Assignment 5 (100 points)

Anirudh Sivaraman

2021/11/20

The goal of this assignment is to familiarize yourself with some concepts in network security. This assignment mostly involves writing small Python programs for each question. For this assignment, you will have to refer to the Python API documentation for the secure sockets layer to help you write the programs: `https://docs.python.org/3/library/ssl.html`. This assignment is intended to give you a taste of networking in the real world, e.g., figuring out how to use a network API and implementing a new network protocol (the Diffie-Hellman key exchange). We use the terms SSL and TLS interchangeably in this assignment.

This assignment also uses Mypy (`http://mypy-lang.org/`). Mypy type hints give you an idea about the expected input and return types of each function signature. Mypy is entirely optional for this assignment, but can help you validate the correctness of your implementation. You can install Mypy using the following command:

```
pip3 install --upgrade mypy
```

You can run mypy on your code with this command:

```
python3 -m mypy [FILE_TO_CHECK].py
```

where `FILE_TO_CHECK` represents the Python file you want to check.

1 Written Questions (50 points)

Refer to Gradescope for the written questions. Some of them may depend on your implementation of the clients and servers in this coding assignment.

2 SSL clients (20 points)

For this question, you need to implement an SSL client to fetch data from an already running HTTPS server (HTTP over TLS/SSL). The server's domain name is www.example.com. For implementing this client, you will have to open a TCP connection to www.example.com, pick the correct transport-layer port, then create a TLS connection on top of this, then issue a HTTP request on this TLS connection. The HTTP request needs to be in the right format to work. Please refer to `https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html` Section 5.1.2, which specifies the correct format. Hint: If you are having trouble with getting the message right, check out the netcat Wikipedia article and its examples.

Your client must read up to 1024 bytes of response from the server. If you have written your program correctly, you should have output that looks like the output below. Our output is formatted to look neat using the Python pprint module, but if your output shows similar content, that's sufficient. If it helps, the Python API docs for SSL (`https://docs.python.org/3/library/ssl.html`) has example Python program snippets for an SSL client that you can reuse for this assignment.

```
['HTTP/1.1 200 OK',
'Accept-Ranges: bytes',
'Age: 436187',
'Cache-Control: max-age=604800',
'Content-Type: text/html; charset=UTF-8',
'Date: Sun, 07 Nov 2021 22:57:11 GMT',
'Etag: "3147526947"',
'Expires: Sun, 14 Nov 2021 22:57:11 GMT',
'Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT',
'Server: ECS (nyb/1D10)',
'Vary: Accept-Encoding',
'X-Cache: HIT',
'Content-Length: 1256',
'',
'']
```

Try running the following query with your client:

```
python3 ssl_web_client.py -w www.w3.org -d /Protocols/rfc2616/rfc2616-sec5.html
```

Then increase your receive buffer from 1024 bytes to 4096 bytes. Does your output change? Why or why not?

3 SSL server (10 points)

We'll now create an SSL server. Creating a server requires creating a certificate, which then needs to be signed by a certificate authority. While considerable progress has been made in automating this process and reducing manual intervention (e.g., `https://letsencrypt.org`), this process still requires you to demonstrate ownership of a domain before getting a certificate. Hence, to simplify things for this assignment, we'll use self-signed certificates, where you yourself will certify that you are who you say you are. This is obviously not secure and not recommended for practical use. To create a self-signed certificate, you can use the instructions provided here: `https://docs.python.org/3/library/ssl.html#certificates`.

Once you have created a self-signed certificate, which consists of both a certificate as well as a private key, use the SSL API reference to create a server SSL socket using that self-signed certificate. To do so, you should bind the server's socket to localhost on a port of your choice. Then connect to this server's socket using a client that creates an SSL client socket.

Because SSL enforces good security practices by default, this won't work out of the box because the client will reject the server's self-signed certificate as invalid. To override this setting, use the following Python snippet:

```
context.check_hostname=False
context.verify_mode=ssl.CERT_NONE
```

At the end of this process, to earn points for this assignment, the client should be able to send "This message is encrypted with SSL." to the server on an encrypted TLS connection and the server should be able to receive and print it out.

4 Diffie-Hellman key exchange (20 points)

Recall that asymmetric encryption and decryption (i.e., with private and public keys) is slow, while symmetric encryption and decryption is much faster. However, for symmetric encryption and decryption, we need a key that is shared between the 2 communicating parties. One way to establish this key is to create the shared key at either of the 2 parties and use asymmetric encryption and decryption to transfer it securely to the other end. However, if the asymmetric private keys are stolen in the future, then the shared key created for past communication sessions between the two parties becomes vulnerable. This is because if the adversary who stole the private keys happened to passively record previous sessions, the adversary can then use these private keys to decrypt the beginning of these sessions to determine the shared key. With the shared key in hand, the adversary can then decrypt the rest of the session, which is encrypted using this shared key.

A solution to this is the Diffie-Hellman key exchange. This is a simple, but very effective algorithm to create a shared key for 2 parties without ever transmitting this shared key between the 2 parties (either as clear text or encrypted text). Diffie-Hellman key exchange relies on 2 parties exchanging some amount of data, but never the shared key. This data exchange is sufficient for each of the 2 parties to then independently compute the shared key.

Your goal for this question is to implement Diffie-Hellman key exchange. The Wikipedia page: `https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange#Cryptographic_explanation` is a good reference for implementing it. Use a communication mechanism of your choice. The only important bit is that your client and server are able to exchange messages using UDP/TCP sockets.

First, execute the server:

```
python3 diffie_hellman_server.py
```

Then, execute the client:

```
python3 diffie_hellman_client.py
```

After both commands are executed, the shared secret computed by the server and client should be printed out on their respective terminals. Your output should look similar to this exchange:

"Client" (A)

```
Base proposal successful.
Base int is 90
Y is 5
Int received from peer is 16
Shared secret is 6
```

"Server" (B)

```
Base int is 90
Y is 4
Int received from peer is 14
Shared secret is 6
```

A should propose a base integer at random, which it will send to *B*. You should also randomize the choice of *A* and *B* by picking *Y* both uniformly from a large range of natural numbers. For this assignment alone, you can pick from the range 1 to 100 to simplify computation of exponents like g^a. In practice, an approach called modular exponentiation is used when the exponents are very large numbers.

The server vs. client distinction is completely arbitrary when using the Diffie-Hellman key exchange. We are using the term server in our implementation to refer to the entity that listens for data from the other entity (the client) before sending data to the other entity. This is so that both entities don't wait around indefinitely for the other to send data first.