# Web Cache Deception

**Web Cache Deception** occurs when an attacker tricks the caching mechanism (CDN, proxy, or application-level cache) into **caching a sensitive, user-specific resource** (like /account, /profile, etc.) by appending **deceptive, cacheable extensions or paths** to the request.

Once cached, the next visitor (often attacker-controlled) can retrieve **private data of a previous victim**.

# Why It Happens:

Caching systems (like CDNs, reverse proxies) often:

* Cache static resources (.css, .jpg, .html)
* Do **not inspect session/auth headers** before caching
* Rely on URL patterns to determine cacheability

If the web app doesn’t validate or route unexpected paths, it may serve **authenticated content** on cacheable paths.

# Scenario:

Victim visits: https://target.com/account

Authenticated request → shows **personal account info**

Attacker tricks victim into visiting: <https://target.com/account.css>

App routes /account.css to the same logic as /account, returns **authenticated data** with Content-Type: text/css

Cache (e.g., Cloudflare, Akamai) stores it and serves it to others.

# Impact

| Area | Risk |
| --- | --- |
| Data Leakage | Attacker retrieves sensitive user info |
| Session Info Disclosure | Full authenticated response cached and exposed |
| Persistent Cache Poisoning | Cached content lives even after logout |
| Exfiltration of PII | Name, email, tokens, session data |

# Example Attack Payloads

## 1. Cacheable Extension

https://target.com/profile → private

https://target.com/profile.css → tricked and cached

## 2. Append Query

/dashboard?callback=evil.com/dashboard.css

## 3. Mixed Headers

Force Content-Type: text/css and get private data in cached response.

# Testing Steps (Manual)

**Identify sensitive endpoints:**  
/account, /settings, /orders, /dashboard, etc.

**Append a static extension:**  
/account.css, /settings.jpg, /orders.php

**Make request while logged in**, observe if content is private (HTML page, JSON with user data)

**Logout and access same URL** — if still served, **response is cached** = vulnerable.

**Try from Incognito or different IP/device**

# Tools for Detection

| Tool | Purpose |
| --- | --- |
| 🧪 Burp Suite (Manual) | Intercept & modify requests, test static extensions |
| 🔧 WCD Scanner (by Dolev Farhi) | Automated WCD scanner (GitHub) |
| 🔍 Cachebuster Extension (Burp) | Detects cacheable behavior |
| 🧬 curl/wget | Script cache checking with headers |

# Mitigation

## 1. Add Anti-Caching Headers

Set headers on **all dynamic pages**, especially authenticated ones:

Cache-Control: no-store, no-cache, must-revalidate

Pragma: no-cache

Expires: 0

## 2. Route Strictness

Validate and **reject unexpected URL extensions** like .css, .jpg, .html for dynamic routes like /account.

/account.css → should return 404 Not Found

## 3. CDN Configuration

* Disable caching for URLs without extensions
* Whitelist cacheable paths only (e.g., /static/, /images/)
* Strip cookies from cache keys
* Enable private session-aware caching (if used)

## 4. No Mixed Routing

Avoid serving **dynamic content** on paths that **look static** (e.g., /home.css, /cart.php)

## 5. Normalize Responses

* If Content-Type is text/css, response should contain **valid CSS only**, not HTML or JSON

# Points

“Web Cache Deception targets **trust boundaries between cache logic and app routing**.”

“It’s dangerous because caches **don’t consider session or auth tokens**, so private data can be **served publicly**.”

“Fixes require a **combination of application-level and CDN-level cache hardening**.”

# Real-World WCD Exploits

| Company | Impact |
| --- | --- |
| Facebook (Bug Bounty) | Cached user profiles via /profile.jpg |
| Twitter (2020) | Cached private tweets using misleading URL extensions |
| Uber | Account data cached via /dashboard.css |
| Several banks | Account info cached via /statements.html |