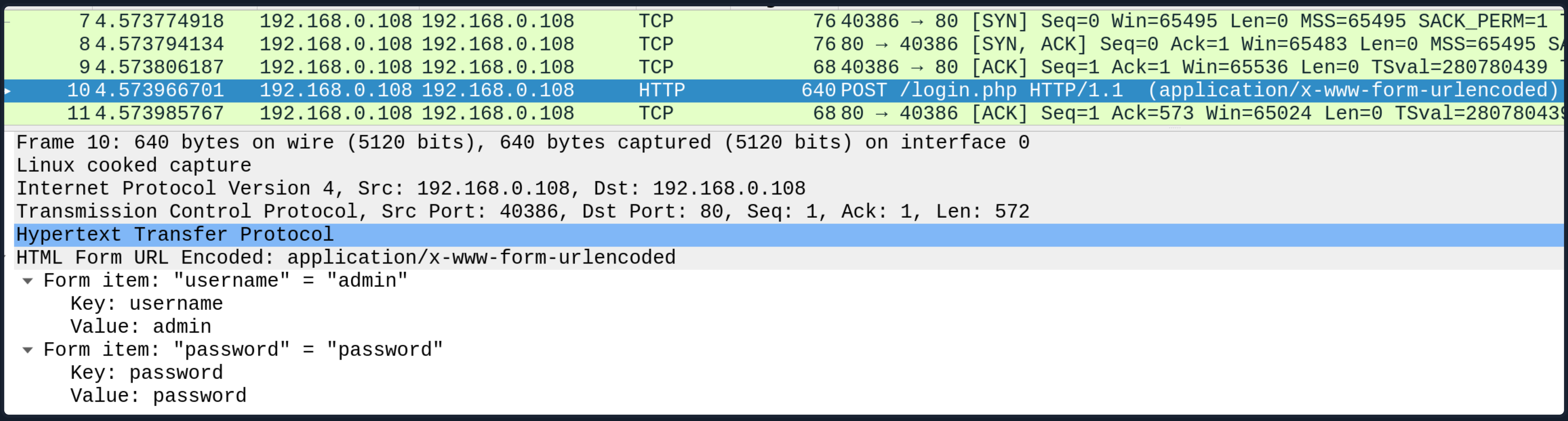
Hypertext Transfer Protocol Secure (HTTPS)

In the previous section, we discussed how HTTP requests are sent and processed. However, one of the significant drawbacks of HTTP is that all data is transferred in clear-text. This means that anyone between the source and destination can perform a Man-in-the-middle (MiTM) attack to view the transferred data.

To counter this issue, the HTTPS (HTTP Secure) protocol was created, in which all communications are transferred in an encrypted format, so even if a third party does intercept the request, they would not be able to extract the data out of it. For this reason, HTTPS has become the mainstream scheme for websites on the internet, and HTTP is being phased out, and soon most web browsers will not allow visiting HTTP websites.

HTTPS Overview

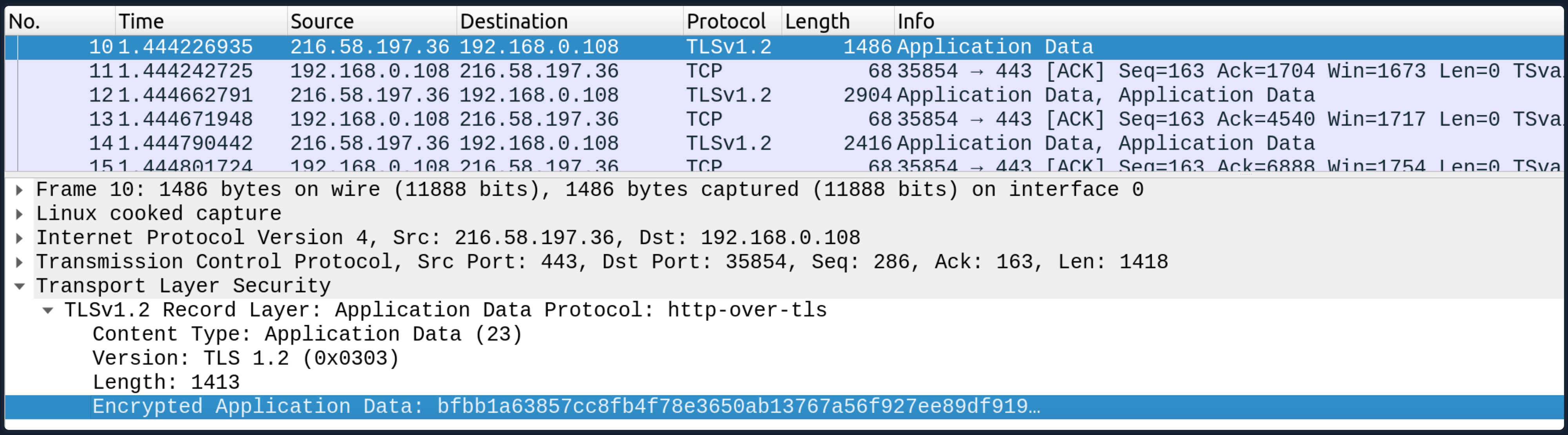
If we examine an HTTP request, we can see the effect of not enforcing secure communications between a web browser and a web application. For example, the following is the content of an HTTP login request:



```
   7 4.573774918   192.168.0.108 192.168.0.108     TCP            76 40386 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1
   8 4.573794134   192.168.0.108 192.168.0.108     TCP            76 80 → 40386 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 S
   9 4.573806187   192.168.0.108 192.168.0.108     TCP            68 40386 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=280780439
  10 4.573966701   192.168.0.108 192.168.0.108     HTTP          640 POST /login.php HTTP/1.1  (application/x-www-form-urlencoded)
  11 4.573985767   192.168.0.108 192.168.0.108     TCP            68 80 → 40386 [ACK] Seq=1 Ack=573 Win=65024 Len=0 TSval=28078043
Frame 10: 640 bytes on wire (5120 bits), 640 bytes captured (5120 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 192.168.0.108, Dst: 192.168.0.108
Transmission Control Protocol, Src Port: 40386, Dst Port: 80, Seq: 1, Ack: 1, Len: 572
Hypertext Transfer Protocol
HTML Form URL Encoded: application/x-www-form-urlencoded
 ▾ Form item: "username" = "admin"
     Key: username
     Value: admin
 ▾ Form item: "password" = "password"
     Key: password
     Value: password
```

We can see that the login credentials can be viewed in clear-text. This would make it easy for someone on the same network (such as a public wireless network) to capture the request and reuse the credentials for malicious purposes.

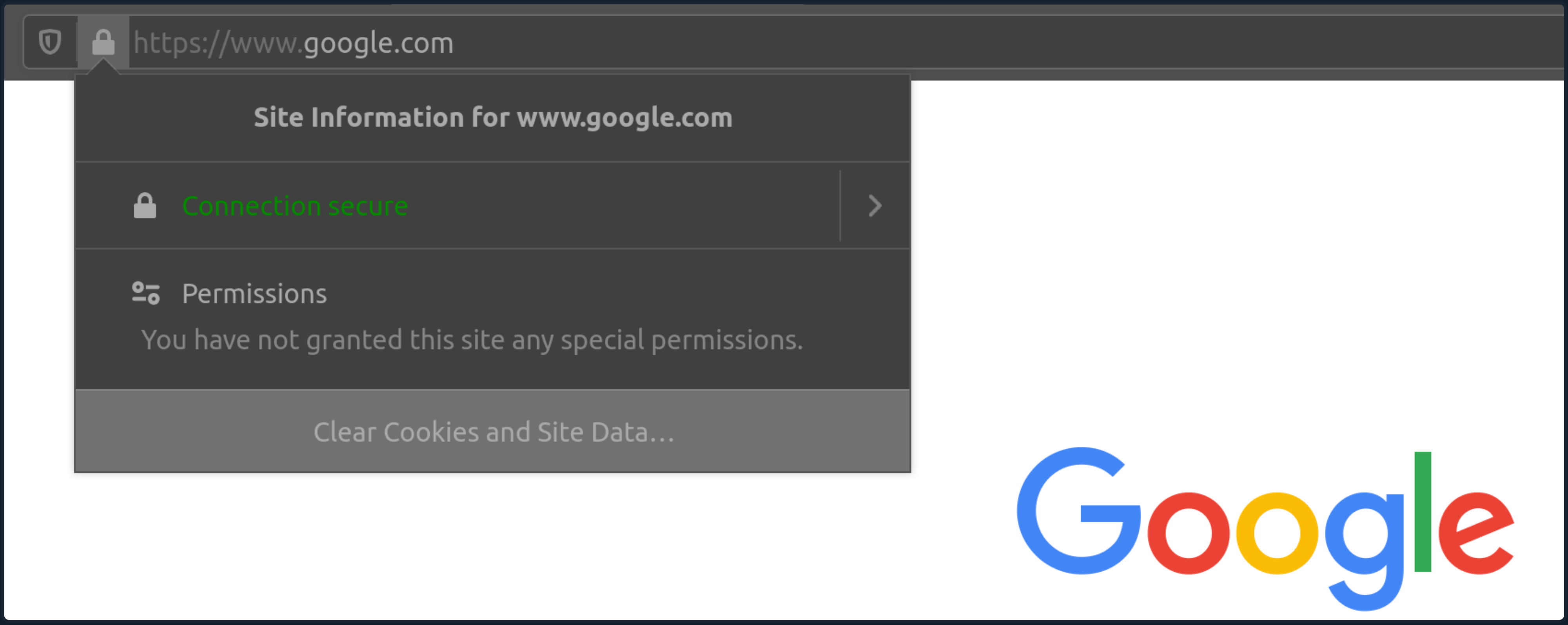
In contrast, when someone intercepts and analyzes traffic from an HTTPS request, they would see something like the following:



```
No.     Time          Source        Destination    Protocol Length Info
     10 1.444226935   216.58.197.36 192.168.0.108  TLSv1.2    1486 Application Data
     11 1.444242725   192.168.0.108 216.58.197.36  TCP          68 35854 → 443 [ACK] Seq=163 Ack=1704 Win=1673 Len=0 TSva
     12 1.444662791   216.58.197.36 192.168.0.108  TLSv1.2    2904 Application Data, Application Data
     13 1.444671948   192.168.0.108 216.58.197.36  TCP          68 35854 → 443 [ACK] Seq=163 Ack=4540 Win=1717 Len=0 TSva
     14 1.444790442   216.58.197.36 192.168.0.108  TLSv1.2    2416 Application Data, Application Data
     15 1.444801724   192.168.0.108 216.58.197.36  TCP          68 35854 → 443 [ACK] Seq=163 Ack=6888 Win=1754 Len=0 TSva
▸ Frame 10: 1486 bytes on wire (11888 bits), 1486 bytes captured (11888 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 216.58.197.36, Dst: 192.168.0.108
▸ Transmission Control Protocol, Src Port: 443, Dst Port: 35854, Seq: 286, Ack: 163, Len: 1418
▾ Transport Layer Security
  ▾ TLSv1.2 Record Layer: Application Data Protocol: http-over-tls
      Content Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 1413
      Encrypted Application Data: bfbb1a63857cc8fb4f78e3650ab13767a56f927ee89df919…
```

As we can see, the data is transferred as a single encrypted stream, which makes it very difficult for anyone to capture information such as credentials or any other sensitive data.

Websites that enforce HTTPS can be identified through `https://` in their URL (e.g. https://www.google.com), as well as the lock icon in the address bar of the web browser, to the left of the URL:

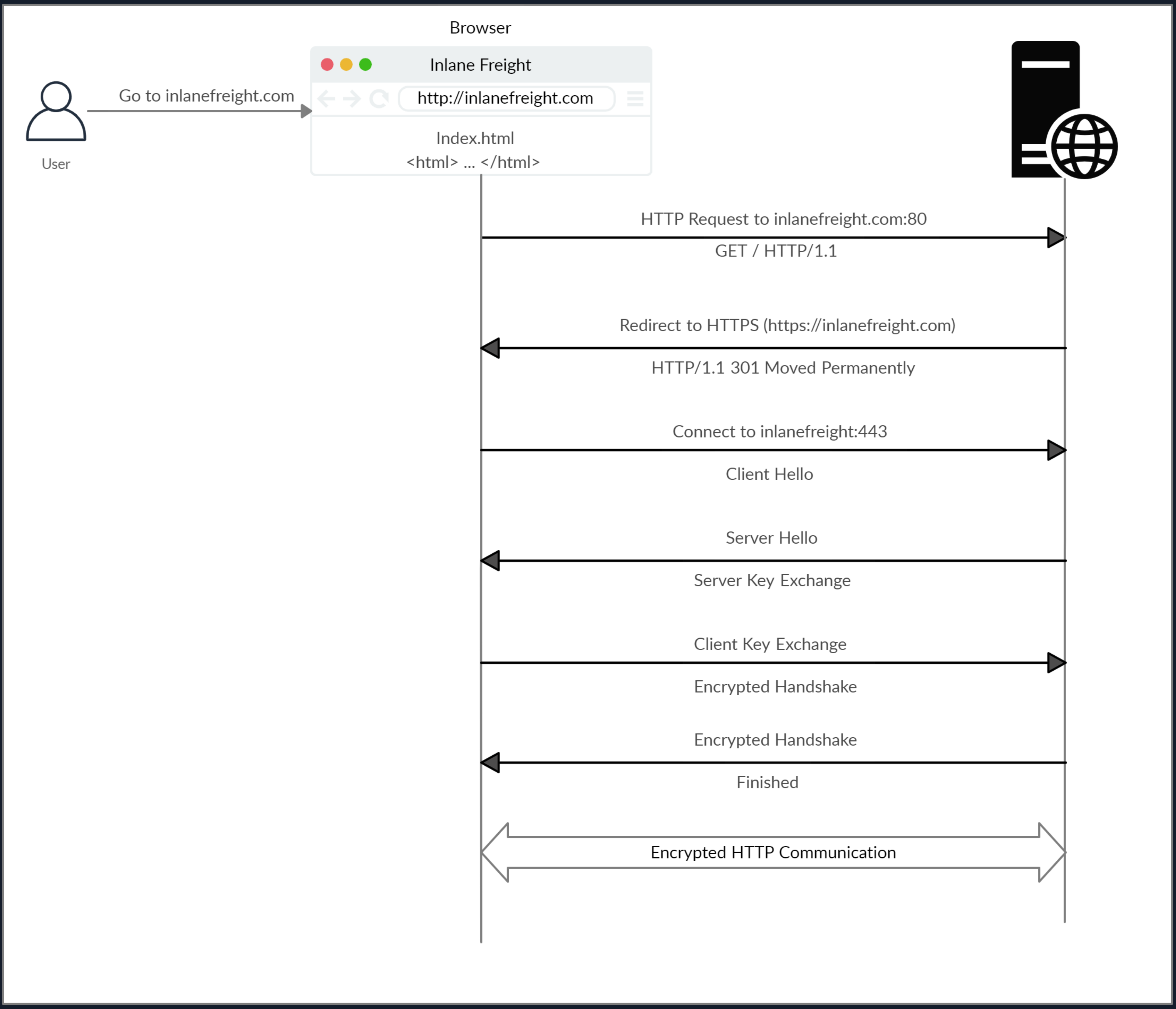

https://www.google.com
Site Information for www.google.com
Connection secure
Permissions
You have not granted this site any special permissions.
Clear Cookies and Site Data…
Google

So, if we visit a website that utilizes HTTPS, like Google, all traffic would be encrypted.

Note: Although the data transferred through the HTTPS protocol may be encrypted, the request may still reveal the visited URL if it contacted a clear-text DNS server. For this reason, it is recommended to utilize encrypted DNS servers (e.g. 8.8.8.8 or 1.2.3.4), or utilize a VPN service to ensure all traffic is properly encrypted.

HTTPS Flow

Let's look at how HTTPS operates at a high level:



If we type `http://` instead of `https://` to visit a website that enforces HTTPS, the browser attempts to resolve the domain and redirects the user to the webserver hosting the target website. A request is sent to port `80` first, which is the unencrypted HTTP protocol. The server detects this and redirects the client to secure HTTPS port `443` instead. This is done via the `301 Moved Permanently` response code, which we will discuss in an upcoming section.

Next, the client (web browser) sends a "client hello" packet, giving information about itself. After this, the server replies with "server hello", followed by a key exchange to exchange SSL certificates. The client verifies the key/certificate and sends one of its own. After this, an encrypted handshake is initiated to confirm whether the encryption and transfer are working correctly.

Once the handshake completes successfully, normal HTTP communication is continued, which is encrypted after that. This is a very high-level overview of the key exchange, which is beyond this module's scope.

Note: Depending on the circumstances, an attacker may be able to perform an HTTP downgrade attack, which downgrades HTTPS communication to HTTP, making the data transferred in clear-text. This is done by setting up a Man-In-The-Middle (MITM) proxy to transfer all traffic through the attacker's host without the user's knowledge. However, most modern browsers, servers, and web applications protect against this attack.

cURL for HTTPS

cURL should automatically handle all HTTPS communication standards and perform a secure handshake and then encrypt and decrypt data automatically. However, if we ever contact a website with an invalid SSL certificate or an outdated one, then cURL by default would not proceed with the communication to protect against the earlier mentioned MITM attacks:

```
MichaelLuka@htb[/htb]$ curl https://inlanefreight.com

curl: (60) SSL certificate problem: Invalid certificate chain
More details here: https://curl.haxx.se/docs/sslcerts.html
...SNIP...
```

Modern web browsers would do the same, warning the user against visiting a website with an invalid SSL certificate.

We may face such an issue when testing a local web application or with a web application hosted for practice purposes, as such web applications may not yet have implemented a valid SSL certificate. To skip the certificate check with cURL, we can use the `-k` flag:

```
MichaelLuka@htb[/htb]$ curl -k https://inlanefreight.com

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
...SNIP...
```

As we can see, the request went through this time, and we received the response data.

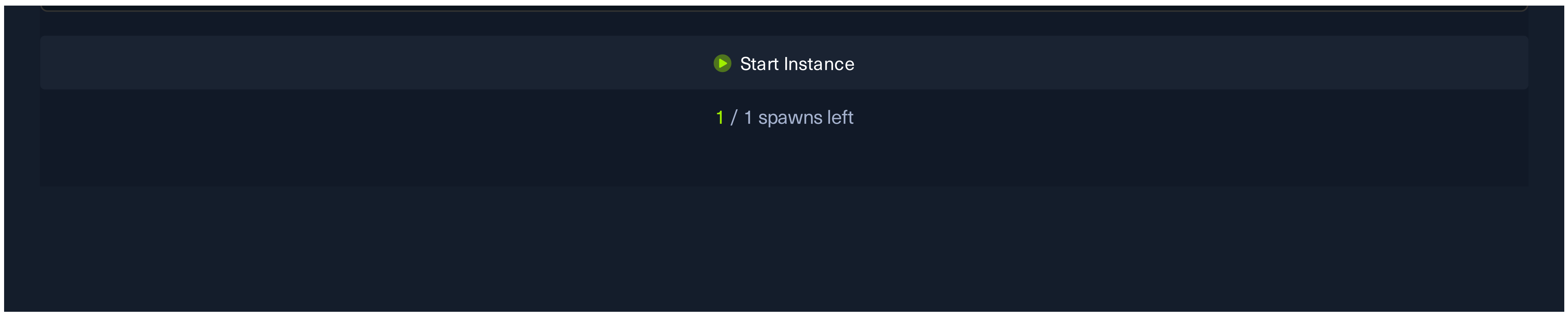
Previous   Next   Mark Complete & Next

Cheat Sheet

Table of Contents

My Workstation

OFFLINE



1 / 1 spawns left