Caching the Un-cacheables - Abusing URL Parser Confusions (Web Cache Poisoning Technique)

Caching the Un-cacheables - Abusing URL Parser Confusions (Web Cache Poisoning Technique) - Author not stated, Harel Security Research.

- Published: 2022-09-02
- Original: <https://nokline.github.io/bugbounty/2022/09/02/Glassdoor-Cache-Poisoning.html>
- Preserved from: https://nokline.github.io/bugbounty/2022/09/02/Glassdoor-Cache-Poisoning.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Here is how I was able to poison the cache of thousands of pages in Glassdoor with reflected & stored XSS

Introduction

Imagine you just picked up a new attractive bug bounty program your friend recommended, and you get excited to try it out because of all the good stories you've heard, but after a few days of intensive recon and research, you are left empty handed and end the day with no findings. The next day you start to doubt yourself, so you start looking for P4s and P5s, when all of the sudden, you find a header XSS. Seems pretty lame, until you remembered that your target has a caching server! You try to find a way to cache the XSS to store it, but keep getting stuck on MISS. Looks like they have web cache poisoning protection on this endpoint, making it un-cacheable, so you're out of luck. Just when you thought you had a P2 or P1, reality HITS you like James Kettle's web cache poisoning payloads. This seems like a dead end, so what do you do now? Have you maybe tried URL parsing confusions?

TL;DR

- For any page under the path https://www.glassdoor.com/Job/?xss all URL parameters are reflected within a Javascript script tag. Lack of sanitization means we can inject </script into the page with https://www.glassdoor.com/Job/?xss=</script, however due to the WAF we cannot simply escape the script tags and execute our own
- The optimizelyEndUserId cookie value is reflected in the page, right after the URL parameters. By combining this with the issue from step 1. we can bypass the WAF by splitting the payload into two parts to execute arbitrary javascript. However this is a self XSS, because we cannot force our victim to send custom cookies.
- We can get past this via cache poisoning. Sadly the pages under https://www.glassdoor.com/Job were not being cached, but all the pages under https://www.glassdoor.com/Award/ were.
- After some testing I found that path traversal characters `/../`, also known as dot segments, were being normalized by the caching frontend server but not being normalized by the backend web application (Dissagreement of RFC 3986 5.2.4). This means for the path https://www.glassdoor.com/Job/../Award/blah?xss=</script it would be seen as https://www.glassdoor.com/Award/blah?xss=</script by the caching server and cached, but the contents for https://www.glassdoor.com/Job/../Award/blah would be returned by the webserver due to lack of normalization
- As a result, by sending the request with our payload to https://www.glassdoor.com/Job/../Award/blah?xss=</script (and rest of the payload in the cookie) we can succeed in obtaining our XSS. The webserver would interpret it as a page under https://www.glassdoor.com/Job/ and the return the contents with our injected XSS payload, while the caching server would see interpret it as https://www.glassdoor.com/Award/blah?xss=</script causing the response to be cached
- By then visiting https://www.glassdoor.com/Award/blah?xss=</script our XSS will fire
- Achieving a stored XSS was also possible using https://glassdoor.com/mz-survey/interview/collectQuestions_input.htm, which behaved very similarly to `/Job`, but the XSS was all in the headers and cookies, so sending a parameter in the URL was not necessary.
- Sending https://www.glassdoor.com/mz-survey/interview/collectQuestions_input.htm/../../../Award/blah with the XSS in the headers and cookies will result in a stored xss under https://www.glassdoor.com/Award/blah

Stored XSS PoC

My XSS methodology

- When testing for XSS, it's important to consider all types of exploitations, and take note of everything that looks interesting.
- Even if something might not be exploitable now, try to see the potential it will have in a chained exploit.
- Many times, exploit chains are made up of unexploitable links that by themselves are useless, but when chained together can be fatal.

- In Glassdoor, I found such endpoint in `/Job/new-york-ny-compliance-officer-jobs-SRCH_IL.0,11_IC1132348_KO12,42069.htm`
- I found that the parameter name (and value too) were reflected in the response unsanitized
- I was very suprised to see this, as this should have been caught very early on. The Glassdoor program has almost 800 submissions, so I didn't make the mistake of thinking I was the only one who noticed it
- The parameter was reflected in a string in a script tag, so to achieve an XSS, I had 2 options
- Escape the string and inject javascript
- Close the script tag and inject a generic XSS payload
- For the first option, the strings seemed to have been escaped with a backslash, and unfortunately bypassing this is hard.
- My second option, however, had much more potential as none of the user input was sanitized, so injecting a closing script tag should do the trick
- However, the moment I put `?</script>` (URL decoded here for readability) my request got immediately swatted down and blocked by the WAF. This was very much expected, however I eat WAFs for breakfast >:)
- Before trying to play against the WAF, we have to understand the rules of the game.
- A common mistake I see people make when trying to bypass WAFs, or just filters in general for the matter, is they copy and paste generic WAF bypass payloads without actually understanding *why* the WAF is blocking their requests. Spraying and praying WAFs is usually a waste of time from my experience, so it's best to test them manually, and most importantly understand them
- So my first step when bypassing WAFs is to start out with a blocked payload and remove a character by character until the WAF lets me pass
- Luckily, it didn't take us long to achieve an agreement with the WAF. All I had to do was remove the greater than `>` sign and I got a 200.
- So now the question is what else doesn't it like? It seemed like any character after `</script` will get the attention of the WAF, like `</scriptaaa` for example
- This would have been a big issue if the WAF truely blocked `</script*`, but luckily the WAF did allow whitespace characters such as `%20` (space), which means that eventually, the script tag will close by the next upcoming greater than `>` sign
- So now, the next step turns to finding a new unsanitized injection point that will allow us to close the script and inject an HTML XSS payload
- I tried to see if I can break down the payload into pieces with other parameters, however it was blocked too. Seemed like the WAF rule applied to the entire URL, not individual parameters. Luckly, I have bypassed these types of WAFs before
- My first goto technique was an alphanumeric based HTTP parameter pollution, which I've already used in the past to bypass a similar WAF in this very program.
- An alphanumeric parameter pollution abuses the alphanumeric ordering of the reflected queries, so it is possible to bypass a WAF like this by breaking down your payload backwards into different parameters

- Unfortunately, it didn't seem like the case here, but I will release a writeup on how I was able to use this technique to achieve reflected XSS
- At this point I was losing a bit of hope in this endpoint, so I decided to look for a chain link vulnerability instead of a stand-alone vulnerability. This is when I started to take a look at the cookies
- This is when I noticed that next to the injection point, there was actually a value that came from the `optimizelyEndUserId` cookie in my request.
- All I needed to do was to close the script tag and inject the HTML. Injecting `><svg>` into the cookie seemed to do the trick.
- Now I needed to actually execute javascript. We already got over the hard part, so now when we were able to smuggle in an svg tag past the WAF, so the rest should be easy
- A pretty generic WAF bypass payload seemed to do the trick: `><svg/onload=a=self['aler'%2B't']%3Ba(document.domain)>`
- And now we got an XSS that looks like this:

```
GET /Job/new-york-ny-compliance-officer-jobs-SRCH_IL.0,11_IC1132348_KO12,42069.htm?attack=VULN%3C/script%20
Host: www.glassdoor.com
Cookie: optimizelyEndUserId=BRUH><svg/onload=a=self['aler'%2B't']%3Ba(document.domain)>
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Content-Length: 0
```

- However it is a self XSS that only exists if we have control over the cookies
- This could escalated to reflected XSS with cache poisoning however, so thats what I started to look for next (I know I said stored XSS in the description, I promise we will get there soon!)

The caching methodology for finding relaxed rules

- When doing my initial recon, I always like testing the cache to see how it behaves.
- If I see a path that gets cached, I always try to test it's limit. Many websites have unique rules to how they cache specific paths and files, so manually testing for these rules is a great way to get familiar with the cache server
- When I first go about manually testing for these caching rules, I usually try to mess with the extensions first. I will remove, add, or change the extension and always carefully observe the caching headers and content of the response
- After I mess around with the extension, I will test the path itself
- For example, in Glassdoor I noticed that `https://www.glassdoor.com/Award/new-york-ny-compliance-officer-jobs-SRCH_IL.0,11_IC1132348_KO12,42069.htm` was getting cached
- I intercepted the request and sent it to the Burp repeater for further inspection
- When I changed the extension, I noticed that while I get a 404 page, I still got MISS/HIT cache headers.

- This immediately got me thinking that there is some sort of pattern for cache or no cache, instead of hardcoded files that get cached
- Then I moved onto the path. I tried `https://www.glassdoor.com/Award/somerandomfile`, and noticed it gave me the same 404 page with the same cache headers.
- I was pretty confident by then I figured out what the rule was, but tested just in case `https://www.glassdoor.com/randompath/somerandomfile`, which gave me a 404 but didn't cache
- So now it was same to assume that the rule was `/Award/*`, meaning everything under the `/Award` path was getting cached
- For a while, I desperately tried to find some sort of header XSS to get Web Cache Poisoning, but unfortunately I ended up empty handed. However, this finding was still pretty great for me. While by itself it is not a vulnerability, it was a very relaxed rule and had a lot of potential to be chained with a vulnerability

Chaining the exploit

- Web Cache Poisoning can be used for many things. The first ones that come to mind are 1) Stored XSS 2) Escalation of unexploitable XSS to Reflected XSS 3) DoS
- At the time of my cache rule finding, I was already aware of the unexploitable XSS in `/Job/new-york-ny-compliance-officer-jobs-SRCH_IL.0,11_IC1132348_KO12,42069.htm?VULN%3C/script%20`, so trying to chain the two bugs into a reflected XSS vulnerability felt like the natrual thing to do
- I went back to the Job path to do a bit more research. I wanted to see if there were any other endpoints that were vulnerable to the self XSS, and there were.
- I found that every page under the Job path was vulnerable to self XSS, which was great! I took it a step further, and noticed that even pages that were *supposed* to be 404s, actually returned 200 and were too vulnerable to the self XSS.
- So to recap the important information:
- The CDN has a rule that will cache `/Award/*`
- There is a self XSS vulnerability on `/Job/*`
- The attack surface of these two bugs were not "static", but relied on a very relaxed wildcard pattern, which got me thinking: "Will these patterns *really* accept anything? Will the server prioritize the pattern over special URL syntax, such as a dot segment `/../`, or will it normalize the URL and *then* match the pattern?"
- Or in other words: "Will both the backend server and frontend server's URL parsers normalize the dot segments?"
- To test this, I tried these two payload with the assumption that the URL parser normalized the dot segments:
- `/Award/../this_should_not_cache`
- `/Job/../this_should_give_a_404`
- And to my suprise, they yielded conflicting results
- The Award payload was NOT cached, meaning the frontend server's URL parser does normalize the dot segment before matching with the cache rule

- The Job path, however, returned a 200, which means that the web server did NOT normalize the dot segment.
- So to conclude this short test, we can say that the frontend server and the backend server have a disagreement over how a dot segment should be parsed
- Knowing this, we can construct the following payload:

```
GET /Job/../Award/RANDOMPATHTATDOESNOTEXIST?cachebuster=046&attack=VULN%3C/script%20 HTTP/2
Host: www.glassdoor.com
Cookie: optimizelyEndUserId=BRUH><svg/onload=a=self['aler'%2B't']%3Ba(document.domain)>
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Content-Length: 0
```

- Since the webserver will NOT normalize the dot segment, we will get the response with the XSS
- But, because the frontend WILL normalize the dot segment, it will be cached (and stored) under `/Award/RANDOMPATHTATDOESNOTEXIST?cachebuster=046&attack=VULN%3C/script%20`
- So now, when the victim will visit `https://glassdoor.com/Award/RANDOMPATHTATDOESNOTEXIST?cachebuster=046&attack=VULN%3C/script%20`, they will get the stored response with the XSS from the CDN

Stored XSS

- So once I was able to get a working PoC of my reflected XSS, I immediately reported.
- However, I was still not satisfied enough, because I knew that a stored XSS should have been possible under the right conditions
- so I kept on looking for an XSS that was truely all header based, and behaved similarly to `/Job/*`, where an XSS was possible under every page under it.
- Thats when I remembered my first report to glassdoor, a reflected XSS in http://glassdoor.com/mz-survey/start_input.htm via an alphanumeric ordered parameter pollution (at the time it was still in triage, so it wasn't fixed).
- I thought that maybe I will be able to find a header XSS there too, so I kept on looking
- Luckily for me, my reflected XSS from the report was also vulnerable to a full header XSS! But it did not behave like `/Job/*` where every page under it was vulnerable, so it was pretty much useless
- I did remember that it was not only one endpoint which was vulnerable, there were quite a few others
- Luckily, I was eventually able to find an endpoint that was both vulnerable to header XSS AND behaved like `/Job/*` after testing each of the vulnerable endpoints I previously reported and got to this one: https://glassdoor.com/mz-survey/interview/collectQuestions_input.htm/
- The payload looked something like this

```
GET /mz-survey/interview/collectQuestions_input.htm/../../../Award/RANDOMPATHTATDOESNOTEXIST123?cachebuste
Host: www.glassdoor.com
X-Forwarded-For: VULN
X-Forwarded-For: VULN><svg/onload=self[`alert`](document.domain)>
Cookie: gdId=VULN%22</script%20
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
```

- The reason for splitting the XSS payload into 2 headers and a cookie is to bypass the WAF, as I was not able to put the entire payload into one cookie or header
- The X-Forwarded-For header is reflected after the cookie, so my opprotunity to continue my payload lied there.
- Unfortunately, the WAF was even stricter for the X-Forwarded-For header, as I was not able to use ANY special characters whatsoever

-

Interestingly enough, there was another cool header confusion where the WAF only blocked the first X-Forwarded-For header, but the webserver interpreted both and reflected both. This allowed me to easily bypass the WAF by giving a valid value for the first X-Forwarded-For header but the rest of my XSS payload in the second X-Forwarded-For header. This can be seen in the above payload

- Due to the tricky nature of the bug, the triage process was a little more complicated than usual. Big thanks to @bxmbn (AKA bombon on h1)) for giving me some help in the triage process

---