

Module 5

Computer Network Security /Part 2

Prepared by Dr. Manal Mizher

Lecture 10

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Network programming

Course ID: CYB615



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

1. Review : HTTP protocol

- What is HTTP?
- HTTP Methods
- Downloading data from an HTTP server (*GET*)
- Requests exceptions (`HTTPError`)
- GET & POST
- Serving HTTP requests from your machine
- Directory Binding for HTTP custom server
- Simple HTTP Web Server and Client in Python
- urllib3 Library (HTTP test server , Simple HTTP_server using requests and urllib3)

2. SSL/TLS & HTTPS

- SSL/TLS definition and properties
- How do SSL/TLS work?
- TLS handshake vs TCP handshake
- Public key & Private key in SSL/TLS
- OpenSSL tool (keys and certification files by OpenSSL, OpenSSL & Convert .pem , .crt , .key)
- .pem from cmd and from Python
- SSLSocket Class In Python (Connection by SSLSocket, SSL/TLS - client / server compatibility,SSLSocket with HTTP server, SSLSocket with loopback client-server)

HTTP

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Network programming

Course ID: CYB615



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

- The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.
- There are many web server software that support HTTP protocol such as *apache*, *nginx*, and *lighttpd*.
- HTTP is based on requests and responses – the client makes requests and servers send responses.
- HTTP : client (access resources) → server (makes services and resources available).
- An URL identifies a resource on the Internet.

- **HTTP methods** such as GET and POST, determine which action you're trying to perform when making an HTTP request.
- The GET, *one of the most common HTTP methods*, method indicates that you're trying to get or retrieve data from a specified resource. To make a GET request, invoke **requests.get()**.
- While POST indicates that you're trying to send data to a specified server. Besides GET and POST, there are several other common methods.
- **Other HTTP Methods** include PUT, DELETE, HEAD, PATCH, and OPTIONS.

- **Installing Requests** : Requests is available on PyPI: *python -m pip install requests*

```
import requests

response = requests.get('https://easyquran.com/video/index.php')
print(response)

if response:
    print('Success!')
else:
    print('An error has occurred.')

print('Status code = ', response.status_code)
print('Headers with content-type = ', response.headers['content-type'])
print('Headers = ', response.headers)
print('Encoding = ', response.encoding)
print('Text = ')
print(response.text)

print('response.content = ', response.content)
```

- **200 status code :**
is by far the most common returned. It means, simply, that the request was received and understood and is being processed.
- **201 status code:**
indicates that a request was successful and as a result, a resource has been created (for example a new page).

Downloading data from an HTTP server

```
import argparse
import urllib.request

REMOTE_SERVER_HOST = 'https://easyquran.com/video/index.php' # try wrong path!
class HTTPClient:
    def __init__(self, host): # For more info. https://stackoverflow.com/questions/625083/what-do-init-and-self-do-in-python
        self.host = host

    def fetch(self):
        response = urllib.request.urlopen(self.host)
        data = response.read()
        text = data.decode('utf-8')
        return text

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='HTTP Client Example')
    parser.add_argument('--host', action="store", dest="host", default=REMOTE_SERVER_HOST)
    # to change default , in terminal , ex: python DownHTTP_2.py --host=http://www.google.com

    given_args = parser.parse_args()
    host = given_args.host
    client = HTTPClient(host)
    print (client.fetch())
```

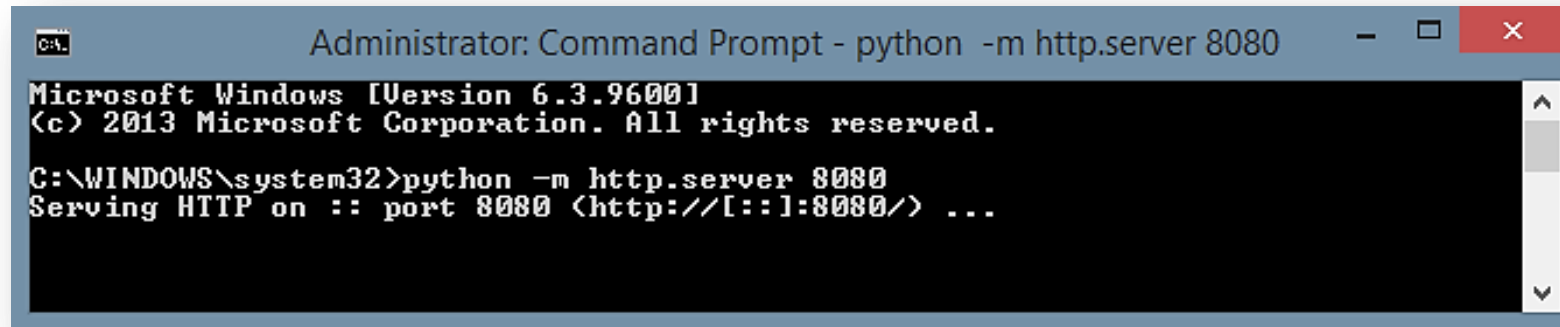
Requests exceptions (HTTPError)

```
import requests
from requests.exceptions import HTTPError

for url in ['https://www.google.com', 'https://www.python.org']:
    try:
        response = requests.get(url)
        # If the response was successful, no Exception will be raised
        response.raise_for_status()
    except HTTPError as http_err:
        print(f'HTTP error occurred: {http_err}') # Python 3.6
    except Exception as err:
        print(f'Other error occurred: {err}') # Python 3.6
    else:
        print('Success!')
```

Serving HTTP requests from your machine

- Python ships with a very simple web server that can be launched from the command line as follows:



```
Administrator: Command Prompt - python -m http.server 8080
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>python -m http.server 8080
Serving HTTP on :: port 8080 (http://[::]:8080/) ...
```

- You can access this web server from your browser by typing `http://localhost:8080`.
<http://net-informations.com/q/mis/8080.html>
- By default, server binds itself to all interfaces and port 8000. The option `-b/--bind` specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. For example, the following command causes the server to bind to localhost only:

`python -m http.server 8080 --bind 127.0.0.1` OR `python -m http.server 8080 -b 127.0.0.1`

- By default, server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory :

```
python -m http.server 8080 --bind 127.0.0.1 -d /tmp/
```

```
python -m http.server 8080 --bind 127.0.0.1 --directory G:/MyHTTPserver/
```

Note: Directory binding is introduced in *python 3.7*

- HTTP server script using BaseHTTPServer module:

Python provides us an HTTP server module, it's called *http.server*. We use two classes from this module, namely *BaseHTTPRequestHandler* and *HTTPServer*. We also need to use *os* module to access file in your system.

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import os
```

Simple HTTP Web Server and Client in Python

```
import argparse
import sys
from http.server import BaseHTTPRequestHandler, HTTPServer
import os

DEFAULT_HOST = '127.0.0.1'
DEFAULT_PORT = 8080

class RequestHandler(BaseHTTPRequestHandler):
    """ Custom request handler """
    def do_GET(self):
        # Handler for the GET requests
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        # Send the message to browser
        self.wfile.write(bytes("Hello from server! .. How are you all? Network programming class :) ", 'utf-8'))
        return

class CustomHTTPServer(HTTPServer):
    # A custom HTTP server
    def __init__(self, host, port):
        server_address = (host, port)
        HTTPServer.__init__(self, server_address, RequestHandler)

def run_server(port):
    try:
        server = CustomHTTPServer(DEFAULT_HOST, port)
        print("Custom HTTP server started on port: %s" % port)
        server.serve_forever()
    except Exception as err:
        print("Error:%s" % err)
    except KeyboardInterrupt:
        print("Server interrupted and is shutting down...")
        server.socket.close()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Simple HTTP Server Example')
    parser.add_argument('--port', action="store", dest="port", type=int, default=DEFAULT_PORT)
    given_args = parser.parse_args()
    port = given_args.port
    run_server(port)
```



- urllib3 is a powerful, user-friendly HTTP client for Python. Much of the Python ecosystem already uses urllib3.
- urllib3 brings many critical features that are missing from the Python standard libraries:
 - Thread safety.
 - Connection pooling.
 - *Client-side TLS/SSL verification.*
 - File uploads with multipart encoding.
 - Helpers for retrying requests and dealing with HTTP redirects.
 - Support for gzip, deflate, and brotli encoding.
 - Proxy support for HTTP and SOCKS.
 - 100% test coverage.

Python urllib3 status

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirects (300–399)
- Client errors (400–499)
- Server errors (500–599)

- PoolManager is used to generate a request. It handles all of the details of connection pooling and thread safety.
- request() returns a HTTPResponse object. The HTTPResponse object provides status, data, and headers attributes:

```
import urllib3

http = urllib3.PoolManager()
resp = http.request("GET", 'https://httpbin.org/forms/post')

.....
r = http.request('POST', 'https://httpbin.org/forms/post', fields={'custname': 'Deaa'})
```

1) <https://httpbin.org/> It echoes the data used in your request for any of these types:

- <https://httpbin.org/anything> Returns most of the below.
- <https://httpbin.org/ip> Returns Origin IP.
- <https://httpbin.org/user-agent> Returns user-agent.
- <https://httpbin.org/headers> Returns header dict.
- <https://httpbin.org/get> Returns GET data.
- <https://httpbin.org/post> Returns POST data.
- <https://httpbin.org/put> Returns PUT data.
- <https://httpbin.org/delete> Returns DELETE data.
- <https://httpbin.org/gzip> Returns gzip-encoded data.
- <https://httpbin.org/status/:code> Returns given HTTP Status code.
- <https://httpbin.org/response-headers?key=val> Returns given response headers.
- <https://httpbin.org/redirect/:n> 302 Redirects n times.
- <https://httpbin.org/relative-redirect/:n> 302 Relative redirects n times.
- <https://httpbin.org/cookies> Returns cookie data.
- <https://httpbin.org/cookies/set/:name/:value> Sets a simple cookie.
- <https://httpbin.org/basic-auth/:user/:passwd> Challenges HTTPBasic Auth.
- <https://httpbin.org/hidden-basic-auth/:user/:passwd> 404'd BasicAuth.
- <https://httpbin.org/digest-auth/:qop/:user/:passwd> Challenges HTTP Digest Auth.
- <https://httpbin.org/stream/:n> Streams n–100 lines.
- <https://httpbin.org/delay/:n> Delays responding for n–10 seconds.

2) <http://ptsv2.com/> Henry's Post Test Server V2! This is a service for developers testing clients that POST and GET things over HTTP.

```
import urllib3

http = urllib3.PoolManager()
# POST
req = http.request('POST', 'http://ptsv2.com/t/4opnx-1659718793', fields={'HTTP Auth': 'Deaa & Nour'})
print(req)
print('request data = ', req.data)

# GET
resp = http.request("GET", 'http://ptsv2.com/t/4opnx-1659718793')
print('resp.status', resp.status)
print('resp.data', resp.data)
print(resp.data.decode('utf-8'))

# Python urllib3 HEAD request
print('Server = ', resp.headers['Server'])
print('Date = ', resp.headers['Date'])
print('Content-Type = ', resp.headers['Content-Type'])
```

Simple HTTP_server using requests and urllib3



```
from http.server import BaseHTTPRequestHandler, HTTPServer
import os

#Create custom HTTPRequestHandler class
class FunHTTPRequestHandler(BaseHTTPRequestHandler):
    #handle GET command
    def do_GET(self):
        rootdir = 'C:/Users/user/PycharmProjects/pythonProject2/lecture10/' # file location
        print('self.path.endswith(.html)', self.path.endswith('.html'))
        try:
            if self.path.endswith('.html'):
                f = open(rootdir + self.path) #open requested file
                #send code 200 response
                self.send_response(200)
                #send header first
                self.send_header('Content-type','text-html')
                self.end_headers()
                #send file content to client
                self.wfile.write(f.read().encode())
                f.close()
            return
        except IOError:
            self.send_error(404, 'file not found')

    def run():
        print('http server is starting...')

        #ip and port of servr
        #by default http server port is 80
        server_address = ('127.0.0.1', 80)
        httpd = HTTPServer(server_address, FunHTTPRequestHandler)
        print('http server is running...')
        httpd.serve_forever()

if __name__ == '__main__':
    run()
```

Simple HTTP_client using requests and urllib3



```
import requests, urllib3.request, sys, os, socket, http.client

#get http server ip
http_server = sys.argv[1]
#create a connection
conn = http.client.HTTPConnection(http_server)
# conn.request('GET', '/')
print('conn = ', conn)
while 1:
    cmd = input('input command (ex. GET index.html): ')
    cmd = cmd.split()
    print('cmd = ', cmd)
    if cmd[0] == 'exit': # enter exit to end it
        break
    # request command to server
    conn.request(cmd[0], cmd[1])
    # get response from server
    rsp = conn.getresponse()
    # print server response and data
    print('rsp.status = ', rsp.reason, ' ||  rsp.reason = ', rsp.status)
    data_received = rsp.read()
    print('data_received', data_received)
conn.close()

# python simple-httpclient.py 127.0.0.1
# GET test.html
```

SSL/TLS

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Network programming

Course ID: CYB615



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

- **Secure Sockets Layer (SSL) and Transport Layer Security (TLS) :**
 - Cryptographic security protocols.
 - Used to make sure that network communication is secure.
 - Main goals are to provide data integrity and communication privacy.
 - The SSL protocol was the first protocol designed for this purpose and TLS is its successor.
 - SSL is now considered obsolete and insecure (even its latest version), so modern browsers such as Chrome or Firefox use TLS instead.

■ SSL and TLS :

- Commonly used by web browsers to protect connections between web applications and web servers.
- Many other TCP-based protocols use TLS/SSL as well, including email (SMTP/POP3), instant messaging (XMPP), FTP, VoIP, VPN, and others.
- Typically, when a service uses a secure connection the letter S is appended to the protocol name, for example, HTTPS, SMTPS, FTPS, SIPS.
- In most cases, SSL/TLS implementations are based on the **OpenSSL** library.

■ SSL and TLS

- Frameworks that use a lot of different cryptographic algorithms, for example, RSA and various Diffie–Hellman algorithms.
- The parties agree on which algorithm to use during initial communication.
- TLS version (TLS 1.3) is specified in the IETF (Internet Engineering Task Force) document RFC 8446 and SSL version (SSL 3.0) is specified in the IETF document RFC 6101.
- SSL and its successor TLS are used widely in email, instant messaging, and voice-over applications.

Privacy & Integrity:

- SSL/TLS protocols allow the connection between two mediums (client-server) to be encrypted.
- Encryption lets you make sure that no third party is able to read the data or tamper with it. Unencrypted communication can expose sensitive data such as user names, passwords, credit card numbers, and more.



- The cookie that we use to authenticate on our website is sent in plain text and anyone who intercepts the connection can see it.
- The attacker can use this information to log into our website administration panel.
- From then on, the attacker's options expand dramatically. However, if we access our website using SSL/TLS, the attacker who is sniffing traffic sees something quite different.

In this case, the information is useless to the attacker.

■ Identification

- SSL/TLS protocols use public-key cryptography. Except for encryption, this technology is also used to authenticate communicating parties.
- This means, that one or both parties know exactly who they are communicating with. This is crucial for such applications as online transactions because must be sure that we are transferring money to the person or company who are who they claim to be.
- When a secure connection is established, the server sends its SSL/TSL certificate to the client.
- The certificate is then checked by the client against a trusted Certificate Authority, validating the server's identity. Such a certificate cannot be falsified, so the client may be one hundred percent sure that they are communicating with the right server.

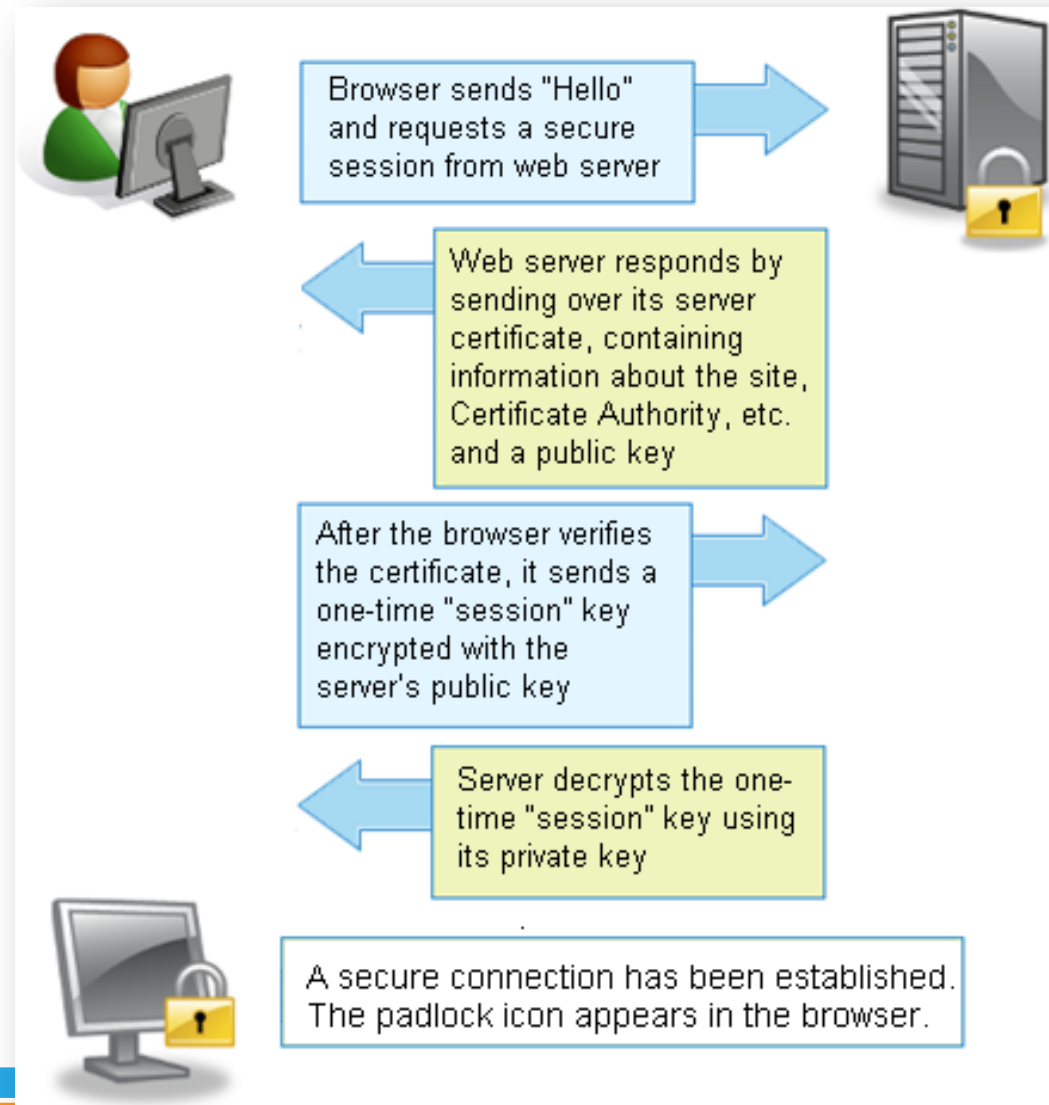
- **Perfect Forward Secrecy**

Perfect forward secrecy (PFS) is a mechanism that is used to protect the client if the private key of the server is compromised. Thanks to PFS, the attacker is not able to decrypt any previous TLS communications. To ensure perfect forward secrecy, we use new keys for every session. These keys are valid only as long as the session is active.

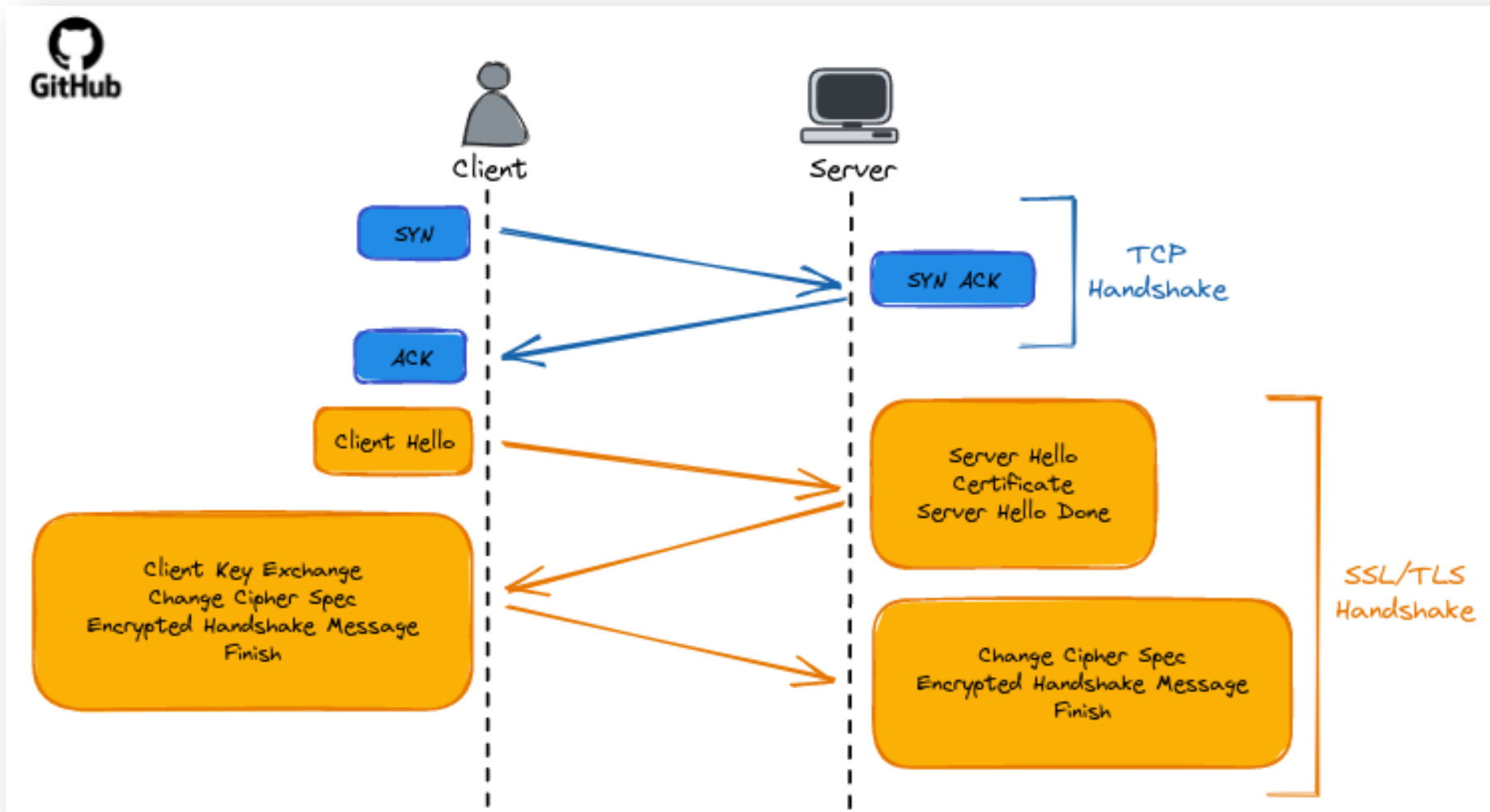
- **What is HTTPS?**

HTTPS is a secure version of the HTTP protocol. It is an extension of the HTTP protocol that implements TLS. It lets you securely access websites and web applications. With HTTPS you can be sure that nobody reads your sensitive information and you can be sure that you are accessing the real website and not a forged one.

How do SSL/TLS work?

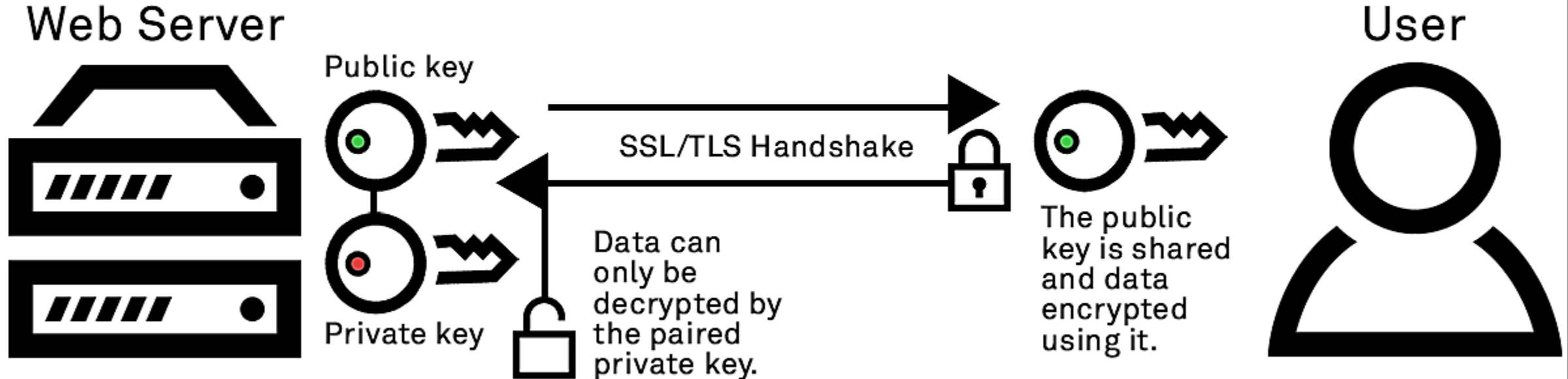


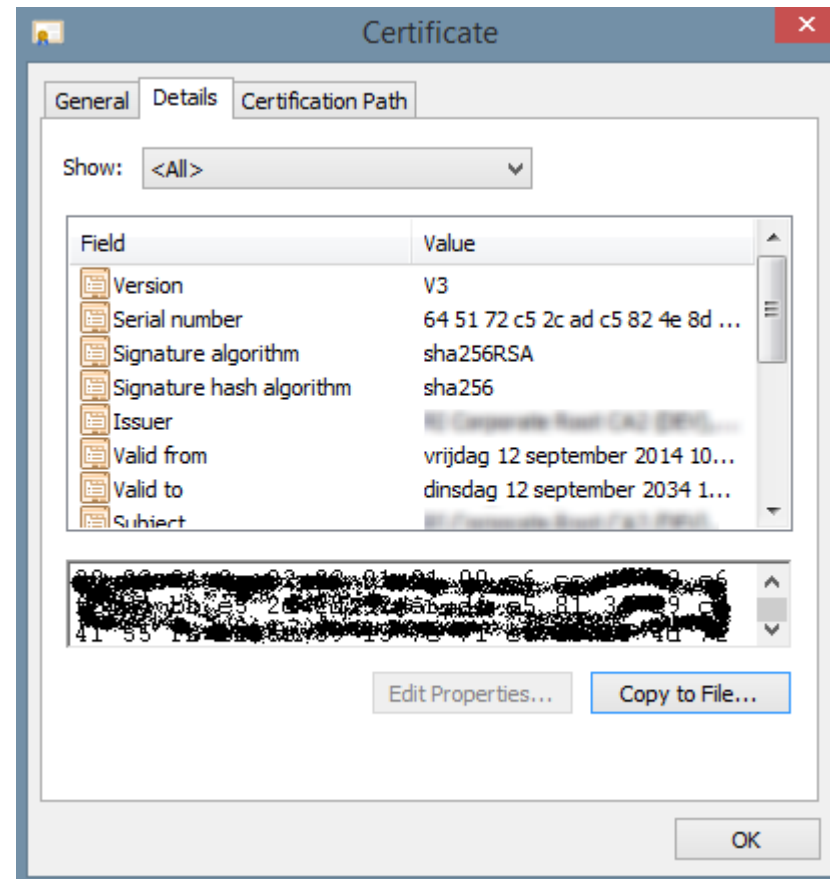
TLS handshake vs TCP handshake



- Keys come in two halves, a public key and a private key. The public key can be distributed publicly and widely, and you can use it to verify, but not replicate, information generated using the private key. The private key must be kept secret.
- When performing authentication, SSL uses a technique called public-key cryptography. Public-key cryptography is based on the concept of a key pair, which consists of a public key and a private key. Data that has been encrypted with a public key can be decrypted only with the corresponding private key.
- The private key is used for both encrypting and decrypting sensitive data. It is shared between the sender and receiver of encrypted data. The public key is used only for the purpose of encrypting the data. The private key mechanism is faster.

Public-Private Key Encryption





- Installing OpenSSL : <https://slproweb.com/products/Win32OpenSSL.html>
- OpenSSL is an open-source command line tool that is commonly used to:
generate private keys, create CSRs, install your SSL/TLS certificate, and identify certificate information.
- With OpenSSL, you can apply for your digital certificate (Generate the Certificate Signing Request) and install the SSL files on your server. You can also convert your certificate into various SSL formats, as well as do all kind of verifications.

<https://www.digicert.com/kb/ssl-support/openssl-quick-reference-guide.htm>

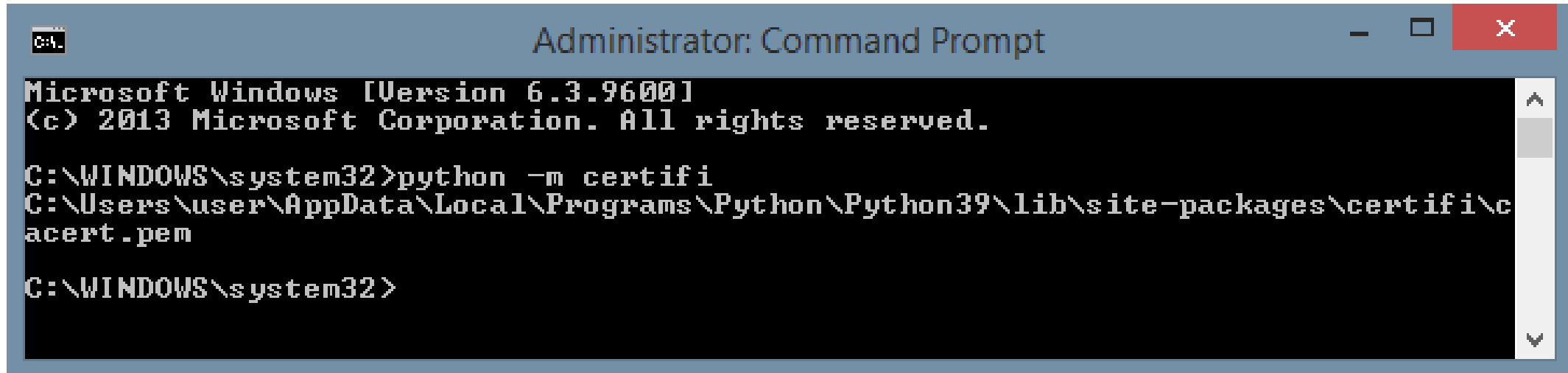
- **.key** files are generally the private key, used by the server to encrypt and package data for verification by clients.
- **.pem** files are generally the public key, used by the client to verify and decrypt data sent by servers. PEM files could also be encoded with private keys, so check the content if you're not sure.
- **.p12** files have both halves of the key embedded so that administrators can easily manage halves of keys.
- **.cert or .crt** files are the signed certificates -- basically the "magic" that allows certain sites to be marked as trustworthy by a third party.
- **.csr** is a certificate signing request. In the self-signed scenario, you will use the certificate signing request with your own private key to verify your private key (thus self-signed). Depending on your specific application, this might not be needed. (*needed for web servers or remote procedural call (RPC) server, but not much else*).

1. Open terminal, go to the your code folder.
2. Create new folder :> mkdir certs and :> cd certs go new created certs folder
3. Install this library :> pip install pyopenssl
4. Copy and past the code , ex :> *openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout key.pem -days 365*
5. Answer the questions asked.
6. Done your '.cert' , '.pem' , '.key' created.

Convert .pem to (.crt , .key):

- *C:\Users\user>echo quit / openssl s_client -showcerts -servername "www.google.com" -connect www.google.com:443 > cacert.pem*
- *C:\Users\user>openssl x509 -outform der -in cacert.pem -out g:\certificate.crt*
- *C:\Users\user>openssl x509 -outform der -in cacert.pem -out g:\certificate.key*

- You can also find the cacert.pem path from the command line using the following command.



```
Administrator: Command Prompt

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>python -m certifi
C:\Users\user\AppData\Local\Programs\Python\Python39\lib\site-packages\certifi\cacert.pem

C:\WINDOWS\system32>
```

- The urllib3 provides client-side TLS/SSL verification. For this, we need to download the certifi module. It is a carefully curated collection of Root Certificates for validating the trustworthiness of SSL certificates while verifying the identity of TLS hosts. It has been extracted from the Requests project. *python -m pip install certifi*
- To reference the installed certificate authority (CA) bundle, we use the built-in where function. *certifi.where()*

```
import urllib3
import certifi

print(certifi.where())

url = 'https://www.google.com'
http = urllib3.PoolManager(ca_certs=certifi.where())
resp = http.request('GET', url)
print(resp.status)
print(http) , print(certifi.where())
```

- We pass the root CA bundle to the PoolManager. Without this CA bundle, the request would issue the following warning: **InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly advised.**

- The SSLSocket class is derived from the socket class and represents a secure socket in TLS context.
- The Python examples given here use SSLSocket instances in both server and client scenarios.
- Normally, an SSL/TLS client verifies the server's certificate. It's also possible for the server to require a signed certificate from the client. These are called Client Certificates. This ensures that not only can the client trust the server, but the server can also trusts the client.
- Traditionally in Python, you'd pass the ca_certs parameter to the ssl.wrap_socket() function on the server to enable client certificates:

```
# Client
ssl.wrap_socket(s, ca_certs="ssl/server.crt", cert_reqs=ssl.CERT_REQUIRED,
                certfile="ssl/client.crt", keyfile="ssl/client.key")

# Server
ssl.wrap_socket(connection, server_side=True, certfile="ssl/server.crt",
                keyfile="ssl/server.key", ca_certs="ssl/client.crt")
```

- `ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version={seedocs}, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)`
- `ssl.wrap_socket(sock)` → Takes an instance `sock` of `socket.socket`, and returns an instance of `ssl.SSLSocket`, a subtype of `socket.socket`, which wraps the underlying socket in an SSL context. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

- The **keyfile** and **certfile** parameters specify optional files which contain a certificate to be used to identify the local side of the connection.
- The parameter **server_side** is a boolean which identifies whether server-side or client-side behavior is desired from this socket.
- The parameter **cert_reqs** specifies whether a certificate is required from the other side of the connection, and whether it will be validated if provided. It must be one of the three values CERT_NONE (certificates ignored), CERT_OPTIONAL (not required, but validated if provided), or CERT_REQUIRED (required and validated). If the value of this parameter is not CERT_NONE, then the ca_certs parameter must point to a file of CA certificates.



- The **ca_certs** file contains a set of concatenated “certification authority” certificates, which are used to validate certificates passed from the other end of the connection.
- The parameter **ssl_version** specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified, the default is `PROTOCOL_SSLv23`; it provides the most compatibility with other versions.

- Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

<i>client / server</i>	SSLv2	SSLv3	SSLv23	TLSv1
SSLv2	yes	no	yes	no
SSLv3	no	yes	yes	no
SSLv23	yes	no	yes	no
TLSv1	no	no	yes	yes

- The certifi package provides Mozilla's CA Bundle in Python.
- Browsers and certificate authorities have finalized that 1024-bit keys are unacceptably weak for certificates, particularly root certificates.
- For the same reason, Mozilla has removed any weak (i.e., 1024-bit key) certificate from its bundle, replacing it with the equivalent robust (i.e., 2048-bit or higher key) certificate from the same CA.
- If you put the additional certificates in the PEM bundle file, you can use these two environment variables to overwrite the default cert stores used by Python OpenSSL and Requests:

```
SSL_CERT_FILE = '/System/Library/OpenSSL/cert.pem'  
REQUESTS_CA_BUNDLE = '/System/Library/OpenSSL/cert.pem'
```

```
import socket, ssl, pprint

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# require a certificate from the server
# set ca_certs to the location of the system root certificates, or else install certifi and use certifi.where()
ssl_sock = ssl.wrap_socket(s, ca_certs=b"C:\\...\\certifi\\cacert.pem", cert_reqs=ssl.CERT_REQUIRED)
ssl_sock.connect(('www.linkedin.com', 443))
cert = ssl_sock.getpeercert()

print('Certification = ', cert)
pprint.pprint(cert)

ssl_sock.sendall(b"HEAD / HTTP/1.0\r\nHost:www.linkedin.com\r\n\r\n")
pprint.pprint(ssl_sock.recv(1024).split(b"\r\n"))

# note that closing the SSLSocket will also close the underlying socket
ssl_sock.close()
```

- **Full example of a client and server who both validate each other's certificates:**

For this example, we'll create Self-signed server and client certificates. Normally you'd use a server certificate from a Certificate Authority such as Let's Encrypt, and would setup your own Certificate Authority so you can sign and revoke client certificates.

- **Self-signed certificates by (OpenSSL)**

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate.

- The simplest way to do this is with the OpenSSL package, using something like the following:

[illegible]

- The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

- **Create server certificate:**

```
openssl req -new -newkey rsa:2048 -days 365 -nodes -x509 -keyout server.key -out server.crt
```

- **Make sure to enter '*example.com*' for the Common Name.**

- **Next, generate a client certificate:**

```
openssl req -new -newkey rsa:2048 -days 365 -nodes -x509 -keyout client.key -out client.crt
```

- **The Common Name for the client certificate doesn't really matter.**

■ A few notes:

1. You can concatenate multiple client certificates into a single PEM file to authenticate different clients.
2. You can re-use the same cert and key on both the server and client. This way, you don't need to generate a specific client certificate. However, any clients using that certificate will require the key, and will be able to impersonate the server. There's also no way to distinguish between clients anymore.
3. You don't need to setup your own Certificate Authority and sign client certificates. You can just generate them with the above mentioned openssl command and add them to the trusted certificates file. If you no longer trust the client, just remove the certificate from the file.
4. I'm not sure if the server verifies the client certificate's expiration date.

- **Example Server program that uses TLS:**

1. The SSL server program creates a server socket and listens on port 15001 on localhost.
2. When a client connection is accepted, a client socket is created which is wrapped into a SSLSocket.
3. The SSLSocket is supplied with the CA Certificate, the certificate of the server and the corresponding private key.
4. A call to the `getpeercert()` on the secure client connection (ie., the SSLSocket instance) gets the certificate of the client.
5. The server program validates the common Name, time validity of the certificate before sending any information over the secure connection with the client.

- **Example Client program that uses TLS:**

1. The client program makes use of the SSLContext instance to load the CA certificate, client certificate and the corresponding private key.
2. The client creates a stream based socket and wraps it around an SSLSocket instance.
3. Through the SSLSocket instance the security aspects of the communication: Privacy, Data Integrity and Authentication are taken care.
4. Once the SSLSocket instance makes a connection to the server listening on a specific IP address and Port it requests for the certificate of the server to which it is connected to.
5. The fields of the certificate like commonName, notBefore and notAfter are validated before any communication is received from the server.