Exploiting web cache poisoning vulnerabilities

BY AYOUB AND RACHID ALLAM · JUNE 24, 2026

Web (or HTTP) caching is a highly adopted practice to effectively optimize web page loading times for clients. However, as with most technologies, when incorrectly implemented, it may open up a new exploitable attack surface for us to look into.

In this article, we'll cover what web cache poisoning vulnerabilities are, how they arise, a few effective ways to enumerate such vulnerabilities, and eventually move on to the practical exploitation part.

Let's dive in!

Special thanks to Zhero

This article was co-authored by Zhero, a researcher who has published several research papers on Web Cache vulnerabilities. If you wish to dive deeper into more web cache poisoning resources, we recommend you have a look at his personal security blog!

https://zhero-web-sec.github.io/

What is web or HTTP caching

Before we can dive into web cache vulnerabilities, we must first have a firm understanding of what web caching is and how it works. HTTP caching is the practice of storing a copy of an HTTP response for a given request. Whenever another client makes the same request, the cached response is returned instead of hitting the origin server again.

Developers rely on caching to reduce server load and improve performance on the origin server. This not only benefits the client with faster load times but also helps a domain rank higher in search results, as page speed is a key factor in SEO.

© Intigriti

How web caching works

Different forms of caching

Caching takes several forms within the network layer. Let's have a look at the 3 relevant types.

DNS cache

The Domain Name System is the service that translates a domain name, like `example.com`, into the server's public IP. Developers are required to contact their DNS registrar any time they want to introduce a new change. However, since most DNS changes are long-term, DNS caching was introduced to further improve overall performance by preventing the DNS resolver from making repeated lookups for every request.

It operates at multiple levels, including the operating system, the ISP's resolver, and dedicated DNS servers. It is crucial to note that DNS caching operates at the network layer and is entirely separate from HTTP caching, so web cache poisoning does not target DNS caches.

Server-side cache

Server-side caches sit between the client and the origin server, intercepting HTTP requests and serving stored responses where possible. This category includes reverse proxies and CDNs (such as Varnish, Nginx, Cloudflare, and Akamai), which cache full HTTP responses, as well as application-level caches (such as Redis and Memcached), which store processed data objects to reduce repeated computation or database queries.

For web cache poisoning, **the reverse proxy and CDN layer are what matter most**. This is the layer targeted by web cache poisoning attacks, as it sits in the request path of all users hitting the same endpoint.

Browser cache

Lastly, browser caching stores assets locally in the client's web browser, such as scripts, stylesheets, images, and HTML responses. It is controlled through response headers such as `Cache-Control` and `Expires`, which instruct the browser on whether and for how long to reuse a cached resource. Unlike server-side caches, browser caches are per-user and local by design.

© Intigriti

Internal cache

zhero

There is a fourth, often overlooked category: internal cache. This refers to caching mechanisms built directly into frameworks and web servers. Unlike CDNs, these caches follow their own rules and internal logic, which can differ significantly from edge caching behavior. They can be particularly interesting from a cache poisoning perspective, as shown by vulnerabilities found in Next.js related to its internal caching mechanism.

Diving deeper into server-side caching

When a request arrives at a cache endpoint, the cache must determine whether it has a cached response it can return. To do this, it constructs a **cache key**, a unique identifier derived from specific components of the incoming HTTP request.

By default, cache keys are typically built from the request's URL path, query string, and `Host` header. These are known as **keyed inputs**: if any of them differ between two requests, the cache treats them as distinct entries and won't serve a cached response across them.

However, not every component of an HTTP request is included in the cache key. Headers such as `X-Forwarded-For`, `Accept-Language`, or other custom headers may be forwarded to and processed by the origin server, yet completely ignored by the cache when constructing the key. These are referred to as **unkeyed inputs**.

It is crucial to understand this distinction, as it is the root cause of web cache poisoning. If an unkeyed input influences the server's response, an attacker can manipulate it to inject malicious content, and since the cache key doesn't account for it, the poisoned response gets served to every user whose request matches the same key.

Cache HIT and cache MISS

When a request arrives at the cache, one of two things happens:

- **Cache HIT**: The cache has a stored response. It returns it directly, without ever reaching the origin server.
- **Cache MISS**: The cache has no matching stored response. It forwards the request to the origin server, if the configuration allows, it'll cache the response it receives before finally returning it to the client.

Most caching layers expose this through response headers such as `X-Cache: HIT` or `X-Cache: MISS`. Targets behind Cloudflare typically use `CF-Cache-Status: HIT` instead. This becomes important during testing. By observing **HIT** and **MISS** responses, you can confirm whether your crafted request has been cached and whether subsequent requests are being served the poisoned response.

© Intigriti

Missing response headers

Guest contributor zhero

Although cache-related response headers are present in most cases, it is important to note that their absence does not necessarily imply the absence of a caching system. A quick look at the difference in response time between an entry that is supposedly cached and another that is supposedly served from origin can help confirm this.

To do so, select a component of the request that is typically part of the cache-key, such as the query string, and append a random value to it. Then send the request two or three times to ensure that the response is, if applicable, properly cached for that specific key. Once this is done, send several more requests using the same key while noting the response time for each request. After that, simply change the key value (*the query string in our example*) and check whether the difference in response time is significant. Repeat this process several times to confirm the trend, and if it is consistent, it is highly likely that a caching layer is present. Also consider repeating the test by selecting other request components as part of the cache key if the first tested component does not yield conclusive results.

Geography-based caching strategy

It sometimes happens that certain CDNs are configured to make only responses from specific geographic regions eligible for caching. You can try sending the same request through IP addresses from different countries or continents to look for differences in caching behavior.

Status code

It is known that status codes can also influence caching behavior, for example by excluding responses with a `200` status code while still caching `500` responses. In such cases, the base page would not be cached, but if there is a way to alter the response and trigger a `500` error, caching may become possible, potentially leading to a CPDoS (*Cache Poisoned Denial of Service*) attack, which is discussed in more detail later in this article.

The discovery of a cache is obviously not a vulnerability in itself, but it opens the door to many potential attack vectors that are likely overlooked by many. This oversight is worth its weight in gold, especially in the highly competitive world of bug bounty.

X-Cache: HIT, HIT, MISS

In some cases, you'll notice that certain applications make use of multiple caching layers, for example, a CDN in front of a reverse proxy in front of the origin. As each layer appends its own status to the `X-Cache` response header, you may come across a target with multiple cache statuses in the X-Cache response header, such as `X-Cache: HIT, HIT, MISS`. This practically means the first two layers served from cache, while the deepest layer had to fetch from the origin.

Caching response headers

Several HTTP response headers depict how caching layers handle stored responses. Here's an overview of the most relevant ones:

© Intigriti

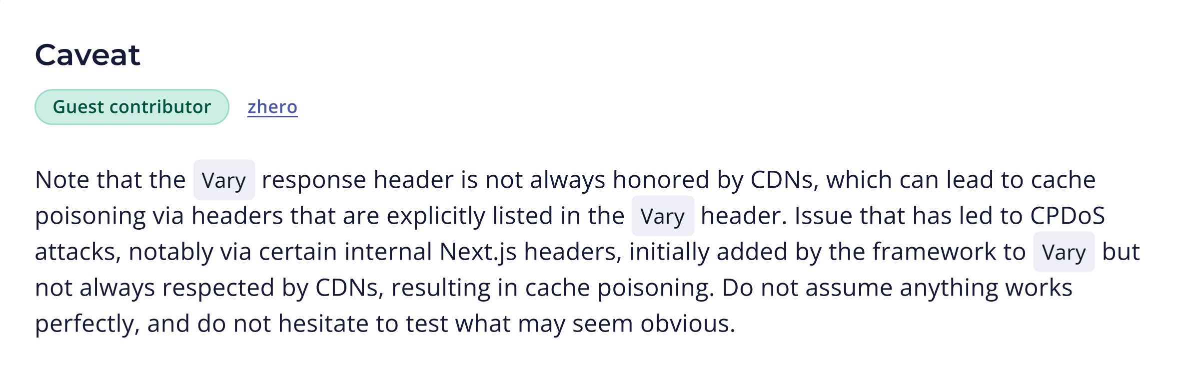
Web Cache Poisoning

HTTP Cache Headers

Header	Purpose	Example
Cache-Control	Primary directive controlling caching behavior, who can cache the response and for how long	public, max-age=86400
Vary	Instructs the cache to include specific request headers as additional cache key components	Accept-Encoding
Age	Indicates how many seconds the response has been stored in cache (in seconds)	120
Expires	Specifies an absolute expiry date after which the response is considered stale. Largely superseded by Cache-Control	Fri, 01 Jan 2027 00:00:00 GMT
X-Cache	Indicates whether the response was served from cache or the origin server	HIT, MISS
CF-Cache-Status	Cloudflare-specific equivalent of X-Cache	HIT, MISS, EXPIRED

HTTP cache headers

The `Vary` header is worth paying close attention to. For example, `Vary: Accept-Encoding` instructs the cache to store separate entries for clients sending `gzip` compared to those that do not. From a security perspective, `Vary` can sometimes limit the exploitability of a cache poisoning vulnerability, but it can also hint at which request headers the origin server is actively processing.



Caveat

Guest contributor zhero

Note that the `Vary` response header is not always honored by CDNs, which can lead to cache poisoning via headers that are explicitly listed in the `Vary` header. Issue that has led to CPDoS attacks, notably via certain internal Next.js headers, initially added by the framework to `Vary` but not always respected by CDNs, resulting in cache poisoning. Do not assume anything works perfectly, and do not hesitate to test what may seem obvious.

What are web cache poisoning vulnerabilities

Web cache poisoning arises when an attacker can control and pollute the cached version of a resource. The polluted cache will then be served to every user whose request matches the same cache key.

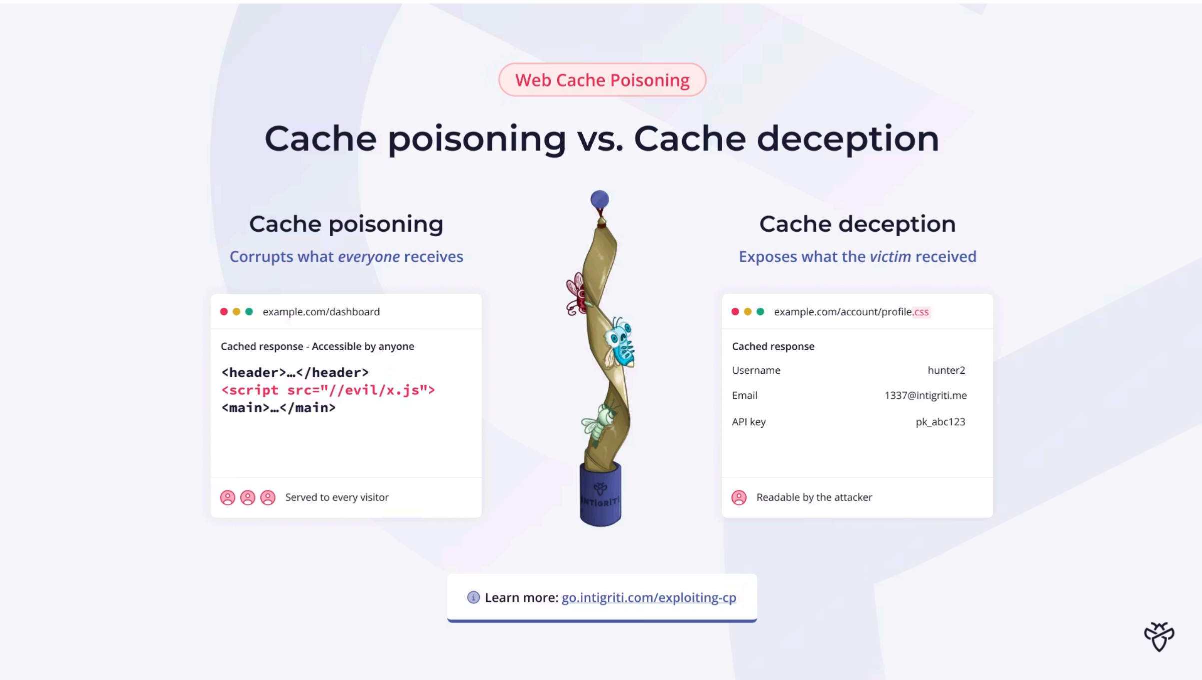
Web cache poisoning vs. web cache deception

These two vulnerability classes are commonly confused with each other. However, they are fundamentally different in both their mechanism and their goal.

In **web cache poisoning**, the attacker manipulates the cache into storing a malicious response, which is then served to other users. The goal is to deliver harmful content to victims.

With **web cache deception**, the attacker tricks the cache into storing a response containing sensitive, user-specific data, and then retrieves it themselves. The objective is to steal private information that was unintentionally cached.

© Intigriti



Understanding HTTP cache poisoning vs cache deception attacks

Throughout this article, we'll focus only on **web cache poisoning**.

How web cache poisoning vulnerabilities arise

Web cache poisoning vulnerabilities arise from a fundamental gap in the developer's understanding of how caching layers interact with the application. Developers often add support for custom request headers, such as `X-Forwarded-Host` or `X-Original-URL`, to handle scenarios like load balancing, URL generation, or infrastructure routing, without accounting for how the cache treats them.

Since these headers are typically unkeyed, the cache ignores them entirely when constructing the cache key. Yet the origin server may process and reflect their values in the response, for instance, dynamically generating a script source URL or an Open Graph tag. From the cache's perspective, two requests are identical if their keyed components match, regardless of what malicious value was passed through an unkeyed header. The poisoned response gets stored, and every subsequent user with a matching cache key receives it.

Unexploitable web cache poisoning vulnerabilities

It is crucial to understand that not every cache-related observation constitutes a valid web cache poisoning vulnerability. There are two common pitfalls to be aware of.

An unkeyed input alone is not a vulnerability. For a finding to be valid, the origin server must actively process the unkeyed input and reflect or act on its value in the response in a meaningful way. Simply identifying that a header is ignored by the cache is not sufficient, you will always need to demonstrate an impact on the polluted, cached response.

Similarly, **a poisoned response with no impact on other users is not a vulnerability.** If the cache key is scoped to something user-specific, such as a session token or user ID, the poisoned response will never be served to another user. Similarly, if the manipulated value produces no exploitable consequence, the finding lacks the impact required to be considered a valid vulnerability.

Additionally, it is important to note that if the **cache is short-lived** (think of invalidation after 1 second), the **impact may be questionable**, as you'd need to poison the cache continuously. It's also worth noting

© Intigriti

that you need to ensure the cache poisoning vulnerability you're about to report can be reliably demonstrated and **reproduced** across **different devices** in **different networks**.

Quick reminder!

Before reporting, you must be able to demonstrate two things: that a poisoned response will realistically be served to other users, and that it carries a meaningful security impact.

Identifying web cache poisoning vulnerabilities

Identifying web cache poisoning vulnerabilities involves finding a way to poison the cache with a malicious payload while ensuring it is served to other users. Let's look into each step individually.

1. Identifying cache keys

Your first step will always be to identify cache keys. As we've mentioned before, keyed inputs are taken into account when the cache server tries to match your response with a cached version. Unkeyed inputs are components (such as request parameters or headers) that are not taken into account. Therefore, we'll need to identify unkeyed inputs that trigger malicious behavior. An example could be the `X-Forwarded-Host` header, which returns a 500 Internal Server Error response when sent with a malicious value.

In practice, you'll likely need to resort to bruteforcing to enumerate potential unkeyed inputs. During initial research, James Kettle, the researcher who originally is behind the concept of Web Cache Vulnerabilities, decided to create Param Miner. A tool that can help us tremendously speed up the process of enumerating keyed inputs for web cache poisoning.

Dive deeper into identifying cache keys

Guest contributor zhero

During the enumeration or brute-force phase, whether performed manually, using the excellent param-miner, or a custom tool, it is important to have first identified a component of the request that is part of the cache-key, so that the cache-buster generated on each request is placed in the correct location.

If, on your target, query strings are not part of the cache key, and the brute forcing tool generates a random URL parameter on every request (having been configured by default to treat query strings as keyed inputs), then the requests will never reach the origin, since the cache will never be bypassed or invalidated. As a result, the enumeration will effectively be performed in vain, leading to potentially false negative results. In the worst case, your malicious request may coincide with the expiration of the previous cached entry, and if that request can alter the response, all users will be affected, since no effective cache buster will be used, which is clearly undesirable (*even forbidden*) in a bug bounty context.

2. Probing for malicious behavior

Once you've identified an unkeyed input, the next step is understanding exactly how the origin server processes it. This is what determines whether the input is exploitable. If, for instance, the value is reflected in the response without proper sanitization (e.g., being used to dynamically generate other data, such as a script URL, a redirect, or a meta tag), you have a potential entry point for poisoning. Similarly, if it fails to handle exceptions and you can reliably induce the server to return an error, it can serve as a potential entry point for cache poisoning that can lead to denial-of-service.

© Intigriti

3. Serving the malicious cached response

Once we have successfully identified an unkeyed input that triggers malicious behavior, the next step is to cache the malicious response and serve it to other users. At this stage, you'll need to identify the keyed inputs to understand when a cached response will be served. Once you've identified these, you can start preparing your request to poison the cache.

> **Use a cache buster!**
>
> During testing, always ensure to use a cache buster. A cache buster is often a unique query parameter, cookie, or request path included as a keyed input, which is appended to your requests (e.g., ?cb=intigriti). This ensures your test traffic hits its own distinct cache entry and doesn't poison responses served to any actual users.

Exploiting web cache poisoning vulnerabilities

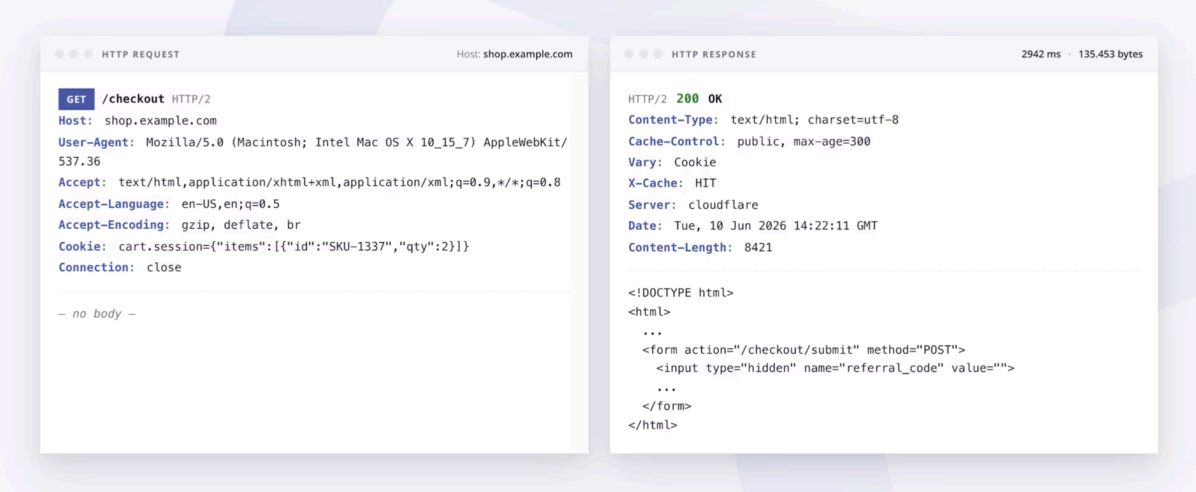
Now that we understand how to identify web cache poisoning vulnerabilities, let's have a look at how we can weaponize them.

Caching malicious responses to exploit logic flaws

Most cache poisoning vulnerabilities are known to escalate to either [cross-site scripting (XSS)] or denial-of-service (DoS) attacks. However, in some cases, the most impactful exploit outcome is exploiting application logic flaws by manipulating what content the cache serves for a given endpoint. As usual, let's have a look at a practical example.

Consider an e-commerce target that allows its customers to enter a referral program. The concept is simple: refer new customers and help drive more sales with your unique code to earn a percentage of the sale. From an attacker's perspective, the ability to require everyone to use our referral code during checkout would be ideal.

Looking further through the app, we can observe that the checkout application route for guests indeed includes a referral code from an unkeyed query parameter into its response. In this instance, it's embedded within a hidden form field:



Caching malicious responses to exploit logic flaws

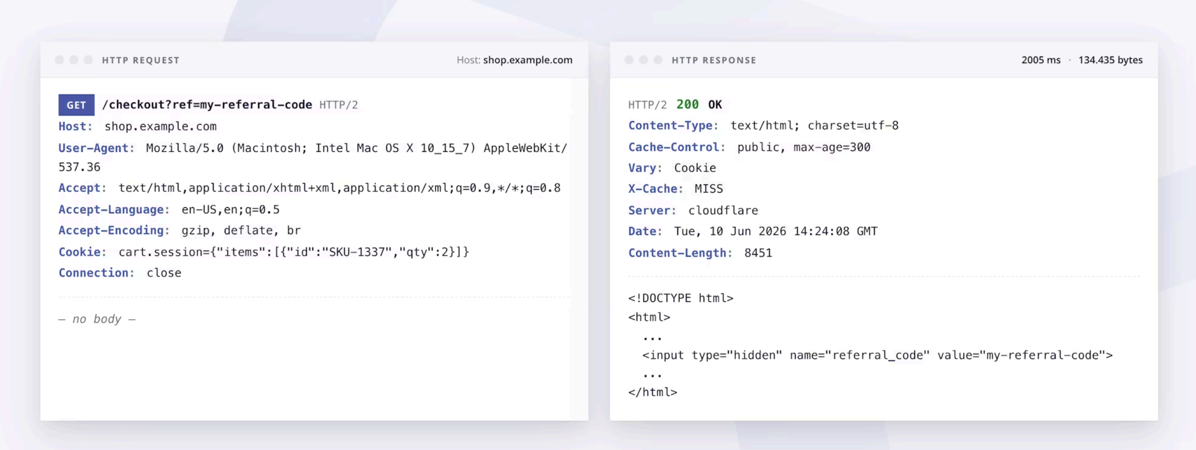
© Intigriti

By poisoning the cache with our own referral code, we can ensure that every user who loads that page uses our code rather than any other referral code. Before we can do so, we first need to verify several attack conditions:

1. First of all, we'll need to ensure that the target indeed caches the checkout page. We figured the application would cache the checkout page, but only when the customer refrains from signing up and continues with a guest checkout. The keyed input here is most likely the full request path and the HTTP cart.session cookie that carries a JSON object of all cart items, and of course.

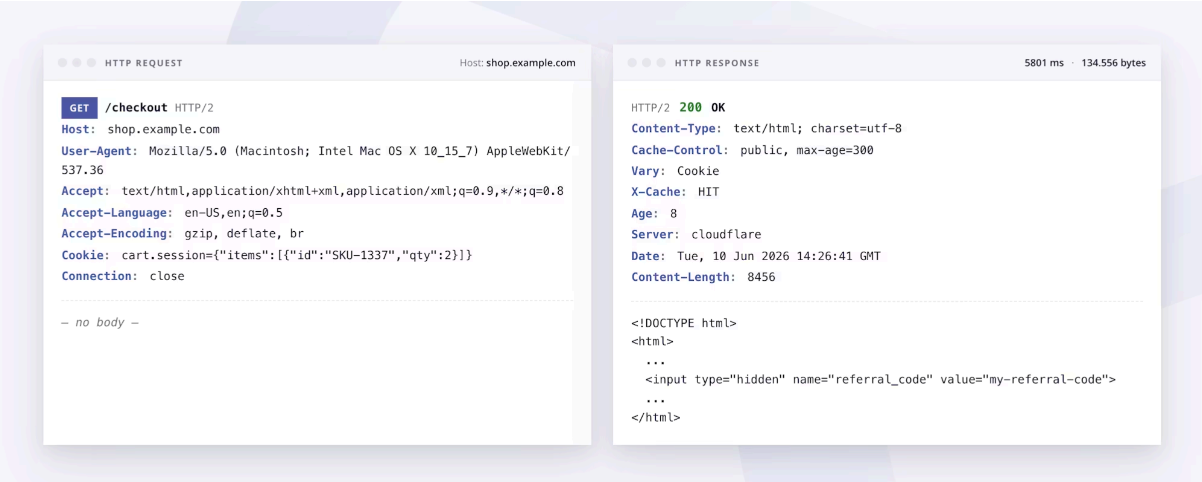
2. The next step is to ensure we identify all unkeyed cache keys to verify whether a cache poisoning is exploitable. In our case, we confirmed that the ref query parameter is the unkeyed cache key. Which is ideal, as in practice, the victim won't need to visit our specially crafted link to trigger cache poisoning. As long as he/she proceeds with the guest checkout, our poisoned response will be served, and the checkout session will hold our referral code.

All that remains now is for us to effectively poison the cache:



Poisoning cache entry with our unique referral code

As shown in the figure above, we've successfully poisoned the response. Any user attempting to check out as a guest with a matching cart object will have the attacker's poisoned referral code auto-applied.



Verifying the poisoned cache response

Aside from caching malicious responses to exploit logic flaws, sometimes, they can also open up an attack vector for injection attacks. Let's have a look at a common scenario.

© Intigriti

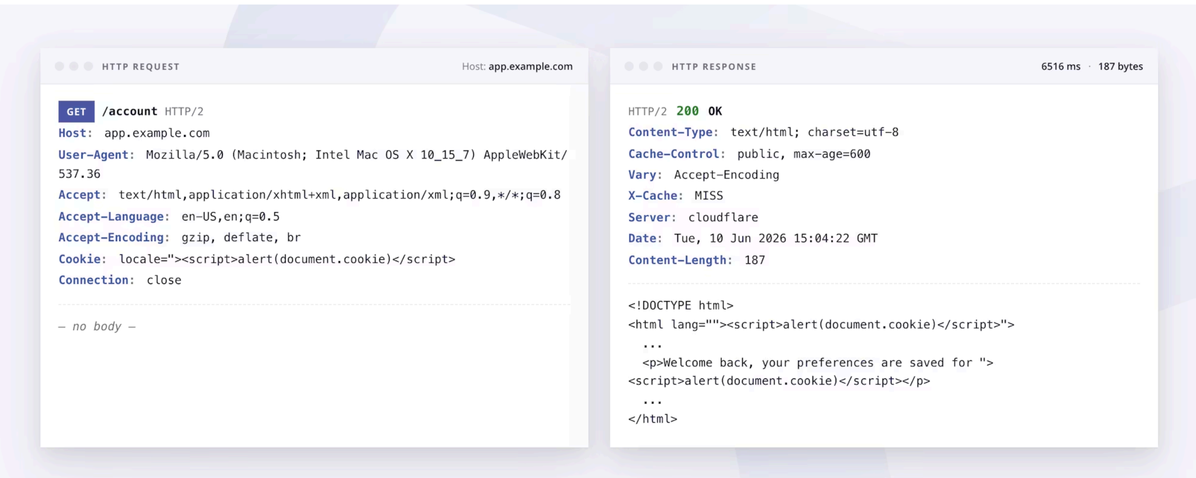
Stored XSS via cache poisoning

With traditional stored XSS, a payload persists because it gets written to a database and later retrieved and rendered for other users. Cache poisoning achieves the same outcome through a somewhat different mechanism. Once an unkeyed input carrying a malicious payload gets cached, every user whose request matches the cache key executes the attacker's script on a completely normal page load. In practice, this allows some forms of self-XSS vulnerabilities to be escalated to exploitable XSS.

Cookie reflection

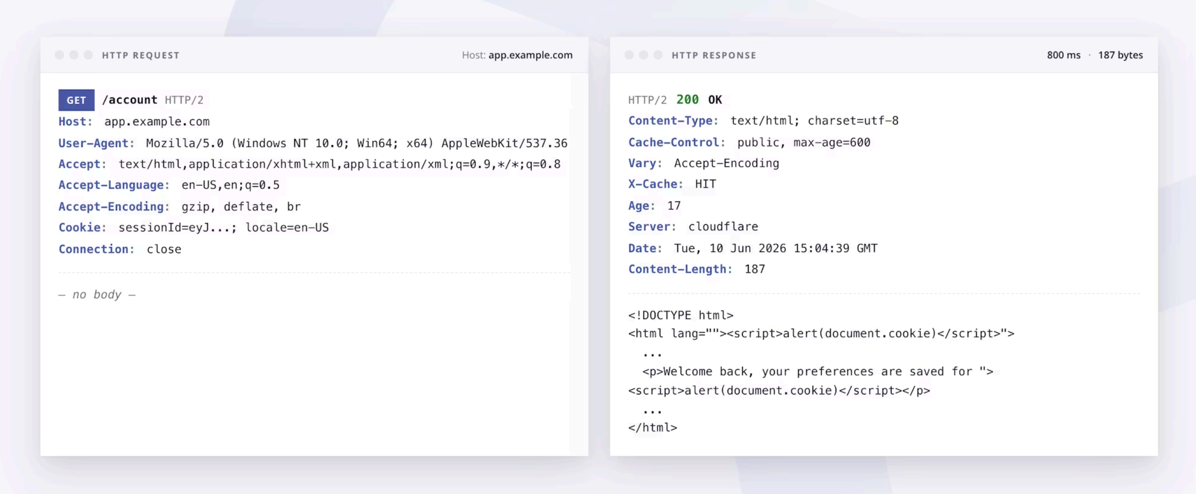
The most common vector is cookie reflection. Applications frequently reflect cookie values directly in the response, such as locale preferences, currency selections, tracking identifiers, A/B feature flags, and theme selections. Depending on the caching configuration, those cookies often don't form part of the cache key.

Consider an application that stores the user's selected locale in a cookie and reflects it into the response body to render the appropriate language:



Poisoning the cache with a malicious locale cookie value

If the value is reflected without sufficient output encoding, an attacker can craft a request with a malicious cookie value:



A victim's clean request served the poisoned XSS payload from the cache

© Intigriti

If the caching layer ignores the `locale` cookie when constructing the cache key, which is common, as caches typically only key on sensitive authentication cookies, the poisoned response gets stored. Every subsequent user requesting the same endpoint receives the attacker's payload as part of a perfectly normal page load.

Minor caveat

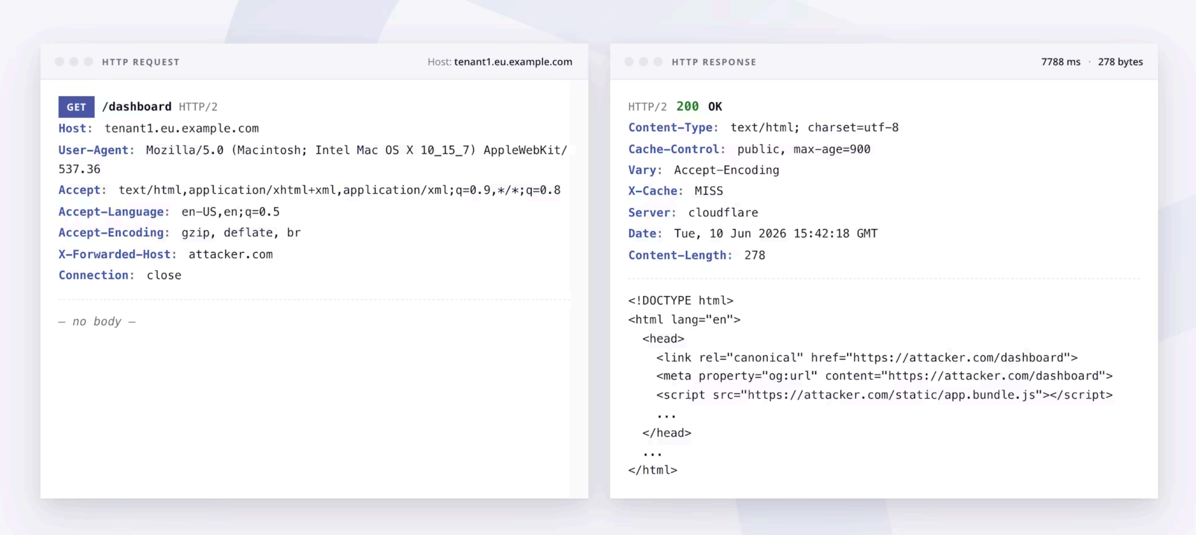
Guest contributor [zhero]

If the base request contains a non-guessable cookie that is part of the cache-key, the attack is not necessarily aborted. It may still be possible to poison the cache for "new users" (*or those visiting the site from a new browser*). Since cookies are set on first visit, poisoning the cache through a request that does not include any will naturally impact all first-time visitors. The impact is of course reduced, but still present.

Host header injection

A less common but worth mentioning vector is host header injection via the `X-Forwarded-Host` header. In setups where a reverse proxy sits in front of a multi-tenant SaaS application, where each tenant is served under a different hostname, the origin server may use the forwarded host value to dynamically generate tenant-specific values, such as canonical tags, Open Graph metadata, absolute URLs in API responses, or self-referencing links for things like password resets and email confirmations.

If an attacker can inject a malicious value via the `X-Forwarded-Host` header and the server reflects it without sufficient output encoding, the poisoned response may be cached and served to other users. It is worth noting that the `Host` header itself is typically included in the cache key by default, making `X-Forwarded-Host` the more practical vector in setups where this kind of dynamic generation is in play.



Poisoning the cache via an unkeyed X-Forwarded-Host header reflected into dynamically generated resource URLs

The two examples above are the most common vectors, but they're far from the only ones. Any reflected request component that falls outside the cache key can possibly lead to exploitable behavior, such as `User-Agent`, `Accept-Language`, custom headers introduced by the application, or even query parameters that the cache excludes from the key. This is also the reason why a thorough enumeration of unkeyed inputs is non-negotiable.

© Intigriti

More about host header injection

Guest contributor zhero

Cache-key

If `X-Forwarded-Host` is part of the cache-key, which can happen quite frequently, it is worth noting how some environments expose request headers internally. In these, a header name is uppercased and its dashes (-) are replaced with underscores (_), sometimes with an `HTTP_` prefix added, so `X-Forwarded-Host` ends up exposed as `X_FORWARDED_HOST` (or `HTTP_X_FORWARDED_HOST`).

The key consequence is that this mapping is many-to-one: both `X-Forwarded-Host` and `X_Forwarded_Host` collapse into the same internal variable, so the application cannot tell them apart. A cache, on the other hand, keys on the literal header name. Injecting `X_Forwarded_Host` (*underscores*) therefore looks like a different, unkeyed header to the cache, while the backend still reads it as `X-Forwarded-Host`, allowing the cache-key to be bypassed while the value is still honored.

Beyond this collapsing behavior, some components apply no such mapping but simply treat underscores as equivalent to dashes during header parsing, and in rarer cases even accept dots (`X.Forwarded.Host`). Both can be abused the same way to bypass a cache-key that was built around the dashed `X-Forwarded-Host`.

The same logic applies to other request components, leveraging discrepancies between the different layers of a stack in order to force the caching of an altered response despite what was originally intended by the cache-key.

Note that this depends on the front-end forwarding underscore headers in the first place: some proxies and servers drop headers containing underscores by default, which can neutralize the technique unless explicitly enabled.

White-list

Sometimes, even though the `X-Forwarded-Host` header is not included in the cache-key, it is incorrectly assumed that adding it to the request has no impact on the response. This is often due to the presence of a validation check ensuring that the header value is consistent. It is common for implementations to attempt to enforce consistency on this header through a whitelist or domain validation, for example by comparing its value against the expected host or by applying a regular expression intended to ensure that the domain matches the target application.

When this validation relies on a simplified or poorly designed regular expression, it can be bypassed by exploiting the structure of domain names. For example, if the validation only checks for the presence of the expected domain as a suffix, the following value can pass the validation while still allowing the attacker-controlled domain to be involved:

```
X-Forwarded-Host: www.example.com.attacker.com
```

In other cases, if the validation only enforces checks at the second-level domain without properly accounting for subdomains, the following value can be accepted:

```
X-Forwarded-Host: non-existent.example.com
```

Once the check is bypassed, the impact will depend on how the value is used/reflected.

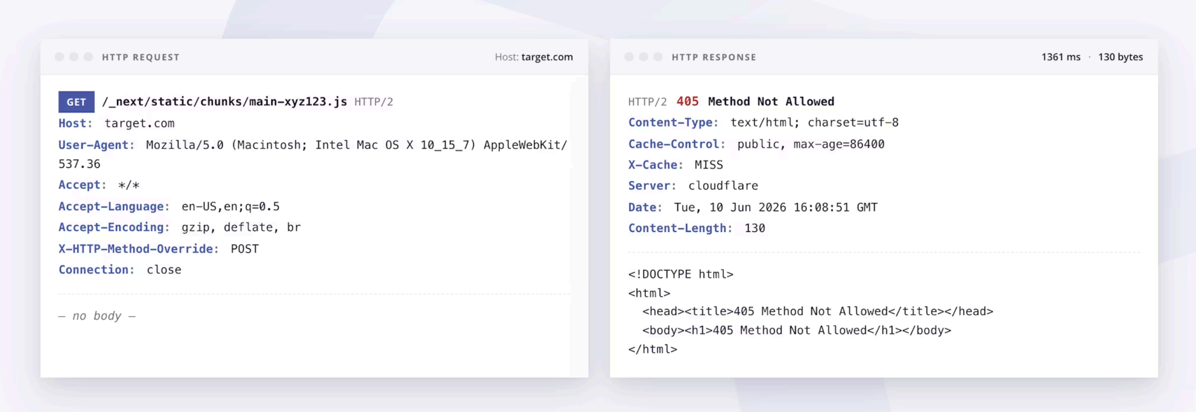
© Intigriti

Cache poisoning denial of service (CPDoS)

Cache-poisoned denial-of-service, or **CPDoS**, reverses the attack's objective. Instead of poisoning the cache with malicious content intended to harm victims, an attacker crafts a request that causes the origin server to return an error response, then caches that error response. Every subsequent user whose request matches the same cache key receives the error instead of the legitimate response, effectively denying access to the endpoint (or even host) until the cache entry expires. As usual, let's have a look at a practical example.

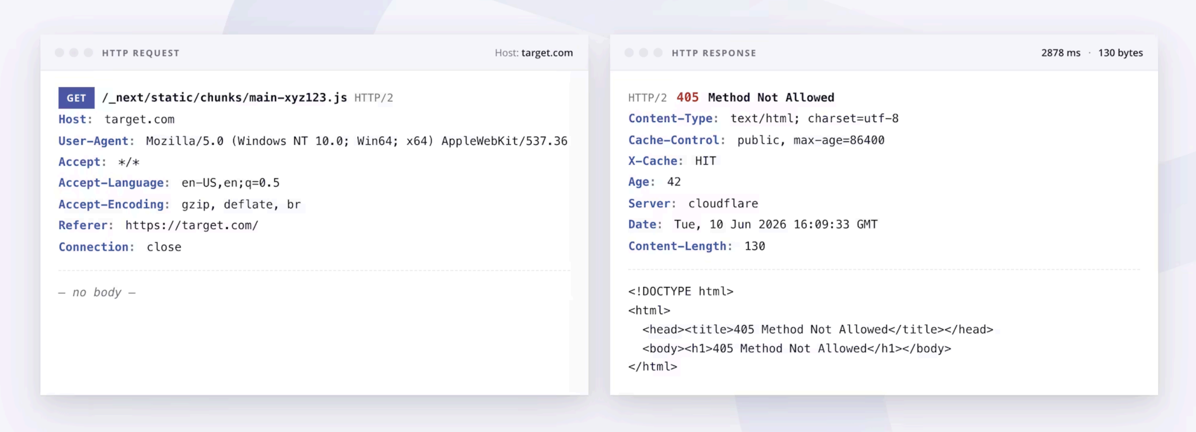
Modern frontend frameworks like React and Next.js compile their applications into chunked JavaScript bundles that are served as static assets, typically from paths like `/_next/static/chunks/main-xyz123.js` or `/static/js/main.xyz123.js`. These JavaScript files will most likely be cached by CDNs, often with long TTLs, as they're treated as static assets that change less frequently.

Take the following NextJS-based target into consideration. The origin server is behind a popular CDN, Cloudflare. Suppose the origin server processes the `X-HTTP-Method-Override` header, but the CDN does not include it in the cache key. An attacker could send the following request:



Triggering a cacheable 405 Method Not Allowed against a Next.js JavaScript chunk via the X-HTTP-Method-Override header

The origin server processes the HTTP request method override, rejects the unsupported method, and returns a **405 Method Not Allowed**.



A clean request from a legitimate user receiving the cached 405 instead of the JavaScript bundle, breaking the application

If the caching layer stores that error response against the JS chunk's cache key, every subsequent client will receive the cached error instead of the correct JavaScript file. This effectively breaks the application and denies access until the cached entry expires and is revalidated.

CPDoS

Guest contributor zhero

A CPDoS attack is fundamentally simple: all it takes is finding a way to break the response and using the cache to preserve that state so that it is served to all users. Knowing, on one hand, the quirks and undesirable behaviors of the targeted technologies, and on the other hand, the status codes cached by default by CDNs, is a combination that can be highly valuable.

Conclusion

Web caching can be extremely useful, but as with any technology, when deployed incorrectly, it can also create a new attack vector. In this article, we've covered how server-side caching works, what qualifies as a valid web cache poisoning vulnerability, how to identify and confirm these issues in practice, and the various ways they can be effectively weaponized.

So, you've just learned something new about exploiting web cache poisoning vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY

AUTHOR

Ayoub

Senior security content developer

CO-AUTHOR

Rachid Allam

External security researcher Rachid Allam (zhero)

REQUEST A DEMO

intigriti.com/demo

VISIT THE WEBSITE

intigriti.com

GET IN TOUCH

hello@intigriti.com