HTTP
Internet-Draft
Intended status: Standards Track
Expires: 9 July 2026

D. Hardt
Hell
S. Goto
Google

5 January 2026

HTTP Redirect Headers
draft-hardt-httpbis-redirect-headers-00

Abstract

This document defines HTTP headers that enable secure parameter passing and mutual authentication during browser redirects. The Redirect-Query header carries parameters in browser-controlled headers instead of URLs, preventing leakage through browser history, Referer headers, server logs, and analytics systems. The Redirect-Origin header provides browser-verified origin authentication that cannot be spoofed or stripped, enabling reliable mutual authentication between parties. The optional Redirect-Path header allows servers to request path-specific origin verification. Together, these headers address critical security and privacy concerns in authentication and authorization protocols such as OAuth 2.0, OpenID Connect, and SAML.

Discussion Venues

Note: This section is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/dickhardt/redirect-headers (https://github.com/dickhardt/redirect-headers).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 9 July 2026.

Table of Contents

1. Introduction

Authentication and authorization protocols (OAuth [RFC6749], OpenID Connect [OIDC], SAML [SAML]) use browser redirects to navigate users between applications and authorization servers. These redirects must carry protocol parameters, which historically appear in URLs or POSTed forms.

This document addresses two distinct problems in redirect-based protocols:

1. *Parameter Leakage*: URL parameters leak sensitive data (authorization codes, tokens, session identifiers) through browser history, Referer headers, server logs, analytics systems, and JavaScript access. POST-based redirects expose parameters in DOM form fields. Both mechanisms make sensitive data visible to unintended parties.

2. *Origin Verification*: Current mechanisms for verifying the origin of redirects are unreliable. The Referer header may be stripped, rewritten, or removed by privacy tools and enterprise proxies, preventing reliable mutual authentication between parties.

This document defines two HTTP headers that address these problems:

* *Redirect-Query* - Carries parameters in a browser-controlled header instead of URLs, preventing leakage while always being paired with origin verification
* *Redirect-Origin* - Provides browser-verified origin authentication that cannot be spoofed by scripts or manipulated by intermediaries

A third header, *Redirect-Path*, allows servers to request path-specific origin verification when finer-grained validation is needed.

Note on independent use: While origin verification without parameter passing is theoretically possible (server sends Redirect-Path, browser responds with Redirect-Origin), no specific use cases have been identified for this configuration.

Incremental Deployment: A key feature of this specification is that deployment does not require coordination between parties. Each party (client application, browser, authorization server) can independently add support for Redirect Headers. Full functionality emerges naturally when all three parties support the mechanism, but partial deployment gracefully degrades to existing URL-based behavior. This allows for organic adoption without requiring synchronized upgrades across the ecosystem.

2. Redirect Headers

Three headers work together during top-level 303 redirects to provide secure parameter passing and mutual authentication:

Redirect-Query carries parameters from servers (client applications or authorization servers) to the browser, which then forwards them to the next party. This keeps sensitive data out of URLs while maintaining the redirect flow.

Redirect-Origin provides browser-verified origin authentication. The browser sets this header when forwarding redirect parameters, allowing the receiving party to verify where the redirect originated. This cannot be spoofed by scripts or stripped by intermediaries.

Redirect-Path (optional) allows servers to request path-specific origin verification for finer-grained validation within an origin.

Browser behavior: Only processes these headers during top-level redirects. Ignores them for normal requests or embedded resources.

2.1. Redirect-Query

The Redirect-Query header carries redirect parameters using URL query string encoding. It is set by servers (either the client application or authorization server) in redirect responses.

Redirect-Query: "code=SplxlOBe&state=123"

Properties:

* Set by server in HTTP 303 redirect response
* Replaces URL query parameters
* Parsed using standard URL query string parsing

* Prevents exposure via browser history, Referer, logs, and analytics
* When present, browser MUST include Redirect-Origin in the subsequent request

2.2. Redirect-Origin

The Redirect-Origin header provides browser-verified origin authentication. It is set ONLY by the browser and contains the origin (and optionally path) of the page from which the redirect originated.

Format: Always ends with /

Properties:

* Set ONLY by the browser (cannot be spoofed by scripts or intermediaries)
* Enables receiving party to verify the origin of the redirect
* Provides mutual authentication between parties
* Always ends with / for consistent parsing
* May include validated path when Redirect-Path is used

Browser behavior:

The browser sets Redirect-Origin when either Redirect-Query or Redirect-Path is present in the redirect response.

2.2.1. Example 1: Origin-only verification (without Redirect-Path)

Current page: https://app.example/some/page

Server sends redirect:

```
HTTP/1.1 303 See Other
Location: https://as.example/authorize
Redirect-Query: "client_id=abc&state=123"
```

Browser forwards to AS:

```
GET /authorize
Host: as.example
Redirect-Origin: "https://app.example/"
Redirect-Query: "client_id=abc&state=123"
```

The Redirect-Origin is set to the origin plus /, without any path component.

2.2.2. Example 2: Origin+path verification (with Redirect-Path)

Current page: https://app.example/mobile/dashboard

Server sends redirect:

```
HTTP/1.1 303 See Other
Location: https://as.example/authorize
Redirect-Query: "client_id=abc&state=123"
Redirect-Path: "/mobile/"
```

Browser validates path claim: - Redirect-Path claim: /mobile/ - Current page path: /mobile/dashboard - Validation: Does /mobile/dashboard start with /mobile/? YES

Browser forwards to AS:

```
GET /authorize
Host: as.example
Redirect-Origin: "https://app.example/mobile/"
Redirect-Query: "client_id=abc&state=123"
```

The Redirect-Origin includes the validated path /mobile/ because the browser confirmed the current page is within that path.

If path validation fails:

If the current page was https://app.example/desktop/page and the server claimed Redirect-Path: "/mobile/", the browser would reject the path claim and send only the origin:

```
Redirect-Origin: "https://app.example/"
```

2.3. Redirect-Path

The Redirect-Path header allows a server to request path-specific origin verification. It is set by the server in the redirect response as a claim about the current page’s path. The browser validates this claim and, if valid, includes the path in Redirect-Origin.

Format: Must start with /

```
Redirect-Path: "/app1"
```

Server behavior:

The server includes Redirect-Path in the redirect response when it wants the receiving party to know not just the origin, but also a specific path within that origin.

Browser validation:

1. Server sends: Redirect-Path: "/api"
2. Browser checks: Does the current page path start with /api?
3. If valid: Include path in Redirect-Origin: "https://example.com/api/"
4. If invalid: Ignore the path claim, use origin only: Redirect-Origin: "https://example.com/"

This mechanism prevents path manipulation attacks where an attacker might try to redirect from an unexpected path within the same origin. The server cannot lie about its path because the browser enforces validation.

3. Feature Discovery

Some protocols may wish to discover browser support for Redirect Headers using Client Hints [RFC8942].

Server advertises support:

Accept-CH: Redirect-Supported

Browser responds with capability:

Redirect-Supported: ?1

Note: Feature discovery is optional and not required for OAuth flows. The incremental deployment model works without explicit discovery - authorization servers detect support by receiving Redirect-Query headers in requests.

4. OAuth Incremental Deployment

Redirect Headers is designed for *incremental adoption* - each party (client, browser, authorization server) can independently add support, with functionality emerging when all parties support it.

4.1. How It Works

Clients can start sending parameters in both locations:

* URL query string (for backward compatibility)
* Redirect-Query header (signaling support)

Browsers forward Redirect- headers when present:

* No special detection needed
* Simply forward the headers during redirects

Authorization Servers detect support and respond accordingly:

* If request includes Redirect-Query → AS knows both client and browser support it
* AS can then send response using *only* Redirect-Query header (no URL parameters)

4.2. Adoption Path

Each party adds support independently, in any order:

Clients add support by sending parameters in both the URL query string and the Redirect-Query header, and by looking for responses in the header while falling back to URL parameters.

Browsers add support by forwarding Redirect-* headers when present during redirects. No configuration is needed.

Authorization Servers add support by detecting when Redirect-Query is received (confirming that both the client and browser support it), then sending responses using only the Redirect-Query header and omitting URL parameters.

Result: Once all three parties support it, the authorization code is sent in the header rather than the URL.

No coordination required - each party adds support independently, and the system naturally converges to the secure behavior once all three support it. The client can immediately start sending both, browsers simply forward headers, and authorization servers detect support from incoming requests.

5. OAuth Redirect Security Threats

Scope: Redirect Headers specifically addresses OAuth and OIDC web-based redirect flows between websites where sensitive parameters are passed via URL query strings. This proposal does NOT address form_post mechanisms where data appears in the DOM, as that attack vector requires different mitigations.

5.1. Authorization Code Theft from URL Query Strings

The primary security weakness is in the *authorization server's response* containing the authorization code in the URL query string. When an AS redirects back to the client with ?code=...&state=..., the authorization code is exposed through multiple vectors:

Known attack vectors:

1. *Browser history leakage* - Authorization codes stored in browser history can be retrieved by attackers with device access
    * Reference: OAuth 2.0 Security Best Current Practice [RFC9700]
2. *Server log exposure* - Authorization codes visible in web server access logs, proxy logs, and load balancer logs
    * Codes can be extracted in real-time or from archived logs
3. *Referer header leakage* - When the callback page loads third-party resources (images, scripts, analytics), the authorization code leaks via the Referer header
    * Reference: OAuth 2.0 authentication vulnerabilities [PORTSWIGGER-OAUTH]
4. *Browser-swapping attacks* - Attackers exploit scenarios where authorization codes leak through shared URLs or when users switch browsers during the flow
    * Discussion: OAuth-WG Browser-Swapping thread
5. *URL sharing* - Users may inadvertently share URLs containing authorization codes after errors or confusion
6. *Analytics and crash reporting* - Authorization codes captured by analytics systems, error tracking, and monitoring tools

What Redirect Headers mitigates:

By moving the authorization code from the URL query string to the Redirect-Query header, *all of these attack vectors are eliminated*. The authorization code never appears in:

* URLs (no browser history exposure)
* Referer headers (no leakage to third parties)
* Server logs (when servers are configured to not log sensitive headers)

* User-visible locations (no accidental sharing)

Important clarification:

The security concern is specifically the *authorization server's response* with the authorization code. The *client's authorization request* to the AS (containing client_id, redirect_uri, state) does not have known security concerns from being in the URL, as these parameters are not sensitive credentials. However, moving them to headers provides consistency and reduces URL clutter.

6. Security Considerations

6.1. Header Confidentiality

Redirect-Query carries sensitive parameters (authorization codes, tokens, session identifiers) that MUST be protected with the same care as credentials. Servers MUST:

* Configure logging systems to exclude or redact Redirect-Query header values
* Treat Redirect-Query with the same confidentiality as Authorization headers
* Use TLS for all redirects carrying Redirect-Query to prevent network observation

Network intermediaries (proxies, load balancers, CDNs) can observe header values in transit. Deployment environments with untrusted intermediaries require additional protection beyond this specification.

6.2. Browser Implementation Requirements

Browsers MUST enforce strict isolation of Redirect headers:

* JavaScript MUST NOT be able to read or set Redirect-* headers via any API
* Browser extensions MUST NOT have access to Redirect-* headers
* Only the browser's redirect handling mechanism can create or consume these headers
* Headers MUST only be processed during top-level navigation redirects, never for subresource requests

Failure to enforce these restrictions would allow malicious scripts to forge origin claims or steal sensitive parameters.

6.3. Origin Verification Limits

Redirect-Origin provides browser-mediated mutual authentication but has limitations:

* It verifies the browser's understanding of origin, not the server's identity
* It does not authenticate the user or establish a secure channel
* It supplements but does NOT replace redirect_uri validation and registration
* Servers MUST continue to validate redirect_uri against registered values

Redirect-Path provides additional validation within an origin but cannot prevent attacks where the attacker controls a legitimate path within the same origin.

6.4. Browser Trust Model

This specification assumes an honest browser implementation. It cannot protect against:

* Compromised or malicious browsers
* Browser bugs that fail to enforce header restrictions
* Browser extensions with elevated privileges
* Debugging tools that modify headers

Servers should monitor for anomalous behavior (e.g., Redirect-Origin values that don't match expected patterns) as potential indicators of browser compromise or implementation bugs.

6.5. Transition Period Risks

During incremental deployment, clients may send parameters in both URL and headers for backward compatibility. This dual-sending pattern preserves URL leakage risks until:

* All parties (client, browser, server) support Redirect Headers
* Clients stop including parameters in URLs

Servers SHOULD detect Redirect-Query presence and warn or reject requests that also include sensitive parameters in URLs, to encourage migration away from URL-based parameters.

7. Privacy Considerations

7.1. URL History and Referer Leakage

When parameters remain in URLs (as during transition or with non-supporting implementations), sensitive data persists in browser history and may leak via Referer headers. Implementers should consider:

* Browser history is persistent storage that may be accessed by malware, forensic tools, or unauthorized users with device access
* Referer headers are sent automatically to third-party resources, potentially leaking parameters to analytics providers, CDNs, or advertisers
* The transition to headers does not eliminate these risks until all parties stop sending parameters in URLs

This specification does not eliminate all tracking vectors - cookies, browser fingerprinting, and other mechanisms remain unaffected.

7.2. Network Observer Privacy

Network intermediaries can still observe Redirect-* headers in transit, just as they can observe URL parameters today. This specification does not enhance privacy against network-level observers (ISPs, proxies, corporate firewalls). TLS remains essential for protecting parameters from network observation.

In some cases, moving parameters to headers may slightly improve privacy: unlike URL paths (which may be visible in TLS SNI or DNS queries), HTTP headers are encrypted by TLS and not visible to passive network observers.

7.3. Redirect-Origin Privacy Implications

Redirect-Origin explicitly reveals the origin (and optionally path) of the redirecting page. Implementers should consider:

* This is similar to the Referer header but more reliable and always present when Redirect-Query is used
* It enables the receiving party to know definitively where the redirect originated
* Unlike Referer (which users/tools can strip for privacy), Redirect-Origin cannot be disabled by the user when Redirect-Query is present
* This trade-off prioritizes security (mutual authentication) over origin hiding

Protocols using Redirect Headers should only be deployed where mutual knowledge of party identities is acceptable and expected (as in OAuth, where the AS and client already know each other).

7.4. User Control and Transparency

Users have limited control over Redirect Headers:

* Users cannot inspect or modify Redirect-* headers (unlike URL parameters which are visible)
* Users cannot selectively disable Redirect-Origin without breaking functionality
* Browser developer tools may or may not expose these headers depending on implementation

Browser implementations SHOULD provide visibility into Redirect Headers in developer tools for transparency, while maintaining the security restriction that JavaScript cannot access them.

7.5. Server Logging Practices

While Redirect Headers remove parameters from URLs (reducing accidental logging via URL-based logs), servers must implement appropriate logging controls:

* Configure web servers and load balancers to exclude Redirect-Query from access logs
* Ensure application logging redacts sensitive header values
* Be aware that default logging configurations may capture all headers

The shift to headers does not automatically prevent logging - it requires conscious configuration changes.

8. IANA Considerations

This document registers four new HTTP header fields in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" defined in [RFC9110].

8.1. Redirect-Query Header Field

Header field name: Redirect-Query

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): [this document]

8.2. Redirect-Origin Header Field

Header field name: Redirect-Origin

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): [this document]

8.3. Redirect-Path Header Field

Header field name: Redirect-Path

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): [this document]

8.4. Redirect-Supported Header Field

Header field name: Redirect-Supported

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): [this document]

Comments: Client Hint for feature discovery

9. Implementation Status

Note to RFC Editor: Please remove this section before publication.

Specification status: Exploratory draft

Browser support: Not yet implemented (proposed specification)

Server support: Reference implementations needed

This specification requires:

* Browser vendors to implement header handling
* Authorization servers to support Redirect-Query
* Client applications to adopt the pattern

Deployment strategy: Backward compatible - clients send both URL and headers during transition.

10. References

10.1. Normative References

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC8942] Grigorik, I. and Y. Weiss, "HTTP Client Hints", RFC 8942, DOI 10.17487/RFC8942, February 2021, <https://www.rfc-editor.org/info/rfc8942>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

10.2. Informative References

[OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

[PORTSWIGGER-OAUTH] PortSwigger, "OAuth 2.0 authentication vulnerabilities", 2024, <https://portswigger.net/web-security/oauth>.

[RFC9700] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "Best Current Practice for OAuth 2.0 Security", BCP 240, RFC 9700, DOI 10.17487/RFC9700, January 2025, <https://www.rfc-editor.org/info/rfc9700>.

[SAML] Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0", March 2005, <http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf>.

Appendix A. OAuth Example: Before and After

This appendix provides a detailed comparison of OAuth flows with and without Redirect Headers to illustrate the differences in security and functionality.

A.1. Without Redirect Headers (current OAuth)

Client Website returns to Browser:

```
HTTP/1.1 303 See Other
Location: https://as.example/authorize?client_id=abc&state=123&redirect_uri=...
```

Browser navigates, sends to AS:

```
GET /authorize?client_id=abc&state=123&redirect_uri=...
Host: as.example
Referer: https://app.example/login
```

The Referer header is unreliable and may be stripped by browsers or proxies.

AS returns code to Browser:

```
HTTP/1.1 303 See Other
Location: https://app.example/cb?code=SplxlOBe&state=123
```

The authorization code is now exposed in the URL.

Browser sends code to Client Website:

```
GET /cb?code=SplxlOBe&state=123
Host: app.example
Referer: https://as.example/consent
```

The authorization code is stored in browser history, server logs, and analytics systems. Third-party resources loaded by this page will receive the code via the Referer header.

Problems:

* Authorization code appears in URL (history, logs, Referer, extensions)
* No cryptographic origin verification (Referer is optional and unreliable)

A.2. With Redirect Headers

Client Website returns to Browser:

```
HTTP/1.1 303 See Other
Location: https://as.example/authorize
Redirect-Query: "client_id=abc&state=123&redirect_uri=https://app.example/portal/callback
"
Redirect-Path: "/portal/"
```

Browser validates and adds origin:

The browser validates the Redirect-Path claim against the current page URL. In this example, the current page is https://app.example/portal/login and the Redirect-Path claim is /portal/. Since the page path starts with /portal/, validation succeeds and the browser sets Redirect-Origin to https://app.example/portal/.

Browser navigates, sends to AS:

```
GET /authorize
Host: as.example
Redirect-Origin: "https://app.example/portal/"
Redirect-Query: "client_id=abc&state=123&redirect_uri=https://app.example/portal/callback
"
```

The Redirect-Origin is browser-supplied and cannot be spoofed. Parameters are transmitted in headers, not in the URL.

AS validates and returns to Browser:

The AS verifies that Redirect-Origin is https://app.example/portal/ and that the redirect_uri starts with https://app.example/portal/.

```
HTTP/1.1 303 See Other
Location: https://app.example/portal/callback
Redirect-Query: "code=SplxlOBe&state=123"
```

The authorization code is transmitted in the header. No parameters appear in the URL.

Browser forwards back to Client Website:

The browser is now on the AS consent page at https://as.example/consent. Since Redirect-Query is present in the response, the browser sets Redirect-Origin to https://as.example/.

```
GET /portal/callback
Host: app.example
Redirect-Origin: "https://as.example/"
Redirect-Query: "code=SplxlOBe&state=123"
```

The URL is clean with no query parameters. The client verifies that Redirect-Origin matches the expected AS. The authorization code never appears in the URL, browser history, or Referer headers.

Benefits:

* Authorization code never appears in URLs
* Mutual origin authentication (browser-verified)
* Backward compatible (browsers/servers without support fall back to URL parameters)

Requirements:

* If Redirect-Query received in request: AS MUST use Redirect-Query for response
* Client MUST verify Redirect-Origin matches expected AS
* AS MUST verify Redirect-Origin matches expected client
* When Redirect-Query is present, client MUST ignore URL parameters and use only header parameters

Appendix B. Acknowledgments

The authors would like to thank early reviewers for their valuable feedback and insights that helped shape this proposal: Jonas Primbs, Warren Parad.

Authors' Addresses

Dick Hardt
Hell
Email: dick.hardt@gmail.com

Sam Goto
Google
Email: goto@google.com