# CL.TE

In our first example of HTTP request smuggling, we will look at a setting where the reverse proxy does not support chunked encoding. Therefore, if a request contains both the CL and TE headers, the reverse proxy will (incorrectly) use the CL header to determine the request length, while the web server will (correctly) use the TE header to determine the request length. As such, this type of HTTP request smuggling vulnerability is called CL.TE vulnerability. We will discuss how to identify and exploit this type of request smuggling vulnerability.

## Foundation

Before jumping into our lab for this section, let's discuss the theory behind a CL.TE vulnerability and how we could exploit it. As described above, this type of vulnerability arises if the reverse proxy does not support chunked encoding. Consider a request like the following:

Code: http

POST / HTTP/1.1  
Host: clte.htb  
Content-Length: 10  
Transfer-Encoding: chunked  
  
0  
  
HELLO

Let's first have a look at the above request from the reverse proxy's perspective. Since the reverse proxy does not support chunked encoding, it uses the CL header to determine the request body's length. The CL header gives the length as 10 bytes, meaning the request body is parsed as the following 10 bytes:

0\r\n\r\nHELLO

In particular, we can see that the reverse proxy consumes all data we sent from the TCP stream such that no data is left (exactly how it should be). The reverse proxy then forwards the bytes we sent in our HTTP request to the web server.

Now let's look at the request from the web server's perspective. The web server correctly prefers the TE header over the CL header, as defined in the RFC shown in the previous section. Since the request body in the chunked encoding is terminated by the empty chunk with size 0, the web server thus parses the request body as:

0\r\n\r\n

In particular, the bytes HELLO are not consumed from the TCP stream. From the web server's perspective, these bytes are thus the beginning of the next HTTP request. However, since the request is not complete yet, the web server waits for more data to arrive on the TCP stream before processing the request. We successfully created a desynchronization between the reverse proxy and the web server since the reverse proxy and web server disagree on the request boundaries.

Now let's discuss what happens if the next request arrives. Keep in mind that this can be a request from an unsuspecting victim that visits the vulnerable site just after we created the desynchronization with our specifically crafted request. Let's assume the next request that hits the reverse proxy looks like this:

Code: http

GET / HTTP/1.1  
Host: clte.htb

Now let's look at the complete TCP stream containing both these subsequent requests. Again, we are going to look at it from the perspective of the reverse proxy first. According to the reverse proxy, the TCP stream should be split into the two requests like this (red marks the first request while green marks the second request):

POST / HTTP/1.1  
Host: clte.htb  
Content-Length: 10  
Transfer-Encoding: chunked  
  
0  
  
HELLOGET / HTTP/1.1  
Host: clte.htb

We can see that the first request's body ends after HELLO and the subsequent request starts with the GET keyword. Now let's look at it from the perspective of the web server:

POST / HTTP/1.1  
Host: clte.htb  
Content-Length: 10  
Transfer-Encoding: chunked  
  
0  
  
HELLOGET / HTTP/1.1  
Host: clte.htb

Since the web server thinks the first request ends after the bytes 0\r\n\r\n, the bytes HELLO are prepended to the subsequent request such that the request method was changed from GET to HELLOGET. Since that is an invalid HTTP method, the web server will most likely respond with an HTTP 405 - Method not allowed error message even though the second request by itself uses a valid HTTP method.

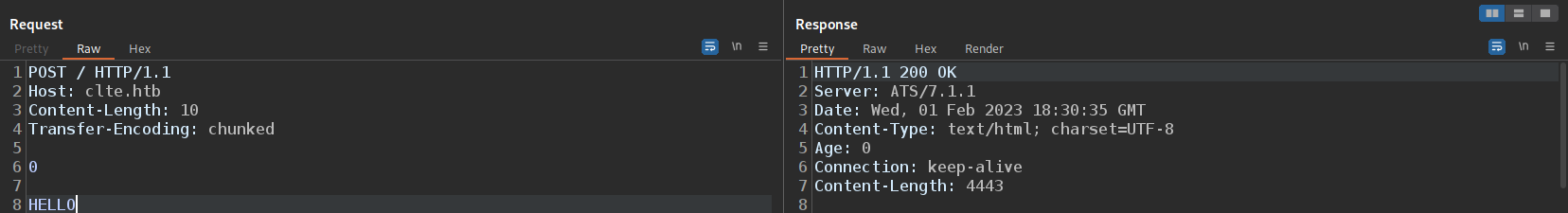
In this example scenario, we successfully created a desynchronization between the reverse proxy and web server to influence the request method of the subsequent request. Since all requests share the same TCP connection between the reverse proxy and web server, the second request does not need to come from the same source as the first request. Therefore, this attack allows an attacker to influence requests by other users without their knowledge.

## Identification

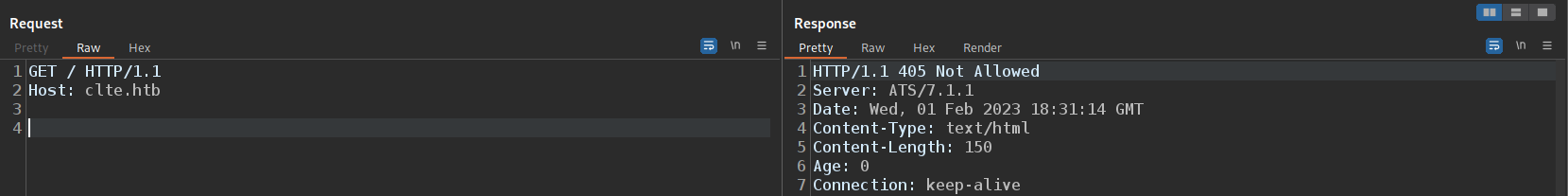
Now let's have a look at a practical example. When we start the exercise below, we can see a simple web application that contains an admin area with an action we are unauthorized to do. In our example, this action is revealing the flag. However, in a real-world application this can be any action that only an admin user is allowed to do such as creating a new user or promoting a user to an admin:



Let's try to confirm that the lab is vulnerable to a CL.TE request smuggling vulnerability. To do so, we can use the two requests shown above. Copy the requests to two separate tabs in Burp Repeater. Quickly send the two requests after each other to observe the behavior described above. The first response contains the index of the web application:



If we now send the second request immediately afterward, we can observe that the web server responds with the HTTP 405 status code for the reasons discussed above:



Thus, we successfully confirmed that the setup is vulnerable to a CL.TE request smuggling vulnerability since we influenced the second request with the first one.

## Exploitation

Now let's discuss how we can exploit a CL.TE vulnerability. Let's assume we want to force the admin user to promote our low-privilege user account to an admin user which can be done with the endpoint /admin.php?promote\_uid=2 where our user id is 2. We can force the admin user to do so by sending the following request:

Code: http

POST / HTTP/1.1  
Host: clte.htb  
Content-Length: 52  
Transfer-Encoding: chunked  
  
0  
  
POST /admin.php?promote\_uid=2 HTTP/1.1  
Dummy:

The request contains both the CL and TE headers. The CL header indicates a request body length of 52 bytes which includes the whole body as displayed above up to the sequence Dummy: . However, the request body starts with the bytes 0\r\n\r\n which represent the empty chunk in chunked encoding thus terminating the request body early if chunked encoding is used. This creates a discrepancy between the reverse proxy and web server as we will discuss in more detail in a bit.

If we wait for the admin user to access the page, which should take about 10 seconds, our user has been promoted. So let's investigate what exactly happened.

The admin user accesses the page using his session cookie with a request like this:

Code: http

GET / HTTP/1.1  
Host: clte.htb  
Cookie: sess=<admin\_session\_cookie>

This request is benign. The admin user simply accesses the index of the website, so no action should be taken. Let's look at the TCP stream from the reverse proxy's view:

POST / HTTP/1.1  
Host: clte.htb  
Content-Length: 52  
Transfer-Encoding: chunked  
  
0  
  
POST /admin.php?promote\_uid=2 HTTP/1.1  
Dummy: GET / HTTP/1.1  
Host: clte.htb  
Cookie: sess=<admin\_session\_cookie>

The reverse proxy uses the CL header to determine the length of the first request such that it ends just after Dummy: . The reverse proxy sees a POST request to / by us and a GET request to / by the admin user.

Now let's look at the TCP stream from the web server's view:

POST / HTTP/1.1  
Host: clte.htb  
Content-Length: 52  
Transfer-Encoding: chunked  
  
0  
  
POST /admin.php?promote\_uid=2 HTTP/1.1  
Dummy: GET / HTTP/1.1  
Host: clte.htb  
Cookie: sess=<admin\_session\_cookie>

Since the web server correctly uses the chunked encoding, it determines that the first request ends with the empty chunk. The web server thus sees a POST request to / by us and a POST request to /admin.php?promote\_uid=2 by the admin user. Since the admin user's request is authenticated and the session cookie sent along the request, we successfully forced the admin user to grant our user admin rights without the admin user even knowing what happened.

**Note:** We need to add a separate line with the Dummy keyword to our first request to "hide" the first line of the admin user's request as an HTTP header value to preserve the syntax of the request's header section.

.