

# CSCI 3403: Project 2

## Emulating SSL and Password Verification

Matt Niemiec and Abigail Fernandes

Due: 3/2/20 at 11:59PM

### Background

While no knowledge of networking is required for this project, you are encouraged to know the basics of sockets. Feel free to refer to the following link and other similar articles to learn about the details that are already implemented [here](#). You are also given code related to Python file I/O, though you should have experience with this from your introductory coursework. Once again, these portions are completely implemented for you.

### Goal

The purpose of this assignment is to give you a deeper insight into how almost everything secure on the internet works, including HTTPS and SSH. So, the purpose is to create a completely encrypted interaction between a client and server that is completely confidential and has complete integrity.

To begin with you're provided a basic client/server program, as well as some placeholder functions to give you an idea of what you'll need to do to implement your program. Please note that the complete solution uses exactly these functions, but you should feel free to add or remove functions if you feel it is necessary for your implementation. Parts of the main function are also provided, but, particularly on the client side, you may need to implement some things yourself. You will not need to change any of the networking functionality, unless your

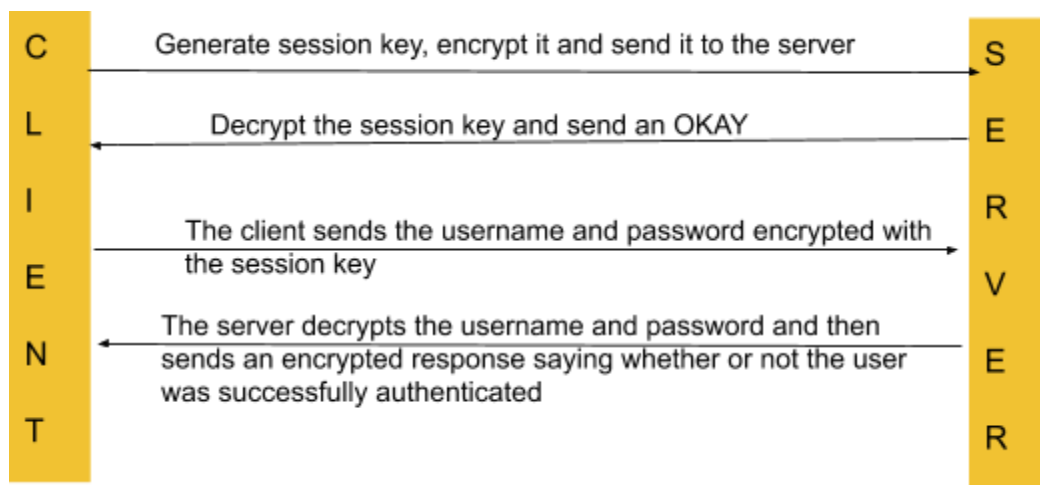


Figure 1

implementation specifically requires you to do so. You are also provided an **add\_user.py**. This should be completed so that it takes in a username and password and stores the username, salt, and hash.

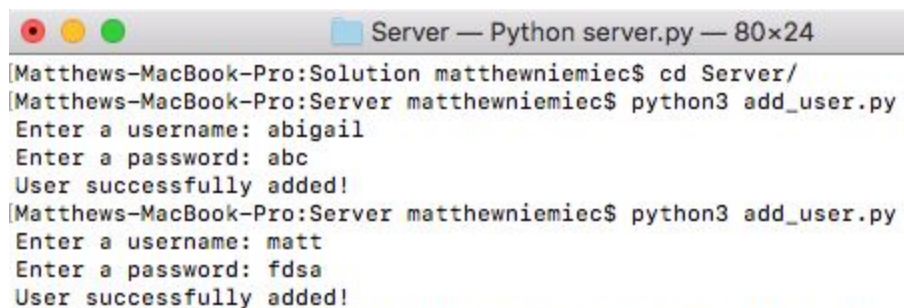
You may refer to Figure 1 for a visual representation of what your code should be doing. First, the server will listen for incoming connections. Next, the client will generate a random AES key, encrypt it with the server's public key, and send that to the server. From this point onward all communication will be done over AES with this key. Next the server will respond with an "okay" - this may be encrypted, but it isn't necessary, since SSL doesn't protect against availability attacks, and this isn't sensitive information. Next the client will send the username and password to the server. The server will take this and compare it to the stored hash file to check if the username and password are correct. The server will send back an encrypted response accordingly.

Of course, in a real-life scenario, this would just be the start of the communication. However, for the purposes of this assignment, after this both the client and the server will tear down the connection. The client process will close and the server process will resume looking for connections. Note that this server, unless you choose to change it, is not multi-threaded, so it can only handle one connection at a time. This is intended to be a rudimentary example.

## Setup

You will be expected to use Python3 for this project. You may also have to download appropriate related libraries. To get started on the project, please refer to the following steps:

1. Create a separate folder for the Client and Server and place the files given to you as well as the files you generate accordingly.
2. You now need to generate RSA keys. You may look into **ssh-keygen** as one way of doing so. You will need the public and private keys to encrypt/decrypt the session key.
3. Create users to be added to the passfile.txt file. You will need this to get your client-server running. You will do this by first running the **add\_user.py**. You can see below what you should expect to see.



```
Server — Python server.py — 80x24
[Matthews-MacBook-Pro:Solution matthewniemiec$ cd Server/
[Matthews-MacBook-Pro:Server matthewniemiec$ python3 add_user.py
Enter a username: abigail
Enter a password: abc
User successfully added!
[Matthews-MacBook-Pro:Server matthewniemiec$ python3 add_user.py
Enter a username: matt
Enter a password: fdsa
User successfully added!]
```

4. Now open the server.py file and client.py. You will find a list of TODOs that you are required to implement. The client - server interaction is depicted in Figure 1.

## Results (50 points)

The screenshots below demonstrate the type of end functionality you should expect to see when running your program. When you're verifying encryption, sniff the traffic with Wireshark to verify that your traffic is encrypted. **Note that they do not explicitly demonstrate every step.** Also note that, while printed in plaintext, everything that is sent over the network is ciphertext (except for the "okay"). When printed in Python, you can expect your ciphertext to look something like this: `b'\xe4n\x91\x06\xa4(\x89\x99\xbc\xf8e\xd3\xd7\xf7uA'`, though the encrypted session key will be much longer.

```
Matthews-MacBook-Pro:Server matthewniemiec$ python3 server.py
starting up on localhost port 10001
waiting for a connection
connection from ('127.0.0.1', 49478)
matt fdsa
waiting for a connection
connection from ('127.0.0.1', 49480)
abigail cba
waiting for a connection
connection from ('127.0.0.1', 49485)
userdne pass
waiting for a connection
█
```

```
Client — -bash — 80x24
Matthews-MacBook-Pro:Solution matthewniemiec$ cd Client/
Matthews-MacBook-Pro:Client matthewniemiec$ python3 client.py
What's your username? matt
What's your password? fdsa
connecting to localhost port 10001
User successfully authenticated!
closing socket
Matthews-MacBook-Pro:Client matthewniemiec$ python3 client.py
What's your username? abigail
What's your password? cba
connecting to localhost port 10001
Password or username incorrect
closing socket
Matthews-MacBook-Pro:Client matthewniemiec$ python3 client.py
What's your username? userdne
What's your password? pass
connecting to localhost port 10001
Password or username incorrect
closing socket
Matthews-MacBook-Pro:Client matthewniemiec$ █
```

## Report (50 points)

Please write a report detailing all your work. Use screenshots as necessary to support what you're saying. In general, please make sure the report is readable (i.e. reads well) and demonstrates understanding. Basically, you should demonstrate that you made educated security decisions, rather than guessing and going with the first thing that didn't throw an error. You don't have to answer the questions in order (that would be rather boring!), but you should include the following:

- 1) How did you hash the passwords in the file? Include information such as the hashing algorithm, the library you used, how you appended the password with the salt before hashing, and other information we've discussed related to password storage. Please provide brief explanations as to why you chose to use what you did. Convince the reader that your password storage algorithm is secure.
- 2) Describe your approach to public/private key use. How did you generate your public/private key pair? What library did you use to encrypt messages? In which folder did you store your keys? Please provide any details you think would be relevant.
- 3) How did you manage symmetric encryption? What encryption mode did you use, and why did you think that was best? What tradeoffs did you make for that encryption mode?
- 4) Provide a *brief* explanation of why your program would be secure from eavesdroppers if you were to run it on a publicly visible network, from start to finish. Write your program like you're protecting yourself from Comcast or the NSA!
- 5) Is your program secure from replay attacks? If not, why not? Use protocols that we discussed in class to show that it isn't vulnerable. If it is vulnerable, what could you do to prevent it? Could you perform a replay attack against yourself?
- 6) What else did you learn from the project? Did you have to do some research? Collectively, or individually, give some of the key takeaways you had when doing this.

## Extra Credit (15 points)

Take your project even further! Don't just add authentication, but add MAC authorization. In your text file you can store the clearance level of the user as well as the categories that the user is cleared for. Then, keep track of some document names and permissions in a separate file. When a user authenticates themselves, they can request to read and/or write to a file. If their permissions are sufficient, then they'll get back a message telling them their permission was granted. Otherwise, they'll get back an error. Exactly how you implement this is up to you, but please ensure that it's simple enough to pick up and use without any knowledge of MAC (and we can verify its correctness!)

## Submission

Upload a zipped folder of your entire project. It should contain the following:

1. Folders for client and server with the required files
2. RSA keys you generated
3. Passfile.txt
4. The Wireshark capture verifying that your connection cannot be sniffed
5. All the other files you may add as part of your implementation
6. The report detailing all your decisions