**Web cache deception**

**\*\* in kali there are 2 good tools for testing this the best being the web-cache-deception scanner. I outfitted it to account for all portswigger delimiters and cacheable file extensions. \*\*Note: it will only use all of the payloads if we do “go run web-cache-scanner.go” we can just run the .sh version if we want a quick not as good scan.**

**\*\* for the static directory cache deception that uses a path traversal we will need to thoroughly test for this manually. Refer to the doc for more details on that.**

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

**IF BY USING THE TECHNIQUES LISTED IN ANY DOC IN THIS DIRECTORY, WE CAN GET AN HTML PAGE TO BE CACHED THERE IS ALMOST SURELY A VULNERABILITY TO BE FOUND. Each site has a huge number of html pages that are dynamically generated to some degree. Keep an eye out for literally any user specific information, username, uuids/user ids, api keys the list goes on forever.**

**VERY IMPORTANT: if we can get a page cached that contains some form we can likely harvest another users CSRF token in our POC. We can chain this with CSRF by simply finding an endpoint that uses a CSRF token, replacing the token with the one we found via cache deception, sending it to CSRF POC generator burp ext and throwing it in a site we control. Almost all forms will result in a POST request being sent to some endpoint, the POST request will contain a CSRF. This is a very simple way to increase impact and potentially gives us a wider cache deception attack surface.**

**Even a reflected user cookie could be impactful (but could potentially be escalated to cache poisoning)**

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

Web cache deception is a vulnerability that enables an attacker to trick a web cache into storing sensitive, dynamic content. It's caused by discrepancies between how the cache server and origin server handle requests.

In a web cache deception attack, an attacker persuades a victim to visit a malicious URL, inducing the victim's browser to make an ambiguous request for sensitive content. The cache misinterprets this as a request for a static resource and stores the response. The attacker can then request the same URL to access the cached response, gaining unauthorized access to private information.

**Cache keys review**

When the cache receives an HTTP request, it must decide whether there is a cached response that it can serve directly, or whether it has to forward the request to the origin server. The cache makes this decision by generating a 'cache key' from elements of the HTTP request. Typically, this includes the URL path and query parameters, but it can also include a variety of other elements like headers and content type.

If the incoming request's cache key matches that of a previous request, the cache considers them to be equivalent and serves a copy of the cached response.

**Cache rules review**

Cache rules determine what can be cached and for how long. Cache rules are often set up to store static resources, which generally don't change frequently and are reused across multiple pages. Dynamic content is not cached as it's more likely to contain sensitive information, ensuring users get the latest data directly from the server.

Web cache deception attacks exploit how cache rules are applied, so it's important to know about some different types of rules, particularly those based on defined strings in the URL path of the request. For example:

* Static file extension rules - These rules match the file extension of the requested resource, for example .css for stylesheets or .js for JavaScript files.
* Static directory rules - These rules match all URL paths that start with a specific prefix. These are often used to target specific directories that contain only static resources, for example /static or /assets.
* File name rules - These rules match specific file names to target files that are universally required for web operations and change rarely, such as robots.txt and favicon.ico.

Caches may also implement custom rules based on other criteria, such as URL parameters or dynamic analysis.

**Constructing a web cache deception attack**

Generally speaking, constructing a basic web cache deception attack involves the following steps:

1. Identify a target endpoint that returns a dynamic response containing sensitive information. Review responses in Burp, as some sensitive information may not be visible on the rendered page. Focus on endpoints that support the GET, HEAD, or OPTIONS methods as requests that alter the origin server's state are generally not cached.
2. Identify a discrepancy in how the cache and origin server parse the URL path. This could be a discrepancy in how they:
   * Map URLs to resources.
   * Process delimiter characters.
   * Normalize paths.
3. Craft a malicious URL that uses the discrepancy to trick the cache into storing a dynamic response. When the victim accesses the URL, their response is stored in the cache. Using Burp, you can then send a request to the same URL to fetch the cached response containing the victim's data. Avoid doing this directly in the browser as some applications redirect users without a session or invalidate local data, which could hide a vulnerability.

We'll explore some different approaches for constructing a web cache deception attack.

**Using a cache buster**

While testing for discrepancies and crafting a web cache deception exploit, make sure that each request you send has a different cache key. Otherwise, you may be served cached responses, which will impact your test results.

As both URL path and any query parameters are typically included in the cache key, you can change the key by adding a query string to the path and changing it each time you send a request. Automate this process using the Param Miner extension. To do this, once you've installed the extension, click on the top-level **Param miner > Settings** menu, then select **Add dynamic cachebuster**. Burp now adds a unique query string to every request that you make. You can view the added query strings in the **Logger** tab.

**Detecting cached responses**

During testing, it's crucial that you're able to identify cached responses. To do so, look at response headers and response times.

Various response headers may indicate that it is cached. For example:

* The X-Cache header provides information about whether a response was served from the cache. Typical values include:
  + X-Cache: hit - The response was served from the cache.
  + X-Cache: miss - The cache did not contain a response for the request's key, so it was fetched from the origin server. In most cases, the response is then cached. To confirm this, send the request again to see whether the value updates to hit.
  + X-Cache: dynamic - The origin server dynamically generated the content. Generally this means the response is not suitable for caching.
  + X-Cache: refresh - The cached content was outdated and needed to be refreshed or revalidated.
* The Cache-Control header may include a directive that indicates caching, like public with a max-age higher than 0. Note that this only suggests that the resource is cacheable. It isn't always indicative of caching, as the cache may sometimes override this header.

If you notice a big difference in response time for the same request, this may also indicate that the faster response is served from the cache.

**Exploiting static extension cache rules**

Cache rules often target static resources by matching common file extensions like .css or .js. This is the default behavior in most CDNs.

If there are discrepancies in how the cache and origin server map the URL path to resources or use delimiters, an attacker may be able to craft a request for a dynamic resource with a static extension that is ignored by the origin server but viewed by the cache.

**Path mapping discrepancies**

URL path mapping is the process of associating URL paths with resources on a server, such as files, scripts, or command executions. There are a range of different mapping styles used by different frameworks and technologies. Two common styles are traditional URL mapping and RESTful URL mapping.

Traditional URL mapping represents a direct path to a resource located on the file system. Here's a typical example:

http://example.com/path/in/filesystem/resource.html

* http://example.com points to the server.
* /path/in/filesystem/ represents the directory path in the server's file system.
* resource.html is the specific file being accessed.

In contrast, REST-style URLs don't directly match the physical file structure. They abstract file paths into logical parts of the API:

http://example.com/path/resource/param1/param2

* http://example.com points to the server.
* /path/resource/ is an endpoint representing a resource.
* param1 and param2 are path parameters used by the server to process the request.

Discrepancies in how the cache and origin server map the URL path to resources can result in web cache deception vulnerabilities. Consider the following example:

http://example.com/user/123/profile/wcd.css

* An origin server using REST-style URL mapping may interpret this as a request for the /user/123/profile endpoint and returns the profile information for user 123, ignoring wcd.css as a non-significant parameter.
* A cache that uses traditional URL mapping may view this as a request for a file named wcd.css located in the /profile directory under /user/123. It interprets the URL path as /user/123/profile/wcd.css. If the cache is configured to store responses for requests where the path ends in .css, it would cache and serve the profile information as if it were a CSS file.

**Exploiting path mapping discrepancies**

To test how the origin server maps the URL path to resources, add an arbitrary path segment to the URL of your target endpoint. If the response still contains the same sensitive data as the base response, it indicates that the origin server abstracts the URL path and ignores the added segment. For example, this is the case if modifying /api/orders/123 to /api/orders/123/foo still returns order information.

To test how the cache maps the URL path to resources, you'll need to modify the path to attempt to match a cache rule by adding a static extension. For example, update /api/orders/123/foo to /api/orders/123/foo.js. If the response is cached, this indicates:

* That the cache interprets the full URL path with the static extension.
* That there is a cache rule to store responses for requests ending in .js.

Caches may have rules based on specific static extensions. Try a range of extensions, including .css, .ico, and .exe.

You can then craft a URL that returns a dynamic response that is stored in the cache. Note that this attack is limited to the specific endpoint that you tested, as the origin server often has different abstraction rules for different endpoints.