**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>