**Other attacks**

* **Subdomain takeover**
* **Insecure cors**
* **SMTP injection**
* **Host Header attack**
* **Cache poisoning attack**
* **CRLF Injection**
* **HTTP Request Smuggling**
* **Insecure Deserialization**
* **Missing SPF record**
* **Type juggling**
* **Php remote xdebug vulnerability**
* **Using components with known vulnerabilities:**
* **Race conditions**
* **HTTP verb tempering**
* **Insufficient logging and Monitoring**
* **Bypass rate limit**
  + **Insecure CORS manipulation**
    - **The same-origin policy**
      * is a restrictive cross-origin specification that limits the ability for a website to interact with resources outside of the source domain. The same-origin policy was defined many years ago in response to potentially malicious cross-domain interactions, such as one website stealing private data from another. It generally allows a domain to issue requests to other domains, but not to access the responses
      * With SOP in place, in order to allow cross-domain communication, developers had to use different techniques to bypass SOP and pass sensitive information. The ‘by-passing’ happened too much to a point that it became a security issue. So, in order to enable information sharing without compromising the security posture of applications, the Cross-Origin Resource Sharing(CORS) was introduced in HTML5.
    - **Cross Origin Resource Sharing (CORS)**
      * is an HTML5 technology  and a browser mechanism which enables controlled access to resources located outside of a given domain using the XMLHttpRequest API . It extends and adds flexibility to the same-origin policy ([SOP](https://portswigger.net/web-security/cors/same-origin-policy)). However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as [cross-site request forgery](https://portswigger.net/web-security/csrf) (CSRF).
    - Many modern websites use CORS to allow access from subdomains and trusted third parties. Their implementation of CORS may contain mistakes or be overly lenient to ensure that everything works, and this can result in exploitable vulnerabilities.
    - There are a number of HTTP headers related to CORS, but the following three response headers are the most important for security:
      * **Access-Control-Allow-**Origin specifies which domains can access a domain’s resources. For instance, if [*requester.com*](http://requester.com) want to access *[provider.com](http://provider.com)’s* resources, then developers can use this header to securely grant *requester.com* access to *provider.com’s* resources.
      * **Access-Control-Allow-Credentials** specifies whether or not the browser will send cookies with the request. Cookies will only be sent if the *allow-credentials* header is set to true.
      * **Access-Control-Allow-Methods** specifies which HTTP request methods (GET, PUT, DELETE, etc.) can be used to access resources. This header lets developers further enhance security by specifying what methods are valid when [requester.com](http://requester.com) requests access to *[provider.com](http://provider.com)’s* resources
    - **Exploiting Misconfigured cors**
      * **Exploiting misconfigured wildcard (\*) in CORS Headers:**
        + One of the most common CORS misconfigurations is incorrectly using wildcards such as (\*) under which domains are allowed to request resources. This is usually set as default, which means any domain can access resources on this site. For example, consider the below request:

GET /api/userinfo.php

Host: www.victim.com

Origin: www.victim.com

* + - * + When you send the above request, you get a response with the Access-Control-Allow-Origin header setting. See the below response code.

HTTP/1.0 200 OK

Access-Control-Allow-Origin: \*

Access-Control-Allow-Credentials: true

* + - * + In this example case, the header is configured with a wildcard(\*). It means any domain can access the resources.
      * **Using XSS to make requests to cross origin sites**
        + One defense mechanism developers use against CORS exploitation is to whitelist domains that frequently requests access for information. However, this isn’t entirely secure, because if even one of the subdomains of the whitelisted domain is vulnerable to other exploits such as XSS, it can enable CORS exploitation.
    - **Preventing CORS manipulation**
      * CORS vulnerabilities arise primarily as misconfigurations.

### Proper configuration of cross-domain requests

* + - * + If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.

### Only allow trusted sites

* + - * + It may seem obvious but origins specified in the Access-Control-Allow-Origin header should only be sites that are trusted. In particular, dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be avoided.

### Avoid whitelisting null

* + - * + Avoid using the header Access-Control-Allow-Origin: null. Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.

### Avoid wildcards (\*)

* + **SMTP HEADER injection (MANIPULATION):**
    - **Definition**
      * SMTP header injection vulnerabilities arise when user input is placed into email headers without adequate sanitization, allowing an attacker to inject additional headers with arbitrary values. This behavior can be exploited to send copies of emails to third parties, attach viruses, deliver phishing attacks, and often alter the content of emails
    - **Vulnerable code**
      * <?php
      * if(isset($\_POST['name'])) {
      * $name = $\_POST['name'];
      * $replyto = $\_POST['replyTo'];
      * $message = $\_POST['message'];
      * $to = 'root@localhost';
      * $subject = 'My Subject';
      * // Set SMTP headers
      * $headers = "From: $name \n" .
      * "Reply-To: $replyto";
      * mail($to, $subject, $message, $headers); }?>
    - **Exploiting** 
      * A typical genuine POST request would be as follows:
        + POST /contact.php HTTP/1.1
        + Host: [www.example2.com](http://www.example2.com/)
        + name=Anna Smith&replyTo=[anna@example.com](mailto:anna@example.com)&message=Hello
      * An attacker could abuse this contact form by sending the following POST request:
        + POST /contact.php HTTP/1.1
        + Host: [www.example2.com](http://www.example2.com/)
        + name=BestProduct**\nbcc:**[**everyone@example3.com**](mailto:everyone@example3.com)&replyTo=[blame\_anna@example.com](mailto:blame_anna@example.com)&message=Buy my product!
  + **Subdomain Takeover**
    - Subdomain takeover is a process of registering a non-existing domain name to gain control over another domain. The most common scenario of this process follows:
      * Domain name (e.g., [*sub.example.com*](http://sub.example.com)) uses a CNAME record to another domain (e.g., [*sub.example.com*](http://sub.example.com) CNAME [*anotherdomain.com*](http://anotherdomain.com)).
      * At some point in time, [*anotherdomain.com*](http://anotherdomain.com) expires and is available for registration by anyone.
      * Since the CNAME record is not deleted from [example.com](http://example.com) DNS zone, anyone who registers or buy [*anotherdomain.com*](http://anotherdomain.com) has full control over [sub.example.com](http://sub.example.com) until the DNS record is present
    - **Exploitation**
      * Subdomain takeover is not limited to CNAME records. NS, MX and even A records
      * **CNAME subdomain takeover**. One of the primary types of CNAME subdomain takeover is the scenario when a canonical domain name is a regular Internet domain (not one owned by cloud providers as will be explained [below](https://0xpatrik.com/subdomain-takeover-basics/#cloudproviders)). The process of detecting whether some source domain name is vulnerable to CNAME subdomain takeover is quite straightforward:
      * **NS subdomain takeover**. The concept of subdomain takeover can be naturally extended to NS records: *If the base domain of canonical domain name of at least one NS record is available for registration, the source domain name is vulnerable to subdomain takeover.* One of the problems in subdomain takeover using NS record is that the source domain name usually has multiple NS records. Multiple NS records are used for redundancy and load balancing. The nameserver is chosen randomly before DNS resolution.
      * **MX subdomain takeover.** Compared to NS and CNAME subdomain takeovers, MX subdomain takeover has the lowest impact. Since MX records are used only to receive e-mails, gaining control over canonical domain name in MX record only allows an attacker to receive e-mails addressed to source domain name. Although the impact is not as significant as for CNAME or NS subdomain takeover, MX subdomain takeover might play a role in spear phishing attacks and intellectual property stealing
    - **Tools**
      * <https://github.com/EdOverflow/can-i-take-over-xyz>
      * <https://github.com/haccer/subjack>
      * [https://github.com/anshumanbh/tko-sub](https://github.com/anshumanbh/tko-subs)
      * <https://github.com/ArifulProtik/sub-domain-takeover>
      * <https://github.com/SaadAhmedx/Subdomain-Takeover>
      * <https://github.com/Ice3man543/SubOver>
      * <https://github.com/m4ll0k/takeover>
      * <https://github.com/antichown/subdomain-takeover>
      * **HostileSubBruteForcer** This is one of the best tools available for testing for subdomain takeover written by Nahamsec
  + **Host header attack**
    - **Definition**
      * It is common practice for the same web server to host several websites or web applications on the same IP address. This why the *host* header exists. The host header specifies which website or web application should process an incoming HTTP request. The web server uses the value of this header to dispatch the request to the specified website or web application. Each web application hosted on the same IP address is commonly referred to as a *virtual host*.
      * What happens if we specify an invalid Host Header?
        + Most web servers are configured to pass the unrecognized host header to the first virtual host in the list. Therefore, it’s possible to send requests with arbitrary host headers to the first virtual host.
    - **Server configuration Ex:**
      * Nano /etc/apache2/sites-enabled/000-default.conf
        + <VirtualHost \*>
        + ServerName [wahh-app1.com](http://wahh-app1.com)
        + DocumentRoot /www/app1
        + </VirtualHost>
        + <VirtualHost \*>
        + ServerName [wahh-app2.com](http://wahh-app2.com)
        + DocumentRoot /www/app2
        + </VirtualHost>

Ex : Any one requesting [wahh-app1.com](http://wahh-app1.com) show the contents of that folder /www/app1

* + - **Testing for host header vulnerability**
      * Initial testing is as simple as supplying another domain (i.e. [attacker.com](http://attacker.com/)) into the Host header field. It is how the web server processes the header value that dictates the impact. The attack is valid when the web server processes the input to send the request to an attacker-controlled host that resides at the supplied domain, and not to an internal virtual host that resides on the web server.
        + GET / HTTP/1.1
        + Host: [www.attacker.com](http://www.attacker.com/)
        + [...]
      * In the simplest case, this may cause a 302 redirect to the supplied domain.
        + HTTP/1.1 302 Found
        + [...]
        + Location: <http://www.attacker.com/login.php>
      * Alternatively, the web server may send the request to the first virtual host on the list.
    - **Exploiting host header**
      * **access local resources**
      * **password reset vuln**
        + It is common for password reset functionality that the developer takes the Host header value when creating password reset links that use a generated secret token.
        + So if the attacker changes the value of the host header , the user will then receive an email with the reset link that will redirect to the attacker website that he had put in the host header field , the victim may click on the link in the email and allow the attacker to obtain the reset token because it will be sent to the attacker server logs , thus the attacker will take this token and resetting the victim’s password.
        + Ex : [www.example.com/resetme.php?token=$^%WRGBSnsdzgrdnshxbdfv](http://www.example.com/resetme.php?token=$%5e%25WRGBSnsdzgrdnshxbdfv)
        + Best exploitation for Host Header attack
      * **xss in host header**
        + if the server return an error in the page that he can’t find this host or any error , the idea here is the the host header value is been print on the screen so if we inject java script code we can get an xss
        + Host Header : www.example.com<svg onload=confirm[1111]>
        + Also this vulnerability is a self-behavior same as unvalidated redirection
      * **unvalidated redirection (redirect to an attacker-controlled domain)**
        + **Vulnerable code**

<? Php

$host =”http://”.$\_SERVER[‘HTTP\_HOST’]

Echo $host.”/login.php” ;

Header (location: $host.”/login.php”) ; ?>

* + - * + redirect to any website that the attacker had put in the host header field , but this only affect the attacker not the victim because you cannot set the host header of anyone else than yourself
      * **perform web cache poisoning**
        + Using this technique, an attacker can manipulate a web-cache to serve poisoned content to anyone who requests it. This relies on the ability to poison the caching proxy run by the application itself, CDNs, or other downstream providers. As a result, the victim will have no control over receiving the malicious content when requesting the vulnerable application.
    - **X-Forwarded Host Header Bypass**
      * Host header injection is mitigated by preventing the tampering of Host header. It means if any request is made with tampered host header, the application responds with an error message like 404 Not Found Another way to pass arbitrary Host headers is to use the **X-Forwarded-Host** **header.** In some configurations, this header will rewrite the value of the Host header Therefore it’s possible to make the following request.
        + GET / HTTP/1.1
        + Host: [www.example.com](http://www.example.com/)
        + X-Forwarded-Host: [www.attacker.com](http://www.attacker.com/)
  + **Cache deception and poisoning** 
    - **Concepts**
      * **What is caching ?**
        + A cache in computing is a temporary store of any content that has been retrieved from its original (master) source. Caches are typically used so that the data can be served faster the next time it is requested, since it needs only be retrieved from the local cache rather than the original source.
      * **Web Caches**
        + Traffic on the server can grow as nb of requests grows and nb of users accessing the web site also Retrieving content from web servers can be both slow – especially if the file being retrieved is large or the server remote topologically – as well as “expensive” to generate for the source web server in terms of computing power if it has to be dynamically generated.
        + Web Caches allow web servers to save and serve copies of responses to certain requests
      * **How web caching works ?**
        + A caching system will sit somewhere in-line between requesters (users) of a service or content and the server. When the first request is made, the request is stored temporarily in the cache. If another request is made (by a user) for the same web address or URL, the web cache can use the response that it stored before, rather than relaying back a fresh request to the origin server
      * **Types of Web Cache**
        + There are more than one type of web caches – they exist at several steps along the way between your browser and the source/origin server:
        + **Browser Caches** – browser caches use a portion of your device’s local disk storage to hold static copies of content such as web pages that you’ve previously visited to help speed up your online experience – when you visit a page again it may be loaded, invisibly to you, direct from your own PC’s cached copy – the request may never even leave your computer.
        + **Proxy web caches (load balancer )** – web proxies are often deployed by organisations to jointly cache requests from all the organisation’s users. They are located at the organisation’s network edge and can be very effective where many users are accessing common resources such as news websites.
        + **Internet Service Providers (ISPs)** – ISPs also typically operate proxy caches via interception proxies on their underlying network, leveraging the scale of possibly hundreds of thousands of users to cache frequently-requested content for all their subscribers.
        + **Content Delivery Networks (CDN)** – CDNs such as cloudflare , Akamai or Speedera are located across the world and are generally leased by commercial organisations who produce content. When configured in DNS, customers performing a DNS lookup for the origin server will receive an IP for a local CDN server operated by the CDN company that is authorised to masquerade as if it were the origin server, caching content from that provider for all users within a given region and drastically reducing the volume of requests made to the origin servers.
        + **Gateway web caches** – also known as surrogate caches, or reverse proxy caches – are typically used by website owners or managers, to make their sites more reliable and scalable.
        + **Server memory caches** – Examples include Memcached and Varnish and can run on same exact local host as the content origin source, generating static cached copies of dynamically-generated pages to save re-generation of them on the next request.
      * **when to cache ?** 
        + So caching sounds like it is massively beneficial, reducing both network delay (lag) and server loads. However, caching has limitations. A caching system needs to have at least two key functionalities in place:

Time limits for how long to cache an item (i.e. File, HTTP Request, etc.) for; and

A system for determining whether or not a given request matches (“hits”) a cached copy of the data (therefore quick response) or “misses” it (therefore needs to ask the application)

* + - * + Time Limits

Resources change and are updated so the cache needs to have a finite lifetime for each cached resource. We wouldn’t be happy if we requested the BBC News webpage on May 5th and found out that the news we were reading was from September the previous year. Cache items therefore have expiry time limits.

* + - * + Hits and misses

Whenever a cache receives a request for a resource, it needs to decide whether it has a copy of this exact resource already saved and can reply with that, or if it needs to forward the request to the application server. For static content such as images, this might be simple – a photo of a cat is a photo of a cat, whoever requests it. However some pages are dynamic – if we access www.ourbank.com/balance then we want to see our own bank balance, not someome else’s (and vice versa) so caches can’t simply cache all content for all users.

* + - * + Caches and their origin sources tackle this problem collaboratively using an agreed arrangement of “cache keys” – a few specific components of a HTTP request that are taken to uniquely identify the resource being requested. This can be done manually through the use of HTTP headers. The owner of the source data can ensure that the origin server returns headers to mark a resource as public or private, set a maximum age to store it, ask the browser to re-validate this resource each time or tell the browser not to cache a specific resource
        + multiple request to the same content will be cached on a key
      * **Caching keys**
        + When the cache receives an HTTP request, it first has to determine whether there is a cached response that it can serve directly, or whether it has to forward the request for handling by the back-end server. Caches identify equivalent or identical requests by comparing a predefined some of the http request's components, known collectively as the "cache key". Such as {user-agent , host header , url parameters , … }.
        + Components of the request that are not seen by the user because they are added by the proxy server are said to be "unkeyed". Such as

X-Forwarded-Host : (Host header attack)

X-Host : (Host header attack)

X-forwarded-Server :

X-Original-Url :

X-Rewrite-Url :

* + - * + if the cache key of an incoming request matches the key of a previous request, then the cache considers them to be equivalent. As a result, it will serve a copy of the cached response that was generated for the original request. This applies to all subsequent requests with the matching cache key, until the cached response expires.
    - **web cache attacks types**
      * **Web Cache Deception (WCD) Attack**
        + WCD attacks arise when there is a contradiction between how a cache and an origin server interpret a given HTTP request. For instance, an attacker can craft a URL that points to the account information on a banking website but append to it a non-existent path component disguised as a static image, such as “/account.php/nonexistent.jpg.” Many origin servers will simply ignore the invalid suffix as spurious and respond with account details.
        + However, a web cache proxying the content may be oblivious to the processing that happens on the origin server to insert dynamic user-specific content, and store the response as if it were an image because of the file extension. If the attacker can trick a user into clicking on this link, the victim’s account information will be cached. The attacker can then access the same URL, retrieving the cached content containing the (dynamic) information personal to that user, and giving the attacker an opportunity to steal it.
      * **Web Cache Poisoning (WCP) Attack**
        + In a nutshell this is when an attacker manages to successfully embed malicious content in a request that gets reflected back into the server response and saved in the cache. The cache server will then unwittingly serve up the malicious response to anyone subsequently requesting the resource.
        + For example, an attacker could find a webpage vulnerable to reflected XSS. However reflected XSS is difficult for the hacker to exploit, since it requires getting a user to click on a specific link with the malicious payload within in. If however the attacker finds that the server in question has a cache and will store the malicious response payload in that cache, then future users requesting the page will have a malicious response returned from the cache. Executing the attacker’s arbitrary JavaScript.
    - **how to test for web cache poisoning:**
      * The first step is to identify unkeyed inputs. Doing this manually is tedious so fortunetly there is a Burp Suite extension called **Param Miner** that automates this step by guessing header/cookie names, and observing whether they have an effect on the application's response.
        + After finding an unkeyed input, the next steps are to assess how much damage you can do with it, then try and get it stored in the cache. If that fails, you'll need to gain a better understanding of how the cache works and hunt down a cacheable target page before retrying. Whether a page gets cached may be based on a variety of factors including the file extension, content-type, route, status code, and response headers.
        + Cached responses can mask unkeyed inputs, so if you're trying to manually detect or explore unkeyed inputs, a cache-buster is crucial. If you have Param Miner loaded, you can ensure every request has a unique cache key by adding a parameter with a value of $randomplz to the query string.
        + When auditing a live website, accidentally poisoning other visitors is a perpetual hazard. Param Miner mitigates this by adding a cache buster to all outbound requests from Burp. This cache buster has a fixed value so you can observe caching behaviour yourself without it affecting other users.
      * Once you have identified an unkeyed input, the next step is to evaluate exactly how the website processes it. Understanding this is essential to successfully eliciting a harmful response. If an input is reflected in the response from the server without being properly sanitized, or is used to dynamically generate other data, then this is a potential entry point for web cache poisoning
      * Finally send the request multiple times (10000) by an intruder such as burp intruder to be cached by the server
    - **cache poisoning attack could lead to :** 
      * stored xss
      * information disclosure
      * ddos attacks
      * unauthenticated access to sensitive pages
      * password reset attack
    - **Preventing web cache poisoning** 
      * Defending yourself against Cache Poisoning attacks can be quite tricky. Disabling caching entirely is one such way which is not feasible for most and understandably so. Some helpful methods however are to:
        + Heavily cache static response, such as \*.js, \*.css, \*.png files, blog posts, landing pages or any page that is always identical.
        + Make sure you are not vulnerable to Cross-site Scripting attacks so that even in the event of such a vulnerability, the user’s browser can’t be exploited.
        + Understand and restrict where caching is done. Are you using frameworks that implement their own caching? If so you may want to disable that and handle caching at a singular point (e.g. CloudFlare).
        + Avoid using user inputs (i.e. HTTP Headers) to be used as the cache key.
  + **CRLF Injection**
    - **Definition**
      * When a browser sends a request to a web server, the web server answers back with a response containing both the HTTP response headers and the actual website content, i.e. the response body. The HTTP headers and the HTML response (the website content) are separated by a specific combination of special characters, namely a carriage return and a line feed. For short they are also known as CRLF.
      * The web server uses the CRLF to understand when new HTTP header begins and another one ends. The CRLF can also tell a web application or user that a new line begins in a file or in a text block. The CRLF characters are a standard HTTP/1.1 message, so it is used by any type of web server, including Apache, Microsoft IIS and all others.
      * The term CRLF refers to Carriage Return (ASCII 13, \r) Line Feed (ASCII 10, \n). They're used to note the termination of a line, however, dealt with differently in today’s popular Operating Systems. For example: in Windows both a CR and LF are required to note the end of a line, whereas in Linux/UNIX a LF is only required. In the HTTP protocol, the CR-LF sequence is always used to terminate a line.
      * In a CRLF injection vulnerability attack the attacker inserts both the carriage return and linefeed characters into user input to trick the server, the web application or the user into thinking that an object is terminated and another one has started This is most commonly done by modifying an HTTP parameter or URL. . As such the CRLF sequences are not malicious characters, however they can be used for malicious intend, for HTTP response splitting etc.
    - **Ex of a normal HTTP response**
      * An HTTP message response includes two parts :
      * Message Headers – metadata that describes a request or response Each terminated by a carriage return (\r) and a linefeed (\n) Then the Message Body which is the raw data of the response
        + GET http://www.google.com/ HTTP/1.1\r\n
        + Host: www.google.com\r\n
        + User-Agent: Mozilla/5.0 (Windows ;U ; Windows NT 5.1; en-US; ) Firefox/3.0.1 Paros/3.2.13\r\n
        + Accept: text/html,application/xhtml+xml,application/xml; q=0.9,\*/\*;q=0.8\r\n
        + Accept-Language: en-us,en;q=0.5\r\n
        + Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7\r\n
        + Keep-Alive: 300\r\n
        + \r\n -> carriage return that separates the response headers and the response body
        + <HTML>\r\n
        + <HEAD>\r\n
        + <TITLE>Your Title Here</TITLE>\r\n
        + </HEAD>\r\n
        + <BODY>\r\n
        + </BODY>\r\n
        + …
        + </HTML>\r\n
      * Those two consecutive carriage-return-linefeed pairs are the source of HTTP response splitting vulnerabilities , The HTTP response splitting vulnerability is not the attack, it is simply the path that makes it possible
    - **CRLF can lead to :**
      * **HTTP Response Splitting**
        + When CRLF injection is used to split an HTTP response header, it is referred to as HTTP Response Splitting
        + The attacker sends a single HTTP request that forces the web server to form an output stream, which is then interpreted by the target as two HTTP responses instead of one response.
        + Since the header of a HTTP response and its body are separated by CRLF characters an attacker can try to inject those. A combination of CRLF CRLF will tell the browser that the header ends and the body begins. That means that he is now able to write data inside the response body where the html code is stored.
        + Remember that you need two \r\n sequences between the headers and the body and %0d%0a is the URL encoding of the \r\n
        + This vulnerability is usually found in case of redirections and when users have control over location headers
        + EX add cookies :

Request

http://[www.example.net](http://www.example.net/)/%0D%0ASet-Cookie:mycookie=myvalue

HTTP Response

Connection: keep-alive

Content-Length: 178

Content-Type: text/html

Date: Mon, 09 May 2016 14:47:29 GMT

Location: https://[www.example.net](http://www.example.net/)/[INJECTION STARTS HERE]

Set-Cookie: mycookie=myvalue

X-Frame-Options: SAMEORIGIN

X-Sucuri-ID: 15016

x-content-type-options: nosniff

x-xss-protection: 1; mode=block

* + - * + Ex CRLF chained with Open Redirect :

//[www.google.com/%2F%2E%2E%0D%0AHeader-Test:test2](http://www.google.com/..%0D%0AHeader-Test:test2)

/[www.google.com/%2E%2E%2F%0D%0AHeader-Test:test2](http://www.google.com/%0D%0AHeader-Test:test2)

/[google.com/%2F..%0D%0AHeader-Test:test2](http://google.com/..%0D%0AHeader-Test:test2)

/%0d%0aLocation:%20[http://example.com](http://example.com/)

* + - * **CRLF to XSS**
        + /%0d%0aContent-Length:35%0d%0a%0d%0a<script>alert(document.domain)</script>
        + /%0d%0aContent-Length:35%0d%0aX-XSS-Protection:0%0d%0a%0d%0a23
        + /%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
      * **CRLF to web cache poisoning,**
      * **CRLF to cross user defacement**
      * **CRLF to log file injection**
        + Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.
        + Writing invalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs. This is called log injection.
        + Log injection vulnerabilities occur when:

Data enters an application from an untrusted source.

The data is written to an application or system log file.

* + - * + Successful log injection attacks can cause:

Injection of new/bogus log events (log forging via log injection)

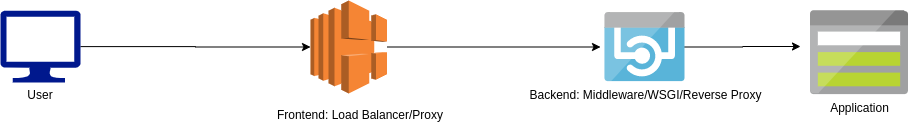
Injection of XSS attacks, hoping that the malicious log event isviewed in a vulnerable web application

Injection of commands that parsers (like PHP parsers) could execute

* + - * + Resources

<https://owasp.org/www-community/attacks/Log_Injection>

* + - **Example**
      * Add a fake HTTP response header: Content-Length: 0. this causes the web browser to treat this as a terminated response and begin parsing a new response.
      * Add a fake HTTP response: HTTP/1.1 200 OK. This begins the new response.
      * Add another fake HTTP response header: Content-Type: text/html. This is needed for the web browser to properly parse the content.
      * Add yet another fake HTTP response header: Content-Length: 25. This causes the web browser to only parse the next 25 bytes.
      * Add page content with an XSS: <script>alert(1)</script>. This content has exactly 25 bytes.
      * Because of the Content-Length header, the web browser ignores the original content that comes from the web server.
      * http://www.example.com/somepage.php?page=%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/html%0d%0aContent-Length:%2025%0d%0a%0d%0a%3Cscript%3Ealert(1)%3C/script%3E
    - **Filter Bypass**
      * %E5%98%8A = %0A = \u560a
      * %E5%98%8D = %0D = \u560d
      * %E5%98%BE = %3E = \u563e (>)
      * %E5%98%BC = %3C = \u563c (<)
    - **Preventing CRLF and HTTP response splitting**
      * You need to restrict CR(0x13) and LF(0x10) from the user input or properly encode the output in order to prevent the injection of custom HTTP headers
      * All inputs must be validated
      * Be aware of any use of input data in HTTP headers and code accordingly
  + **HTTP Request smuggling**
    - **Overview**



* + - * **Keep-Alive and pipelining**
        + The Keep-Alive header is a hop-by-hop header that provides information about a persistent connection. In web servers, Keep-Alive can be specified within the “Connection” header which allows a web server to keep a TCP socket/connection open. By using this header, multiple requests and responses can use a single connection which can reduce overhead and improve performance for a web server. This feature is supported by all browsers and servers today.
        + Pipelining is another feature that was introduced in RFC 2616. This allows a web server to process requests asynchronously—as a first-in-first-out stream rather than processing each request individually, allowing it to send a request without waiting for a previous response to arrive.
      * **Content-Length and Transfer-Encoding**
        + HTTP requests can have a message body. The presence of a message body in a request is signaled by a Content-Length or Transfer-Encoding header field. These headers are used for message framing, telling a server where a message ends and another begins.
        + The Content-Length, , is an HTTP header that indicates the size of the entity-body of the request. This is commonly seen in HTTP POST requests which have a body of data. It should be noted that GET requests typically shouldn’t contain the Content-Length header since they have no body.

The Content-Length header is straightforward: it specifies the length of the message body in bytes. For example:

POST /search HTTP/1.1

Host: [normal-website.com](http://normal-website.com)

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

Content-Length: 11

q=smuggling

* + - * + Transfer-Encoding, also specified in RFC 7230, was created to allow the sending of binary data over HTTP. Transfer-Encoding has numerous directives, this blog will focus on the chunked directive.
        + The chunked directive allows data to be sent in a series of chunks along with the length of these chunks specified in hexadecimal format, followed by carriage return and a line feed. The end of a chunked directive is stated by 0 and an empty sequence

POST / HTTP/1.1

Host: snyk.io

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

Transfer-Encoding: chunked

7 (length of chunks)

foo=bar (series of chunks)

0 (0 to terminate request followed by rn)

(rn)

* + - * HTTP Request Smuggling.is a security exploit on the HTTP protocol , . Most HTTP request smuggling vulnerabilities arise because the HTTP specification provides two different ways to specify where a request ends: the Content-Length header and the Transfer-Encoding header.
      * There are also different variations of Request Smuggling, which are known by the abbreviations symbolising the headers used in the attack. These are: CL:CL CL:TE TE:TE and TE:CL. The CL stands for the header value Content-Length and the TE value stands for the header Transfer-Encoding. To reduce complexity will only be providing detail about one method of Request Smuggling.
        + CL:CL: the front-end server uses the Content-Length header and the back-end server uses Content-Length the header.
        + CL.TE: the front-end server uses the Content-Length header and the back-end server uses the Transfer-Encoding header.
        + TE.CL: the front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header.
        + TE.TE: the front-end and back-end servers both support the Transfer-Encoding header, but one of the servers can be induced not to process it by obfuscating the header in some way
      * This vulnerability can be exploited to conduct phishing attacks, cache poisoning, [Cross-Site Scripting (XSS)](https://snyk.io/learn/cross-site-scripting/), bypass security controls, gain unauthorized access to sensitive data, and directly compromise other application users and more
    - **Explanation**
      * Today's web applications frequently employ chains of HTTP servers between users and the ultimate application logic. Users send requests to a front-end server (sometimes called a load balancer or reverse proxy) and this server forwards requests to one or more back-end servers. This type of architecture is increasingly common, and in some cases unavoidable, in modern cloud-based applications.
      * When the front-end server forwards HTTP requests to a back-end server, it typically sends several requests over the same back-end network connection, because this is much more efficient and performant. The protocol is very simple: HTTP requests are sent one after another, and the receiving server parses the HTTP request headers to determine where one request ends and the next one begins
      * In this situation, it is crucial that the front-end and back-end systems agree about the boundaries between requests. Otherwise, an attacker might be able to send an ambiguous request that gets interpreted differently by the front-end and back-end systems:
      * Since the HTTP specification provides two different methods for specifying the length of HTTP messages, it is possible for a single message to use both methods at once, such that they conflict with each other. The HTTP specification attempts to prevent this problem by stating that if both the Content-Length and Transfer-Encoding headers are present, then the Content-Length header should be ignored. This might be sufficient to avoid ambiguity when only a single server is in play, but not when two or more servers are chained together. In this situation, problems can arise for two reasons:
        + Some servers do not support the Transfer-Encoding header in requests.
        + Some servers that do support the Transfer-Encoding header can be induced not to process it if the header is obfuscated in some way.
      * If the front-end and back-end servers behave differently in relation to the (possibly obfuscated) Transfer-Encoding header, then they might disagree about the boundaries between successive requests, leading to request smuggling vulnerabilities
    - **Prevent HTTP Request Smuggling**
      * HTTP request smuggling vulnerabilities arise in situations where a front-end server forwards multiple requests to a back-end server over the same network connection, and the protocol used for the back-end connections carries the risk that the two servers disagree about the boundaries between requests. Some generic ways to prevent HTTP request smuggling vulnerabilities arising are as follows:
        + Disable reuse of back-end connections, so that each back-end request is sent over a separate network connection.
        + Use HTTP/2 for back-end connections, as this protocol prevents ambiguity about the boundaries between requests.
      * Use exactly the same web server software for the front-end and back-end servers, so that they agree about the boundaries between requests.
      * In some cases, vulnerabilities can be avoided by making the front-end server normalize ambiguous requests or making the back-end server reject ambiguous requests and close the network connection. However, these approaches are potentially more error-prone than the generic mitigations identified above.
    - **Resources**
      * <https://snyk.io/blog/demystifying-http-request-smuggling/>
      * <https://portswigger.net/web-security/request-smuggling>
      * <https://blog.detectify.com/2020/05/28/hiding-in-plain-sight-http-request-smuggling/#:~:text=What%20is%20HTTP%20request%20smuggling,servers%20together%20with%20different%20configurations.>
      * <https://portswigger.net/web-security/request-smuggling/finding>
  + **Insecure Desirialization**
    - **Terms and Definitions**
      * Serialization (marshaling):
        + It is the process of translating data structures or object state into bytes format that can be stored on disk or database or transmitted over the network.
      * Deserialization (marshaling):
        + It is the opposite process, which means to, extract data structure or object from series of bytes
      * Risk for using serialization:
        + The risk raisers, when an untrusted deserialization user inputs by sending malicious data to be de-serialized and this could lead to logic manipulation or arbitrary code execution.
    - **How to identify insecure deserialization**
      * Identifying insecure deserialization is relatively simple regardless of whether you are whitebox or blackbox testing.
      * During auditing, you should look at all data being passed into the website and try to identify anything that looks like serialized data. Serialized data can be identified relatively easily if you know the format that different languages use. In this section, we'll show examples from both PHP and Java serialization. Once you identify serialized data, you can test whether you are able to control it.
      * Tip: For users of Burp Suite Professional, Burp Scanner will automatically flag any HTTP messages that appear to contain serialized objects.
    - **Java serialization format**
      * Some languages, such as Java, use binary serialization formats. This is more difficult to read, but you can still identify serialized data if you know how to recognize a few tell-tale signs. For example, serialized Java objects always begin with the same bytes, which are encoded as ac ed in hexadecimal and rO0 in Base64.
      * Any class that implements the interface java.io.Serializable can be serialized and deserialized. If you have source code access, take note of any code that uses the readObject() method, which is used to read anddeserialize data from an InputStream.
    - **PHP serialization format**
      * PHP uses a mostly human-readable string format, with letters representing the data type and numbers representing the length of each entry. For example, consider a User object with the attributes:
        + $user->name = "carlos";
        + $user->isLoggedIn = true;
        + echo serialize($user);
      * When serialized, this object may look something like this:
        + O:LENTH\_OF\_NAME:"CLASS\_NAME":NUMBER\_OF\_PROPERTIES:{PROPERTIES}​
        + O:4:"User":2:{s:4:"name":s:6:"carlos"; s:10:"isLoggedIn":b:1;}
      * This can be interpreted as follows:

O:4:"User" - An object with the 4-character class name "User"

2 - the object has 2 attributes

s:4:"name" - The key of the first attribute is the 4-character string "name"

s:6:"carlos" - The value of the first attribute is the 6-character string "carlos"

s:10:"isLoggedIn" - The key of the second attribute is the 10-character string "isLoggedIn"

b:1 - The value of the second attribute is the boolean value true

* + - * The native methods for PHP serialization are serialize() and unserialize(). If you have source code access, you should start by looking for unserialize() anywhere in the code and investigating further.
      * **Exploiting Php deserialization (php object injection)**
        + When you control a serialized object that is passed into unserialize(), you control the properties of the created object and its values this is called object injection
        + One possible way of exploiting a PHP object injection vulnerability is variable manipulation and Controlling variable values

Ex : In this serialize string, you can try to change the value of “status” to “admin”, and see if the application grants you admin privileges.

O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:9:"not admin";}

O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:5:"admin";}

* + - * + PHP Object Injection could allow an attacker to perform different kinds of malicious attacks, such as [Code Injection](https://owasp.org/www-community/vulnerabilities/Code_Injection), [SQL Injection](https://owasp.org/www-community/attacks/SQL_Injection), [Path Traversal](https://owasp.org/www-community/vulnerabilities/Path_Traversal) and [Application Denial of Service](https://owasp.org/www-community/vulnerabilities/Application_Denial_of_Service), depending on the context
        + In order to successfully exploit a PHP Object Injection vulnerability two conditions must be met:

The application must have a class which implements a **PHP magic method** (such as \_\_wakeup or \_\_destruct) that can be used to carry out malicious attacks, or to start a “POP chain”.

The unserializacion of an object which manipulation is feasible from the user’s side (i.e. a cookie storing data as a serialized object)

* + - **Resources**
      * <https://www.exploit-db.com/docs/english/44756-deserialization-vulnerability.pdf>
      * https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/
  + **Missing SPF Record**
    - **Definition**
      * Sender Policy Framework (SPF) is used to authenticate the sender of an email. An SPF record is a type of Domain Name Service (DNS) record that identifies which mail servers are permitted to send email on behalf of your domain. The purpose of an SPF record is to prevent spammers from sending messages with sender addresses of your domain , Missing SPF record allows hackers to send spam emails by using an email address that includes your domain name as its suffix
      * Together with the DMARC related information, this gives the receiver (or receiving systems) information on how trustworthy the origin of an email is. SPF is, just like DMARC, an email authentication technique that uses DNS (Domain Name Service). This gives you, as an email sender, the ability to specify which email servers are permitted to send email on behalf of your domain.
      * An SPF record is a DNS record that has to be added to the DNS zone of your domain. In this SPF record you can specify which IP addresses and/or hostnames are authorized to send email from the specific domain. The mail receiver will use the “envelope from” address of the mail (mostly the Return-Path header) to confirm that the sending IP address was allowed to do so. This will happen before receiving the body of the message. When the sending email server isn’t included in the SPF record from a specific domain the email from this server will be marked as suspicious and can be rejected by the email receiver.
    - **SPF record structure**
      * Each spf record contains 3 parts
        + the declaration part that its an spf
        + the ip addresses that are allowed to send emails on behalf of your domain other than their mail server
        + the enforcement rule

the enforcement rule have the following options

-all (Hard fail) = the mail should be rejected

~all (soft fail) = (recommended) Typically, messages that return a SOFTFAIL are accepted but tagged.or found in junk

?all (neutral) =

+all = means any host can send mails on behalf of the domain (this should never be used)

* + - * v=spf <ip 1> <ip 2> <ip 3> <enforcement rule>
    - **check for vulnerability** 
      * search in dns searching websites such as mxtools.com for {spf www.example.com} , if the result is {Dns record not found thats means it has no spf record and its vulnerable}
      * to exploit this vulnerability we can use a phishing website such as "emkei.cz" and send fake emails with the domain of the vulnerable organization admin@example.com to anyone and this mail won’t be in the spam
    - **Fixing the vulnerability**
      * Add an SPF record for your domain name at your DNS provider.
        + Create a new TXT record.
        + Set the Host field to the name of your subdomain (for example, mail if your email address is contact@mail.example.com), or to @ if you do not use a subdomain.
        + Enter your SPF record for the TXT Value field (for example, v=spf1 a mx include:secureserver.net ~all)
  + **php type juggling**
    - **Explanation**
      * php Has two main comparison modes, let’s call them loose (==) and strict (===)
      * When using the "*==*" operator, PHP attempts something called loose comparison (or 'type juggling'). With loose comparison, it's possible for a PHP developer to compare values even if they have a different data type, such as integers and strings
      * Comparing a string to an integer is a little bit like comparing apples and oranges. It is impossible, and the final food metaphor you'll read in this post! PHP somehow has to convert both values to the same data type. When PHP does a comparison, it always attempts to convert the string to an integer first. PHP has a feature called “type juggling”, or “type coercion”. This means that during the comparison of variables of different types, PHP will first convert them to a common, comparable type.
      * To do this, it checks the beginning of the string to see if there are numbers it can use for comparison. The string '*test*' doesn't begin with a number. In PHP, this means that it is equal to the integer '*0*', which explains why the comparison returns true. It's a little different for strings and boolean values. PHP always tries to convert a string to boolean if both are compared. It does this by treating strings with any given content as *TRUE*. For example '*test*' *== TRUE*, while an empty string is always treated as *FALSE*
    - *For example,*
      * *when PHP needs to compare the string “Password2” to the integer 2, PHP will attempt to extract the integer from the string. So this comparison will evaluate to True.*
        + *(“Password2” == 2) -> True*
      * *But what if the string that is being compared does not contain an integer? The string will then be converted to a “0”. So the following comparison will also evaluate to True:*
        + *(“Password” == 0) -> True*
    - *True conditions*
      * 0xABCdef' == ' 0xABCdef' // true
      * '0010e2' == '1e3' // true
      * '123' == 123 // true
      * 'abc' == 0 // true
      * '123a' == 123 // true
      * '0x01' == 1 // true, php7.0 부터는 false
      * '' == 0 == false == NULL // all the same
      * 1.0000000000000001 == 1 // true
    - ***Exploiting type juggling***
      * The most common way that this particularity in PHP is exploited is by using it to bypass authentication.
      * Authentication bypass - Type juggling
        + Vulnerable code:
        + $data = unserialize($\_COOKIE['auth']);
        + if ($data['username'] == $adminName && $data['password'] == $adminPassword) {
        + $admin = true;
        + } else {
        + $admin = false; }
      * **Payload:**
        + a:2:{s:8:"username";b:1;s:8:"password";b:1;}
    - ***Avoid Type Juggling***
      * The behavior I have just described is obviously undesirable in the many instances in which you want to have an exact comparison of two given values. For this reason, PHP also has the === operator. Since good things come in threes, three equal signs mean that the comparison should take the data type into consideration. If the type differs, PHP will not attempt any conversions; it will return *FALSE* instead. This is a relatively easy way to deal with PHP's often confusing behavior when it comes to type comparisons. Simply add a single character.
  + **Php remote xdebug vulnerability**
    - **Introduction**
      * The PHP Xdebug module installed on the remote host is configured in a vulnerable manner and is less than or equal to version 2.5.5
      * Xdebug is a php extension that allows to debug php pages, remotely by using DGBp protocoL. PHP programmers or web security researchers always setup a local PHP debugging environments for convenience. If the debugging server can be directly accessed by an attacker, there is a Remote Code Execution vulnerability.
    - **How to find the vulnerability** 
      * looking into the response header Xdebug: 2.5.5, we can get to know that Xdebug is enabled in the server,
    - **Exploit the vulnerability** 
      * By default xdebug will listen on port 9000 on the developer’s machine. In order to activate xdebug the paremeter ‘XDEBUG\_SESSION=name’ has to be present in a get/post-parameter or as a cookie. The value of the parameter can be randomly chosen since it is only used for development environments where multiple developers work on same machine simultaneously and each developer has its own session. Once xdebug is active, it connects back to the configured IP on port 9000 and waits for instructions
      * To start exploit this service, add XDEBUG\_SESSION\_START to query parameters. Remote server will try to connect back to user on port tcp/9000. The endpoint where to connect to can be altered using X-Forwarded-For header.. (which points to attacker's server )Then we can get Remote Code Execution using the eval function in php
      * For php programmers, there are always index.php or config.php in their local debugging environment.
        + http://127.0.0.1/index.php?XDEBUG\_SESSION\_START or
        + http://127.0.0.1/config.php?XDEBUG\_SESSION\_START
      * Steps :
        + 1. The remote attacker sends an HTTP request to a PHP resource on the target server containing the XDebug parameter or cookie.
        + 2. The server makes an outbound DBGP connection to the attacker on port 9000 using the specified session identifier.
        + 3. The attacker executes DBGP commands to have malicious PHP code evaluated on the server.
        + 4. After the DBGP session is closed, the server sends the response to the original HTTP request.
    - **Remedation**
      * Upgrade to Xdebug version 2.60 or later. Additionally, the following line may be removed from the Xdebug configuration:
      * xdebug.remote\_connect\_back= true Or Set xdebug.remote\_connect\_back to 0 (the default value).
* If it’s set to true it connects back to any host that activates xdebug.
  + - **Resources**
      * https://fir3wa1-k3r.github.io/2018/09/22/Olympus-writeup.html
      * https://xlab.tencent.com/en/2018/04/02/pwning-php-developers/
      * https://github.com/vulhub/vulhub/tree/master/php/xdebug-rce
  + **Using components with known vulnerabilities:** 
    - If a component (e.g. libraries, frameworks, and other software modules) is not kept up to date or patched, it may be exposed to any number of known vulnerabilities, most of which have public exploits available
  + **Race Condition Vulnerability?**
    - A race condition attack happens when a computing system that’s designed to handle tasks in a specific sequence is forced to perform two or more operations simultaneously. without locking or synchronization
    - race condition vulnerabilities are an artifact of parallel processing. The ability to run multiple different execution threads in parallel can create vulnerabilities that would not exist in single-threaded programs
  + **HTTP Verb Tampering** 
    - is an attack that exploits vulnerabilities in HTTP verb (also known as HTTP method) authentication and access control mechanisms. Many authentication mechanisms only limit access to the most common HTTP methods, thus allowing unauthorized access to restricted resources by other HTTP methods.
  + **Insufficient logging and Monitoring**
  + It isn’t a vulnerability in itself, insufficient logging and Monitoring is an OWASP category that covers the lack of various best practices that could in turn prevent or damage control security breaches.
  + The category includes everything from unlogged events, logs that are not stored properly and warnings where no action is taken within reasonable time.
  + **When is an application vulnerable?**
    - According to OWASP policy, insufficient logging, detection, and monitoring occur in the following cases:
      * Verifiable events such as logins, failed logins and high-value transactions are not logged.
      * Warnings and errors result in no, insufficient or unclear log messages. This includes obscure error logging without sufficient detail for forensics to understand.
      * Application and API logs are not monitored for suspicious activity.
      * Logs are only stored locally. Logs that are not backed up run the risk of being deleted by intruders accessing a system. In this way, the intruders conceal their traces, so that the source of the intrusion is not traceable.
      * Adequate alarm thresholds and reaction escalation processes are absent or ineffective.
      * Penetration tests and scans by DAST tools (e.g. OWASP ZAP) do not trigger any warnings.
      * The application cannot detect, escalate or warn against active attacks in real time.
  + **Prevention**
    - Make sure the logs are backed up and synced to another server. The attacker should not be able to clear all the logs after hacking the server and by doing so preventing any forensics.
    - Go over the system and make sure sensitive actions are logged. This would include logins, high value transactions, password changes, and so on. This is valuable when investigating a hack afterwards.
    - Make it a routine to actually look at the most important logs and automate the process for the rest. There should be a system in place that alerts you if a specific warning has been triggered or if a certain warning threshold has been reached, so that proper action can be taken
  + **Bypass Rate Limit**
  + **What is rate Limit ?**
    - In computer networks, rate limiting is used to control the rate of requests sent or received by a network interface controller and is used to prevent DoS attacks
    - A rate limiting algorithm is used to check if the user session (or IP-address) has to be limited based on the information in the session cache. In case a client made too many requests within a given timeframe, HTTP-Servers can respond with status code 429: Too Many Requests.
    - Rate limiting is often employed to stop bad bots from negatively impacting a website or application. Bot attacks that rate limiting can help mitigate include:
      * Brute force attacks
      * DoS and DDoS attacks
      * Web scraping
    - Before we jump into bypass, let's dig into places where we can find it.
      * Login
      * Reset Password
      * 2FA
      * Sign up
      * Sending messages, Forums, comments, etc.
  + **Bypass rate limit methods**
    - **Bypassing Rate Limit with Header**
      * There are some headers which can be used to Bypass Rate Limitation. All you have to do is to Use theses Headers just under the Host Header in the Request.
        + X-Originating-IP: 127.0.0.1
        + X-Forwarded-For: 127.0.0.1
        + X-Remote-IP: 127.0.0.1
        + X-Remote-Addr: 127.0.0.1
        + X-Client-IP: 127.0.0.1
        + X-Host: 127.0.0.1
        + X-Forwared-Host: 127.0.0.1
      * or use double X-Forwared-For header
        + X-Forwarded-For:
        + X-Forwarded-For: 127.0.0.1
      * Change the IP whenever the Request gets Blocked Again.
      * TIP: Try adding multiple headers sometimes can bypass a rate limit too.
      * Try random words in this header  X-Forwarded-For: asdfgd
    - **Changing user-agents, cookies and IP address**
    - **Using null byte and CRLF**
      * If you are attacking the /api/v3/sign-up endpoint try to perform bruteforce to /Sing-up, /SignUp, /singup... Also try appending to the original endpoint bytes like %00, %0d%0a, %0d, %0a, %09, %0C, %20