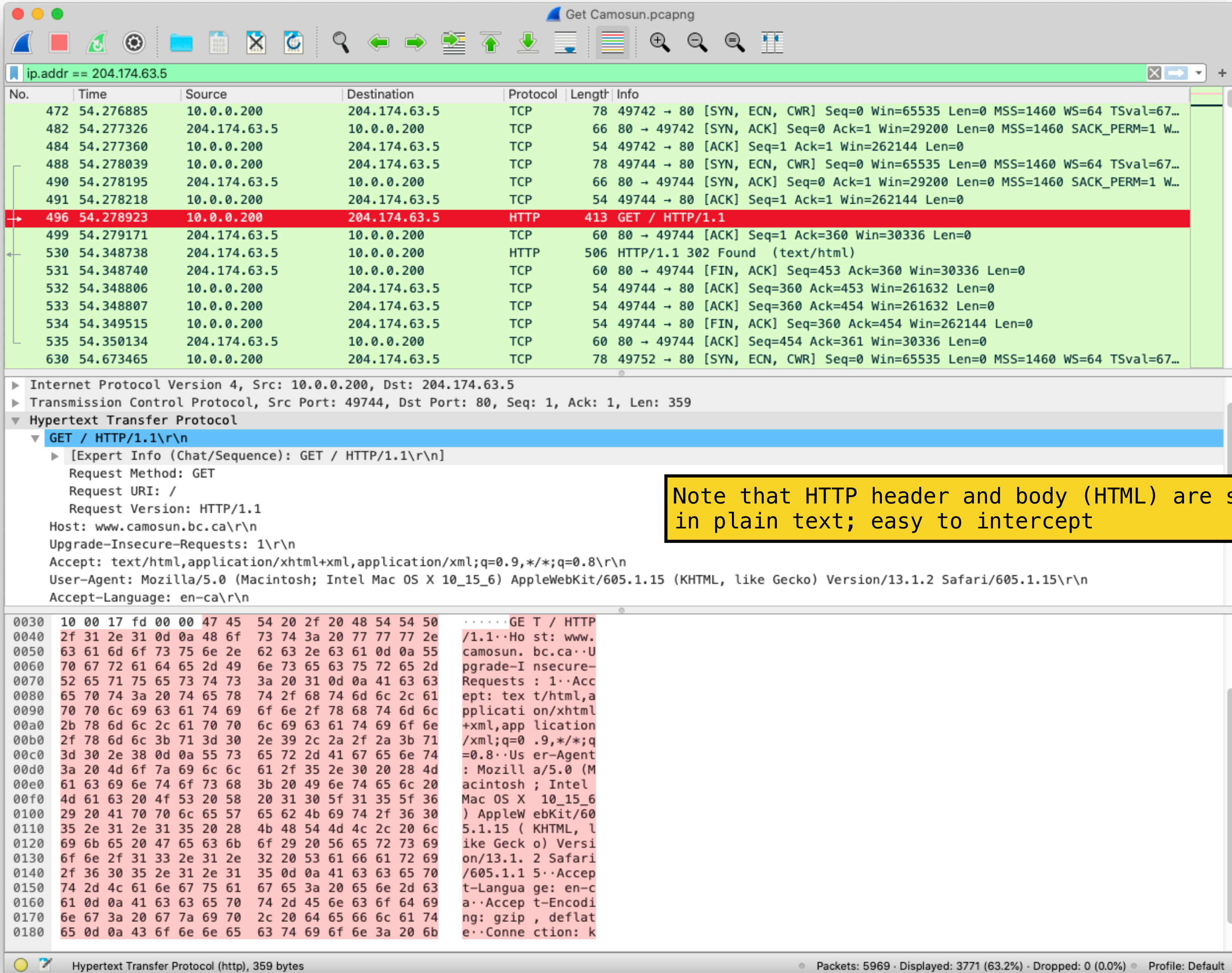
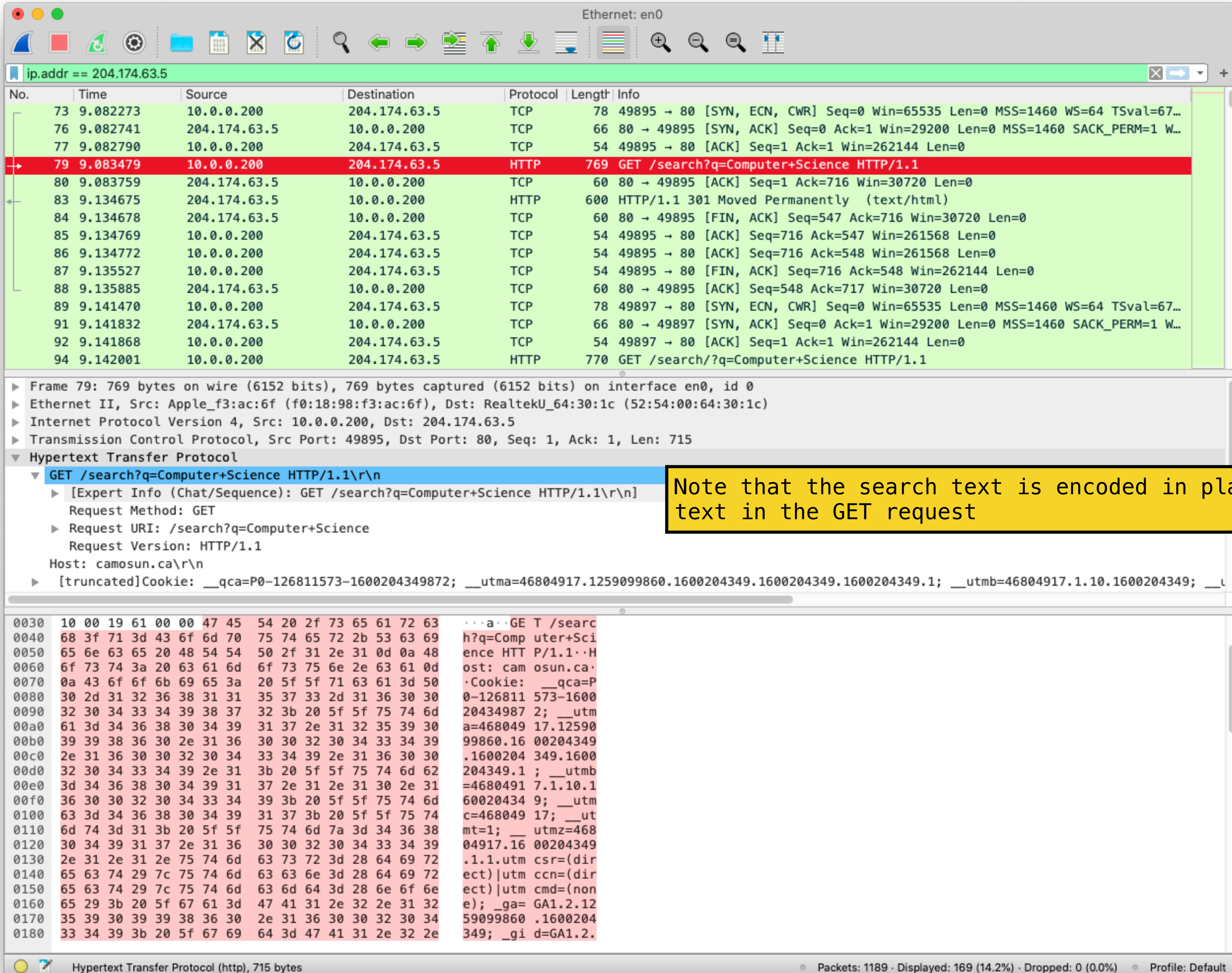
Encryption

HTTP Request



Note that HTTP header and body (HTML) are sent in plain text; easy to intercept

HTTP Search



Note that the search text is encoded in plain text in the GET request

Security, Privacy, Integrity Issues

- When transmitting data in plain text over a network:
  - Online banking credentials can be intercepted and used to make fraudulent transactions
  - Email content can be read and disclosed to unauthorized parties
  - Text messages can be modified or forged
  - etc.

Ethical, Legal, Social Implications

- There are many software tools that make it easy to watch and/or modify traffic, both in wired and wireless environments
- Point to Ponder:
  - I can vs. I should vs. I should not
  - parental control vs. crime prevention vs. social protest vs. eliminating dissent
- Many jurisdictions (including Canada) view privacy as a right and make it illegal to spy on others (some exceptions apply)

Symmetric Encryption in Python

Setup on Ubuntu

```
sudo apt-get install python3-pip # if not already installed
pipenv install cryptography # from inside the PyCharm terminal
```

Symmetric Encryption

```
#!/usr/bin/python3.11

from cryptography.fernet import Fernet

key = Fernet.generate_key()
print(key)

f = Fernet(key)
encrypted = f.encrypt(b"Hello World")
print(encrypted)

decrypted = f.decrypt(encrypted)
print(decrypted)
```

Symmetric Encryption

```
#!/usr/bin/python3.11

from cryptography.fernet import Fernet

key = Fernet.generate_key()
```

```
b'RE_9SKxNg5Aj8x7TteJTyVZ87EDW5xM5c7XjDdazrGE='
```

```
f = Fernet(key)
encrypted = f.encrypt(b"Hello World")
print(encrypted)

decrypted = f.decrypt(encrypted)
print(decrypted)
```

Symmetric Encryption

```
#!/usr/bin/python3.11

from cryptography.fernet import Fernet

key = Fernet.generate_key()
print(key)

f = Fernet(key)
encrypted = f.encrypt(b"Hello World")
```

```
b'gAAAAABfX_WN08Z0U3pZXwzMI6wEKDPUALB_l6kafBEujKl8SZFCSHmlolGACV5kLU4hkI-PjBtOPd8KSdhPhauxmCIcj8qwsA=='
```

```
decrypted = f.decrypt(encrypted)
print(decrypted)
```

Symmetric Encryption

```
#!/usr/bin/python3.11

from cryptography.fernet import Fernet

key = Fernet.generate_key()
print(key)

f = Fernet(key)
encrypted = f.encrypt(b"Hello World")
print(encrypted)

decrypted = f.decrypt(encrypted)
print(decrypted)
```

```
b'Hello World'
```

Symmetric Encryption

- Can now encrypt communication between a server and a client
- This was the easy part

Symmetric Encryption

- Problem: We must keep the key secret at all times
  - How do we distribute the key to clients?
  - How do we protect the key once it is shared with a client?
- Problem: Without a timestamp or other nonce, the same plain text string will always return the same encrypted string, e.g.,
  - 'Yes', given the previous key, will **ALWAYS** generate b'gAAAAABfX_Z0dZC-PO0maWc9TM-LGWxuTKUwK1N6mHixvwjR6FBAMC3WW3rpbKcNge3WtcJcBho33NsI3G7Eceb67EnhaMNK_Q=='
  - Given time, an attacker can figure this out
- These are hard problems! How do we solve them?

Expert Advice

- DON'T!

Expert Advice

- Solving these issues requires in-depth analysis
- Cryptographic expertise is required
- Involves the use of mathematical proofs, verification tools, and expert peer review
- Better to use established, well-tested libraries

Public Key Encryption in Python

Basic Setup

- For demonstration purposes, we'll create a self-signed certificate
- In a production environment, we would have to obtain a certificate from an official source (e.g., https://letsencrypt.org/)

Create a Configuration File (server.cnf)

```
[ req ]
prompt = no
distinguished_name = req_distinguished_name

[ req_distinguished_name ]
countryName = ca
stateOrProvinceName = British Columbia
localityName = Victoria
organizationName = Camosun College
organizationalUnitName = Department of Computer Science
emailAddress = horiem@camosun.bc.ca
commonName = horiem.ca
```

Modify /etc/hosts

```
127.0.0.1 localhost
127.0.1.1 ics226
127.0.0.1 horiem.ca

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Generate a Self-Signed Certificate

```
openssl req -x509 -nodes -days 1095 -newkey rsa:4096 -config
server.cnf -out server.crt -keyout server.key
```

Review the Certificate

```
openssl x509 -text -in server.crt -noout
```

Create a PEM File for the Server

```
cat server.crt server.key > server.pem
```

Protect this file! It contains the private key AND the public key

TLS Server

```
#!/usr/bin/python3.11

from socket import socket, AF_INET, SOCK_STREAM, SOL_SOCKET,
SO_REUSEADDR
from ssl import create_default_context, Purpose

HOST = ''
PORT = 12345
NUM_CLIENTS = 1

purpose = Purpose.CLIENT_AUTH
context = create_default_context(purpose, cafile=None)
context.load_cert_chain('server.pem')

listener = socket(AF_INET, SOCK_STREAM)
listener.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
listener.bind((HOST, PORT))
listener.listen(NUM_CLIENTS)
raw_sock, address = listener.accept()

ssl_sock = context.wrap_socket(raw_sock, server_side=True)
ssl_sock.sendall('Hello World!'.encode('utf-8'))

ssl_sock.close()
```

Based on Rhodes and Goerzen, Foundations of Python Network Programming

TLS Client

```
#!/usr/bin/env python3.11

from socket import socket, AF_INET, SOCK_STREAM
from ssl import create_default_context, Purpose

HOST = 'horiem.ca'
PORT = 12345
BUF_SIZE = 1024

purpose = Purpose.SERVER_AUTH
context = create_default_context(purpose, cafile='server.crt')

raw_sock = socket(AF_INET, SOCK_STREAM)
raw_sock.connect((HOST, PORT))

ssl_sock = context.wrap_socket(raw_sock, server_hostname=HOST)
while True:
    data = ssl_sock.recv(BUF_SIZE)
    if not data:
        break
    print(data)
```

Based on Rhodes and Goerzen, Foundations of Python Network Programming

Expert Advice

- DON'T!

Python Requests

Alternatives

- Prime Directive of Security: Only cryptography experts should write cryptography code; everybody else should use that code
- Can use *stunnel* to set up secure tunnels; Python socket code needs not be modified, except to make sure that the correct port is chosen so that packets travel through the tunnel
- Can also use *ssh* to tunnel traffic
- Can place *nginx* in front of the Python socket code to handle encryption and load balancing of HTTPS traffic

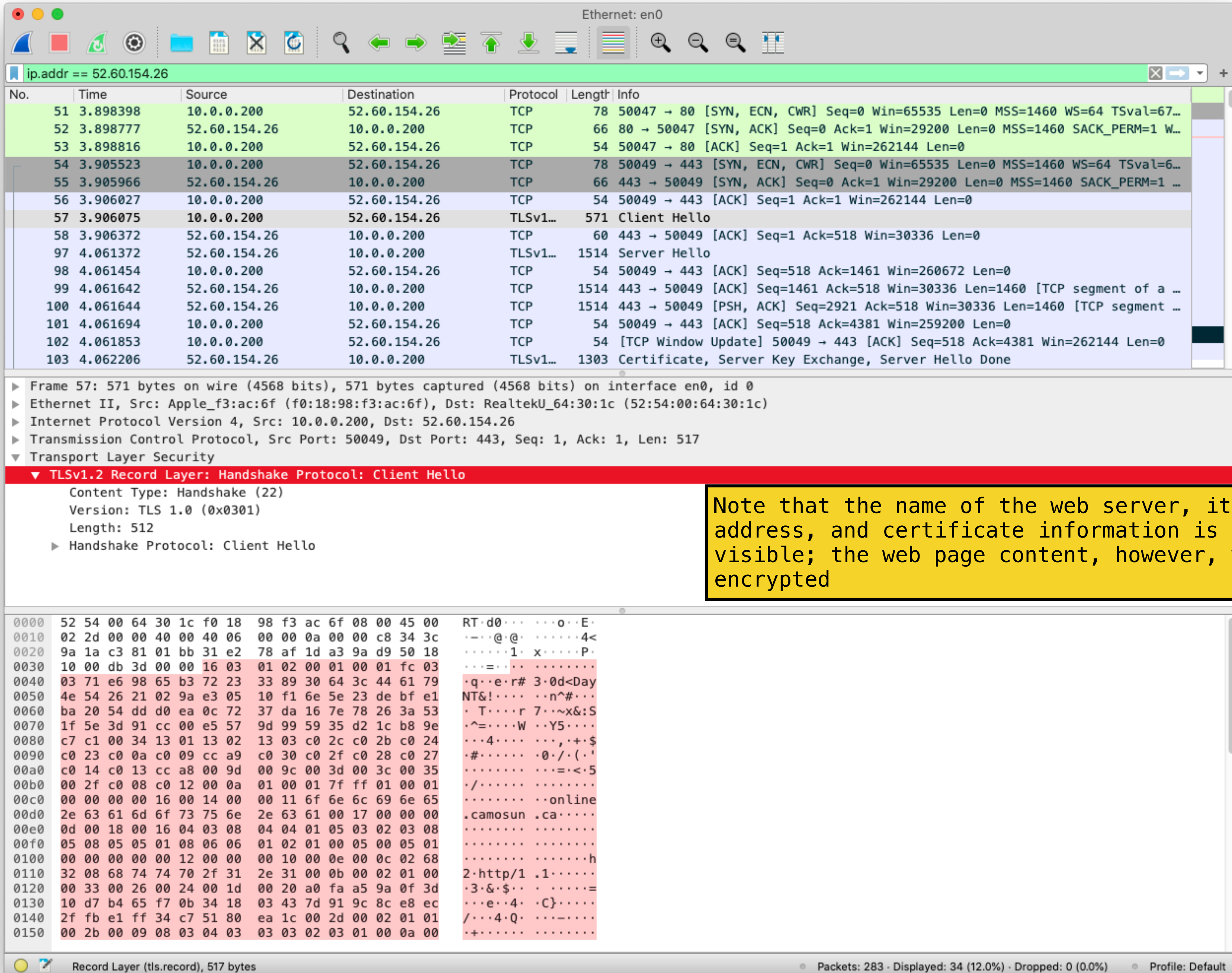
requests Library

- Can install and then use the *requests* library to make HTTPS client requests

```
import requests
r = requests.get('https://www.camosun.ca')
print(r.status_code)
print(r.text)
```

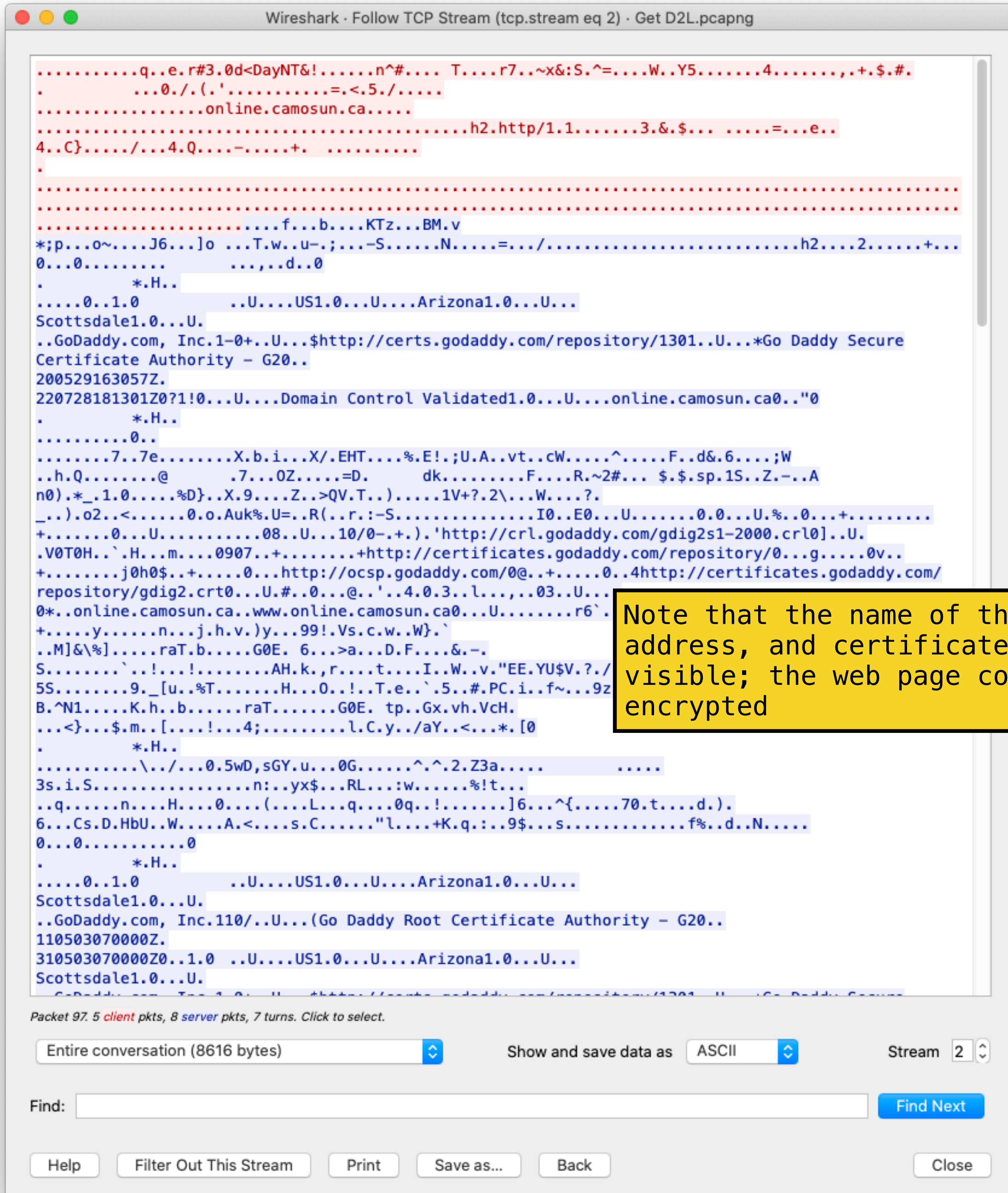
TLS

HTTPS Request



Note that the name of the web server, its IP address, and certificate information is still visible; the web page content, however, will be encrypted

HTTPS Request



Note that the name of the web server, its IP address, and certificate information is still visible; the web page content, however, will be encrypted

Notes on DNS

- Even though we went to an HTTPS site, the DNS request for www.camosun.ca is not encrypted, meaning that a network observer knows that we are visiting a Camosun web site
- Even if we encrypted DNS traffic, the fact that your computer shortly thereafter connects to an IP associated with Camosun is not hidden

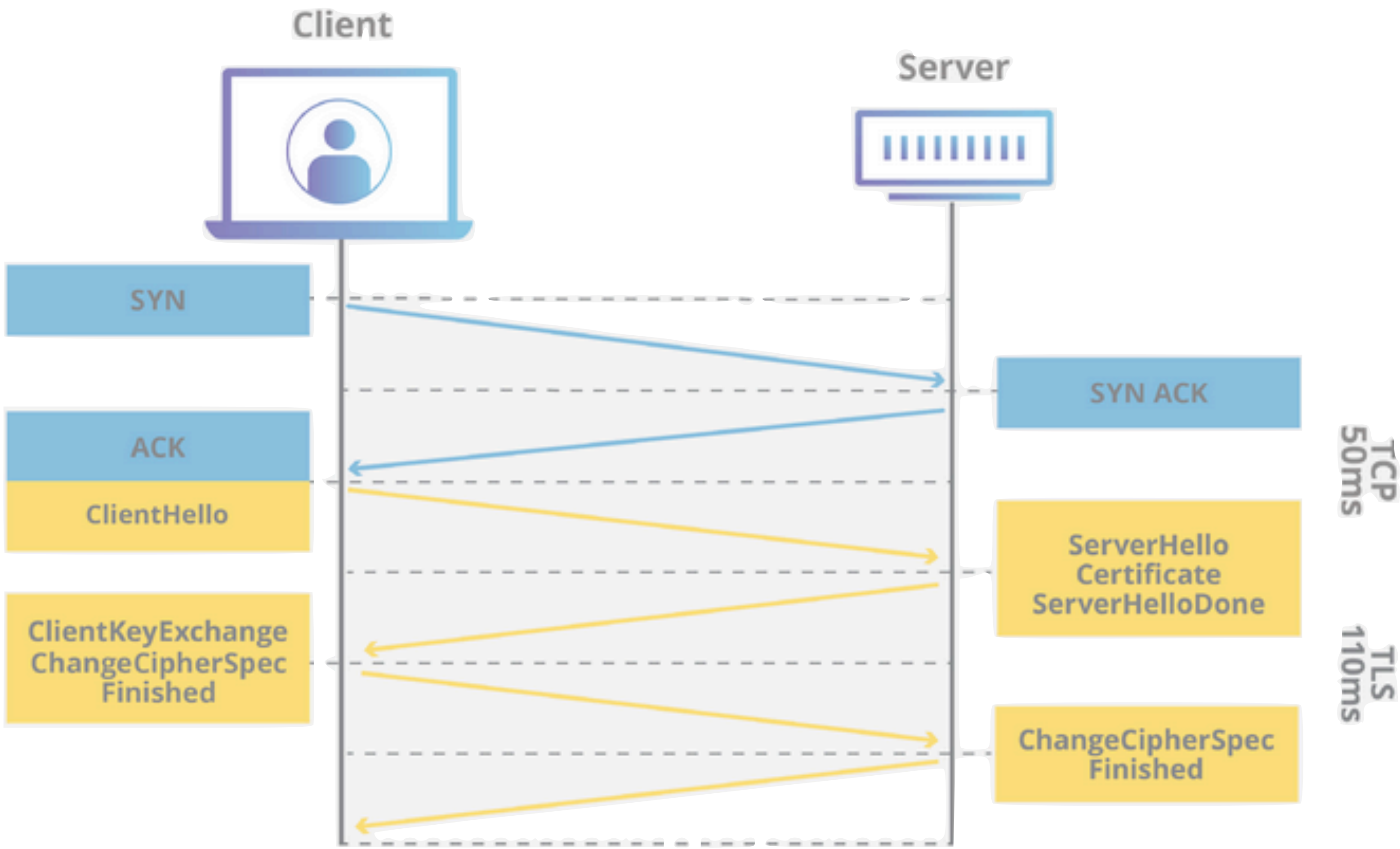
Notes on HTTPS

- Note that www.camosun.ca appears in plain text!
- The certificate check also shows that a GoDaddy certificate is being used
- The actual payload is encrypted, but based on data lengths, an observer may still figure out which web pages are being viewed
- ICS 228 is about exploring further weaknesses

TLS Handshake

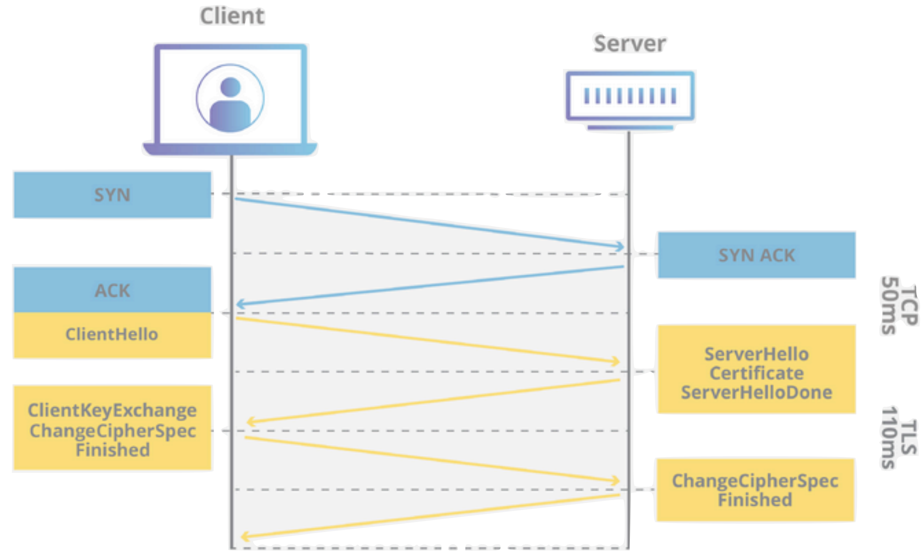
- When establishing an HTTPS connection, modern websites rely on TLS to provide the actual authentication and encryption
- During the actual setup (the TLS Handshake), the client and server establish
  - what version of TLS to use
  - what cipher suite to use
  - that the server could be authenticated successfully
  - session keys
- Details may differ, depending on what cipher was chosen

TLS 1.2 Handshake



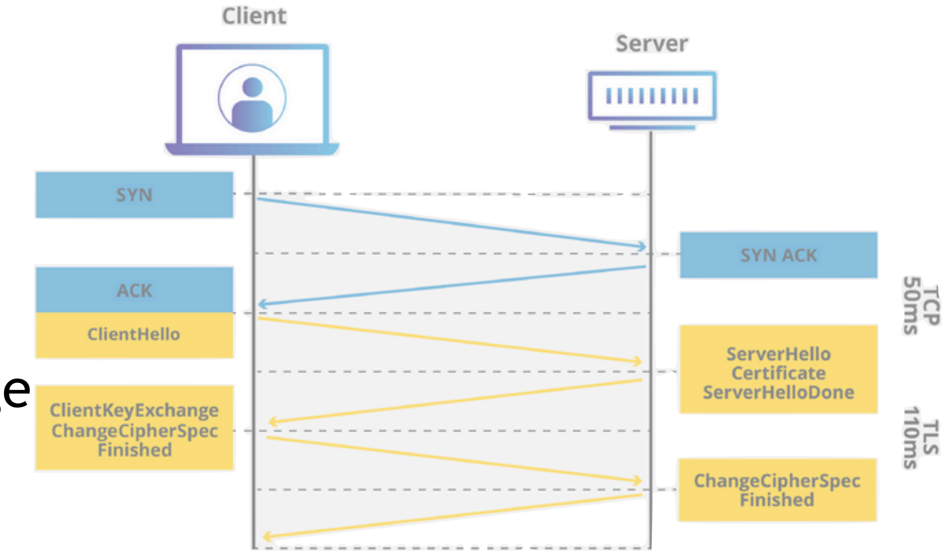
https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/

Client Hello



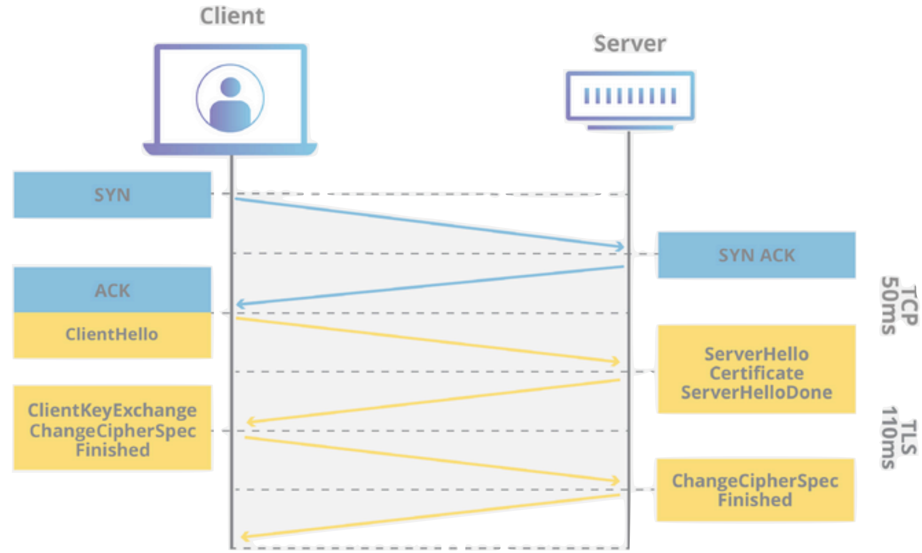
- Initiates the handshake
- Includes
  - TLS version that the client uses
  - Cipher suites that the client supports
  - A random byte string for later use

Server Hello, Certificate, Server Hello Done



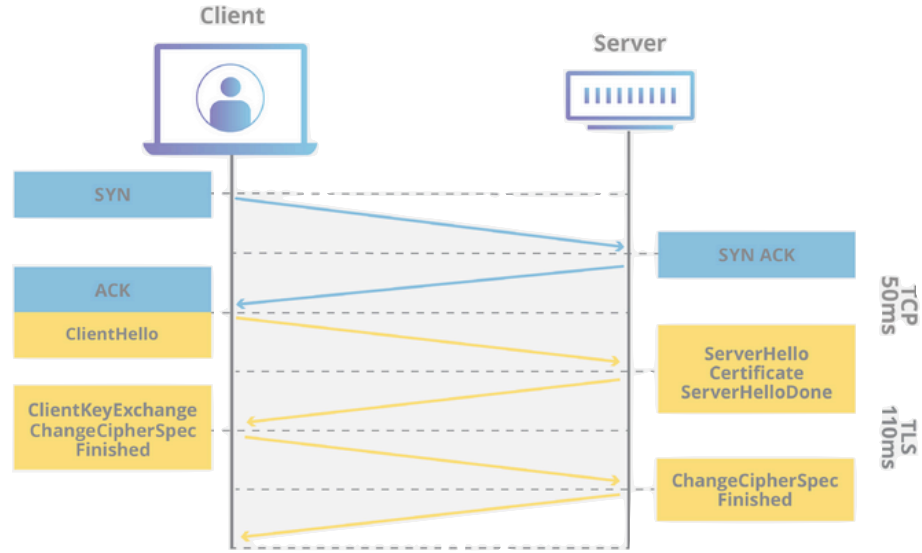
- Replies to the client hello message
- Includes
  - Another random byte string for later use
  - The chosen cipher suite
  - The server certificate

Client Key Exchange



- Provides the server with information needed to generate an encryption key for the session

Change Cipher Spec Finished



- Ready for encrypted session
- Change Cipher Spec possibly piggybacked on top of data
- This part of the handshake is already encrypted

Alert

- Encrypted
- Can be as simple as notifying a close of the connection
- Can also indicate a range of errors

TLS 1.3

- Reduces the number of exchanges
- Client Hello and Server Hello are still used
- Client Hello includes client key information based on an expected cipher protocol
- Assuming the server is willing to support the expected cipher, it will start encrypting data as part of its reply
- Subsequent connections even allow the Client Hello to include encrypted data. This is also known as 0-RTT (zero round trip time); some replay issues exist; see http://blog.cloudflare.com/introducing-0-rtt

Starting Points

- https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/
- http://blog.fourthbit.com/2014/12/23/traffic-analysis-of-an-ssl-slash-tls-session/
- Rhodes and Goerzen, Foundations of Python Network Programming, Apress, 2014

Key Skills

- Explain how a TLS session is established and terminated
- Explain the role of stunnel, nginx, requests
- Explain how to ensure security, privacy, and integrity of communication while recognizing the ethical, legal, and social implications of network programming