# Web Cache Poisoning Attacks

Web cache poisoning is typically used to distribute a different underlying vulnerability in the web application to a large number of users. This makes the exploitation of web cache poisoning very dependent on the specific web application. Additionally, poisoning the cache is often not trivial. In real-world scenarios, many users are accessing the web application at the same time as we are. Therefore, when we request a page it is most likely already cached and we are only served the cached response. In these cases, we need to send our malicious request at the exact time that the cache expires to get the web cache to store the response. Getting this timing right involves a lot of trial and error. However, it can be significantly easier if the web server reveals information about the expiry of the cache.

## Exploitation & Impact of Web Cache Poisoning

#### XSS

The exploitation of web cache poisoning depends on the underlying issue in the web application itself. In the previous section, we have seen an example of how web cache poisoning using an unkeyed GET parameter can be used to distribute reflected XSS vulnerabilities to unknowing users, eliminating the need for user interaction. Furthermore, we have seen that an XSS vulnerability via an unkeyed HTTP header that was unexploitable on its own can be weaponized with the help of web cache poisoning. XSS is one of the most common ways of exploiting web cache poisoning, though there are other ways as well.

#### Unkeyed Cookies

Another example is the exploitation of unkeyed cookies. If a web application utilizes a user cookie to remember certain user choices and this cookie is unkeyed, it can be used to poison the cache and force these choices upon other users. For instance, assume the cookie consent is used to remember if the user consented to something being displayed on the page. If this cookie is unkeyed, an attacker could send the following request:

Code: http

GET /index.php HTTP/1.1  
Host: webcache.htb  
Cookie: consent=1;

If the response is cached, all users that visit the website are served the content as if they already consented. Similar attacks are possible if the web application uses unkeyed cookies to determine the layout of the application, for instance in a color=blue cookie, or the language of the application in a language=en cookie. While these types of cache poisoning vulnerabilities do occur, they are often caught by the website maintainers relatively quickly since the cache is poisoned during normal interaction with the website. For instance, if a user sets the layout to blue via the color=blue cookie and the response is cached, all subsequent requests made by other users that have chosen a different color will still get served the blue layout. Therefore, it is quite obvious to notice that something is not working correctly in these cases.

#### Denial-of-Service

Another type of web cache poisoning vulnerability revolves around the host header. We are going to go into more detail about host header attacks in the upcoming section, so we are not discussing it in detail here. However, we can think of a scenario where web cache poisoning can lead to a Denial-of-Service (DoS) attack. Consider a setting where a faulty web cache is used that includes the host header in its cache key but applies normalization before caching by stripping the port. The underlying web application then uses the host header to construct an absolute URL for a redirect. A request similar to this:

Code: http

GET / HTTP/1.1  
Host: webcache.htb:80

would result in a response like this:

Code: http

HTTP/1.1 302 Found  
Location: <http://webcache.htb:80/index.php>

While the port is present in the response, it is not considered part of the cache key due to the flawed behavior of the web cache. This means we could achieve a DoS by sending a request like this:

Code: http

GET / HTTP/1.1  
Host: webcache.htb:1337

If the response is cached, all users that try to access the site are redirected to port 1337. Since the web application runs on port 80 however, the users are unable to access the site.

#### Remarks

One of the hardest parts about web cache poisoning is to ensure that a response is cached. When many users use a website, it is unlikely that the cache is empty when we send our malicious payload, meaning we are served an already cached response and our request never hits the web server.

We can try to bypass the web cache by setting the Cache-Control: no-cache header in our request. Most caches will respect this header in the default configuration and will check with the web server before serving us the response. This means we can force our request to hit the web server even if there is a cached entry with the same cache key. If this does not work, we could also try the deprecated Pragma: no-cache header.

However, these headers cannot be used to force the web cache to refresh the stored copy. To force our poisoned response to be cached, we need to wait until the current cache expires and then time our request correctly for it to be cached. This involves a lot of guesswork. However, in some cases, the server informs us about how long a cached resource is considered fresh. We can look for the Cache-Control header in the response to check how many seconds the response remains fresh.

#### Impact

The impact of web cache poisoning is hard to state in general. It depends highly on the vulnerability that can be distributed, how reliably the attacker can poison the cache, how long the payload is cached for, and how many potential victims access the page in that time frame.

The impact can also depend on the cache configuration. If certain HTTP headers such as the User-Agent are included in the cache key, an attacker needs to poison the cache for each target group separately, since the web cache will serve different cached responses for different User-Agents.

## Cache Busters

In real-world scenarios, we have to ensure that our poisoned response is not served to any real users of the web application. We can achieve this by adding a cache buster to all of our requests. A cache buster is a unique parameter value that only we use to guarantee a unique cache key. Since we have a unique cache key, only we get served the poisoned response and no real users are affected.

As an example, let's revisit the web application from the previous section. We know that the admin user is German and thus visits the website using the language=de parameter. To ensure that he does not get served a poisoned response until we completed our payload, we should use a cache buster in the language parameter to guarantee that we don't affect the admin user. For instance, when we determined that the ref parameter is unkeyed, we built a proof of concept for the XSS. We did this using the following request:

Code: http

GET /index.php?language=unusedvalue&ref="><script>alert(1)</script> HTTP/1.1  
Host: webcache.htb

Since the value we sent in the language parameter is a unique value that no real user would ever set, it is our cache buster.

Keep in mind that we have to use a different cache buster in a follow-up request, as the cache key with the language=unusedvalue parameter already exists due to our previous request. So, we would have to adjust the value of the language parameter slightly to get a new unique and unused cache key.