**\*\*\* when testing use the dev tools network tab this will show how long a file one that site takes to load which will help to clearly visualize exactly what is being cached we can also turn off the cache from this tab. \*\*\***

**Background:**

In our previous labs, you learned how to exploit web cache poisoning vulnerabilities by manipulating typical unkeyed inputs, such as HTTP headers and cookies. While this approach is effective, it only scratches the surface of what is possible with web cache poisoning.

In this section, we'll demonstrate how you can access a much greater attack surface for web cache poisoning by exploiting quirks in specific implementations of caching systems. In particular, we'll look at why flaws in how cache keys are generated can sometimes leave websites vulnerable to cache poisoning via separate vulnerabilities that are traditionally considered unexploitable. We'll also show how you can take classic techniques even further to potentially poison application-level caches, often with devastating results.

**Cache key flaws**

The behavior of individual caching systems is not always as you would expect. In practice, many websites and CDNs perform various transformations on keyed components when they are saved in the cache key. This can include:

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

These transformations may introduce a few unexpected quirks. These are primarily based around discrepancies between the data that is written to the cache key and the data that is passed into the application code, even though it all stems from the same input. These cache key flaws can be exploited to poison the cache via inputs that may initially appear unusable.

In the case of fully integrated, application-level caches, these quirks can be even more extreme. In fact, internal caches can be so unpredictable that it is sometimes difficult to test them at all without inadvertently poisoning the cache for live users.

**Probing Methodology:**

The methodology of probing for cache implementation flaws differs slightly from the classic web cache poisoning methodology. These newer techniques rely on flaws in the specific implementation and configuration of the cache, which may vary dramatically from site to site. This means that you need a deeper understanding of the target cache and its behavior.

The methodology generally involves the following steps:

1. [Identify a suitable cache oracle](https://portswigger.net/web-security/web-cache-poisoning/exploiting-implementation-flaws#identify-a-suitable-cache-oracle)
2. [Probe key handling](https://portswigger.net/web-security/web-cache-poisoning/exploiting-implementation-flaws#probe-key-handling)
3. [Identify an exploitable gadget](https://portswigger.net/web-security/web-cache-poisoning/exploiting-implementation-flaws#identify-an-exploitable-gadget)

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

**Step 1: Identify a suitable cache oracle**

The first step is to identify a suitable "cache oracle" that you can use for testing. A cache oracle is simply a page or endpoint that provides feedback about the cache's behavior. This needs to be cacheable and must indicate in some way whether you received a cached response or a response directly from the server. This feedback could take various forms, such as:

* An HTTP header that explicitly tells you whether you got a cache hit
* Observable changes to dynamic content
* Distinct response times

Ideally, the cache oracle will also reflect the entire URL and at least one query parameter in the response. This will make it easier to notice parsing discrepancies between the cache and the application, which will be useful for constructing different exploits later.

If you can identify that a specific third-party cache is being used, you can also consult the corresponding documentation. This may contain information about how the default cache key is constructed. You might even stumble across some handy tips and tricks, such as features that allow you to see the cache key directly. For example, Akamai-based websites may support the header Pragma: akamai-x-get-cache-key, which you can use 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

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

**Step 2: Probe key handling**

The next step is to investigate whether the cache performs any additional processing of your input when generating the cache key. You are looking for an additional attack surface hidden within seemingly keyed components.

You should specifically look at any transformation that is taking place. Is anything being excluded from a keyed component when it is added to the cache key**? Common examples are excluding specific query parameters, or even the entire query string, and removing the port from the Host header.**

If you're fortunate enough to have direct access to the cache key, you can simply compare the key after injecting different inputs. Otherwise, you can use your understanding of the cache oracle to infer whether you received the correct cached response. For each case that you want to test, you send two similar requests and compare the responses.

Let's say that our hypothetical cache oracle is the target website's home page. This automatically redirects users to a region-specific page. It uses 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

To test whether the port is excluded from the cache key, we first need to request an arbitrary port and make sure that we receive a fresh response from the server that reflects this input:

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

Next, we'll send another request, but this time we won't specify a port:

GET / HTTP/1.1

Host: vulnerable-website.com

HTTP/1.1 302 Moved Permanently

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

Cache-Status: hit

As you can see, we have been served our cached response even though the Host header in the request does not specify a port. This proves that the port is being excluded from the cache key. Importantly, the full header is still passed into the application code and reflected in the response.

**In short, although the Host header is keyed, the way it is transformed by the cache allows us to pass a payload into the application while still preserving a "normal" cache key that will be mapped to other users' requests. This kind of behavior is the key concept behind all of the exploits that we'll discuss in this section.**

You can use a similar approach to investigate any other processing of your input by the cache. Is your input being normalized in any way? How is your input stored? Do you notice any anomalies?

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

Step 3: **Identify an exploitable gadget**

By now, you should have a relatively solid understanding of how the target website's cache behaves and might have found some interesting flaws in the way the cache key is constructed. The final step is to identify a suitable gadget that you can chain with this cache key flaw. This is an important skill because the severity of any web cache poisoning attack is heavily dependent on the gadget you are able to exploit.

These gadgets will often be classic client-side vulnerabilities, such as [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected), DOM-based vulnerabilities and open redirects. By combining these with web cache poisoning, you can massively escalate the severity of these attacks, turning a reflected vulnerability into a stored one. Instead of having to induce a victim to visit a specially crafted URL, your payload will automatically be served to anybody who visits the ordinary, perfectly legitimate URL.

Perhaps even more interestingly, these techniques enable you to exploit a number of unclassified vulnerabilities that are often dismissed as "unexploitable" and left unpatched. This includes the use of dynamic content in resource files, and exploits requiring malformed requests that a browser would never send.

**Unkeyed Ports:**

The Host header is often part of the cache key and, as such, initially seems an unlikely candidate for injecting any kind of payload. However, some caching systems will parse the header and exclude the port from the cache key.

In this case, you can potentially use this header for web cache poisoning. For example, consider the case we saw earlier where a redirect URL was dynamically generated based on the Host header. This might enable you to construct a denial-of-service attack by simply adding an arbitrary port to the request. All users who browsed to the home page would be redirected to a dud port, effectively taking down the home page until the cache expired.

This kind of attack can be escalated further if the website allows you to specify a non-numeric port. You could use this to inject an XSS payload, for example.

**Unkeyed query string**

Like the Host header, the request line is typically keyed. However, one of the most common cache-key transformations is to exclude the entire query string.

**Detecting an unkeyed query string**

If the response explicitly tells you whether you got a cache hit or not, this transformation is relatively simple to spot - but what if it doesn't? This has the side-effect of making dynamic pages appear as though they are fully static because it can be hard to know whether you are communicating with the cache or the server.

To identify a dynamic page, you would normally observe how changing a parameter value has an effect on the response. But if the query string is unkeyed, most of the time you would still get a cache hit, and therefore an unchanged response, regardless of any parameters you add. Clearly, this also makes classic cache-buster query parameters redundant.

Fortunately, there are alternative ways of adding a cache buster, such as adding it to a keyed header that doesn't interfere with the application's behavior. Some typical examples include:

Accept-Encoding: gzip, deflate, cachebuster

Accept: \*/\*, text/cachebuster

Cookie: cachebuster=1

Origin: https://cachebuster.vulnerable-website.com

If you use Param Miner, you can also select the options "Add static/dynamic cache buster" and "Include cache busters in headers". It will then automatically add a cache buster to commonly keyed headers in any requests that you send using Burp's manual testing tools.

Another approach is to see whether there are any discrepancies between how the cache and the back-end normalize the path of the request. As the path is almost guaranteed to be keyed, you can sometimes exploit this to issue requests with different keys that still hit the same endpoint. For example, the following entries might all be cached separately but treated as equivalent to GET / on the back-end:

Apache: GET //  
Nginx: GET /%2F  
PHP: GET /index.php/xyz  
.NET GET /(A(xyz)/

This transformation can sometimes mask what would otherwise be glaringly obvious reflected XSS vulnerabilities. If penetration testers or automated scanners only receive cached responses without realizing, it can appear as though there is no reflected XSS on the page.

**Exploiting an unkeyed query string**

Excluding the query string from the cache key can actually make these reflected XSS vulnerabilities even more severe.

Usually, such an attack would rely on inducing the victim to visit a maliciously crafted URL. However, poisoning the cache via an unkeyed query string would cause the payload to be served to users who visit what would otherwise be a perfectly normal URL. This has the potential to impact a far greater number of victims with no further interaction from the attacker.

\*\*\*\*\*\*\*\*\* So essentially we can try to add XSS payloads as a random parameter to the query we just have to insure we are not getting a cached response when we try this as if we are we would not see the XSS . This is likely to be blocked, but it is defintly worth trying if the qery string is reflected in the response. Payload ex: GET /?evil='/><script>alert(1)</script>

\*\*\* very important, param miner will not really help with this particular vulnerability. Often we will ONLY see any changes from our unkeyed query string when we get a cache miss. If were just seeing the cached response of course it will not show up. Thus, if we cannot find a cachebuster in the form of another header, we will have to carefully analysis if we are seeing a cached response or not. If we do not have a cache oracle this will be more difficult as we will need to analyze this based on time. Still, this is a great and relatively easy vuln to exploit and find, be sure to test for this. Also be sure your XSS is properly escaping the context, double and triple check. Even if it is partially escaped eg breaking out of quotes it may not have broken out of the html tag its in.

**Unkeyed Query Parameters:**

Sometimes sites will only “unkey” specific query parameters that are unimportant to the backend as opposed to the entire query string.

This is often the case for parameters that help with analytics or serving targeted advertisements. UTM parameters like ‘utm\_content’ are good candidates to check during testing.

Parameters that have been excluded from the cache key are unlikely to have a significant impact on the response. The chances are there won't be any useful gadgets that accept input from these parameters. That said, some pages handle the entire URL in a vulnerable manner, making it possible to exploit arbitrary parameters.

\*\*\*\* **use param miners guess query params feature to automatically find accepted query parameters. There is very often parameters not included in the normal request that will still be excepted and often unkeyed are they are sort of unexpected to the server.** \*\*\*\*

\*\*\* this is a must test on all applications. First check if the query string is keyed by adding a random param to it and seeing if you get a cache hit or miss. If the entire query string is unkeyed try to exploit according to the PREVIOUS section in this doc. Else, have param miner guess query params. If found, we can test by adding a arbitrary param to the query as a cache buster(as usually the query string is keyed besdies very specific params as found by param miner). Then add the param found by param miner and try to escape the context its reflected in. Once done send a normal request to the endpoint to repeater, and add the arbitrary param were using as a cache buster. Get the malicious response cached w cache buster and then try to get the malicious response with the “normal” response with the added cache buster. If you see your malicious payload you have exploited it successfully.

**Cache parameter cloaking:**

This is a way to obfuscate url params to get them keyed even when they are usually excluded from being keyed.

If the cache excludes a harmless parameter from the cache key, and you can't find any exploitable gadgets based on the full URL, you'd be forgiven for thinking that you've reached a dead end. However, this is actually where things can get interesting. (essentially once we’ve tested for all vulnerabilities in this doc up to this point)

Parameter cloaking is essentially hiding a parameter within another parameter that is excluded from the cache key. This will work when there are pasring descrepoencies between the application and the cache on how exactly they parse the url. We must test how the cache parses the url to exclude parameters.

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

For example, 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.

Let's assume that the algorithm for excluding parameters from the cache key behaves in this way, but the server's algorithm only accepts the first ? as a delimiter. Consider the following request:

GET /?example=123?excluded\_param=bad-stuff-here

In this case, 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. If the value of example is passed into a useful gadget, we have successfully injected our payload without affecting the cache key

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

**Exploiting parameter parsing quirks**

This specific example is useful when there is at least 1 keyed parameter in the url.

Similar parameter cloaking issues can arise in the opposite scenario, where the back-end identifies distinct parameters that the cache does not. The Ruby on Rails framework, for example, interprets both ampersands (&) and semicolons (;) as delimiters. When used in conjunction with a cache that does not allow this, you can potentially exploit another quirk to override the value of a keyed parameter in the application logic.

Consider the following request:

GET /?keyed\_param=abc&excluded\_param=123;keyed\_param=bad-stuff-here

As the names suggest, keyed\_param is included in the cache key, but excluded\_param is not. Many caches will only 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

But now there is a duplicate keyed\_param. This is where the second quirk comes into play. If there are duplicate parameters, each with different values, Ruby on Rails gives precedence to the final occurrence. The end result is that the cache key contains an innocent, expected parameter value, allowing the cached response to be served as normal to other users. On the back-end, however, the same parameter has a completely different value, which is our injected payload. It is this second value that will be passed into the gadget and reflected in the poisoned response.

This exploit can be especially powerful if it gives you control over a function that will be executed. For example, if a website is using JSONP to make a cross-domain request, this will often contain a callback parameter to execute a given function on the returned data:

GET /jsonp?callback=innocentFunction

In this case, you could use these techniques to override the expected callback function and execute arbitrary JavaScript instead.

**Exploiting fat GET support**

In select cases, the HTTP method may not be keyed. This might allow you to poison the cache with a POST request containing a malicious payload in the body. Your payload would then even be served in response to users' GET requests. Although this scenario is pretty rare, you can sometimes 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

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

\*\*\* this is literally just adding a body to a get request. This body should contain a param that works. Just run param miner fat get on it and it will show params that work. Use that param we found to throw it in the body of the get request and give it a payload. \*\*\*

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

This is only possible if a website accepts GET requests that have a body, but there are potential workarounds. You can sometimes encourage "fat GET" handling by overriding the HTTP method, for example:

GET /?param=innocent HTTP/1.1

Host: innocent-website.com

X-HTTP-Method-Override: POST

…

param=bad-stuff-here

As long as the X-HTTP-Method-Override header is unkeyed, you could submit a pseudo-POST request while preserving a GET cache key derived from the request line.

**Exploiting dynamic content in resource imports**

Imported resource files are typically static but some reflect input from the query string. This is mostly considered harmless because browsers rarely execute these files when viewed directly, and an attacker has no control over the URLs used to load a page's subresources. However, by combining this with web cache poisoning, you can occasionally inject content into the resource file.

For example, consider a page that reflects the current query string in an import statement:

GET /style.css?excluded\_param=123);@import… HTTP/1.1

HTTP/1.1 200 OK

…

@import url(/site/home/index.part1.8a6715a2.css?excluded\_param=123);@import…

You could exploit this behavior to inject malicious CSS that exfiltrates sensitive information from any pages that import /style.css.

If the page importing the CSS file doesn't specify a doctype, you can maybe even exploit static CSS files. Given the right configuration, browsers will simply scour the document looking for CSS and then execute it. This means that you can occasionally poison static CSS files by triggering a server error that reflects the excluded query parameter:

GET /style.css?excluded\_param=alert(1)%0A{}\*{color:red;} HTTP/1.1

HTTP/1.1 200 OK

Content-Type: text/html

…

This request was blocked due to…alert(1){}\*{color:red;}

**Normalized cache keys**

Any normalization applied to the cache key can also introduce exploitable behavior. In fact, it can occasionally enable some exploits that would otherwise be almost impossible.

For example, when you find reflected XSS in a parameter, it is often unexploitable in practice. This is because modern browsers typically URL-encode the necessary characters when sending the request, and the server doesn't decode them. The response that the intended victim receives will merely contain a harmless URL-encoded string.

Some caching implementations normalize keyed input when adding it to the cache key. In this case, both of the following requests would have the same key:

GET /example?param="><test>

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

This behavior can allow you to exploit these otherwise "unexploitable" XSS vulnerabilities. If you send a malicious request using Burp Repeater, you can poison the cache with an unencoded XSS payload. When the victim visits the malicious URL, the payload will still be URL-encoded by their browser; however, once the URL is normalized by the cache, it will have the same cache key as the response containing your unencoded payload.

As a result, the cache will serve the poisoned response and the payload will be executed client-side. You just need to make sure that the cache is poisoned when the victim visits the URL.

**Poisoning internal caches**

**\*\*\* so to find this we see if old inputs (from previous requests) are persistently being reflected on the page. Once we think weve identified an internal cache use param miner with the dynamic cache buster query param option or in some way w param miner add a dynamic cache buster. This will bypass the external cache. Once identified it can take MANY requests before the internal cache is persistently poisoned. Once poisoned we should see the poisoned response with any cache buster. We can further test this by removing our reflected input and observing if it is still reflected and where. This will indicate exactly what parameters are being used as a cache key by the internal server.**

So far, we've looked at how you can exploit flaws in the way external web caches are implemented to expose an extended attack surface hidden within seemingly keyed components. However, some websites implement caching behavior directly into the application in addition to using a distinct, external component. This can have several advantages, such as avoiding the kind of parsing discrepancies we looked at earlier.

As these integrated caches are purpose-built for the specific application, this also gives developers the freedom to tailor their behavior to a greater degree. As a result, these caches can sometimes behave in unusual ways that you wouldn't typically see from a more standardized, external cache that needs to be compatible with multiple applications. Sometimes, these strange behaviors can also provide an opportunity for some high-severity cache poisoning exploits.

Instead of caching entire responses, some of these caches break the response down into reusable fragments and cache them each separately. For example, a snippet for importing a widely used resource might be stored as a standalone cache entry. Users might then receive a response comprising a mixture of content from the server, as well as several individual fragments from the cache.

As these cached fragments are intended to be reusable across multiple distinct responses, the concept of a cache key doesn't really apply. Every response that contains a given fragment will reuse the same cached fragment, even if the rest of the response is completely different. In a scenario like this, poisoning the cache can have wide-reaching effects, especially if you poison a fragment that is used on every page. As there is no cache key, you would have poisoned every page, for every user, with a single request.

This will often only require you to use basic web cache poisoning techniques, such as manipulating the Host header.

**How to identify internal caches**

One of the challenges posed by integrated, application-level caches is that they can be difficult to identify and investigate because there is often no user-facing feedback. To identify these caches, you can look for a few tell-tale signs.

For example, if the response reflects a mixture of both input from the last request you sent and input from a previous request, this is a key indicator that the cache is storing fragments rather than entire responses. The same applies if your input is reflected in responses on multiple distinct pages, in particular on pages in which you never tried to inject your input.

Other times, the cache's behavior may simply be so unusual that the most logical conclusion is that it must be a unique and specialized internal cache.

When a website implements multiple layers of caching, it can make it difficult to comprehend what is happening behind the scenes and understand how the website's caching system behaves.