RFC 9126
OAuth 2.0 Pushed Authorization Requests

Abstract
This document defines the pushed authorization request (PAR) endpoint, which allows clients to push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and provides them with a request URI that is used as reference to the data in a subsequent call to the authorization endpoint.

Status of This Memo
This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9126.

Copyright Notice
Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
Table of Contents

1. Introduction
   1.1. Introductory Example
   1.2. Conventions and Terminology

2. Pushed Authorization Request Endpoint
   2.1. Request
   2.2. Successful Response
   2.3. Error Response
   2.4. Management of Client Redirect URIs

3. The "request" Request Parameter

4. Authorization Request

5. Authorization Server Metadata

6. Client Metadata

7. Security Considerations
   7.1. Request URI Guessing
   7.2. Open Redirection
   7.3. Request Object Replay
   7.4. Client Policy Change
   7.5. Request URI Swapping

8. Privacy Considerations

9. IANA Considerations
   9.1. OAuth Authorization Server Metadata
   9.2. OAuth Dynamic Client Registration Metadata
   9.3. OAuth URI Registration

10. References
   10.1. Normative References
   10.2. Informative References

Acknowledgements
1. Introduction

This document defines the pushed authorization request (PAR) endpoint, which enables an OAuth [RFC6749] client to push the payload of an authorization request directly to the authorization server. A request URI value is received in exchange; it is used as reference to the authorization request payload data in a subsequent call to the authorization endpoint via the user agent.

In OAuth [RFC6749], authorization request parameters are typically sent as URI query parameters via redirection in the user agent. This is simple but also yields challenges:

- There is no cryptographic integrity and authenticity protection. An attacker could, for example, modify the scope of access requested or swap the context of a payment transaction by changing scope values. Although protocol facilities exist to enable clients or users to detect some such changes, preventing modifications early in the process is a more robust solution.
- There is no mechanism to ensure confidentiality of the request parameters. Although HTTPS is required for the authorization endpoint, the request data passes through the user agent in the clear, and query string data can inadvertently leak to web server logs and to other sites via the referrer. The impact of such leakage can be significant, if personally identifiable information or other regulated data is sent in the authorization request (which might well be the case in identity, open banking, and similar scenarios).
- Authorization request URLs can become quite large, especially in scenarios requiring fine-grained authorization data, which might cause errors in request processing.

JWT-Secured Authorization Request (JAR) [RFC9101] provides solutions for the security challenges by allowing OAuth clients to wrap authorization request parameters in a Request Object, which is a signed and optionally encrypted JSON Web Token (JWT) [RFC7519]. In order to cope with the size restrictions, JAR introduces the request_uri parameter that allows clients to send a reference to a Request Object instead of the Request Object itself.

This document complements JAR by providing an interoperable way to push the payload of an authorization request directly to the authorization server in exchange for a request_uri value usable at the authorization server in a subsequent authorization request.

PAR fosters OAuth security by providing clients a simple means for a confidential and integrity-protected authorization request. Clients requiring an even higher security level, especially cryptographically confirmed non-repudiation, are able to use JWT-based Request Objects as defined by [RFC9101] in conjunction with PAR.
PAR allows the authorization server to authenticate the client before any user interaction happens. The increased confidence in the identity of the client during the authorization process allows the authorization server to refuse illegitimate requests much earlier in the process, which can prevent attempts to spoof clients or otherwise tamper with or misuse an authorization request.

Note that HTTP POST requests to the authorization endpoint via the user agent, as described in Section 3.1 of [RFC6749] and Section 3.1.2.1 of [OIDC], could also be used to cope with the request size limitations described above. However, it’s only optional per [RFC6749], and, even when supported, it is a viable option for conventional web applications but is prohibitively difficult to use with installed mobile applications. As described in [RFC8252], those apps use platform-specific APIs to open the authorization request URI in the system browser. When a mobile app launches a browser, however, the resultant initial request is constrained to use the GET method. Using POST for the authorization request would require the app to first direct the browser to open a URI that the app controls via GET while somehow conveying the sizable authorization request payload and then having the resultant response contain the content and script to initiate a cross-site form POST towards the authorization server. PAR is simpler to use and has additional security benefits, as described above.

1.1. Introductory Example

In conventional OAuth 2.0, a client typically initiates an authorization request by directing the user agent to make an HTTP request like the following to the authorization server’s authorization endpoint (extra line breaks and indentation for display purposes only):

```plaintext
GET /authorize?response_type=code
&client_id=CLIENT1234&state=duk681S8n00GsJpe7n9boxdzen
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcba HTTP/1.1
Host: as.example.com
```
Such a request could instead be pushed directly to the authorization server by the client with a POST request to the PAR endpoint as illustrated in the following example (extra line breaks and spaces for display purposes only). The client can authenticate (e.g., using JWT client assertion-based authentication as shown) because the request is made directly to the authorization server.

```plaintext
POST /as/par HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

)&response_type=code
&client_id=CLIENT1234&state=duk681S8n00Gsjpe7n9boxdzen
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&client_assertion_type=
urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJraWQiOiI0MiIsImFsZyI6IkVTMjU2In0.eyJpc3MiOiJDTE
lFTlQxMjMOiwic3ViIjoiOQxJRU5UMTIzNCIsImF1ZCI6ImlhodHBzOi8vc2VydvLy
mV4YW1wbGUuY29tIiwiaXN0b3J5IjoiQ0xJRU5UMTIzNCIsImF1ZCI6ImlhodHBzOi8vc2VydvLy
wbLjeIywqWUu-ywgyvvef1_0sQJftNs3bzjIrP0BV9rRG-3eI1Ksh0kQ1CwvzA

The authorization server responds with a request URI:

```json
HTTP/1.1 201 Created
Cache-Control: no-cache, no-store
Content-Type: application/json

```
"request_uri": "urn:example:bwc4JK-ESC0w8acc191e-Y1LTC2",
"expires_in": 90
}

The client uses the request URI value to create the subsequent authorization request by directing the user agent to make an HTTP request to the authorization server's authorization endpoint like the following (extra line breaks and indentation for display purposes only):

```plaintext
GET /authorize?client_id=CLIENT1234
&request_uri=urn%3Aexample%3Abwc4JK-ESC0w8acc191e-Y1LTC2 HTTP/1.1
Host: as.example.com

1.2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.
This specification uses the terms "access token", "authorization server", "authorization endpoint", "authorization request", "token endpoint", and "client" defined by "The OAuth 2.0 Authorization Framework" [RFC6749].

2. Pushed Authorization Request Endpoint

The pushed authorization request endpoint is an HTTP API at the authorization server that accepts HTTP POST requests with parameters in the HTTP request message body using the application/x-www-form-urlencoded format. This format has a character encoding of UTF-8, as described in Appendix B of [RFC6749]. The PAR endpoint URL MUST use the "https" scheme.

Authorization servers supporting PAR SHOULD include the URL of their pushed authorization request endpoint in their authorization server metadata document [RFC8414] using the pushed_authorization_request_endpoint parameter as defined in Section 5.

The endpoint accepts the authorization request parameters defined in [RFC6749] for the authorization endpoint as well as all applicable extensions defined for the authorization endpoint. Some examples of such extensions include Proof Key for Code Exchange (PKCE) [RFC7636], Resource Indicators [RFC8707], and OpenID Connect (OIDC) [OIDC]. The endpoint MAY also support sending the set of authorization request parameters as a Request Object according to [RFC9101] and Section 3 of this document.

The rules for client authentication as defined in [RFC6749] for token endpoint requests, including the applicable authentication methods, apply for the PAR endpoint as well. If applicable, the token_endpoint_auth_method client metadata parameter [RFC7591] indicates the registered authentication method for the client to use when making direct requests to the authorization server, including requests to the PAR endpoint. Similarly, the token_endpoint_auth_methods_supported authorization server metadata [RFC8414] parameter lists client authentication methods supported by the authorization server when accepting direct requests from clients, including requests to the PAR endpoint.

Due to historical reasons, there is potential ambiguity regarding the appropriate audience value to use when employing JWT client assertion-based authentication (defined in Section 2.2 of [RFC7523] with private_key_jwt or client_secret_jwt authentication method names per Section 9 of [OIDC]). To address that ambiguity, the issuer identifier URL of the authorization server according to [RFC8414] SHOULD be used as the value of the audience. In order to facilitate interoperability, the authorization server MUST accept its issuer identifier, token endpoint URL, or pushed authorization request endpoint URL as values that identify it as an intended audience.

2.1. Request

A client sends the parameters that comprise an authorization request directly to the PAR endpoint. A typical parameter set might include: client_id, response_type, redirect_uri, scope, state, code_challenge, and code_challenge_method as shown in the example below. However, the pushed authorization request can be composed of any of the parameters applicable
for use at the authorization endpoint, including those defined in [RFC6749] as well as all applicable extensions. The request_uri authorization request parameter is one exception, and it **MUST NOT** be provided.

The request also includes, as appropriate for the given client, any additional parameters necessary for client authentication (e.g., client_secret or client_assertion and client_assertion_type). Such parameters are defined and registered for use at the token endpoint but are applicable only for client authentication. When present in a pushed authorization request, they are relied upon only for client authentication and are not germane to the authorization request itself. Any token endpoint parameters that are not related to client authentication have no defined meaning for a pushed authorization request. The client_id parameter is defined with the same semantics for both authorization requests and requests to the token endpoint; as a required authorization request parameter, it is similarly required in a pushed authorization request.

The client constructs the message body of an HTTP POST request with parameters formatted with x-www-form-urlencoded using a character encoding of UTF-8, as described in Appendix B of [RFC6749]. If applicable, the client also adds its authentication credentials to the request header or the request body using the same rules as for token endpoint requests.

This is illustrated by the following example (extra line breaks in the message body for display purposes only):

```
POST /as/par HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

response_type=code&state=af0ifjsldkj&client_id=s6BhdRkqt3
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&code_challenge=K2-1tc83acc4h0c9w6ESC_rEMT3bwu-uCHaoeK1t8U
&code_challenge_method=S256&scope=account-information
&client_assertion_type=

urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJraWQiOiJrMmJkYyIsImFsZyI6IlI1JTmjU2In0.eyJpc3Mi
O1jzNkJoZFJrcXQzIwic3V1jioiczCAgRSa3F6MyIsmF1ZCI6ImlhBzdZi8vc
2VydmlvbW9yWCBuYW5rIzY9IiwiX1ZhbWJjciI6IjI1ODY5Njc3fQ.te4IdnP_DK4hWrh
TWa6fyhy3f xlA0QZAhfaA4lmRdpp5uzb-E90R5YxzN1YDA8mmVdpj_Bx1l05r6se
f5Tlc0ka3ahahhOB44dq1E4naEmL1sNa1pds2WxTM0UzZY8akKSDzNTDqhyTgcE-K
d7B3RafRj7td2b9zW7S_cmo0vfcQOiy5zzi1BvLQKWY8Js3yvdp2AVpxRPbcP8W
yeW9B6PL6_fy3pXYKG3e-qUcvPa9kan-mo9EoSgt-YSRD0jK1nZMdxIxITluK9caVE
RWW0fD1Y11_tl0cJn-ya7v7d8YmFypkhZfm8x1FoeH0djEicXTixKedRuzsgUCm6
GQ
```

The authorization server **MUST** process the request as follows:

1. Authenticate the client in the same way as at the token endpoint (Section 2.3 of [RFC6749]).
2. Reject the request if the request_uri authorization request parameter is provided.
3. Validate the pushed request as it would an authorization request sent to the authorization endpoint. For example, the authorization server checks whether the redirect URI matches
one of the redirect URIs configured for the client and also checks whether the client is authorized for the scope for which it is requesting access. This validation allows the authorization server to refuse unauthorized or fraudulent requests early. The authorization server **MAY** omit validation steps that it is unable to perform when processing the pushed request; however, such checks **MUST** then be performed when processing the authorization request at the authorization endpoint.

The authorization server **MAY** allow clients with authentication credentials to establish per-authorization-request redirect URIs with every pushed authorization request. Described in more detail in Section 2.4, this is possible since, in contrast to [RFC6749], this specification gives the authorization server the ability to authenticate clients and validate client requests before the actual authorization request is performed.

### 2.2. Successful Response

If the verification is successful, the server **MUST** generate a request URI and provide it in the response with a 201 HTTP status code. The following parameters are included as top-level members in the message body of the HTTP response using the application/json media type as defined by [RFC8259].

- **request_uri**
  The request URI corresponding to the authorization request posted. This URI is a single-use reference to the respective request data in the subsequent authorization request. The way the authorization process obtains the authorization request data is at the discretion of the authorization server and is out of scope of this specification. There is no need to make the authorization request data available to other parties via this URI.

- **expires_in**
  A JSON number that represents the lifetime of the request URI in seconds as a positive integer. The request URI lifetime is at the discretion of the authorization server but will typically be relatively short (e.g., between 5 and 600 seconds).

The format of the **request_uri** value is at the discretion of the authorization server, but it **MUST** contain some part generated using a cryptographically strong pseudorandom algorithm such that it is computationally infeasible to predict or guess a valid value (see Section 10.10 of [RFC6749] for specifics). The authorization server **MAY** construct the **request_uri** value using the form `urn:ietf:params:oauth:request_uri:<reference-value>` with `<reference-value>` as the random part of the URI that references the respective authorization request data.

The **request_uri** value **MUST** be bound to the client that posted the authorization request.
The following is an example of such a response:

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "request_uri":
    "urn:ietf:params:oauth:request_uri:6esc_11ACC5bwc014ltc14eY22c",
  "expires_in": 60
}
```

### 2.3. Error Response

The authorization server returns an error response with the same format as is specified for error responses from the token endpoint in Section 5.2 of [RFC6749] using the appropriate error code from therein or from Section 4.1.2.1 of [RFC6749]. In those cases where Section 4.1.2.1 of [RFC6749] prohibits automatic redirection with an error back to the requesting client and hence doesn't define an error code (for example, when the request fails due to a missing, invalid, or mismatching redirection URI), the invalid_request error code can be used as the default error code. Error codes defined by the OAuth extension can also be used when such an extension is involved in the initial processing of the authorization request that was pushed. Since initial processing of the pushed authorization request does not involve resource owner interaction, error codes related to user interaction, such as consent_required defined by [OIDC], are never returned.

If the client is required to use signed Request Objects, by either the authorization server or the client policy (see [RFC9101], Section 10.5), the authorization server MUST only accept requests complying with the definition given in Section 3 and MUST refuse any other request with HTTP status code 400 and error code invalid_request.

In addition to the above, the PAR endpoint can also make use of the following HTTP status codes:

- **405**: If the request did not use the POST method, the authorization server responds with an HTTP 405 (Method Not Allowed) status code.
- **413**: If the request size was beyond the upper bound that the authorization server allows, the authorization server responds with an HTTP 413 (Payload Too Large) status code.
- **429**: If the number of requests from a client during a particular time period exceeds the number the authorization server allows, the authorization server responds with an HTTP 429 (Too Many Requests) status code.
The following is an example of an error response from the PAR endpoint:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "error": "invalid_request",
  "error_description": "The redirect_uri is not valid for the given client"
}
```

### 2.4. Management of Client Redirect URIs

OAuth 2.0 [RFC6749] allows clients to use unregistered `redirect_uri` values in certain circumstances or for the authorization server to apply its own matching semantics to the `redirect_uri` value presented by the client at the authorization endpoint. However, the OAuth security BCP [OAUTH-SECURITY-TOPICS] as well as the OAuth 2.1 specification [OAUTH-V2] require an authorization server to exactly match the `redirect_uri` parameter against the set of redirect URIs previously established for a particular client. This is a means for early detection of client impersonation attempts and prevents token leakage and open redirection. As a downside, this can make client management more cumbersome since the redirect URI is typically the most volatile part of a client policy.

The exact matching requirement MAY be relaxed when using PAR for clients that have established authentication credentials with the authorization server. This is possible since, in contrast to a conventional authorization request, the authorization server authenticates the client before the authorization process starts and thus ensures it is interacting with the legitimate client. The authorization server MAY allow such clients to specify `redirect_uri` values that were not previously registered with the authorization server. This will give the client more flexibility (e.g., to mint distinct `redirect_uri` values per authorization server at runtime) and can simplify client management. It is at the discretion of the authorization server to apply restrictions on supplied `redirect_uri` values, e.g., the authorization server MAY require a certain URI prefix or allow only a query parameter to vary at runtime.

Note: The ability to set up transaction-specific redirect URIs is also useful in situations where client IDs and corresponding credentials and policies are managed by a trusted third party, e.g., via client certificates containing client permissions. Such an externally managed client could interact with an authorization server trusting the respective third party without the need for an additional registration step.
3. The "request" Request Parameter

Clients **MAY** use the request parameter as defined in JAR [RFC9101] to push a Request Object JWT to the authorization server. The rules for processing, signing, and encryption of the Request Object as defined in JAR [RFC9101] apply. Request parameters required by a given client authentication method are included in the application/x-www-form-urlencoded request directly and are the only parameters other than request in the form body (e.g., mutual TLS client authentication [RFC8705] uses the client_id HTTP request parameter, while JWT assertion-based client authentication [RFC7523] uses client_assertion and client_assertion_type). All other request parameters, i.e., those pertaining to the authorization request itself, **MUST** appear as claims of the JWT representing the authorization request.

The following is an example of a pushed authorization request using a signed Request Object with the same authorization request payload as the example in Section 2.1. The client is authenticated with JWT client assertion-based authentication [RFC7523] (extra line breaks and spaces for display purposes only):

```
POST /as/par HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJraWQiOiJrMmJkYyIsImFsZyI6I1JTmjU2In0.eyJpc3MiOiJzNkJoZJrcQzQiic3ViIjoiczZCaGRsa3F0MyIsImFiZCI6ImhdHdHzOiI8vc2VydmYyLmV4YW1wbGUuY29tIiwiZXhvIjoxNjI0ODY5NjczfQ.te4I4nPg_Dk4whWHRwTWA6fhy3fx1AQAzAfA41zmRdpoP5uZb-E90R5yzN1YDA8mmVdpqj_BxI1G5rsef5T1ckAPa3hahhCBQ4dcqLe4naEmLiSmN1pds2WxTMUozZYaKKSdzNTDklyTgE-KdTb3RafRj7tdZb9zW7sc_moOvpfCQIOy5zzBvLQKWy1YJSyvipdu2ApvxRPbchP8yw9B6PL6_f+y3pYKGYe-qOcVpa9kan-mo9Eosgt-YTDQkj1nZMdxIqTluK9caVJERWW0fdY111t0cNn-ya7v7d8yMfYJpkhZfms8x1FoeH8djEicXTtExkDruzgUcm6GQ
&request=eyJraWQiOiJrMmJkJkYyIsImFsZyI6I1JTmjU2In0.eyJpc3MiOiJzNkJoZJrcQzQiic3ViIjoiczZCaGRsa3F0MyIsImFiZCI6ImhdHdHzOiI8vc2VydmYyLmV4YW1wbGUuY29tIiwiZXhvIjoxNjI0ODY5NjczfQ.te4I4nPg_Dk4whWHRwTWA6fhy3fx1AQAzAfA41zmRdpoP5uZb-E90R5yzN1YDA8mmVdpqj_BxI1G5rsef5T1ckAPa3hahhCBQ4dcqLe4naEmLiSmN1pds2WxTMUozZYaKKSdzNTDklyTgE-KdTb3RafRj7tdZb9zW7sc_moOvpfCQIOy5zzBvLQKWy1YJSyvipdu2ApvxRPbchP8yw9B6PL6_f+y3pYKGYe-qOcVpa9kan-mo9Eosgt-YTDQkj1nZMdxIqTluK9caVJERWW0fdY111t0cNn-ya7v7d8yMfYJpkhZfms8x1FoeH8djEicXTtExkDruzgUcm6GQ
```

---

Lodderstedt, et al. Standards Track Page 11
The authorization server **MUST** take the following steps beyond the processing rules defined in Section 2.1:

1. If applicable, decrypt the Request Object as specified in JAR [RFC9101], Section 6.1.
2. Validate the Request Object signature as specified in JAR [RFC9101], Section 6.2.
3. If the client has authentication credentials established with the authorization server, reject the request if the authenticated client_id does not match the client_id claim in the Request Object. Additionally, requiring the iss claim to match the client_id is at the discretion of the authorization server.

The following RSA key pair, represented in JSON Web Key (JWK) format, can be used to validate or recreate the Request Object signature in the above example (extra line breaks and indentation within values for display purposes only):

```json
{
  "kty": "RSA",
  "kid": "k2bdc",
  "n": "y9Lqv4fCp6Ei-u2-ZCKq83YvbFEk6JMspSj76eMkddWRuWX2aBKGHAtK1E5P7_vn_-PKZKWept3vGBK6ePgzAfuO88NkmewmE5bQI0e6kIChtt_6kzT5oa
aXDFI6qCLJmk51cc44VYFqaxgqevMncYrzaw_.50mZ1yG5fIQzLYP8bijAHGvjdEFGaZaEN91sn_GdWLajHprB3R0lS50E45wxrl9gMncVb8QDuXzryvhGL0HzoUyRradBJvoWZowDNTPkpk2Rk17QaB07XDv3uR7s_sf2g-bAjSyXYUGsqkNA9b3xW53am_uz3zTbFIh557JICWKhIwJ5uzeJXaw",
  "e": "AQAB",
  "d": "LNgw_pCKrwawALCpRcdOkLsvy1SurzhE6CpkRi9eC9dpGkIK09CxPIxOL
zzjxuQcu8ModMrQ0ZTnAwdg7Hh86Gncrrv3NewI-XQV8ckldTjqNFOTZt1V
Rsje-57kAXI3YBiHu-0YpIdZdk_wBaAk661Svn6GsPQe7m9Doxzenu9Q
0_soewUhlPzRrTH0EeiQyi715ri3TYszoWBmEPD2fIcYji8Bf0MPy_Sqz
K3ovUJUJfzflNnJsiW_y-v63QBCMqJrOShHdMniz91vIpJwWqj305n_21C2-
cSwjmnKsFZdBQNCj7qM616EssJUwg5JxzTeAUFFw4YAQ",
  "q": "J-mG9swRFTy3atrcQ7dd0hhYntE9QndN-
-sD4E0Q00rnFj6weFCwIc4
7HctVeFnCTPYJncJyv-mu-9vlz5S5GSNuyR5qdpMsMZXUMpEvQwcwTk3tfFZ
YGaqFkEyEemfW_i88FfCeE689RE0jnnKrXm7w2-Iu_GrVJA90X-u",
  "q": "4h1MYAg0dvogdKlgjnxQ7J-Lqppq199-aeoFvoYpmPhctHzwFZ091qMuo
BpMqVQTws_s7VWgmtZABB3ywtkurf0V7BD0fwejiUZrW4KkJxtnp=auxr
jms3s2FI9n60wRY52H8ij9A7C2DnYJcuZiJOMYCWdsZ8-d-lA-s",
  "p": "5d9Er312F9TR-gy84-oakEyCmgwOv3bE_BfYEEOQwsvi27uICVK3bSEL
5CWQ69Ng3BH3Whqy8AyiPirwnqmq8k9mX1E-4xM10WZW-rP3XjyPQ6S9nSru5
LFYwusAzT-FX7dOBiBAxlaLEGxcxa441b0ie_bPD3dx1u5_1YoE",
  "dq": "NzZPF3M6ebcE4xSluK2078ErYdKgdITJReUR7Rno_t0zpejwIPBGYYW19
zpAeeYtcBT2jrxr2BQxLhxGELxEPQPzsz2rTJKSe4BhHy2mluxSDGdcsrb
NoKUKaNiCuyZszhW1n0AT_bE14bJgq_ofH0UeSjYBBU7tgr_k",
  "dp": "Zc877jirkkLOtyTs2vxyNe9KnMNAMo1dUc2te-89gAL4OpohSwKfKtwe
ZJ-gqtk1hT-dwNw_0Xtg_-NXsadMRMwnzBMYyYAfjApUkfqABC8yUClJ13
KqzRcUGf1WXUK96ZAHZ2-x8P0UpdNUEa70ISowPRh04HANKX4fkjWAE"
}
```
4. Authorization Request

The client uses the request_uri value returned by the authorization server to build an authorization request as defined in [RFC9101]. This is shown in the following example where the client directs the user agent to make the following HTTP request (extra line breaks and indentation for display purposes only):

```
GET /authorize?client_id=s6BhdRkqt3&request_uri=urn%3Aietf%3Aparams
    %3Aoauth%3Arequest_uri%3A6esc_11ACC5bwc014ltc14eY22c HTTP/1.1
Host: as.example.com
```

Since parts of the authorization request content, e.g., the code_challenge parameter value, are unique to a particular authorization request, the client MUST only use a request_uri value once. Authorization servers SHOULD treat request_uri values as one-time use but MAY allow for duplicate requests due to a user reloading/refreshing their user agent. An expired request_uri MUST be rejected as invalid.

The authorization server MUST validate authorization requests arising from a pushed request as it would any other authorization request. The authorization server MAY omit validation steps that it performed when the request was pushed, provided that it can validate that the request was a pushed request and that the request or the authorization server's policy has not been modified in a way that would affect the outcome of the omitted steps.

Authorization server policy MAY dictate, either globally or on a per-client basis, that PAR be the only means for a client to pass authorization request data. In this case, the authorization server will refuse, using the invalid_request error code, to process any request to the authorization endpoint that does not have a request_uri parameter with a value obtained from the PAR endpoint.

Note: Authorization server and clients MAY use metadata as defined in Sections 5 and 6 to signal the desired behavior.

5. Authorization Server Metadata

The following authorization server metadata parameters [RFC8414] are introduced to signal the server's capability and policy with respect to PAR.

pushed_authorization_request_endpoint

The URL of the pushed authorization request endpoint at which a client can post an authorization request to exchange for a request_uri value usable at the authorization server.
require_pushed_authorization_requests
Boolean parameter indicating whether the authorization server accepts authorization request data only via PAR. If omitted, the default value is false.

Note that the presence of pushed_authorization_request_endpoint is sufficient for a client to determine that it may use the PAR flow. A request_uri value obtained from the PAR endpoint is usable at the authorization endpoint regardless of other authorization server metadata such as request_uri_parameter_supported or require_request_uri_registration [OIDC.Disco].

6. Client Metadata
The Dynamic Client Registration Protocol [RFC7591] defines an API for dynamically registering OAuth 2.0 client metadata with authorization servers. The metadata defined by [RFC7591], and registered extensions to it, also imply a general data model for clients that is useful for authorization server implementations even when the Dynamic Client Registration Protocol isn't in play. Such implementations will typically have some sort of user interface available for managing client configuration. The following client metadata parameter is introduced by this document to indicate whether pushed authorization requests are required for the given client.

require_pushed_authorization_requests
Boolean parameter indicating whether the only means of initiating an authorization request the client is allowed to use is PAR. If omitted, the default value is false.

7. Security Considerations

7.1. Request URI Guessing
An attacker could attempt to guess and replay a valid request URI value and try to impersonate the respective client. The authorization server MUST account for the considerations given in JAR [RFC9101], Section 10.2, clause (d) on request URI entropy.

7.2. Open Redirection
An attacker could try to register a redirect URI pointing to a site under their control in order to obtain authorization codes or launch other attacks towards the user. The authorization server MUST only accept new redirect URIs in the pushed authorization request from authenticated clients.

7.3. Request Object Replay
An attacker could replay a request URI captured from a legitimate authorization request. In order to cope with such attacks, the authorization server SHOULD make the request URIs one-time use.
7.4. Client Policy Change

The client policy might change between the lodging of the Request Object and the authorization request using a particular Request Object. Therefore, it is recommended that the authorization server check the request parameter against the client policy when processing the authorization request.

7.5. Request URI Swapping

An attacker could capture the request URI from one request and then substitute it into a different authorization request. For example, in the context of OpenID Connect, an attacker could replace a request URI asking for a high level of authentication assurance with one that requires a lower level of assurance. Clients SHOULD make use of PKCE [RFC7636], a unique state parameter [RFC6749], or the OIDC "nonce" parameter [OIDC] in the pushed Request Object to prevent this attack.

8. Privacy Considerations

OAuth 2.0 is a complex and flexible framework with broad-ranging privacy implications due to its very nature of having one entity intermediate user authorization to data access between two other entities. The privacy considerations of all of OAuth are beyond the scope of this document, which only defines an alternative way of initiating one message sequence in the larger framework. However, using PAR may improve privacy by reducing the potential for inadvertent information disclosure since it passes the authorization request data directly between the client and authorization server over a secure connection in the message body of an HTTP request rather than in the query component of a URL that passes through the user agent in the clear.

9. IANA Considerations

9.1. OAuth Authorization Server Metadata

IANA has registered the following values in the IANA "OAuth Authorization Server Metadata" registry of [IANA.OAuth.Parameters] established by [RFC8414].

- Metadata Name: pushed_authorization_request_endpoint
  Metadata Description: URL of the authorization server's pushed authorization request endpoint.
  Change Controller: IESG
  Specification Document(s): Section 5 of RFC 9126

- Metadata Name: require_pushed_authorization_requests
  Metadata Description: Indicates whether the authorization server accepts authorization requests only via PAR.
  Change Controller: IESG
9.2. OAuth Dynamic Client Registration Metadata

IANA has registered the following value in the IANA "OAuth Dynamic Client Registration Metadata" registry of [IANA.OAuth.Parameters] established by [RFC7591].

Client Metadata Name: require_pushed_authorization_requests
Client Metadata Description: Indicates whether the client is required to use PAR to initiate authorization requests.
Change Controller: IESG
Specification Document(s): Section 5 of RFC 9126

9.3. OAuth URI Registration

IANA has registered the following value in the "OAuth URI" registry of [IANA.OAuth.Parameters] established by [RFC6755].

URN: urn:ietf:params:oauth:request_uri:
Common Name: A URN Sub-Namespace for OAuth Request URIs.
Change Controller: IESG
Specification Document(s): Section 2.2 of RFC 9126

10. References

10.1. Normative References


10.2. Informative References


Acknowledgements

This specification is based on the work on Pushed Request Object conducted at the Financial-grade API Working Group at the OpenID Foundation. We would like to thank the members of the WG for their valuable contributions.

We would like to thank Vladimir Dzhuvinov, Aaron Parecki, Justin Richer, Sascha Preibisch, Daniel Fett, Michael B. Jones, Annabelle Backman, Joseph Heenan, Sean Glencross, Maggie Hung, Neil Madden, Karsten Meyer zu Selhausen, Roman Danyliw, Meral Shirazipour, and Takahiko Kawasaki for their valuable feedback on this document.

Authors' Addresses

Torsten Lodderstedt
yes.com
Email: torsten@lodderstedt.net

Brian Campbell
Ping Identity
Email: bcampbell@pingidentity.com

Nat Sakimura
NAT.Consulting
Email: nat@sakimura.org

Dave Tonge
Moneyhub Financial Technology
Email: dave@tonge.org

Filip Skokan
Auth0
Email: panva.ip@gmail.com