# HTTP Response Splitting

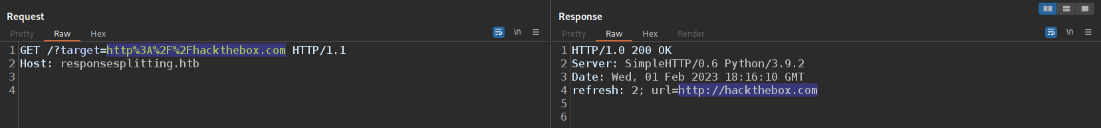
[HTTP Response Splitting](https://owasp.org/www-community/attacks/HTTP_Response_Splitting) is a serious vulnerability that arises when web servers reflect user input in HTTP headers without proper sanitization. Since HTTP headers are separated only by newline characters, an injection of the CRLF character sequence breaks out of the intended HTTP header and allows an attacker to append further arbitrary HTTP headers and even manipulate the response. This can lead to reflected XSS vulnerabilities.

## Identification

The exercise below contains a simple web application that implements a redirection service:



It works by setting the user-supplied target domain in the [Refresh](https://codingshower.com/http-refresh-header-meta-refresh/) header, which tells the client's browser to load the specified URL after the given amount of seconds (in this case 2):

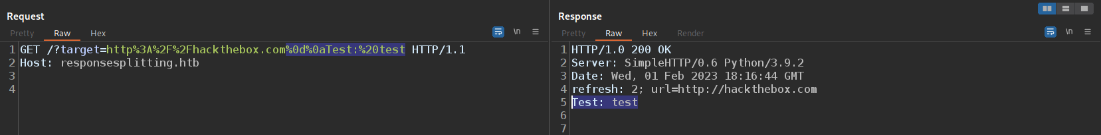


We can simply confirm that no sanitization is implemented by injecting the CRLF sequence and attempting to append our own header to the response with a request like the following:

Code: http

GET /?target=http%3A%2F%2Fhackthebox.com%0d%0aTest:%20test HTTP/1.1  
Host: responsesplitting.htb

Looking at the response, we successfully injected our own header into the response:



The injection works as the response contains the newline sequence we injected and treats the appended data as a separate HTTP header.

## Exploitation

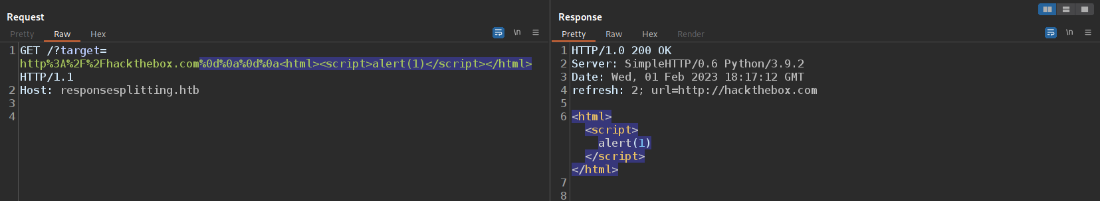
HTTP response splitting can be exploited in multiple ways. The simplest and most generic approach would be to construct a reflected XSS attack. Since we can append arbitrary lines to the HTTP header our payload is reflected in, we can effectively modify the entire response without any restrictions. The original page is of course appended to our payload but this does not prevent us from executing any injected JavaScript code.

Let's construct a simple proof of concept. To do so, we need to inject two new lines since these separate the HTTP response body from the HTTP headers section. We can then inject our XSS payload which will be treated as the response body by our browser. This results in a request like this:

Code: http

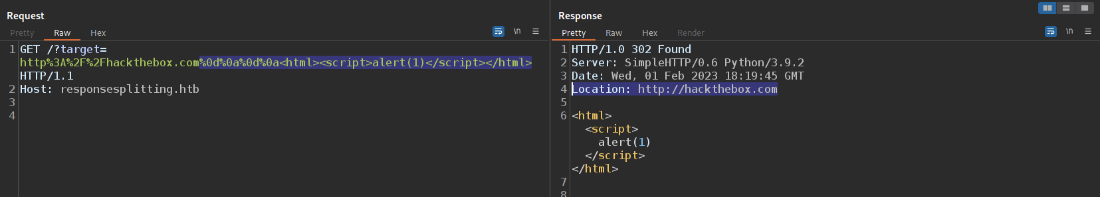
GET /?target=http%3A%2F%2Fhackthebox.com%0d%0a%0d%0a<html><script>alert(1)</script></html> HTTP/1.1  
Host: responsesplitting.htb

Our XSS payload is reflected in the response body and successfully executed by our web browser:



#### Exploitation of HTTP 302 Redirects

It is probably more common to see a redirect via an HTTP 302 status code and the Location header rather than the Refresh header. In this case, the web browser immediately redirects the user without displaying the content. Thus, our previous payload would not work as the web browser simply ignores it:

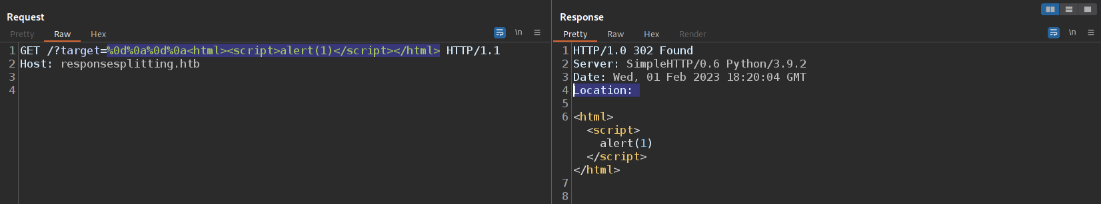


In this case, the browser reads the Location header and redirects the user to the new location without ever executing our malicious XSS payload. Luckily for us, there is an easy workaround for this. We can simply supply an empty Location header:

Code: http

GET /?target=%0d%0a%0d%0a<html><script>alert(1)</script></html> HTTP/1.1  
Host: responsesplitting.htb

Since an empty location is invalid, the browser does not know where to navigate and displays the response body, thus executing our XSS payload:



#### Exploitation Remarks

HTTP Response Splitting can be exploited in other ways than reflected XSS. For instance, we can easily deface the website by injecting arbitrary HTML content in the response. If the web application is deployed in an incorrectly configured setting, we might be able to exploit a vulnerability like web cache poisoning to further escalate HTTP response splitting. For more details on web cache poisoning, check out the [Abusing HTTP Misconfigurations](https://academy.hackthebox.com/module/details/189) module. Lastly, if the web application implements custom headers or uses headers to implement security measures such as Clickjacking protection or a Content-Security-Policy (CSP), HTTP response splitting can lead to bypasses of these security measures as well.

OBS: sometimes is good idea to check for Content-Type so if content-type=text/plain it will not execute or parse the HTML/JS code correctly but only as merely text file. To bypass this is enough to inject a %0d%0aContent-Type:%20text/html%0d%0a