Mutual TLS Authentication (mTLS) De-Mystified

John Tucker Follow
Jun 13, 2020 · 7 min read

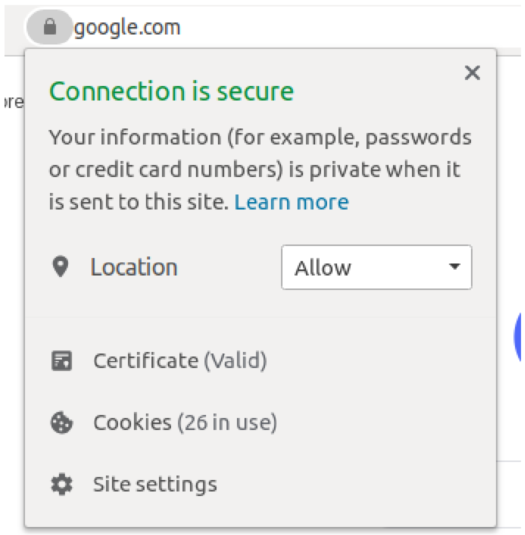
A walk-through of a simplified implementation of mTLS.

First, what is TLS?

> *Transport Layer Security (TLS), and its now-deprecated predecessor, Secure Sockets Layer (SSL),[1] are cryptographic protocols designed to provide communications security over a computer network.[2] Several versions of the protocols find widespread use in applications such as web browsing, email, instant messaging, and voice over IP (VoIP). Websites can use TLS to secure all communications between their servers and web browsers.*

— Wikipedia — Transport Layer Security

Yes, it is the mechanism by which our web browsers create secure connections to web servers. Just click on the lock in your browser's address bar when visiting most any web site and you will get an informational popup.



google.com

Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more

Location: Allow

Certificate (Valid)

Cookies (26 in use)

Site settings

At the heart of TLS is Public Key Infrastructure (PKI) and in particular X.509 certificates.

> *In cryptography, X.509 is a standard defining the format of public key certificates.[1] X.509 certificates are used in many Internet protocols, including TLS/SSL, which is the basis for HTTPS[2], the secure protocol for browsing the web. They are also used in offline applications, like electronic signatures. An X.509 certificate contains a public key and an identity (a hostname, or an organization, or an individual), and is either signed by a certificate authority or self-signed. When a certificate is signed by a trusted certificate authority, or validated by other means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.*

— Wikipedia — X.509

To inspect a X.509 certificate, click on the *Certificate* entry in the informational popup (shown when we clicked on the lock above).

So then, what is mTLS?

> *By default the TLS protocol only proves the identity of the server to the client using X.509 certificate and the authentication of the client to the server is left to the application layer. TLS also offers client-to-server authentication using client-side X.509 authentication.[1] As it requires provisioning of the certificates to the clients and involves less user-friendly experience, it's rarely used in end-user applications.*
>
> *Mutual TLS authentication (mTLS) is much more widespread in business-to-business (B2B) applications, where a limited number of programmatic and homogeneous clients are connecting to specific web services, the operational burden is limited, and security requirements are usually much higher as compared to consumer environments.*

— Wikipedia — Mutual authentication

With all this in mind, let us walk through a mTLS example of using the cURL web browser (the client) to connect to a Node.js web server (the server) serving on the DNS name *localhost*. In doing so:

- The client will validate that the server is trusted to serve up content for the DNS name *localhost*
- The server will validate the client is known, i.e., it will authenticate it

The first step is to create a certificate authority (CA) that both the client and server trust. The CA is just a public and private key with the public key wrapped up in a self-signed X.509 certificate. The command we use to do this is:

```
openssl req \
  -new \
  -x509 \
  -nodes \
  -days 365 \
  -subj '/CN=my-ca' \
  -keyout ca.key \
  -out ca.crt
```

This outputs two files, *ca.key* and *ca.crt,* in the PEM format (base64 encoding of the private key and X.509 certificate respectively).

Looking at the *openssl req* documentation, we see that the *-new* and *-x509* options enable the creation of a self-signed root CA X.509 certificate. The *nodes* (No DES) option disables securing the private key with a pass-code; this option is optional. The *subj* option provides the CA's identity; in this case the Common Name (CN) of *my-ca*. The remaining options are self-explanatory.

We can turn-around and inspect the certificate using the following command:

```
openssl x509 \
  --in ca.crt \
  -text \
  --noout
```

The options here are self-explanatory as documented in the *openssl x509* documentation.

Looking at the output, we can confirm a number of things:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            [OBMITTED]
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = my-ca
        Validity
            Not Before: Jun 13 00:49:48 2020 GMT
            Not After : Jun 13 00:49:48 2021 GMT
        Subject: CN = my-ca
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    [OBMITTED]
                Exponent: [OBMITTED]
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                [OBMITTED]
            X509v3 Authority Key Identifier:
                keyid:[OBMITTED]

            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         [OBMITTED]
```

- Both the *Subject* and *Issuer* have the value *CN = my-ca;* this indicates that this certificate is self-signed
- The *Validity* indicates that the certificate is valid for a year
- The *X509v3 Basic Constraints* value *CA:TRUE* indicate that this certificate can be used as a CA, i.e., can be used to sign certificates

Next we create the server's key and certificate; starting with the key:

```
openssl genrsa \
  -out server.key 2048
```

The options here are self-explanatory as documented in the *openssl genrsa* documentation.

Remember, our goal here is to create a server certificate for the DNS name *localhost* signed by the CA. We now create a Certificate Signing Request (CSR) with the Common Name (CN) *localhost*:

```
openssl req \
  -new \
  -key server.key \
  -subj '/CN=localhost' \
  -out server.csr
```

Using the CSR, the CA (really using the CA key and certificate) creates the signed certificate:

```
openssl x509 \
  -req \
  -in server.csr \
  -CA ca.crt \
  -CAkey ca.key \
  -CAcreateserial \
  -days 365 \
  -out server.crt
```

The output is the signed server certificate, *server.crt*, in the PEM format.

Most of the options are either familiar (from above) or self-explanatory as documented in the *openssl x509* documentation. The one exception is the *CAcreateserial* option that manages a newly created file, *ca.srl*, that enables each certificate created by this CA to have a unique serial number.

As before we inspect the certificate using the following command:

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

Looking at the output, we can confirm a number of things:

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            [OBMITTED]
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = my-ca
        Validity
            Not Before: Jun 13 00:50:18 2020 GMT
            Not After : Jun 13 00:50:18 2021 GMT
        Subject: CN = localhost
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    [OBMITTED]
                Exponent: [OBMITTED]
    Signature Algorithm: sha256WithRSAEncryption
         [OBMITTED]
```

- The *Issuer* has the value *CN = my-ca;* this indicates that this certificate is signed by the *my-ca* certificate authority
- The *Validity* indicates that the certificate is valid for a year
- The *Subject* has the value *CN = localhost*; this indicates that this certificate can be served to a client to validate that the server is trusted to serve up content for the DNS name *localhost*

We essentially repeat the process to create the client's key and certificate; starting by creating the client's key:

```
openssl genrsa \
  -out client.key 2048
```

Creating the CSR with the arbitrary Common Name of *my-client:*

```
openssl req \
  -new \
  -key client.key \
  -subj '/CN=my-client' \
  -out client.csr
```

And finally the creating the client's certificate:

```
openssl x509 \
  -req \
  -in client.csr \
  -CA ca.crt \
  -CAkey ca.key \
  -CAcreateserial \
  -days 365 \
  -out client.crt
```

note: If you inspect this certificate, you will observe that the *Serial Number* is indeed different than the server's certificate.

One observation is that both the server and client certificates are simpler X.509 v1 certificates; the CA certificate however is a X.509 v3 certificate. This is because OpenSSL automatically creates X.508 v3 self-signed certificates (CA certificate) and we did not supply any v3 extensions when signing the server and client certificates (using the *extfile* and *extensions* options).

note: The use of various X.509 v3 extensions is outside the scope of this article; besides I have not found any simple explanations of them myself.

With all of our keys and certificates (ca, server, and client) created we can configure our server and client.

The server is the basic Hello World example, provided by Node.js, enhanced to support mTLS.

```
const https = require('https');
const fs = require('fs');

const hostname = 'localhost';
const port = 3000;

const options = {
    ca: fs.readFileSync('ca.crt'),
    cert: fs.readFileSync('server.crt'),
    key: fs.readFileSync('server.key'),
    rejectUnauthorized: true,
    requestCert: true,
};

const server = https.createServer(options, (req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

index.js hosted with ❤ by GitHub view raw

Here the *requestCert*, *rejectUnauthorized*, and *ca* options are used to require the browser (client) to supply a certificate signed by the CA certificate to interact with the server.

The *key* and *cert* options enable the server to serve up the CA signed server certificate.

The client is simply the cURL web browser with options:

```
curl \
  --cacert ca.crt \
  --key client.key \
  --cert client.crt \
  https://localhost:3000
```

Here the *cacert* option is used so that the client (cURL) can validate the server supplied certificate. The *key* and *cert* are used so the client sends the CA signed client certificate with the request.

Indeed, we observe that this request successfully returns *hello world*. If, however, we leave off the *cacert* option, we get the error:

```
curl --key client.key --cert client.crt  https://localhost:3000
curl: (60) SSL certificate problem: self signed certificate in
```

```
certificate chain
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore
could not
establish a secure connection to it. To learn more about this
situation and
how to fix it, please visit the web page mentioned above.
```

On the other hand, if we leave off the key and cert options, we get a different error:

```
curl --cacert ca.crt https://localhost:3000
curl: (56) OpenSSL SSL_read: error:1409445C:SSL
routines:ssl3_read_bytes:tlsv13 alert certificate required, errno 0
```

Wrap Up

Nothing spectacularly new here. At the same time, hope you learned something.

Tls Web Development Security