HTTPs-TLS Attacks

Introduction to HTTPS/TLS

The Hypertext Transfer Protocol (HTTP) is an application layer protocol used to access resources on the internet. Since HTTP transmits data in plaintext, it cannot provide confidentiality, integrity, or authenticity of the transmitted data. To overcome these shortcomings of HTTP, the Hypertext Transfer Protocol Secure (HTTPS), also called `HTTP over TLS` was created. The Transport Layer Security (TLS) protocol and its predecessor, the Secure Sockets Layer (SSL) protocol, are cryptographic protocols that provide secure communication over the internet by encrypting traffic.

Encryption can generally be applied at different levels. These include `encryption-at-rest`, `encryption-in-transit`, and `end-to-end encryption`. `Encryption-at-rest` means that data is stored in an encrypted form to prevent unauthorized access. An example would be hard drive encryption. When `encryption-in-transit` is applied, data that is transmitted is encrypted before transmission and decrypted after reception to prevent unauthorized access during the transmission. This module focuses on `encryption-in-transit` since TLS applies `encryption-in-transit`. Lastly, `end-to-end encryption` encrypts data from the true sender to the final recipient such that no other party can access the data.

To illustrate the difference to `encryption-in-transit`, consider Alice who wants to send an email to Bob. If they use `end-to-end encryption`, Alice encrypts the e-mail and sends it to Bob who decrypts it to access the e-mail. No intermediary servers that the encrypted e-mail is routed over can access it. When TLS and thus `encryption-in-transit` is used, Alice encrypts the e-mail and sends it to her mail server, which decrypts it, and re-encrypts it to forward it to the next server, and so on until the final server sends it to Bob. This protects the email from any unauthorized access during transit but all intermediary servers can access the e-mail in plaintext, while only Alice and Bob can access the e-mail if `end-to-end encryption` is used.

The main purpose of this module is to provide insights into web cryptography protocols, how they work, and what vulnerabilities can arise when using them. Generally speaking, finding vulnerabilities in protocols is more challenging compared to finding vulnerabilities in individual web applications. That is because protocols such as HTTPS and TLS have been designed with security in mind and revised multiple times to tackle potential security issues. However, if there are security issues in protocols, the impact is generally much higher as well since a huge number of services are affected. Oftentimes, security issues on HTTPS or TLS are not specification flaws but implementation flaws. That means that specific implementations of the

protocol do not implement the protocol correctly or deviate slightly which can create security issues.

TLS Overview and Version History

What is TLS?

TLS and before it SSL are widely used to secure communication on the internet, including email, file transfer, and web browsing. TLS was developed to address the weaknesses in SSL and has undergone several revisions over the years, each of which has introduced new features and improvements to the protocol. Today, TLS is the standard protocol for secure communication on the internet.

In the network protocol stack, TLS sits between TCP and the application layer, which can be any application layer protocol such as HTTP, SMTP, or FTP. TLS is transparent for the application layer protocol, meaning the application layer protocol does not need to know if TLS is implemented or not. In particular, TLS takes care of all cryptographic operations, the application layer protocol can operate the same regardless of whether TLS is used or not.

Version History

SSL was first developed by Netscape in the mid-1990s as a way to secure communication over the internet. It quickly became the standard protocol for secure communication and was widely adopted by web browsers and servers. There are three major versions of SSL:

- SSL 1.0: This was the initial version of SSL. It was never released to the public due to serious security flaws.
- SSL 2.0: This was the first SSL version that became widely used. It was released in 1995. However, it suffered from multiple serious specification flaws that made it impractical to use in some cases and susceptible to cryptographic attacks.
- SSL 3.0: This was the last version of SSL. It is a full redesign of the 2.0 version that fixed the specification flaws. However, from today's perspective, it relies on deprecated cryptographic algorithms and is vulnerable to a variety of attacks.

In response to weaknesses in SSL, the TLS protocol was developed to replace it. TLS was designed to address the vulnerabilities in SSL and to provide stronger encryption and authentication for secure communication. Like SSL, TLS has undergone several revisions, each of which has introduced new features and improvements to the protocol. Some of the key versions of TLS include:

- TLS 1.0: This was the first version of TLS and was released in 1999. It was based on SSL 3.0 and included many of the same features as SSL, but with additional security enhancements.
- TLS 1.1: This version of TLS was released in 2006 and introduced several important improvements to the protocol, including support for new cryptographic algorithms and protection against attacks such as man-in-the-middle attacks (aka `On-Path Attacks`).
- TLS 1.2: This version of TLS was released in 2008 and introduced further security enhancements, including support for stronger cryptographic algorithms and better protection against attacks. It also introduced new features such as the ability to negotiate the use of compression during the handshake process.
- TLS 1.3: This is the latest version of TLS, released in 2018. It includes significant improvements to the protocol, including faster performance, stronger encryption, and better protection against attacks. It also includes a simplified handshake process and the ability to negotiate the use of encryption during the handshake process.

In this module, we will discuss attacks that broke certain SSL/TLS protocol versions completely, including SSL 2.0 and SSL 3.0.

What is HTTPS?

Now that we have a basic understanding of what TLS is, let's discuss how TLS relates to HTTPS. HTTPS works the same as HTTP, however in HTTPS, TLS is contained in the protocol stack. That means HTTPS traffic is encrypted and integrity protected thus preventing attackers from eavesdropping on or manipulating data. While HTTP uses the protocol scheme `http://` and targets port 80 by default, HTTPS uses `https://` and targets port 443. Although there are different HTTP versions, HTTPS only means that the HTTP traffic is encapsulated in TLS. Thus, there are no dedicated HTTPS versions.

Introduction to TLS Attacks

The Transport Layer Security (TLS) protocol and its predecessor, the Secure Sockets Layer (SSL) protocol, are cryptographic protocols that provide secure communication over the internet. TLS protects the confidentiality, integrity, and authenticity of transmitted data. To provide these security services, TLS utilizes a combination of cryptographic algorithms such as symmetric encryption, asymmetric encryption, and Message Authentication Codes (MACs).

In this module, we will take a closer look at TLS to gain a broad understanding of how TLS works and what things to look out for when testing TLS configurations. We will discuss common TLS security vulnerabilities to understand what misconfiguration or bugs caused them. Finally,

we will discuss how to detect, exploit, and prevent each of these attacks as well as common misconfigurations regarding TLS servers.

Padding Oracle Attacks

The first type of TLS attacks discussed in this module are Padding Oracle attacks. Padding oracle attacks exploit vulnerable servers that leak information about the correctness of the padding after decrypting a received ciphertext. These attacks can enable an attacker to fully decrypt a ciphertext without knowledge of the encryption key. Examples of Padding Oracle attacks on TLS are the POODLE, DROWN, and Bleichenbacher attacks.

Compression Attacks

The second type of TLS attacks discussed in this module are compression attacks. Compression can be applied at the HTTP level or TLS level to increase the performance of data transmission. However, incorrectly configured servers can be exploited, resulting in the leakage of encrypted information such as session cookies or CSRF tokens. Examples of compression-based attacks on TLS are the CRIME and BREACH attacks.

Misc Attacks & Misconfigurations

The last type of TLS attacks discussed in this module are various other attacks that exploit misconfigurations or bugs. A famous example is the Heartbleed bug that exploits a missing length validation in the OpenSSL library, which can lead to a complete server takeover via private key leakage. We will also discuss different TLS misconfigurations that can weaken TLS security by using insecure cryptographic primitives.

Public Key Infrastructure

TLS utilizes both symmetric and asymmetric cryptography. Asymmetric cryptography typically relies on public key infrastructure. To fully understand how TLS works, we must therefore get a basic understanding of certain terminologies such as public key infrastructure, certificates, and certificate authorities.

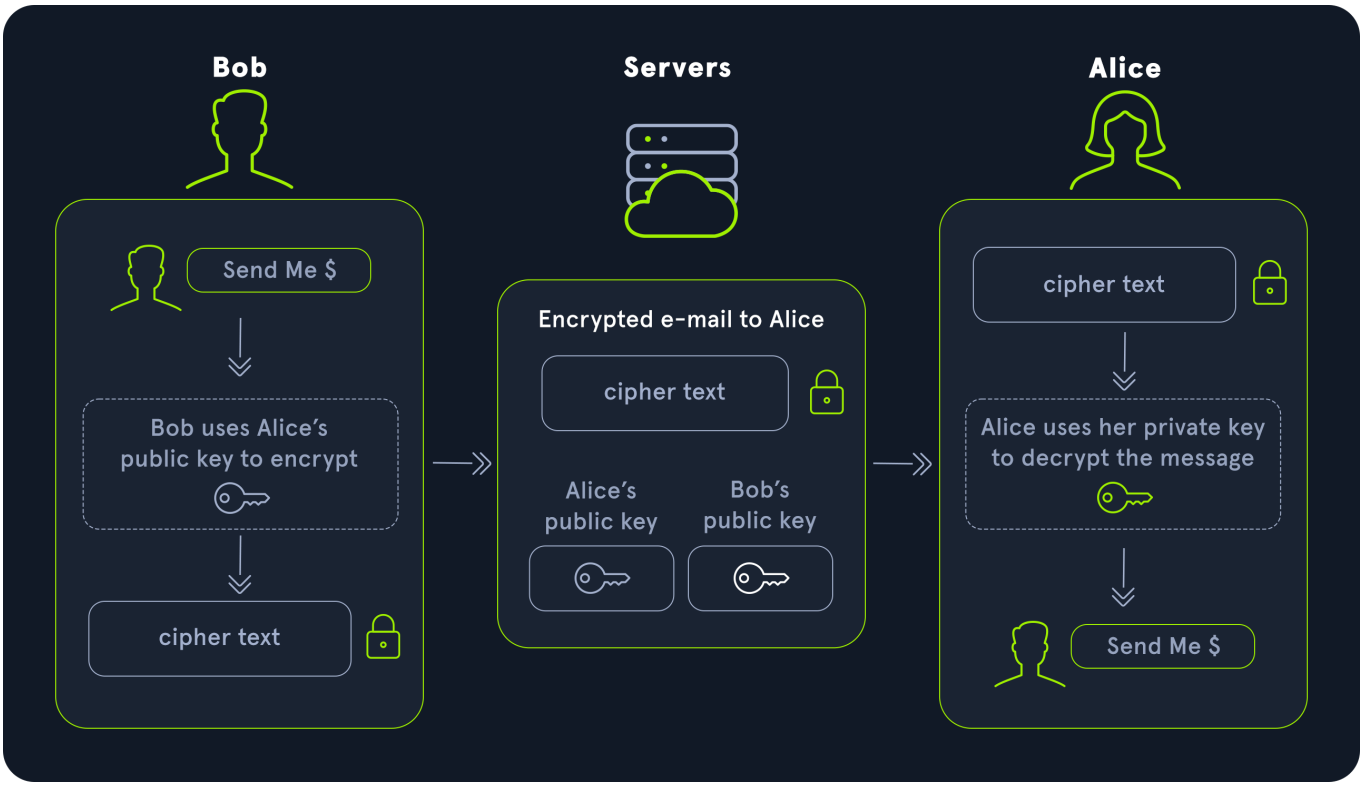
Public Key Infrastructure

A public key infrastructure (PKI) comprises roles and processes responsible for the management of digital certificates. This includes the distribution, creation, and revocation of certificates. Without a proper PKI, public key cryptography would be impractical.

In public key cryptography, the encryption key is different from the decryption key, which is why it is also called `asymmetric` encryption. Each participant owns a key pair consisting of a `public key` that is used for encryption, and a `private key` or `secret key` that is used for decryption. As its name suggests, the public key is public knowledge, thus everyone can use it to encrypt messages for the owner of the corresponding private key. Since messages encrypted with a public key can only be decrypted with the corresponding private key, only the intended receiver can decrypt the message. This inherently protects the messages from unauthorized actors. Here is an overview of commonly used encryption algorithms and their type:

Algorithm	Type
RSA	asymmetric
DSA	asymmetric
AES	symmetric
DES	symmetric
3DES	symmetric
Blowfish	symmetric

However, there is a conceptual problem that comes from the acquisition of other actors' public keys since it is impossible to verify their validity. For instance, consider an example where the user `Alice` wants to communicate with `hackthebox.com` privately. To do so, she obtains the public key of `hackthebox.com`, encrypts her message with it, and sends it to the target. Since only `hackthebox.com` knows the corresponding private key for decryption, the message cannot be decrypted by unauthorized actors. But how can Alice know that the public key actually belongs to `hackthebox.com` and not an attacker that wants to steal Alice's HackTheBox credentials? Assume an attacker intercepts Alice's request to obtain HackTheBox's public key and instead sends Alice his own public key while spoofing the origin to make Alice think it's actually HackTheBox's public key. She would then encrypt her message with the attacker's public key thinking it was HackTheBox's public key, enabling the attacker to decrypt and access the message. Certificates exist precisely to solve this problem.

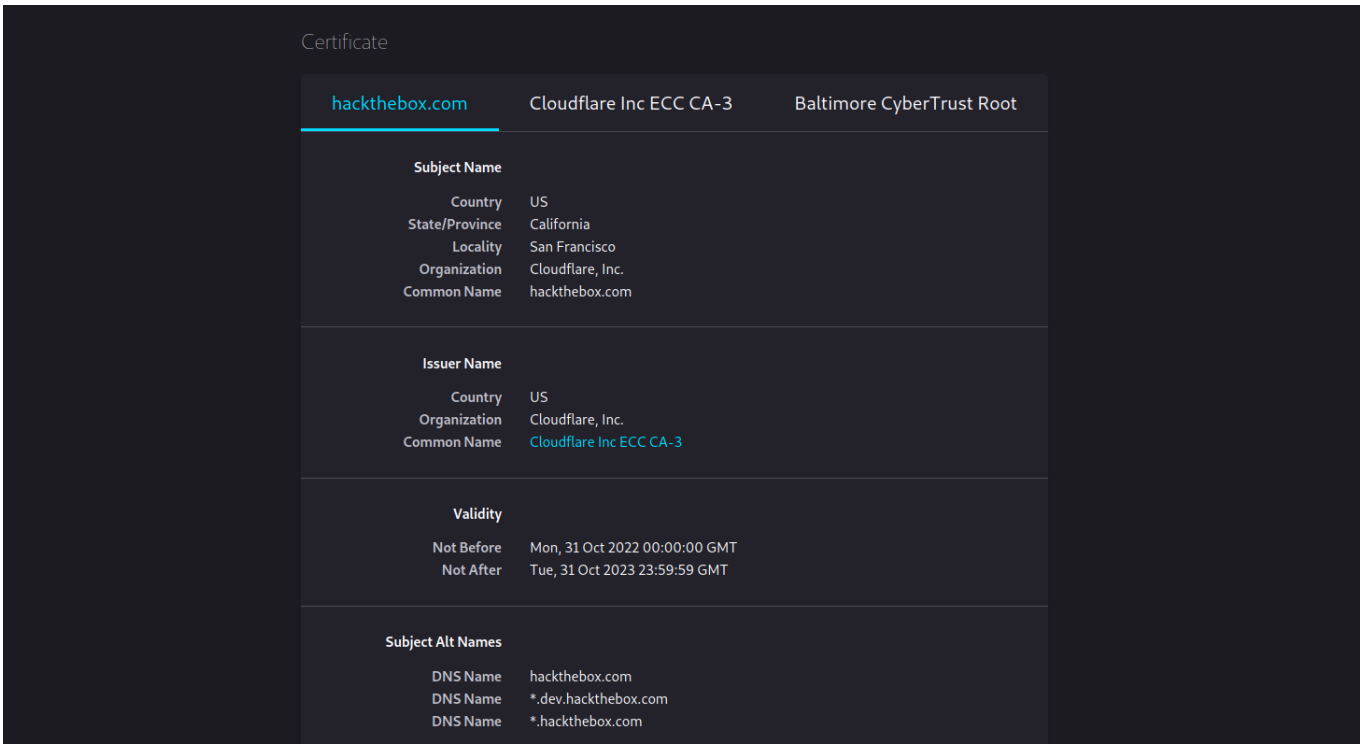


Certificates

The purpose of certificates is to bind public keys to an identity. This proves the identity of the public key owner and thus solves the previously discussed problem.

When accessing a website, we can check the certificate of the web server. In Firefox we can do this by clicking on the lock next to the URL bar, and then `Connection Secure > More Information > View Certificate`.

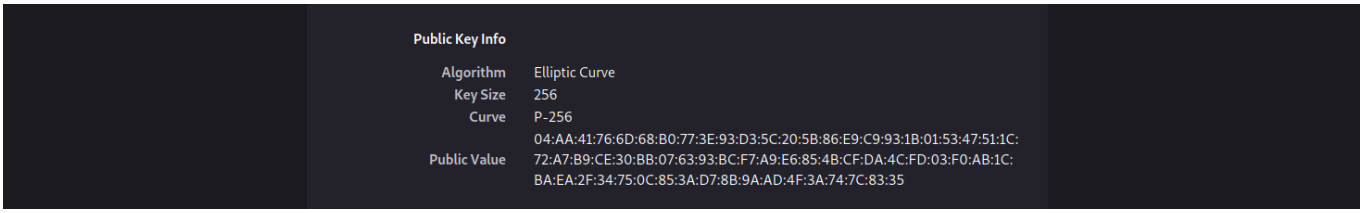
Let's have a look at the contents of the certificate for `hackthebox.com`:



The certificate contains information about the subject. Most importantly the `Common Name`, which is the domain name the public key belongs to. Additionally, each certificate has an expiry date and needs to be renewed before it expires to remain valid.

Note: Additional domain names can be specified in the `Subject Alt Names` section.

If we scroll down a bit, we can see that the certificate also contains the public key:



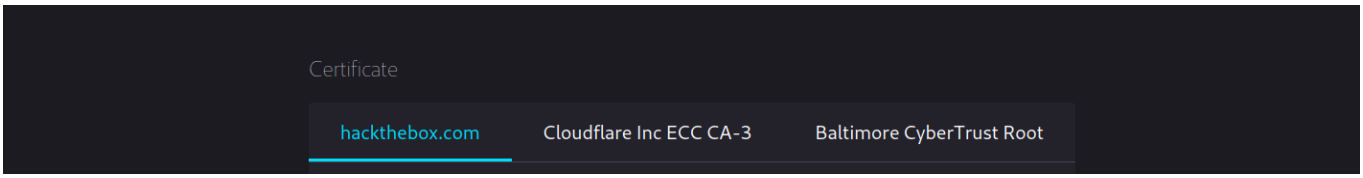
This certificate ensures that when we encrypt a message with the public key shown in the screenshot, only `HackTheBox` will be able to decrypt it.

So certificates are used to tie an identity to a public key. But who can issue certificates? And what prevents an attacker from just creating a certificate with his own public key and the domain `hackthebox.com`, thus impersonating `HackTheBox` with a forged certificate? That is where `Certificate Authorities` come into play.

Certificate Authorities

Certificate authorities (CAs) are entities that are explicitly allowed to issue certificates. They do this by cryptographically signing a certificate. The identity of the CA is proven by a `CA`

`Certificate`. Just like any other certificate, CA certificates are signed by another CA. This continues until a `root CA` is reached. The chain from the root CA to the end-user's certificate is called the `certificate chain`. When we look again at `HackTheBox`'s certificate, we can see the certificate chain consisting of three total certificates at the top:



When accessing a website, the browser validates the whole certificate chain. If any of the certificates contained in the chain are invalid or insecure, the browser displays a warning to the user. The `root CA`'s identity is checked against a hardcoded set of trusted CAs in the so-called `certificate store` to prevent forgery of root CA certificates.

OpenSSL

OpenSSL is a project that implements cryptographic algorithms for secure communication. Many Linux distributions rely on OpenSSL, making it essential for encrypted communication on the internet. Security vulnerabilities and bugs in OpenSSL lead to millions of affected web servers. We can use the OpenSSL client which is preinstalled on many Linux distributions to generate our own keys and certificates, convert them to different formats, and perform encryption.

Key Generation & Certificate Conversion

We can generate a new RSA key-pair with `2048` bit length and store it in a file using the following command:

```
mayala@htb[/htb] $ openssl genrsa -out key.pem 2048 Generating RSA private key, 2048 bit long modulus (2 primes) ..+++++ .....................................................................+++++ e is 65537 (0x010001)
```

When we cat the file, it shows us the private key:

```
mayala@htb[/htb] $ cat key.pem -----BEGIN RSA PRIVATE KEY----- MIIEowIBAAKCAQEAzCBnKqY7/6joFncQwuMfn9jRJmA4KX3rvAeN9/Zo4ItLWZ6q <SNIP> rfrwXh8ZgFAuDx75kxnuzKzzHg+uV2YiS2RMuid03qlW2iKHWUV8 -----END RSA PRIVATE KEY-----
```

We can use openssl to print out the public key as well:

mayala@htb[/htb] `$ openssl rsa -in key.pem -pubout writing RSA key -----BEGIN PUBLIC KEY----- MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzCBnKqY7/6joFncQwuMfn9jRJmA4KX3rvAeN9/Zo4ItLWZ6qnax1RXmAM986GVcTqIlBp7vzzYY3VulcYM/k78QwQr//nU3zKlPoMOhhhoeuq98tj76dmUYWl9gfyHyg3FIk0yWZuvVSOor5D0Rmr5PhCu5B7+EpbnbXcfRuyoPJqq78Bs9HOfg0HO++R7Ilpcr6t6WD/ftkr1zEXaWOcYhYCPdkpouCTsBe8QbRAy6B/E9kbENeRFdTkkX0cM5BSy4MKj9VezygK6kynzE6KKY1I8pGYChyfWjskbXbaJF9ocBnPAvzM2RzMrw1RhiAT8ErubuMqPYdRQS4RtHV GQIDAQAB -----END PUBLIC KEY----`

Furthermore, we can download the certificate of any web server:

mayala@htb[/htb] `$ openssl s_client -connect hackthebox.com:443 | openssl x509 > hackthebox.pem`

This stores the certificate in the `PEM` format. However, there are other formats such as `DER` and `PKCS#7`. We can convert from `PEM` to these formats using openssl:

mayala@htb[/htb] `# PEM to DER $ openssl x509 -outform der -in hackthebox.pem -out hackthebox.der # PEM to PKCS#7 $ openssl crl2pkcs7 -nocrl -certfile hackthebox.pem -out hackthebox.p7`

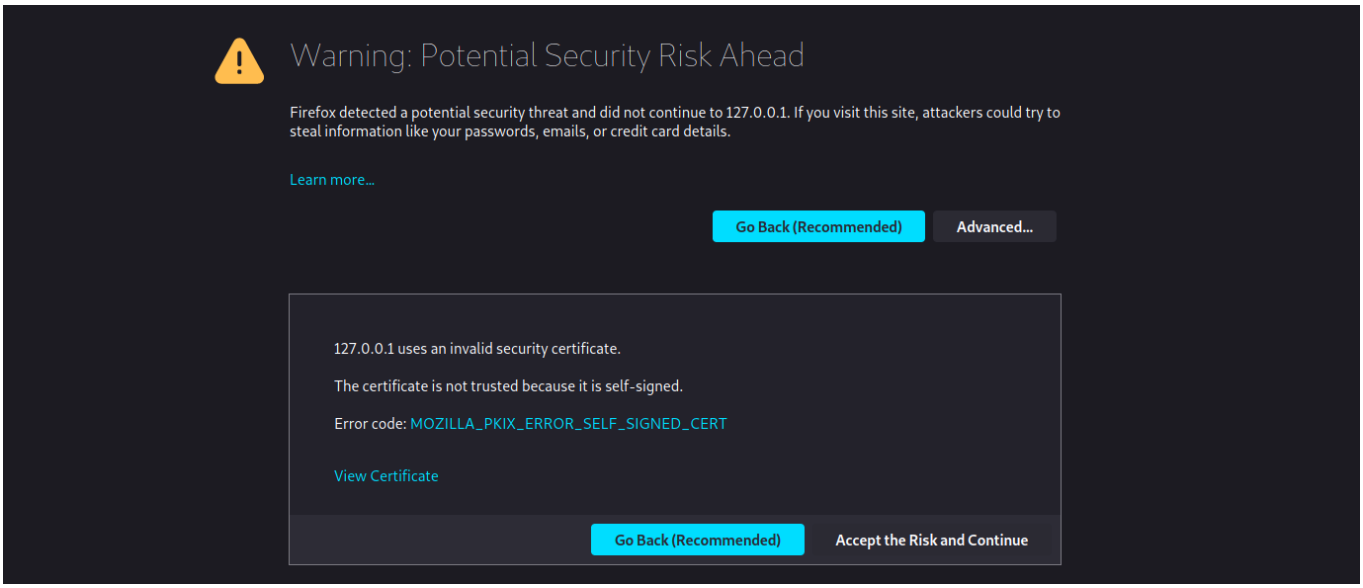
Creating a Self-Signed Certificate

Finally, we can create our own certificate and sign it ourselves. This means that the signature of a CA is not required. We can specify the type of key that is created, as well as the algorithm and expiry date of the certificate. We are also asked to enter a passphrase to protect the private key file and provide subject information for the certificate. We can provide any information we want, including impersonating `HackTheBox` by copying the information from their certificate:

mayala@htb[/htb] `$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out selfsigned.pem -sha256 -days 365 Generating a RSA private key ..++++ ....++++ writing new private key to 'key.pem' Enter PEM pass phrase: Verifying - Enter PEM pass phrase: ----- You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter '.', the field will be left blank. ----- Country Name (2 letter code) [AU]:US State or Province Name (full name) [Some-State]:California Locality Name (eg, city) []:San Francisco Organization Name (eg, company) [Internet Widgits Pty Ltd]:Cloudflare, Inc. Organizational Unit`

```
Name (eg, section) []: Common Name (e.g. server FQDN or YOUR name)
[]:hackthebox.com Email Address []:
```

Does this mean we hacked the system and are now able to impersonate HackTheBox? No, because the certificate is self-signed, the web browser does not trust it and displays a warning:



However, if we ever got our hands on the private key of a CA, we could use it to sign certificates with an arbitrary subject. This would allow us to effectively impersonate anyone. Therefore, private keys of CAs are one of the most protected resources when it comes to secure online communication.

Performing Encryption

Finally, we can also use openssl to perform encryption. For that, we first create a new key-pair and extract the public key to a separate file:

mayala@htb[/htb]
```
# create new keypair $ openssl genrsa -out rsa.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes) ..+++++
.............................+++++ e is 65537 (0x010001) # extract public key $
openssl rsa -in rsa.pem -pubout > rsa_pub.pem writing RSA key
```

We can then use the extracted public key to encrypt a file. Inspecting the encrypted file reveals the binary ciphertext:

mayala@htb[/htb]
```
$ openssl pkeyutl -encrypt -inkey rsa_pub.pem -pubin -in msg.txt
-out msg.enc $ cat msg.enc | xxd | head 00000000: 0550 eea0 8b79 ba5e a933 0539
6175 4834 .P...y.^.3.9auH4 00000010: 26dd a435 d4c1 bc18 f1c2 075f 8d51 2d2d
&..5.......Q-- 00000020: 5e13 fa33 d65f 4d59 fb87 26e2 6a29 a8e9
^..3._MY..&.j).. 00000030: 017c 39d4 f43c 210b fbb5 921e 2763 4512 .|9..
```

```
<!.....'cE. 00000040: b68e 3b41 c77d 948e c720 eb35 f104 a428 ..;A.}... .5...(
00000050: 2191 e2bd 2638 f6da ce87 93fa 80ca e32c !...&8.........,  00000060:
3b2b 3c89 61cf 366f ff8b 933d 5dda 1299 ;+<.a.6o...=]...  00000070: 5760 3849
bad3 891d d3cb 0c94 c299 2de4 W`8I..........-.  00000080: 5c13 5a6a 25ea b645
c891 2894 995d ed7a \.Zj%..E..(..].z  00000090: ad41 a5fc 79c3 6beb 7670 5f00
3d30 052d .A..y.k.vp_.=0.-
```

Lastly, we can decrypt the encrypted file using the corresponding private key:

```
mayala@htb[/htb] $ openssl pkeyutl -decrypt -inkey rsa.pem -in msg.enc >
decrypted.txt $ cat decrypted.txt Hello.World
```

TLS 1.2 Handshake

The TLS handshake is the process in which the client and server negotiate all the parameters for the TLS session. It always follows a predefined scheme with the exception of minor deviations depending on the concrete parameters chosen for the connection.

Cipher Suites

In TLS, cipher suites define the cryptographic algorithms used for a connection. That includes the following information:

- The key exchange algorithm
- The method used for authentification
- The encryption algorithm and mode, which provide confidentiality
- The MAC algorithm, which provides integrity protection

As an example, let's have a look at the following TLS 1.2 cipher suite: `TLS_DH_RSA_WITH_AES_128_CBC_SHA256`

From the name, we can identify the algorithms used by this cipher suite:

- The key exchange algorithm is `Diffie-Hellman (DH)`
- Server authentification is performed via `RSA`
- The encryption is `AES-128` in `CBC` mode
- The MAC algorithm is a `SHA256` HMAC

All TLS 1.2 cipher suites follow this naming scheme. The encryption algorithm is always a symmetric algorithm. The symmetric key for this algorithm is exchanged using the key exchange algorithm, which is always an asymmetric algorithm. Thus, TLS encrypts data using a

symmetric key due to significant performance advantages compared to asymmetric encryption. The cipher suite used by a specific connection is negotiated in the handshake.

Cipher Suites using the `TLS_DHE` and `TLS_ECDHE` key exchange algorithms provide `Perfect Forward Secrecy (PFS)`, meaning an attacker is unable to decrypt past messages even after obtaining a future session key. In particular, this protects past communication from leaks potentially occurring in the future. Therefore, PFS cipher suites are preferable if they are supported by the client.

Handshake Overview

During the handshake, the client and server establish a connection and negotiate all the required parameters to establish a secure channel for application data. The handshake follows a well-defined schema and varies slightly depending on the cipher suite that is negotiated.

The handshake begins with the client sending the `ClientHello` message. This message informs the server that the client wants to establish a secure connection. It contains the latest TLS version supported by the client, as well as a list of cipher suites the client supports among other information.

The server responds with a `ServerHello` message. The server chooses a TLS version that is equal to or lower than the version provided by the client. Additionally, the server chooses one of the cipher suites provided in the `ClientHello`. This information is included in the `ServerHello` message.

After agreeing on the TLS version and cryptographic parameters, the server provides a certificate in the `Certificate` message, thereby proving the server's identity to the client.

If a PFS cipher suite was agreed upon, the server proceeds to share fresh key material in the `ServerKeyExchange` message. It contains a key share as well as a signature. This is followed by the `ServerHelloDone` message.

The client responds with the `ClientKeyExchange` message, containing the client's key share. After this, the key exchange is concluded and both parties share a secret that is used to derive a shared symmetric key. Both parties transmit a `ChangeCipherSpec` message to indicate that all following messages are encrypted using the computed symmetric key. From here on, all data is encrypted and MAC-protected.

Analyzing a TLS 1.2 Handshake in Wireshark

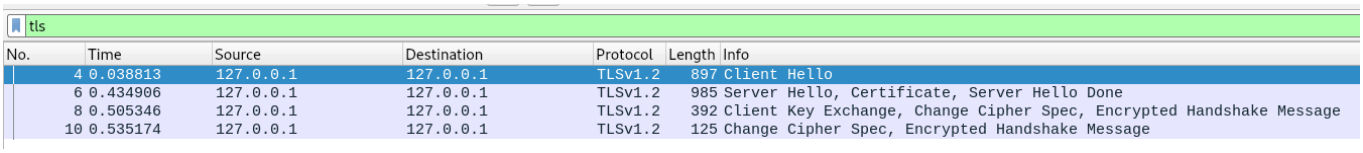
Let's have a look at a TLS 1.2 handshake in Wireshark, which is a network protocol analyzer. It can typically be installed from the package manager:

mayala@htb[/htb] `$ sudo apt install wireshark`

We can then start Wireshark with a path to a packet capture (or `pcap`) file to analyze the packets:

mayala@htb[/htb] `$ Wireshark /path/to/file.pcap`

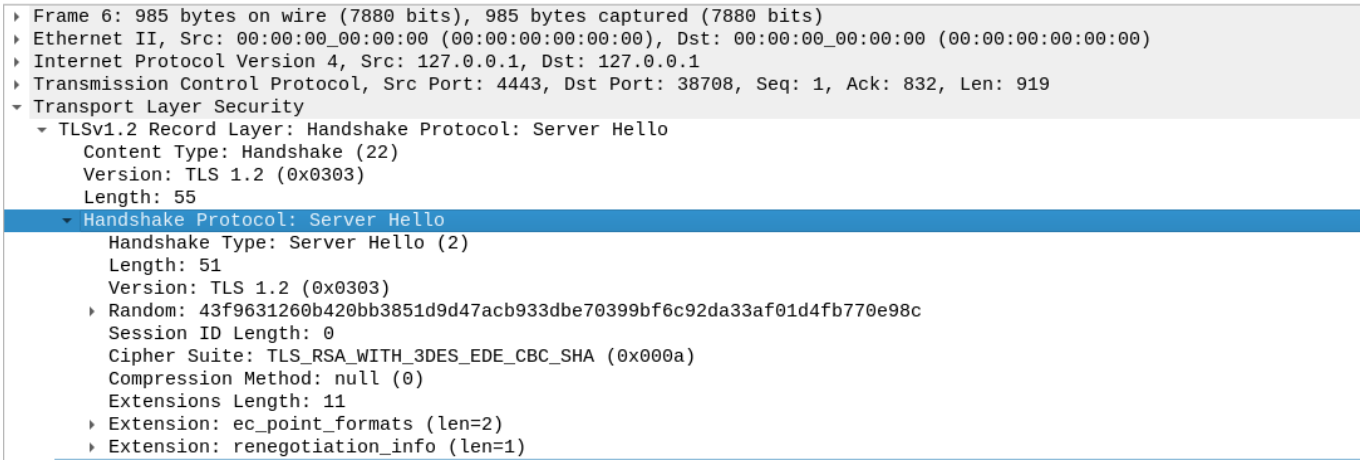
After entering the protocol name `tls` in the filter bar, we can see the TLS handshake in Wireshark:



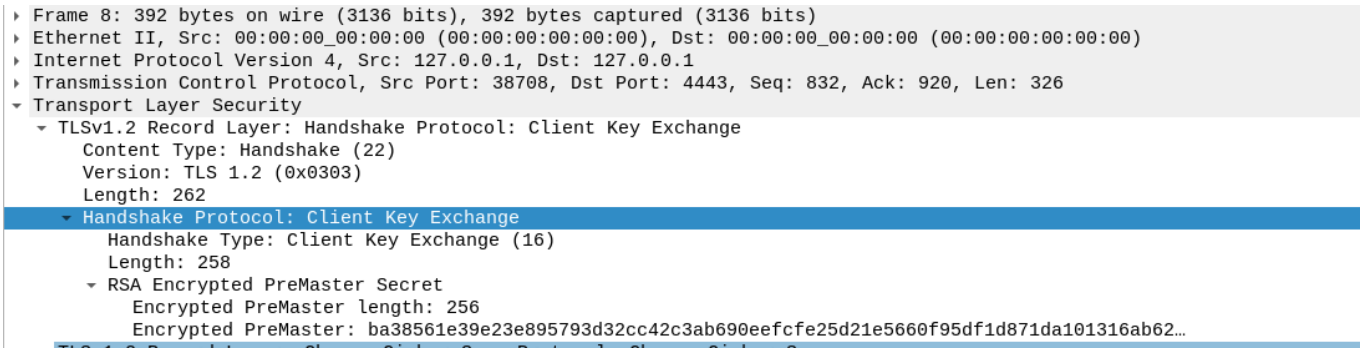
When expanding the `ClientHello` message, we can inspect the TLS version and supported cipher suites sent by the client:

```
▸ Frame 4: 897 bytes on wire (7176 bits), 897 bytes captured (7176 bits)
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 38708, Dst Port: 4443, Seq: 1, Ack: 1, Len: 831
▾ Transport Layer Security
  ▾ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 826
    ▾ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 822
        Version: TLS 1.2 (0x0303)
      ▸ Random: 43f962bf60b420bb3851d9d47acb933dbe70399bf6c92da33af01d4fb770e98c
        Session ID Length: 0
        Cipher Suites Length: 654
      ▾ Cipher Suites (327 suites)
          Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
          Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
          Cipher Suite: TLS_RSA_WITH_NULL_MD5 (0x0001)
          Cipher Suite: TLS_RSA_WITH_NULL_SHA (0x0002)
          Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
          Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)
          Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
```

Doing the same in the `ServerHello` message reveals the TLS version and cipher suite chosen by the server for this TLS connection:



Lastly, we can inspect the key information sent by the client in the `ClientKeyExchange` message. In this case, a `TLS_RSA` cipher suite was chosen, thus the key information sent by the client is the shared key encrypted with the server's public key.



Note: There is no `Server Key Exchange` message since the cipher suite does not provide PFS.

TLS 1.3

TLS 1.3 made several improvements over TLS 1.2. That includes dropping support for insecure cryptographic parameters and thereby reducing complexity. Furthermore, improvements to the handshake were made to allow for faster session establishment.

Cipher Suites and Cryptography

Several cryptographic improvements have been made with the new version TLS 1.3. These enhancements include the removal of older, less secure cryptographic techniques and the addition of newer, more secure techniques. TLS 1.3 also includes improved key exchange algorithms and support for post-quantum cryptography. In particular, TLS 1.3 only supports key exchange algorithms that support PFS.

For instance, a TLS 1.3 cipher suite looks like this:

```
TLS_AES_128_GCM_SHA256
```

It is significantly shorter than TLS 1.2 cipher suites since it only specifies the encryption algorithm and mode as well as the hash function used for the HMAC algorithm. TLS 1.3 cipher suites do not specify the method used for server authentication and the key exchange algorithm.

Handshake

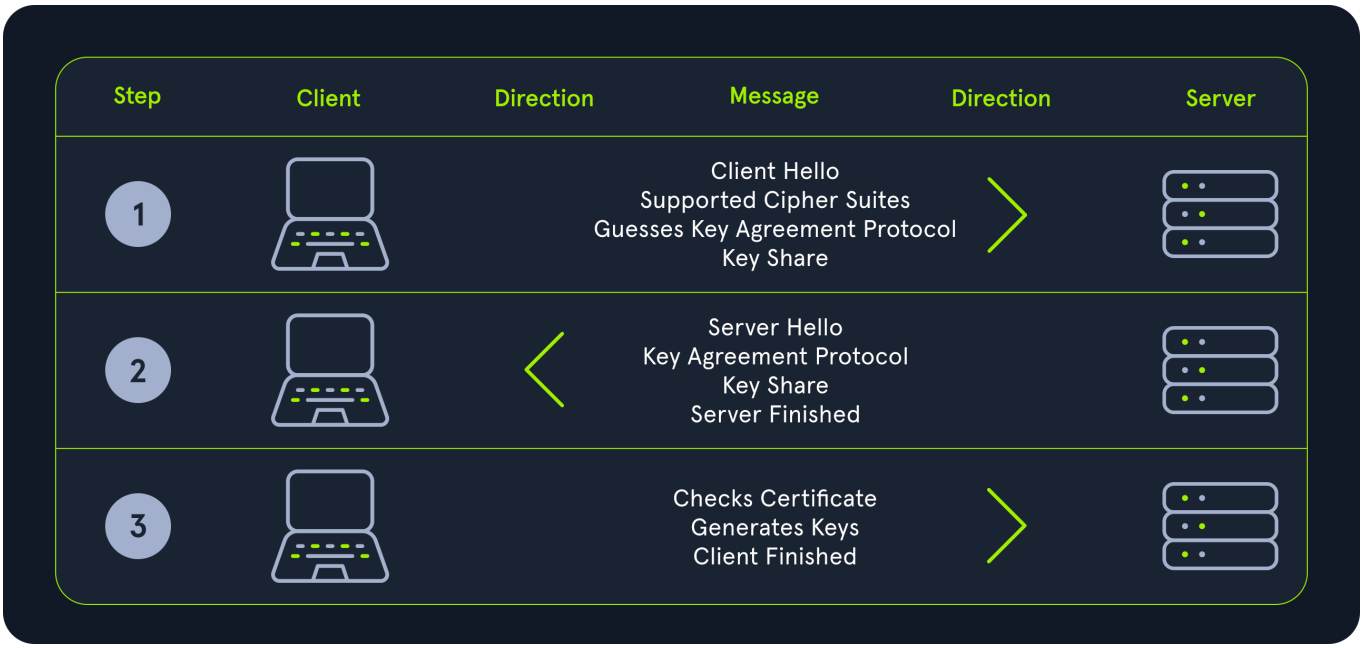
Several changes were introduced in the TLS 1.3 handshake process. Some messages have been redesigned for efficiency, while other messages have been eliminated completely to reduce the latency and overhead of the handshake which enables a faster connection establishment.

Just like in TLS 1.2, the TLS 1.3 handshake begins with the `ClientHello` message. However, in TLS1.3 this message contains the client's key share in addition to the supported cipher suites. This eliminates the need for the `ClientKeyExchange` message later on in the handshake. This key share is contained in an extension that is sent with the `ClientHello` message.

The server responds with the `ServerHello` message that confirms the key agreement protocol and specifies the chosen cipher suite, just like in TLS 1.2. This message also contains the server's key share. A fresh key share is always transmitted here to guarantee PFS. This replaces the need for the `ServerKeyExchange` message in TLS1.2, which was required when PFS cipher suites were used. The server's certificate is also contained within the `ServerHello` message.

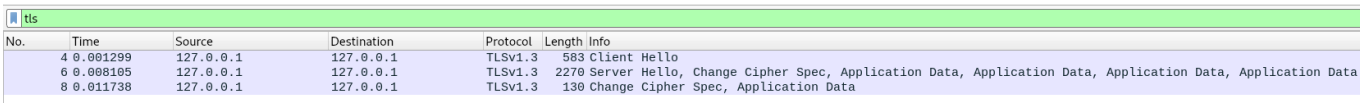
The handshake concludes with a `ServerFinished` and `ClientFinished` message.

Note: All messages after the `ServerHello` are already encrypted. Therefore, the TLS 1.3 handshake is significantly shorter than the TLS 1.2 handshake.



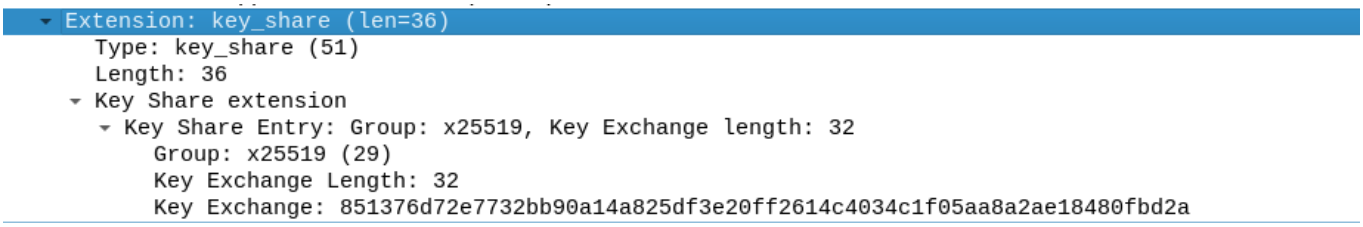
Analyzing a TLS 1.3 Handshake in Wireshark

When looking at a TLS 1.3 handshake in Wireshark, the differences to a TLS 1.2 handshake become apparent. In particular, we can see that there are no `Certificate` and `ClientKeyExchange` messages since they have been removed:



We can find the client's key share in the `key_share` extension in the `ClientHello` message. In this case, the client chooses two different shares for different groups. That is because the group is chosen by the server in the `ServerHello` message, which the client has not received yet. Therefore, the client transmits multiple shares to increase the chance of agreement on a group with the server:

The server's key share can be inspected in the `key_share` extension in the `ServerHello` message. The server chooses a group and only transmits its key share for that group:



From this point on, all transmitted data is encrypted, as indicated by the `EncryptedApplicationData` tag in Wireshark.