Stream: Internet Engineering Task Force (IETF)
RFC: 9126
Category: Standards Track
Published: September 2021
ISSN: 2070-1721
Authors: T. Lodderstedt, *yes.com*; B. Campbell, *Ping Identity*; N. Sakimura, *NAT.Consulting*; D. Tonge, *Moneyhub Financial Technology*; F. Skokan, *Auth0*

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.

Table of Contents

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):

```
GET /authorize?response_type=code
 &client_id=CLIENT1234&state=duk681S8n00GsJpe7n9boxdzen
 &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb 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.

```
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
 lFTlQxMjM0Iiwic3ViIjoiQ0xJRU5UMTIzNCIsImF1ZCI6Imh0dHBzOi8vc2VydmVyL
 mV4YW1wbGUuY29tIiwiZXhwIjoxNjI1ODY4ODc4fQ.Igw8QrpAWRNPDGoWGRmJumLBM
 wbLjeIYwqWUu-ywgvvufl_0sQJftNs3bzjIrP0BV9rRG-3eI1Ksh0kQ1CwvzA
```

The authorization server responds with a request URI:

```
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):

```
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-ltc83acc4h0c9w6ESC_rEMTJ3bww-uCHaoeK1t8U
&code_challenge_method=S256&scope=account-information
&client_assertion_type=
 urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJraWQiOiJrMmJkYyIsImFsZyI6IlJTMjU2In0.eyJpc3Mi
 OiJzNkJoZFJrcXQzIiwic3ViIjoiczZCaGRSa3F0MyIsImF1ZCI6Imh0dHBzOi8vc
 2VydmVyLmV4YW1wbGUuY29tIiwiZXhwIjoxNjI1ODY5Njc3fQ.te4IdnP_DK4hWrh
 TWA6fyhy3fxlAQZAhfA4lmzRdpoP5uZb-E90R5YxzN1YDA8mnVdpgj_Bx1lG5r6se
 f5TlckApA3hahhC804dcqlE4naEmLISmN1pds2WxTMOUzZY8aKKSDzNTDqhyTgE-K
 dTb3RafRj7tdZb09zWs7c_moOvfVcQIoy5zz1BvLQKW1Y8JsYvdpu2AvpxRPbcP8W
 yeW9B6PL6_fy3pXYKG3e-qUcvPa9kan-mo9EoSgt-YTDQjK1nZMdXIqTluK9caVJE
 RWW0fD1Y11_tlOcJn-ya7v7d8YmFyJpkhZfm8x1FoeH0djEicXTixEkdRuzsgUCm6
 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=eyJraWQiOiJrMmJkYyIsImFsZyI6IlJTMjU2In0.eyJpc3Mi
 OiJzNkJoZFJrcXQzIiwic3ViIjoiczZCaGRSa3F0MyIsImF1ZCI6Imh0dHBzOi8vc
 2VydmVyLmV4YW1wbGUuY29tIiwiZXhwIjoxNjI1ODY5Njc3fQ.te4IdnP_DK4hWrh
 TWA6fyhy3fxlAQZAhfA4lmzRdpoP5uZb-E90R5YxzN1YDA8mnVdpgj_Bx1lG5r6se
 f5TlckApA3hahhC804dcqlE4naEmLISmN1pds2WxTMOUzZY8aKKSDzNTDqhyTgE-K
 dTb3RafRj7tdZb09zWs7c_moOvfVcQIoy5zz1BvLQKW1Y8JsYvdpu2AvpxRPbcP8W
 yeW9B6PL6_fy3pXYKG3e-qUcvPa9kan-mo9EoSgt-YTDQjK1nZMdXIqTluK9caVJE
 RWW0fD1Y11_tlOcJn-ya7v7d8YmFyJpkhZfm8x1FoeH0djEicXTixEkdRuzsgUCm6
 GQ
&request=eyJraWQiOiJrMmJkYyIsImFsZyI6IlJTMjU2In0.eyJpc3MiOiJzNkJoZ
 FJrcXQzIiwiYXVkIjoiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLCJleHAiOj
 E2MjU4Njk2NzcsInJlc3BvbnNlX3R5cGUiOiJjb2RlIiwiY2xpZW50X2lkIjoiczZ
 CaGRSa3F0MyIsInJlZGlyZWN0X3VyaSI6Imh0dHBzOi8vY2xpZW50LmV4YW1wbGUu
 b3JnL2NiIiwic2NvcGUiOiJhY2NvdW50LWluZm9ybWF0aW9uIiwic3RhdGUiOiJhZ
 jBpZmpzbGRraiIsImNvZGVfY2hhbGxlbmdlIjoiSzItbHRjODNhY2M0aDBjOXc2RV
 NDX3JFTVRKM2J3dy11Q0hhb2VLMXQ4VSIsImNvZGVfY2hhbGxlbmdlX21ldGhvZCI
 6IlMyNTYifQ.l9R3RC9bFBHry_8acObQjEf4fX5yfJkWUPfak3J3iiBm0aaQznPw5
 BZ0B3VQZ9_KYdPt5bTkaflS5fSDklM3_7my9MyOSKFYmf46INk6ju_qUuC2crkOQX
 ZWYJB-0bnYEbdHpUjazFSUvN49cEGstNQeE-dKDWHNgEojgcuNA_pjKfL9VYp1dEA
 6-WjXZ_OlJ7R_mBWpjFAzc0UkQwqX5hfOJoGTqB2tE4a4aB2z8iYlUJp0DeeYp_hP
 N6svtmdvte73p5bLGDFpRIlmrBQIAQuxiS0skORpXlS0cBcgHimXVnXOJG7E-A_lS
 _5y54dVLQPA1jKYx-fxbYSG7dp2fw
&client_id=s6BhdRkqt3
```

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 [RFC7517], 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):

```
{
  "kty": "RSA",
  "kid":"k2bdc",
  "n": "y9Lqv4fCp6Ei-u2-ZCKq83YvbFEk6JMs_pSj76eMkddWRuWX2aBKGHAtKlE
        5P7_vn__PCKZWePt3vGkB6ePgzAFu08NmKemwE5bQI0e6kIChtt_6KzT5Oa
        aXDFI6qCLJmk51Cc4VYFaxgqevMncYrzaW_50mZ1yGSFIQzLYP8bijAHGVj
        dEFgZaZEN9lsn_GdWLaJpHrB3ROlS50E45wxrlg9xMncVb8qDPuXZarvghL
        L0HzOuYRadBJVoWZowDNTpKpk2RklZ7QaBO7XDv3uR7s_sf2g-bAjSYxYUG
        sqkNA9b3xVW53am_UZZ3tZbFTIh557JICWKHlWj5uzeJXaw",
  "e": "AQAB",
  "d": "LNwG_pCKrwowALpCpRdcOKlSVqylSurZhE6CpkRiE9cpDgGKIkO9CxPlXOL
        zjqxXuQc8MdMqRQZTnAwgd7HH0B6gncrruV3NewI-XQV0ckldTjqNfOTz1V
        Rs-jE-57KAXI3YBIhu-_0YpIDzdk_wBuAk661Svn0GsPQe7m9DoxdzenQu9
        O_soewUhlPzRrTH0EeIqYI715rwI3TYaSzoWBmEPD2fICyj18FF0MPy_SQz
        k3noVUUIzfzLnnJiWy_p63QBCMqjRoSHHdMnI4z9iVpIwJWQ3jO5n_2lC2-
        cSgwjmKsFzDBbQNJc7qMG1N6EssJUwgGJxz1eAUFf0w4YAQ",
  "qi": "J-mG0swR4FTy3atrcQ7dd0hhYn1E9QndN-
        -sDG4EQO0RnFj6wIefCvwIc4
        7hCtVeFnCTPYJNc_JyV-mU-9vlzS5GSNuyR5qdpsMZXUMpEvQcwKt23ffPZ
        YGaqfKyEesmf_Wi8fFcE68H9REQjnniKrXm7w2-IuG_IrVJA9Ox-uU",
  "q": "4hlMYAGa0dvogdK1jnxQ7J_Lqpqi99e-AeoFvoYpMPhthChTzwFZO9lQmUo
        BpMqVQTws_s7vWGmt7ZAB3ywkurf0pV7BD0fweJiUzrWk4KJjxtmP_auuxr
        jvm3s2FUGn6f0wRY9Z8Hj9A7C72DnYCjuZiJQMYCWDsZ8-d-L1a-s",
  "p": "5sd9Er3I2FFT9R-gy84_oakEyCmgw036B_nfYEEOCwpSvi2z7UcIVK3bSEL
        5WCW6BNgB3HDWhq8aYPirwQnqm0K9mX1E-4xM10WWZ-rP3XjYpQeS0Snru5
        LFVWsAzi-FX7BOqBibSAXLdEGXcXa44l08iec_bPD3xduq5V_1YoE",
  "dq": "Nz2PF3XM6bEc4XsluKZO70ErdYdKgdtIJReUR7Rno_tOZpejwlPGBYVW19
        zpAeYtCT82jxroB2XqhLxGeMxEPQpsz2qTKLSe4BgHY2ml2uxSDGdjcsrbb
        NoKUKaN1CuyZszhWl1n0AT_bENl4bJgQj_Fh0UEsQj5YBBUJt5gr_k",
  "dp": "Zc877jirkkLOtyTs2vxyNe9KnMNAmOidlUc2tE_-0gAL4Lpo1hSwKCtKwe
        ZJ-gkqt1hT-dwNx_0Xtg_-NXsadMRMwJnzBMYwYAfjApUkfqABc0yUCJJl3
        KozRCugf1WXkU9GZAH2_x8PUopdNUEa70ISowPRh04HANKX4fkjWAE"
 }
```

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
Specification Document(s): Section 5 of RFC 9126

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 6 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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[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>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC9101] Sakimura, N., Bradley, J., and M. Jones, "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)", RFC 9101, DOI 10.17487/RFC9101, August 2021, <https://www.rfc-editor.org/info/rfc9101>.

10.2. Informative References

[IANA.OAuth.Parameters] IANA, "OAuth Parameters", <http://www.iana.org/assignments/oauth-parameters>.

[OAUTH-SECURITY-TOPICS] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", Work in Progress, Internet-Draft, draft-ietf-oauth-security-topics-18, 13 April 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-18>.

[OAUTH-V2] Hardt, D., Parecki, A., and T. Lodderstedt, "The OAuth 2.1 Authorization Framework", Work in Progress, Internet-Draft, draft-ietf-oauth-v2-1-03, 8 September 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-03>.

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

[OIDC.Disco] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0 incorporating errata set 1", November 2014, <http://openid.net/specs/openid-connect-discovery-1_0.html>.

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <https://www.rfc-editor.org/info/rfc6755>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/info/rfc7523>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <https://www.rfc-editor.org/info/rfc8252>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/info/rfc8707>.

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