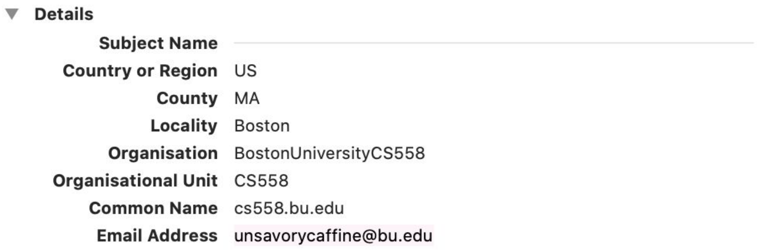
Tricking Your Browser

- Step 0: First we start out by doing some reading on the process. Links used:
    - https://gist.github.com/Soarez/9688998
- Step 1: Generate a public key corresponding to the given private key
    - Started out by making a privatekey.pem file with Textedit and the private key text from the assignment instructions
    - Generated public key from private.key pem
        - Using this command: `openssl rsa -in privatekey.pem -pub out`
            - Reference from: https://security.stackexchange.com/questions/172274/can-i-get-a-public-key-from-an-rsa-private-key
        - Output:

```
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA3ak32ELiqRv/7SQ1r1Hn
DKVfnepIMuQ4n1OKjhx0IMJ9MjZtnf2UA4P/M8LS9cfAoKCbl3GNVjfkzajCDIay
m4u+8FLRgau5UCDaEPcwvMasQsOzuBBvrSkra5bQzMZDTiBxrrSuUVJ1G7lmghP4
4Eu4VD1D6U9VtZbyrDLmGGQT5YJphlF5hphYK3MQndJZxmdLK66BdtuOAg9gNg6n
Sm+balUdRFBZy4TIVp2GR2Bc/XxVukekb6rIMPHIfVZv1Q/U3ZcTV7myk4hW3ewJ
/Sm793ZcxbLW+uIogSd2mjUTfNGK1t24Xw8bXyL8mz3H5Lb9KTFhZ1pNJ0XscgnZ
GrEeg+OcqX6n5i35VJHNzuDaN7jgVSFDDzKOxWGVkd2GTPdBXWJArna92MEHBIkR
3MQu5gOODpKKTnvylO62s0FLMBwWofNVahC6s1F0dHxMP/vEIGP/+FUIfHh8XAml
kvZ3hMZBwJF1+zp5Pi9n+A3uGQ0sXztch7ARTRzvZ/XcPR78vkp4kVfwBcIL2t6q
TA1SbP2IsoYnb9hHOoHwhkBt8DnsksDmgoZOSLeBALHVbgpyhich8rSlUN5nX1XP
NUYZl42MugOE8hrfZ0187/fDxM3VSFdwzyy+mYRcZWKctlmnvRtpFopYxHsxj3/a
DQT48GdOOOItzLsL49ZlEUMCAwEAAQ==
-----END PUBLIC KEY-----
```

- Step 2: Find contact email of CA
    - Contact email: unsavorycaffine@bu.edu

    - Using Keychain Access



▼ Details

Subject Name	
Country or Region	US
County	MA
Locality	Boston
Organisation	BostonUniversityCS558
Organisational Unit	CS558
Common Name	cs558.bu.edu
Email Address	unsavorycaffine@bu.edu

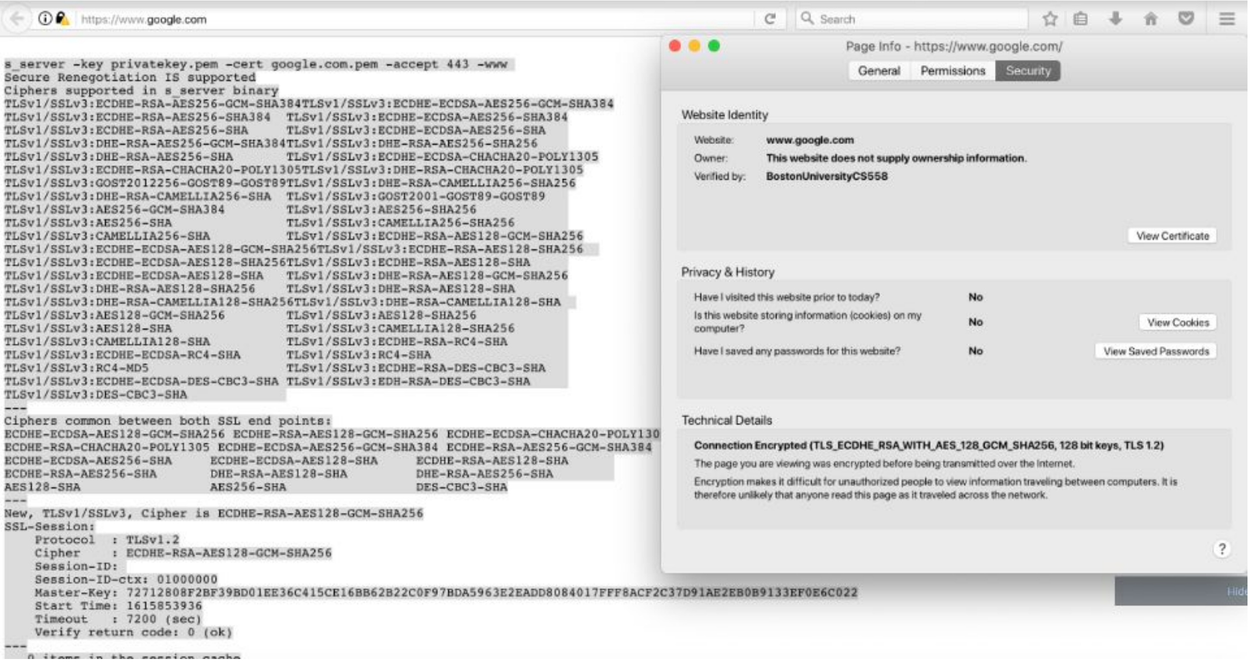
- Step 3: Choose a domain and generate a certificate signing request
    - First domain that comes to mind is google.com
    - How to generate a certificate signing request with a pre-existing private key:
        - https://gist.github.com/Soarez/9688998
    - Command used: `openssl req -new -key privatekey.pem -out google.com.csr`
    - Terminal prompts us to enter a Distinguished Name (DN), which consists of some parameters. We'll be filling this out for the domain "google.com" following the examples here: https://gist.github.com/Soarez/9688998 ('.' indicates blank)
        - Country Name (2 letter code) [AU]:US
        - State or Province Name (full name) [Some-State]:Massachusetts
        - Locality Name (eg, city) []:Boston
        - Organization Name (eg, company) [Internet Widgits Pty Ltd]:Google
        - Organizational Unit Name (eg, section) []:.
        - Common Name (e.g. server FQDN or YOUR name) []:google.com
        - Email Address []:.
        - A challenge password []:punked
        - An optional company name []:Google
    - Now we have generated a .csr file.
- Step 4: Sign this CSR using the cs558 key
    - Following this guide again: https://gist.github.com/Soarez/9688998
    - Made a .crt file with the certificate given in the assignment document
    - Assuming we are making a X.509 certificate
    - Used this command: `openssl x509 -req -in google.com.csr -CA cs558.crt -CAkey privatekey.pem -CAcreateserial -out google.com.crt`
- Step 5: Checking that certificate is valid with respect to CS558 CA certificate
    - Reference: https://stackoverflow.com/questions/25482199/verify-a-certificate-chain-using-openssl-verify
        - We used this command to verify that our certificate has its anchor by the certificate provided for us in the assignment document: `openssl verify -verbose -CAfile cs558.crt google.com.crt`
        - The output we got was: `google.com.crt: OK`
        - So we know that we have verified the certificate successfully.

    - Reference: https://support.acquia.com/hc/en-us/articles/360004119234-Verifying-the-validity-of-an-SSL-certificate
        - According to this, apparently if we run these two commands:

            -
              ```
              openssl x509 -noout -modulus -in certificate.pem | openssl md5
              ```

            -
              ```
              openssl rsa -noout -modulus -in ssl.key | openssl md5
              ```
        - And get the same output, it means that we have verified that the private key and server certificate match.
            - Successfully received the same output:
              ```
              (stdin) = 74abfe17885e61922cf907ff175d8621
              ```
- Step 6: Setting up a local TLS server of some kind
    - From the assignment document: “You want this server to generate TLS connections (as the server) using the certificates and server key that you have generated in the previous steps. It can serve any arbitrary html content.”
    - We’ll be trying out OpenSSL’s S_Server tool.
        - Reference: https://www.openssl.org/docs/man1.0.2/man1/openssl-s_server.html
        - There are a lot of parameters to enter, but the only one’s we’ll be putting in for now are:
            - -cert certname
                - Where certname is the name of the certificate we produced for google.com
                - Converted the certificate to PEM using this command:
                  ```
                  openssl x509 -in google.com.crt -out google.com.pem -outform PEM
                  ```
            - -key keyfile
                - Where keyfile is the private key from the assignment
            - -www
                - To simulate HTTPS server
        - Command used:
          ```
          openssl s_server -key privatekey.pem -cert google.com.pem -accept 443 -www
          ```
        - Terminal says “ACCEPT”. We think this means we should be good to go!
- Step 7: Setting up a proxy
    - Since we are working with MAC OS, we’ll try to make changes in System Preferences so that we’ll redirect requests to our local webserver
    - System Preferences → Network → Advanced → Proxies
        - Not too sure how exactly to set this up, but first we’ll go with this: https://www.howtogeek.com/293444/how-to-configure-a-proxy-server-on-a-mac/
        - And set up according to the instructions there
            - More details in Step 9

- Step 8: Add cs558 certificate to appropriate keystore
  - Reference: https://support.apple.com/en-gb/guide/keychain-access/kyca2431/mac
  - Added cs558 certificate to Systems keystore, as instructed
  - Trusted the certificate (had to manually verify this certificate in order for Keychain Access to allow us to do this).
- Step 9: Connect to target domain
  - Went on to MacOS Network Preferences -> Advanced -> Proxies. Enabled Auto Proxy Discovery on the protocols configurations, and then added https://www.google.com to the list of proxy settings to bypass.
  - Using GNU Nano, added the localhost and https://www.google.com to /private/etc/hosts
  - Cleared MacOS DNS Cache and also cleared Firefox's cache in order to avoid reconnecting to a previously cached version of Google Chrome
  - Started up the SSL server using openssl s_server -key privatekey.pem -cert google.com.pem -accept 443 -www
  - Success:



```
s_server -key privatekey.pem -cert google.com.pem -accept 443 -www
Secure Renegotiation IS supported
Ciphers supported in s_server binary
TLSv1/SSLv3:ECDHE-RSA-AES256-GCM-SHA384TLSv1/SSLv3:ECDHE-ECDSA-AES256-GCM-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA384  TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA     TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA
TLSv1/SSLv3:DHE-RSA-AES256-GCM-SHA384TLSv1/SSLv3:DHE-RSA-AES256-SHA256
TLSv1/SSLv3:DHE-RSA-AES256-SHA       TLSv1/SSLv3:ECDHE-ECDSA-CHACHA20-POLY1305
TLSv1/SSLv3:ECDHE-RSA-CHACHA20-POLY1305TLSv1/SSLv3:DHE-RSA-CHACHA20-POLY1305
TLSv1/SSLv3:GOST2012256-GOST89-GOST89TLSv1/SSLv3:DHE-RSA-CAMELLIA256-SHA256
TLSv1/SSLv3:DHE-RSA-CAMELLIA256-SHA  TLSv1/SSLv3:GOST2001-GOST89-GOST89
TLSv1/SSLv3:AES256-GCM-SHA384        TLSv1/SSLv3:AES256-SHA256
TLSv1/SSLv3:AES256-SHA               TLSv1/SSLv3:CAMELLIA256-SHA256
TLSv1/SSLv3:CAMELLIA256-SHA          TLSv1/SSLv3:ECDHE-RSA-AES128-GCM-SHA256
TLSv1/SSLv3:ECDHE-ECDSA-AES128-GCM-SHA256TLSv1/SSLv3:ECDHE-RSA-AES128-SHA256
TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA256TLSv1/SSLv3:ECDHE-RSA-AES128-SHA
TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA   TLSv1/SSLv3:DHE-RSA-AES128-GCM-SHA256
TLSv1/SSLv3:DHE-RSA-AES128-SHA256    TLSv1/SSLv3:DHE-RSA-AES128-SHA
TLSv1/SSLv3:DHE-RSA-CAMELLIA128-SHA256TLSv1/SSLv3:DHE-RSA-CAMELLIA128-SHA
TLSv1/SSLv3:AES128-GCM-SHA256        TLSv1/SSLv3:AES128-SHA256
TLSv1/SSLv3:AES128-SHA               TLSv1/SSLv3:CAMELLIA128-SHA256
TLSv1/SSLv3:CAMELLIA128-SHA          TLSv1/SSLv3:ECDHE-RSA-RC4-SHA
TLSv1/SSLv3:ECDHE-ECDSA-RC4-SHA      TLSv1/SSLv3:RC4-SHA
TLSv1/SSLv3:RC4-MD5                  TLSv1/SSLv3:ECDHE-RSA-DES-CBC3-SHA
TLSv1/SSLv3:ECDHE-ECDSA-DES-CBC3-SHA TLSv1/SSLv3:EDH-RSA-DES-CBC3-SHA
TLSv1/SSLv3:DES-CBC3-SHA
---
Ciphers common between both SSL end points:
ECDHE-ECDSA-AES128-GCM-SHA256 ECDHE-RSA-AES128-GCM-SHA256 ECDHE-ECDSA-CHACHA20-POLY130
ECDHE-RSA-CHACHA20-POLY1305 ECDHE-ECDSA-AES256-GCM-SHA384 ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES256-SHA     ECDHE-ECDSA-AES128-SHA     ECDHE-RSA-AES128-SHA
ECDHE-RSA-AES256-SHA       DHE-RSA-AES128-SHA         DHE-RSA-AES256-SHA
AES128-SHA                 AES256-SHA                 DES-CBC3-SHA
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID:
    Session-ID-ctx: 01000000
    Master-Key: 72712808F2BF39BD01EE36C415CE16BB62B22C0F97BDA5963E2EADD8084017FFF8ACF2C37D91AE2EB0B9133EF0E6C022
    Start Time: 1615853936
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
   0 items in the session cache
```

Page Info - https://www.google.com/

Website Identity

Website: www.google.com
Owner: This website does not supply ownership information.
Verified by: BostonUniversityCS558

Privacy & History

Have I visited this website prior to today? No
Is this website storing information (cookies) on my computer? No
Have I saved any passwords for this website? No

Technical Details

Connection Encrypted (TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, 128 bit keys, TLS 1.2)

The page you are viewing was encrypted before being transmitted over the Internet.

Encryption makes it difficult for unauthorized people to view information traveling between computers. It is therefore unlikely that anyone read this page as it traveled across the network.