Module 7.3 – HTTPS Servers

Refer to Module 7.1 and 7.2 for a review of HTTPS/TLS, and OpenSSL.

Module 7.1 Required Reading Material:

- [1] Lewis Van Winkle, “Hands-On Network Programming with C". Packt Publishing. May 2019. ISBN: 9781789349863. https://learning.oreilly.com/library/view/hands-on-network-programming/9781789349863/
    - Read chapter 10.

- [2] Brandon Rhodes, John Goerzen, “Foundations of Python Network Programming, Third Edition”. Apress. August 2014. https://learning.oreilly.com/library/view/foundations-of-python/9781430258551/
    - Read chapter 6.

https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning

https://security.stackexchange.com/questions/20803/how-does-ssl-tls-work

https://harshityadav.in/posts/What-is-TLS-SSL/

This module covers HTTPS/TLS from the perspective of a server.

Topics covered in this module:

- HTTPS overview
- HTTPS certificates
- HTTPS server setup with OpenSSL
- Accepting HTTPS connections
- Common problems
- OpenSSL alternatives
- Direct TLS termination alternatives

How are HTTPS Servers different from HTTPS clients?

* Unlike HTTPS clients, HTTPS servers are expected to identify themselves with certificates.

Recall, HTTPS connections are first made using TCP sockets. Once the TCP connection is established, OpenSSL is used to negotiate a TLS connection over the open TCP connection. From that point forward, OpenSSL functions are used to send and receive data over the TLS connection.

Overview of an HTTPS session setup:

1. Client makes DNS request for the hostname/IP address of server.

2. TCP connection is established: 3-way handshake.

3. TLS/HTTPS protocol initiation takes over.

4. Client (TLS) requests a certificate from the server; The certificate includes a public key.

5. Server responds with the requested information.

6. Client verifies identify of the server by verifying validity of the certificate.

7. If everything checks out, HTTPS session is completed, and secure communication can commence.

8. TLS session is torn down.

9. TCP connection is closed: 4-way close-down process.

For an in-depth explanation of TLS Certificates read:

https://harshityadav.in/posts/What-is-TLS-SSL/

Foundations of Python Programming. Chapter 6, pages: 94-68. and

Hands-On Network Programming in C - Packt. By Lewis Van Winkle Chapter 10, pages: 284-288.

What is the purpose of a certificate in the context of TLS?

* While encryption protects the contents of the data, it does not prevent against machine-in-the-middle attacks, eavesdroppers, or connecting to impostors.

* Certificates are used to prove the server's identity, and the client's if desired (though not common). Without a certificate, a client wouldn't be able to trust whether it was connected to the intended server or an impostor.

How are certificates used to provide proof of identity?

* TLS needs to verify both to whom you are talking and how you and the peer to whom you are speaking will protect data from prying eyes.

Certificate Authorities are entities that provide validation of an entity's legitimacy. Entities, servers, and companies looking to host secured servers using HTTPS/TLS will typically pay a fee in order for a CA to validate and digitally sign their certificates.

TLS clients have a few certificate authorities that they explicitly trust, using the CA they are able to validate the identity of a server before establishing a secure connection.

Client certificate verification process:

The TLS client will request a certificate from the server to prove its identity.

* Certificates contain a public key, among other data, which can be used to encrypt data. Only the peer with the private key can decrypt the data encoded by this private key.

* Client encrypts sends some information across the wire to the server, this information has been encrypted with the public key which can only be decrypted by the legitimate server holding the private key (MiTM attacks aside).

* Server must proof decryption/identify by providing a checksum demonstrating that the data was decrypted successfully with the secret key.

How does the client trust that the certificate is legitimate?

Certificate Authorities (CA)

* Contain a signature to demonstrate the CA's approval of a given certificate.

* TLS verifies the validity of the certificate's signature before accepting that the certificate is valid.

* TLS checks the CA's public key to perform validation.

Types of Validation:

* Domain validation:

* Where a signed certificate is issued after simply verifying that the certificate recipient can be reached at the given domain.

* Most common type of validation.

* Extended Validation:

* Issued after the CA verifies the recipient's identity; usually done using public records and a phone call.

* Self-Signed certificates:

* Typically used for private applications not exposed to the internet and where no public/untrusted systems will be accessing the server.

We will require a self-signed certificate for our HTTPS server example. These are the easiest to obtain, especially for test and development environment. We will use our self-signed certs to explore the different fields and values of a certificate.

Creating a self-signed cert: (Can use default values.)

#Run the following command on a terminal:

openssl req -x509 -newkey rsa:2048 -nodes -sha256 -keyout key.pem -out cert.pem -days 365

The new certificate is placed in cert.pem and the key for it in key.pem

The server will require both files:

* cert.pem: is the certificate that gets sent to the connected client.

* key.pem: provides our server with the encryption key that proves that it owns the certificate.

* SHOULD BE KEPT SECRET in a production environment.

#View certificate

openssl x509 -text -noout -in cert.pem

#view key

openssl rsa -text -noout -in key.pem

HTTPS and OpenSSL Functions for server programming:

OpenSSL API:

Simlar to module 7.2, before using OpenSSL in your program you need to initialize it, load the supported algorithms, and SSL error support functions.

Refer to module 7.2 for a full description of each of the functions.

Step 1: Initialize OpenSSL

```
SSL_library_init();
OpenSSL_add_all_algorithms();
SSL_load_error_strings();
```

Step 2: Create an SSL context

Once OpenSSL has been initialized, we are ready to create an SSL context.

Note that unlike the HTTP client program in module 7.2, we use TLS_server_method() instead of TLS_client_method() when creating a new context.

```
SSL_CTX *ctx = SSL_CTX_new(TLS_server_method());
if (!ctx)
{
   fprintf(stderr, "SSL_CTX_new() failed.\n");
   return 1;
}
```

Step 3: Specify certificate and key to be used. Note: This step differs from the previous module.

```
if (!SSL_CTX_use_certificate_file(ctx, "cert.pem" , SSL_FILETYPE_PEM)||
!SSL_CTX_use_PrivateKey_file(ctx,"key.pem", SSL_FILETYPE_PEM))
{
   fprintf(stderr, "SSL_CTX_use_certificate_file() failed.\n");
```

```
    ERR_print_errors_fp(stderr);
    return 1;
}
```

SSL_CTX_use_certificate_file()

https://www.openssl.org/docs/man3.1/man3/SSL_CTX_use_certificate_file.html

int SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char *file, int type);

Loads the first certificate stored in file into ctx. The formatting type of the certificate must be specified from the known types of SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1.

Params:

See man page.

Returns:

On success, the functions return 1. Otherwise check out the error stack to find out the reason.

SSL_CTX_use_PrivateKey_file()

https://www.openssl.org/docs/man3.1/man3/SSL_CTX_use_PrivateKey_file.html

int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char *file, int type)

Adds the first private key found in file to ctx. The formatting type of the private key must be specified from the known types of SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1.

Params:

See man page.

Returns:

On success, the functions return 1. Otherwise check out the error stack to find out the reason.

That concludes the minimal OpenSSL setup needed for an HTTPS server. The rest of the steps follow the same pattern as the one covered in module 7.2, revisit that section for a refresher.

Step 4: Create a TCP connection.

After initializing OpenSSL and creating an SSL context, we proceed to create a TCP connection in the normal way (refer to Module 2):

getaddrinfo()

socket()

connect()

Step 5: Initiate a TLS connection.

After a new TCP connection is established, we use the socket returned by accept() to create our TLS/SSL socket.

Note that unlike the HTTP client program in module 7.2, we do not need to set the hostname of the host when writing a server program.

```
SSL *ssl = SSL_new(ctx);
if (!ctx)
{
    fprintf(stderr, "SSL_new() failed.\n");
    return 1;
}
```

The SSL object is then linked to our open TCP socket using SSL_set_fd().

Note that unlike the HTTP client program in module 7.2, we use SSL_accept() instead of SSL_connect.

```
SSL_set_fd(ssl, socket_client);
if (SSL_accept(ssl) <= 0)
{
   fprintf(stderr, "SSL_accept() failed.\n");
   ERR_print_errors_fp(stderr);
   return 1;
}
printf ("SSL connection using %s\n", SSL_get_cipher(ssl));
```

Step 6: Send/Receive Data using SSL_write() and SSL_read(). See module 7.2

Step 7: Tear down SSL connection, TCP socket, and free SSL resources.

```
SSL_shutdown(ssl);
close(socket);
SSL_free(ssl);
```

Step 8: free SSL_CTX.

```
SSL_CTX_free(ctx)
```

We've now covered all the basic APIs needed to write an HTTPS server. See module 7.2 for a refresher of the functions covered here.

See tls_time_server.c for a sample programming implementing the functions discussed.