24  1GB.bin
645310e9779b69835c5cf81b3c5461813cdbf761889106ecf61221738a42a63e  1KB.bin
a081ddc175b3404304436aa87aec115a1e754787643c5371f177f3baf139bf00  1MB.bin
```

Here is an example where you test whether your system works with a 1 GB file with AES-256:

```
$ sha256sum 1GB.bin
5c258626a2a49167d3d645062dec214065065ace3599ca25acbfd67819d99e24  1GB.bin
$ ./client write test localhost:9999 aes256 secret123 < 1GB.bin
OK
$ ./client read test localhost:9999 aes256 secret123 | sha256
OK
5c258626a2a49167d3d645062dec214065065ace3599ca25acbfd67819d99e24
```

If the checksums match, your system is likely working correctly.

You should use your assignment 2 program to verify that your communication is encrypted when encryption is requested. You should also verify that your communication is not encrypted when cipher=none is used. You should also verify that sending the same file twice is encrypted differently. This should be true if you use different initialization vectors for each communication session.

Similar tests will be used by your TAs to verify correct operation of your implementation. Be prepared to show any of these tests during demos.

Timing

Part of this assignment involves testing the speed of file uploads and downloads with and without encryption. I am interested to hear your opinions on the following: Are there any significant differences in how long it takes to transfer data back and forth depending on which cipher is used?

To answer this question, you need to perform some timing tests. To make these timings meaningful, you need to run the client and the server on two different machines. To time how long it takes to upload/download a file, you can use the UNIX **`time`** command and record the 'real time'. For example, to test how long it would take to send a 1KB file to the server using 256-bit AES, you can execute this command:

```
$ time ./client write 1KB.bin localhost:9999 aes256 secret123 < 1KB.bin
real    0m0.016s
user    0m0.000s
sys     0m0.006s
```

You need to run the timing tests for each of the 3 file sizes, for each of the 3 supported ciphers, and for upload and download. Altogether there should be 3x3x2 = 18 different tests. Run each of the 18 tests 10 times, and record the results. Then eliminate the two slowest and two fastest runs, and calculate the average of the remaining timings. Try to plot the data, and write a 1 page summary and discussion of the results.

Additional notes

- You may implement your program in Python, C or C++.
- You may not call external programs.
- You do not need to handle multiple clients simultaneously. Handling one client at a time is sufficient.
- You are required to demo your assignments individually. The time for your demo will be arranged by your TA.
- You are allowed to work on this assignment with another student (max. group size is 2 students). But beware that during the demo you will be asked to demonstrate your familiarity with all of the code. So if you do decide to group up, both of you should understand the code 100%.

Where to start

You are free to start working on the assignment any way you like. But in case you are curious, here is how I would go about it:

1. Start by designing the protocol. Make a diagram – it might help.
2. Implement the system only using the 'none' cipher.
   a) Implement the server application and test it with *netcat*.
   b) Implement the client and make it work with your server.
   c) Use your assignment 2 to debug your protocol.
3. Test that your implementation is correct.
4. Add AES128 with a fixed IV = 0, and test.
5. Add dynamically generated IVs, and test.
6. Add AES256, and test one last time.

Submission

You need to submit your source code and a **`readme.pdf`** file to D2L. Please use ZIP or TAR archives. If you decide to work in a group, each group member needs to submit the assignment. The **`readme.pdf`** file must include:

- your name, ID and tutorial section;
- name of your group partner if applicable;
- a section that describes how to compile/run your code;
- a section that shows at least one test for correctness;
- a section describing your communication protocol, include a diagram if you made one;

- a section describing your timing tests, including tabulated results and a discussion of those results.

You must submit the above to D2L to receive any marks for this assignment.

Marking

Source code formatting / documentation	5 marks
Readme.pdf – how to compile and/or run your code	5 marks
Readme.pdf – documented testing (test AES256 upload/download with checksums before and after)	5 marks
Readme.pdf – communication protocol description	5 marks
Readme.pdf – timing report	5 marks
Basic file upload/download working with cipher = none	20 marks
Working cipher = aes128 and cipher = aes256	15 marks
Error reporting on wrong shared key	5 marks
Error reporting on reading non-existent file	5 marks
Encryption working with auto-generated key	10 marks
Using random IVs for each session	10 marks
Server logging	5 marks
Ability to handle very large files that don't fit in memory/disk.	5 marks

General information about all assignments:

1. Late assignments or components of assignments will not be accepted for marking without approval for an extension beforehand. What you have submitted in D2L as of the due date is what will be marked.
2. **Extensions** may be granted for reasonable cases, but only by the course instructor, and only with the receipt of the appropriate documentation (e.g., a doctor's note). Typical examples of reasonable cases for an extension include: illness or a death in the family. Cases where extensions will not be granted include situations that are typical of student life, such as having multiple due dates, work commitments, etc. Forgetting to hand in your assignment on time is not a valid reason for getting an extension.
3. After you submit your work to D2L, make sure that you check the content of your submission. It's your responsibility to do this, so make sure that your submit your assignment with enough time before it is due so that you can double-check your upload, and possibly re-upload the assignment.
4. All assignments should include contact information, including full name, student ID and tutorial section, at the very top of each file submitted.
5. Although group work is allowed, you are not allowed to copy the work of others. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
6. You can and should submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It's better to submit incomplete work for a chance of getting partial marks, than not to submit anything.
7. Only one file can be submitted per assignment. If you need to submit multiple files, you can put them into a single container. Supported container types are TAR or gzipped TAR. No other formats will be accepted.
8. Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have question after you have talked to your TA then you can contact your instructor.