CPSC 526 - Assignment 4

Due date: November 12, 2017 @ 23:59.
Weight: 16% of your final grade.
Group work allowed, max. group size of 2.

For this assignment you will implement a client/server system that will allow a user to upload and download data to/from a remote computer over a network. You need to implement both the client and server portion of this system. To this end, you will need to design and implement a suitable protocol. The protocol will need to support features such as: symmetric encryption, negotiation of ciphers/session-keys/initialization-vectors, client authentication, error codes and the ability to handle upload of data with unpredictable sizes.

Client

The client will connect to the server application, and either upload data to the server, or download data already stored on the server. To make the client flexible, it will read/write the data to/from standard output. Error messages will be sent to standard error. The client will protect its communication with the server using a symmetric cipher. The client will accept 5 command line arguments:

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

Where:

- 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 hostname can be specified as a domain name or an IPv4 address. The port will be an integer in range 0-65535.
- The `cipher` argument specifies which cipher is to be used for encrypting the communication with the server. Valid values are `aes256`, `aes128` and `null`.
- The `key` parameter specifies a secret key that must match the server's secret key. This key will be also used to derive both the session keys and the initialization vectors.

Connecting to server

Once started, the client will connect to the server running on `hostname` on the given `port`. The client will then inform the server that it intends to use `cipher` for encryption. After this, all communication between the server and client will be encrypted using this cipher. The next step is for client to authenticate itself to the server, by proving its knowledge of the `key`. If the client fails to prove it has the right key, an error message should be displayed and the client will terminate. After successful authentication, the client will either try to upload data to the server, or download data from the server, depending on the value of the `command` argument. Once the upload or download is successfully finished, the client will indicate success and exit.

Uploading

When `command=write`, the client will read data from the standard input and send it (encrypted) to the server. The server will receive (and decrypt) the data, and write the results to a file called `filename`. The client will read 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 and written to a file, the client will disconnect. If the destination file already exists, the server should overwrite it.

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 must not try to read and buffer all this data into memory/disk, and then send it to the server later. Instead, the client must 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. Also, this means that you will not be able to determine ahead of time how much data your client will be uploading. You will need to design your protocol that reflects this.

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.

Error handling

The client should write the final status of the requested operation to standard error. When successful (either uploading or downloading a file), the client will simply write “OK” to stderr. If there is an error of any kind, the client should display an appropriate error message on stderr. Below are some examples of status messages:

Message	**Meaning**
OK	upload or download was successful
Error: wrong command line arguments	bad command line arguments
Error: wrong key	client could not connect to server because of key mismatch
Error: file could not be read by server	server could not read the requested file (download)
Error: file could not be written by server	server could not write the data to the requested file (upload)

Server

The server application should take two command line arguments:

```
$ server port key
```

The `port` argument will specify the listening port. The `key` argument will be the secret key. When the server starts, it will listen for clients on the given port. When a client connects, the server will authenticate the client by making it prove it knows the secret key. If the authentication is successful, the client will be allowed to request a single read or a single write operation. The server will then perform the operation, and communicate to the client any applicable errors or success.

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. The communication with the client must be encrypted using the cipher selected by the client.

Logging

During its operation, the server application should log client activity to standard output. This will be useful during debugging. The following events must be logged by the server:

- when a new client connects, display the IP address of the client, requested cipher and nonce
- the generated initialization vector and session-key;
- command (write or read) and the filename;
- final status: either an error or success

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

```
$ ./server 9999 mySecret
 Listening on port 9999
 Using secret key: mySecret
 19:05:33: new connection from 136.159.55.221 cipher=aes256
 19:05:33: nonce=P0BHL2WSZ0KJVA56
 19:05:33: IV=d923506ed7aeeabffcf787b73e9bad117a39b39c570ec9625556e4897868bcc4
 19:05:33: SK=19a279489f92ef998abb71cb63b21cf05b010633bb8700b7bff4b2894ebd2e0b
 19:05:33: command:write, filename:f1.txt
 19:05:34: status: success
 19:07:16: new connection from 136.159.55.221 cipher=null
 19:05:33: nonce=022GE2UKB54LB5UL
 19:05:33: IV=0f0a77c6ad1f4ca31e32e26b03b9de80683167954432eba8857680592739dea7
 19:05:33: SK=8ea62560aae8dc9d71ac9c0271aa70effc1c6ffe767f2ffa8159b45993276207
 19:07:16: status: error - bad key
```

During development, you may decide to include additional debugging information in the server log.

Ciphers

You need to implement support for 3 different ciphers:

`cipher = null`	With this cipher *ciphertext* = *plaintext*. Useful for debugging.
`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. Same goes for the SHA256 functionality. Instead, you should use a library that already implements them. If you decide to writer C/C++ code, you may want to consider using 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
https://wiki.openssl.org/index.php/EVP_Message_Digests

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

To use AES in CBC mode, you will need to generate matching initialization vectors (IV) and a session-keys on both sides (client and server). You will compute these using SHA256 cryptographic hash function:

IV = SHA256(key | nonce | "IV")

session-key = SHA256(key | nonce | "SK")

where `key` is the secret key specified on the command line argument to both the server and the client, and '|' represents string concatenation. The `nonce` is a random string of 16 alphanumeric characters. It must be generated by the client and sent to the server together with the cipher (see Protocol section below for more information). This means that the IV and the session keys will be different for every communication session.

Padding

The *libcrypto* library implements padding automatically for you. Just beware that your ciphertext may end up being larger than the original plaintext. If you decide to use another library, consult the library's documentation on padding options.

The protocol

You will need to design and implement a communication protocol to support all of the functionality described above. With the exception of the very first message, the **entire** communication must be encrypted by the selected cipher.

The first message

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) a nonce.

client → server: cipher, nonce

Only this first message is sent unencrypted. All subsequent communication after this first message must be encrypted by the selected cipher. Naturally, if the requested encryption is a null cipher, then all communication will end up being sent in the clear.

Authentication

The next few messages of the protocol should implement some form of authentication, where the client must prove to the server the knowledge of the key. Even though these messages must be encrypted by the selected cipher, the cipher could be null. This means that these message might travel over the wire in the clear. Your authentication scheme should take that into consideration – you must devise a scheme that will verify the secret key without actually revealing it, even if a null cipher is selected. I suggest some form of challenge-response scheme.

> *server → client: random challenge*
>
> *client → server: compute and send back a reply that can only be computed if secret key is known*
>
> *server → client: verify the reply, send success/failure message to client*

If the server is unable to verify that the client knows the secret key, it should disconnect the client.

Request

The next message should come from the client, indicating the type of request. The server should try to determine whether the request can be satisfied, and indicate so to the client.

> *client → server: operation, filename*
>
> *server → client: response indicating whether operation can proceed*

Data exchange

The next part of the protocol consists of messages that exchange data between the client and the server. If client is uploading a file, the client will be sending the messages. If client is downloading a file, the server should be sending the messages.

> *client → server: data chunk*
>
> *server → client: data chunk*

In case of any errors, the server should indicate so to the client and then disconnect.

> *server → client: optional error message*

Final success

You may want to implement a last message sent from the server to the client, indicating the operation was a success.

> *server → client: final success*

At this point the server and the client should both disconnect.

Examples

The following examples describe how your system should work in more detail.

Example: starting the server

Start the server on port `9999` and set the secret key to `secret123`:

```
$ ./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: uploading a file to the server – using NULL encryption

Since the client reads data from standard input, we can use this to upload a file to the server using a shell pipe:

```
$ cat file1.txt | ./client write f1.txt csx1.cs.ucalgary.ca:9999 null secret123
OK
```

In the example above the client connects to a server running on csx1 on port 9999, and the server saves the data received to a file called `f1.txt`. No encryption is used (null cipher). 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 csx1:9999 aes128 secret123
OK
$ ./client write f1.txt csx1:9999 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 client 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 136.159.5.7:9999 null secret123 > 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

To test your system, for each of the three ciphers you should try to upload and then download at many different files of different sizes, and compare their checksums before and after. I recommend you test files with 0 bytes, 1 byte, 1027 bytes, 2^16+31 bytes and 1GB. You can use `dd` and `/dev/urandom` to create these, eg.

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

In case you do not have enough disk quota to store large files, I prepared some test files in my directory:

`/home/profs/pfederl/Teaching/CPSC526/A4`

The directory also contains two python scripts (`tgen.py` and `bgen.py`), which can generate pseudorandom data of any size. The `tgen.py` script generates text data, and `bgen.py` generates binary data. Both take two arguments: data size in bytes and a seed for the pseudorandom number generator.

To compute checksums for your data, you can use sha256 like this:

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

Or when using the bgen/tgen scripts:

```
$ ./bgen.py 1024 3 | sha256sum
59031a5fd8e9022fa3bf08413f574173dfdd0b1c5d83aae71f9cb948c186c414  -
```

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 3 proxy program to verify that your communication is encrypted when encryption is requested. You should also verify that your communication is not encrypted when cipher=null is used. You should also verify that sending the same file twice is encrypted differently.

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 to the server using 256-bit AES, you can execute this command:

```
$ time ./bgen.py 1024 6 | ./client write 1KB.bin localhost:9999 aes256 secret123
 real    0m0.016s
 user    0m0.000s
 sys     0m0.006s
```

You need to run the timing tests for 3 data sizes (1KB, 1MB, 1GB), for each of the 3 supported ciphers, and for upload and download. Altogether there should be 3*3*2 = 18 different tests. Run each of the 18 tests 10 times, record the results and calculate the median of the timings. Write a one page summary and discussion of the results. You may want to include graphs 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. Demo time 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, although I would recommend these steps:

1. Design the protocol. Make a diagram, it might help. Consider designing a human readable protocol (i.e. a text protocol). It will be less efficient than a binary protocol, but much easier to debug.
2. Implement the server program that understands your protocol. Only implement support for the null cipher, and no authentication. You should be able to test your server with netcat.
3. Implement the client and make it work with your server. You can use your proxy program from assignment 3 to debug your protocol.
4. Make sure your implementation is correct. Manually verify checksums on transferred files. Make sure your error messages are working.
5. Add authentication. Test again.
6. Add support for AES128 with fixed IV = 0 and a fixed session-key = 0. Ignore the nonce. Test.
7. Add support for AES256. Test.
8. Generate IVs from nonce. Test.
9. Generate session-keys from nonce. Test.

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;
  - upload then download a 1MB file using AES256, show that file digests are the same
- 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;
  - include graphs if you made them.

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

Marking

Source code formatting / documentation	required / 20 mark penalty
Readme – how to compile and/or run your code	required / 50 mark penalty
Readme – testing (AES256 upload/download/checksums/1MB file)	required / 20 mark penalty
Readme – communication protocol description	10 marks
Readme – timing report & conclusions	10 marks
File upload working with NULL cipher	10 marks
File download working with NULL cipher	10 marks

Authentication over null cipher without revealing secret key	5 marks
Detecting and reporting wrong shared key	10 marks
Detecting and reporting when reading non-existent/unreadable file	5 marks
Detecting and reporting when writing to non-writable file	5 marks
Working AES128 cipher	10 marks
Working AES256 cipher	10 marks
Using random IVs and session-keys for each session	10 marks
Server logging	5 marks
Ability to handle very large files that don’t fit in memory/disk.	required / 20 mark penalty

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.