W

Web Cache Poisoning: How It Works and Prevention

Web cache poisoning causes a shared cache to store a harmful response and serve it to other users. Learn the prerequisites, impact, testing, and fixes.

PUBLISHER
Gridinsoft Support

UPDATED
Oct 8, 2026

READ TIME
4 min

CONTENTS

In this guide

01 What is web cache poisoning?
02 How an attack works
03 Cache key, Cache-Control, and Vary
04 How to test safely
05 How to prevent web cache poisoning
06 Incident response
07 Cache poisoning vs. cache deception

Quick answer: Web cache poisoning occurs when an attacker makes an origin server generate a harmful response that a shared cache stores under a key used by other visitors. It usually requires a difference between the inputs the application processes and those the cache includes in its cache key. Prevent it by aligning cache keys with

application behavior, rejecting unnecessary inputs, setting correct cache directives, and testing with unique cache busters in an authorized environment.

01 What is web cache poisoning?

A web cache stores responses so that equivalent requests can be served without contacting the origin every time. The cache key commonly includes the request method and target URI and may include selected headers. If two requests have the same key, the cache may treat them as equivalent.

Poisoning becomes possible when the application uses an input that the cache ignores. An attacker manipulates that unkeyed input, causes the origin to return a malicious or incorrect response, and gets that response cached. Later visitors receive the stored response even though they never supplied the attacker's input.

02 How an attack works

1. **Find a cacheable endpoint:** the attacker identifies a route and response that a CDN, reverse proxy, or application cache will store.
2. **Identify a key mismatch:** a header, cookie, query parameter, path normalization difference, or port affects the origin response but not the cache key.
3. **Create a harmful response:** the input changes a redirect, script URL, markup, security header, or content.
4. **Store it:** the crafted request populates the shared cache under a key normal visitors use.
5. **Deliver at scale:** subsequent cache hits serve the poisoned response until it expires, is replaced, or is purged.

Possible effects include cross-site scripting, malicious redirects, credential theft, defacement, disclosure of another variant, or denial of service. Impact depends on what the origin does with the unkeyed input and how widely the cache entry is shared.

03 Cache key, Cache-Control, and Vary

The cache key defines which requests are considered equivalent. The HTTP `Vary` response header tells compliant caches that selected request headers distinguish representations. `Cache-Control` directives influence whether and how long a response may be stored or reused. These controls must match behavior across the browser, CDN, reverse proxy, framework, and application cache.

Adding every possible value to a cache key can damage hit rates or create denial-of-service opportunities, so the better design is often to reject or normalize inputs the application does not need. Sensitive, personalized, authentication, error, and administrative responses require explicit cache review.

04 How to test safely

Only test systems you are authorized to assess. On a live service, an ordinary probe can poison content for real users. Use a staging environment when possible. If production testing is approved, create a unique cache key for every test—such as a dedicated hostname, path, or known keyed cache-buster—and confirm it cannot collide with visitor traffic. Observe cache status, age, key behavior, and the origin response at every layer.

Do not assume a random query parameter is a safe cache buster: some caches intentionally exclude all or selected query parameters. Validate that the chosen component is actually in the key before testing inputs.

05 How to prevent web cache poisoning

- Inventory every cache layer and document the exact key, normalization, and storage rules for important routes.
- Reject, strip, or canonicalize unnecessary request headers, ambiguous hosts, duplicate parameters, and unsupported methods before they reach the application.

- Do not reflect or trust forwarding headers unless they come from an authenticated, controlled proxy path.
- Ensure every input that legitimately changes a cacheable response is represented in the cache key or appropriate `Vary` behavior.
- Mark personalized, sensitive, and unsafe-to-share responses with suitable cache directives and verify CDN overrides do not defeat them.
- Patch application-side injection and redirect flaws; caching can amplify a vulnerability that otherwise appears limited.
- Test configuration changes end to end and monitor unusual cache misses, variants, redirects, and purge events.

06 Incident response

Purge the affected cache keys at every layer, temporarily bypass caching on vulnerable routes, and remove the input-to-response behavior or key mismatch. Preserve the crafted request, headers, cached response, timestamps, cache status, and relevant logs. Search for other entries created through the same discrepancy, then validate from multiple locations that clean content is served.

07 Cache poisoning vs. cache deception

Cache poisoning makes a harmful attacker-influenced response get served to other users. Web cache deception tricks a cache into storing a victim's private response at a location the attacker can retrieve. Both involve incorrect caching, but the data flow, victim interaction, and fixes differ.

NEED MORE HELP?

We are here when the guide is not enough.

Visit the Gridinsoft Support Center for current product guidance or submit a support request with the details of your case.

Open this guide online at support.gridinsoft.com